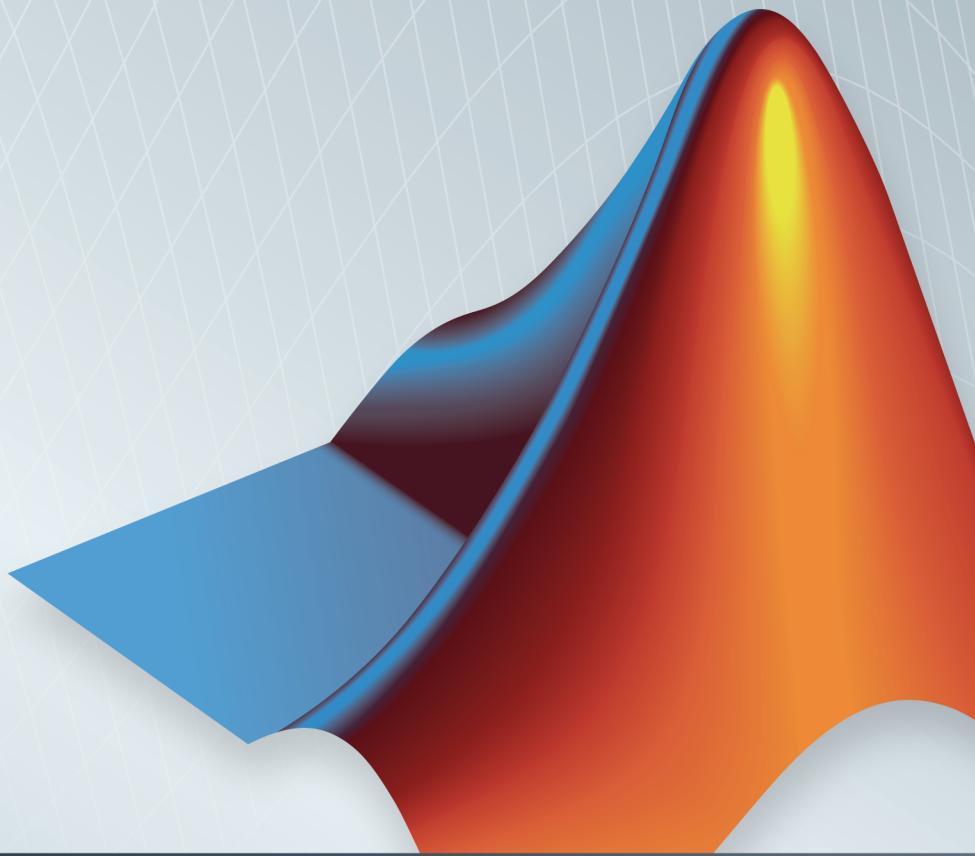
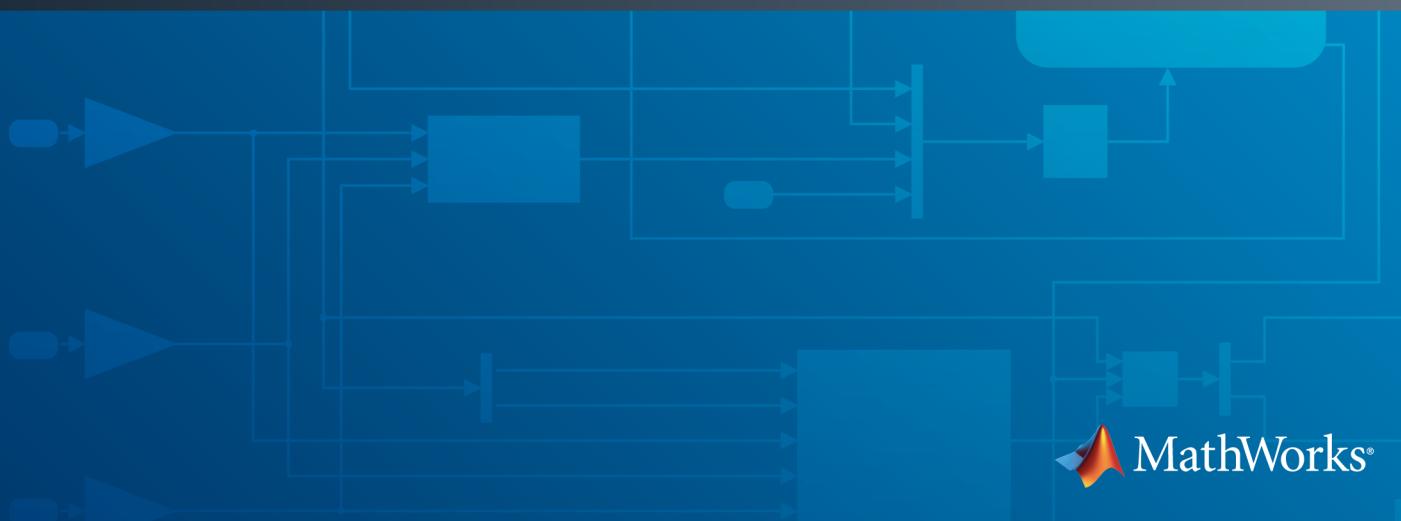


MATLAB® Graphics

R2014b



MATLAB®



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB® Graphics

© COPYRIGHT 1984–2014 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2006	Online only	New for MATLAB® 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB® 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB® 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB® 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB® 7.6 (Release 2008a) This publication was previously part of the Using MATLAB® Graphics User Guide.
October 2008	Online only	Revised for MATLAB® 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB® 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB® 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB® 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB® 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB® 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB® 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB® 7.14 (Release 2012a)
September 2012	Online only	Revised for Version 8.0 (Release 2012b)
March 2013	Online only	Revised for Version 8.1 (Release 2013a)
September 2013	Online only	Revised for Version 8.2 (Release 2013b)
March 2014	Online only	Revised for Version 8.3 (Release 2014a)
October 2014	Online only	Revised for Version 8.4 (Release 2014b)

Plots and Plotting Tools

1

Types of MATLAB Plots	1-2
Two-Dimensional Plotting Functions	1-2
Three-Dimensional Plotting Functions	1-3
Create Graph Using Plots Tab	1-6
Customize Graph Using Plot Tools	1-8
Open Plot Tools	1-8
Customize Objects in Graph	1-9
Control Visibility of Objects in Graph	1-10
Add Annotations to Graph	1-10
Close Plot Tools	1-10
Create Subplots Using Plot Tools	1-11
Create Simple Line Plot and Open Plot Tools	1-11
Create Upper and Lower Subplots	1-11
Add Data to Lower Subplot	1-12
Add New Plot Without Overwriting Existing Plot	1-13

Basic Plotting Commands

2

Create 2-D Graph and Customize Lines	2-2
Create 2-D Line Graph	2-2
Create Graph in New Figure Window	2-3
Plot Multiple Lines	2-4
Colors, Line Styles, and Markers	2-5
Specify Line Style	2-6
Specify Different Line Styles for Multiple Lines	2-6

Specify Line Style and Color	2-7
Specify Line Style, Color, and Markers	2-8
Plot Only Data Points	2-10
Add Title, Axis Labels, and Legend to Graph	2-12
Change Axis Limits of Graph	2-18
Change Tick Marks and Tick Labels of Graph	2-22
Display Grid Lines on 2-D Graph	2-26
Display Major and Minor Grid Lines	2-26
Display Grid Lines in Single Direction	2-29
Change Grid Line Style	2-31
Add Plot to Existing Graph	2-34
Create Figure with Multiple Graphs Using Subplots	2-37
Create Graph with Two y-Axes	2-42
Display Markers at Specific Data Points on Line Graph	2-47
Plot Imaginary and Complex Data	2-49

Data Exploration Tools

3

Ways to Explore Graphical Data	3-2
Introduction	3-2
Types of Tools	3-2
Display Data Values Interactively	3-4
What Is a Data Cursor?	3-4
Enabling Data Cursor Mode	3-4
Display Style — Datatip or Cursor Window	3-12
Selection Style — Select Data Points or Interpolate Points on Graph	3-13
Exporting Data Value to Workspace Variable	3-13

Zooming in Graphs	3-15
Zooming in 2-D and 3-D	3-15
Zooming in 2-D Views	3-15
Panning — Shifting Your View of the Graph	3-17
Rotate in 3-D	3-18
Enabling 3-D Rotation	3-18
Selecting Predefined Views	3-18
Rotation Style for Complex Graphs	3-19
Undo/Redo — Eliminating Mistakes	3-21

Annotating Graphs

4

Change Mapping of Data Values into the Colormap	4-2
Change Colorbar Width	4-5
Include Subset of Objects in Graph Legend	4-8
Display One Legend Entry for Group of Objects	4-11
Specify Legend Descriptions During Line Creation	4-14
Add Text to Specific Points on Graph	4-17
Add Text to Three Data Points on Graph	4-17
Determine Minimum and Maximum Points and Add Text ..	4-20
Include Variable Values in Graph Text	4-23
Include Variable Value in Axis Label	4-23
Include Loop Variable Value in Graph Title	4-24
Text with Mathematical Expression Using LaTeX	4-26
Add Text with Integral Expression to Graph	4-26
Add Text with Summation Symbol to Graph	4-28
Text with Greek Letters and Special Characters	4-31
Include Greek Letters in Graph Text	4-31
Include Superscripts and Annotations in Graph Text	4-32

Add Annotations to Graph Interactively	4-35
Add Annotations	4-35
Pin Annotations to Points in Graph	4-36
Add Text to Graph Interactively	4-38
Add Title and Axis Labels	4-38
Add Legend	4-40
Add Annotations to Graph	4-42
Add Colorbar to Graph Interactively	4-44
Add Colorbar	4-44
Change Colorbar Location	4-45
Change Colormap	4-46
Align Objects in Graph Using Alignment Tools	4-48

Creating Specialized Plots

5

Types of Bar Graphs	5-2
Modify Baseline of Bar Graph	5-9
Overlay Bar Graphs	5-13
Overlay Line Plot on Bar Graph Using Different Y-Axes ..	5-16
Color 3-D Bars by Height	5-20
Compare Data Sets Using Overlayed Area Graphs	5-23
Offset Pie Slice with Greatest Contribution	5-28
Add Legend to Pie Chart	5-30
Label Pie Chart With Text and Percent Values	5-33
Create Pie Chart	5-33
Store Precalculated Percent Values	5-34
Combine Percent Values and Additional Text	5-35
Determine Horizontal Distance to Move Each Label	5-36

Position New Label	5-37
Data Cursors with Histograms	5-39
Combine Stem Plot and Line Plot	5-41
Overlay Stairstep Plot and Line Plot	5-46
Display Quiver Plot Over Contour Plot	5-49
Projectile Path Over Time	5-51
Label Contour Plot Levels	5-53
Change Fill Colors for Contour Plot	5-55
Highlight Specific Contour Levels	5-57
Contour Plot in Polar Coordinates	5-60
Animation Techniques	5-66
Updating the Screen	5-66
Optimizing Performance	5-66
Trace Marker Along Line	5-68
Move Group of Objects Along Line	5-71
Animate Graphics Object	5-75
Line Animations	5-79
Record Animation for Playback	5-82
Record and Play Back Movie	5-82
Capture Entire Figure for Movie	5-83

Working with Images in MATLAB Graphics	6-2
What Is Image Data?	6-2
Supported Image Formats	6-3
Functions for Reading, Writing, and Displaying Images	6-4
Image Types	6-5
Indexed Images	6-5
Intensity Images	6-7
RGB (Truecolor) Images	6-8
8-Bit and 16-Bit Images	6-10
Indexed Images	6-10
Intensity Images	6-11
RGB Images	6-11
Mathematical Operations Support for uint8 and uint16	6-12
Other 8-Bit and 16-Bit Array Support	6-12
Converting an 8-Bit RGB Image to Grayscale	6-13
Summary of Image Types and Numeric Classes	6-16
Read, Write, and Query Image Files	6-18
Working with Image Formats	6-18
Reading a Graphics Image	6-19
Writing a Graphics Image	6-19
Subsetting a Graphics Image (Cropping)	6-20
Obtaining Information About Graphics Files	6-21
Displaying Graphics Images	6-22
Image Types and Display Methods	6-22
Controlling Aspect Ratio and Display Size	6-24
The Image Object and Its Properties	6-27
Image CData	6-27
Image CDataMapping	6-27
XData and YData	6-28
Add Text to Image Data	6-30
Additional Techniques for Fast Image Updating	6-32
Printing Images	6-34

Printing and Saving**7**

Overview of Printing and Exporting	7-2
Print and Export Operations	7-2
Graphical User Interfaces	7-2
Command Line Interface	7-3
Specifying Parameters and Options	7-4
Default Settings and How to Change Them	7-5
Bitmap vs. Vector Formats	7-8
Choosing a Format	7-8
How Renderer Affects Format	7-9
Controlling Graphics Output	7-9
How to Print or Export	7-10
Using Print Preview	7-10
Printing a Figure	7-13
Printing to a File	7-15
Exporting to a File	7-17
Exporting to the Windows or Macintosh Clipboard	7-25
Printing and Exporting Use Cases	7-30
Printing a Figure at Screen Size	7-30
Printing with a Specific Paper Size	7-31
Printing a Centered Figure	7-31
Exporting in a Specific Graphics Format	7-32
PostScript and PDF Font Translations	7-33
Exporting in EPS Format with a TIFF Preview	7-34
Exporting a Figure to the Clipboard	7-34
Change Figure Settings	7-37
Parameters that Affect Printing	7-37
Selecting the Figure	7-39
Selecting the Printer	7-39
Setting the Figure Size and Position	7-41
Setting the Paper Size or Type	7-44
Setting the Paper Orientation	7-46

Selecting a Renderer	7-48
Setting the Resolution	7-50
Setting the Axes Ticks and Limits	7-52
Setting the Background Color	7-54
Setting Line and Text Characteristics	7-55
Setting the Line and Text Color	7-58
Specifying a Colorspace for Printing and Exporting	7-61
Excluding User Interface Controls from Printed Output	7-63
Producing Uncropped Figures	7-64
Troubleshooting	7-65
Introduction	7-65
Common Problems	7-65
Printing Problems	7-66
Exporting Problems	7-69
General Problems	7-72
Saving Figures	7-75
Saving and Loading Graphs	7-75
FIG-File Format	7-76
Saving Figures From the Menu	7-76
Saving to a Different Format — Exporting Figures	7-77
Printing Figures	7-78
Generating a MATLAB File to Recreate a Graph	7-79

Axes Active Position

8

Axes Resize to Accommodate Titles and Labels	8-2
Axes Layout	8-2
Properties Controlling Axes Size	8-2
Using OuterPosition as the ActivePositionProperty	8-5
ActivePositionProperty = OuterPosition	8-5
ActivePositionProperty = Position	8-6
Axes Resizing in Subplots	8-7

Controlling Graphics Output

9

Control Graph Display	9-2
What You Can Control	9-2
Targeting Specific Figures and Axes	9-2
Prepare Figures and Axes for Graphs	9-5
Behavior of MATLAB Plotting Functions	9-5
How the NextPlot Properties Control Behavior	9-5
Control Behavior of User-Written Plotting Functions	9-7
Use newplot to Control Plotting	9-9
Responding to Hold State	9-12
Prevent Access to Figures and Axes	9-14
Why Prevent Access	9-14
How to Prevent Access	9-14

Default Values

10

Default Property Values	10-2
Predefined Values for Properties	10-2
Specify Default Values	10-2
Where in Hierarchy to Define Default	10-3
List Default Values	10-3
Set Properties to the Current Default	10-4
Remove Default Values	10-4
Set Properties to Factory-Defined Values	10-4
List Factory-Defined Property Values	10-4
Reserved Words	10-5
Default Values for Automatically Calculated Properties . .	10-6
What Are Automatically Calculated Properties	10-6
Default Values for Automatically Calculated Properties . .	10-6
How MATLAB Finds Default Values	10-8

Factory-Defined Property Values	10-9
Define Default Line Styles	10-10
Multilevel Default Values	10-12

Graphics Object Callbacks

11

Callbacks — Programmed Response to User Action	11-2
What Are Callbacks?	11-2
Window Callbacks	11-2
Callback Definition	11-4
Ways to Specify Callbacks	11-4
Callback Function Syntax	11-4
Related Information	11-5
Define a Callback as a Default	11-6
Button Down Callback Function	11-7
When to Use a Button Down Callback	11-7
How to Define a Button Down Callback	11-7
Define a Context Menu	11-9
When to Use a Context Menu	11-9
How to Define a Context Menu	11-9
Define an Object Creation Callback	11-11
Related Information	11-12
Define an Object Deletion Callback	11-13
Capturing Mouse Clicks	11-14
Properties That Control Response to Mouse Clicks	11-14
Combinations of PickablePart/HitTest Values	11-14
Passing Mouse Click Up the Hierarchy	11-15
Pass Mouse Click to Group Parent	11-18
Objective and Design	11-18
Object Hierarchy and Key Properties	11-18

MATLAB Code	11-19
-------------------	-------

Pass Mouse Click to Obscured Object	11-21
---	-------

Graphics Objects

12

Graphics Objects	12-2
MATLAB Graphics Objects	12-2
Graphs Are Composed of Specific Objects	12-2
Organization of Graphics Objects	12-2
Features Controlled by Graphics Objects	12-7
Purpose of Graphics Objects	12-7
Figures	12-7
Axes	12-8
Objects That Represent Data	12-9
Group Objects	12-10
Annotation Objects	12-11

Group Objects

13

Object Groups	13-2
Create Object Groups	13-3
Parent Specification	13-4
Visible and Selected Properties of Group Children	13-4
Transforms Supported by hgtransform	13-5
Transforming Objects	13-5
Rotation	13-5
Translation	13-6
Scaling	13-6
The Default Transform	13-7
Disallowable Transforms: Perspective	13-7
Disallowable Transforms: Shear	13-7

Absolute vs. Relative Transforms	13-8
Combining Transforms into One Matrix	13-8
Undoing Transform Operations	13-9
Rotate About an Arbitrary Axis	13-10
Translate to Origin Before Rotating	13-10
Rotate Surface	13-10
Nest Transforms for Complex Movements	13-14

Control Legend Content

14

Control Legend Content	14-2
Properties for Controlling Legend Content	14-2
Updating a Legend	14-3

Working with Graphics Objects

15

Graphics Object Handles	15-2
What You Can Do with Handles	15-2
What You Cannot Do with Handles	15-3
Preallocate Arrays	15-4
Test for Valid Handle	15-5
Handles in Logical Expressions	15-6
If Handle Is Valid	15-6
If Result Is Empty	15-6
If Handles Are Equal	15-7
Graphics Arrays	15-9

Special Object Identifiers	16-2
Getting Handles to Special Objects	16-2
The Current Figure, Axes, and Object	16-2
Callback Object and Callback Figure	16-4
Find Objects	16-5
Find Objects with Specific Property Values	16-5
Find Text by String Property	16-5
Use Regular Expressions with <code>findobj</code>	16-7
Limit Scope of Search	16-9
Copy Objects	16-11
Copying Objects with <code>copyobj</code>	16-11
Copy Single Object to Multiple Destinations.	16-11
Copying Multiple Objects	16-12
Delete Graphics Objects	16-14
How to Delete Objects	16-14
Handles to Deleted Objects	16-15

Optimize Performance of Graphics Programs

Finding Code Bottlenecks	17-2
What Affects Code Execution Speed	17-4
Potential Bottlenecks	17-4
How to Improve Performance	17-4
Judicious Object Creation	17-6
Object Overhead	17-6
Do Not Create Unnecessary Objects	17-6
Use NaNs to Simulate Multiple Lines	17-7
Modify Data Instead of Creating New Objects	17-7

Avoid Repeated Searches for Objects	17-8
Limit Scope of Search	17-8
Screen Updates	17-10
MATLAB Graphics System	17-10
Managing Updates	17-11
Getting and Setting Properties	17-12
Automatically Calculated Properties	17-12
Inefficient Cycles of Sets and Gets	17-13
Changing Text Extent to Rotate Labels	17-14
Avoid Updating Static Data	17-15
Segmenting Data to Reduce Update Times	17-15
Animating Line Graphs	17-17
Interactive Lines with Markers	17-17
Transforming Objects Efficiently	17-18
Use Low-Level Functions for Speed	17-19
Using drawnow Efficiently	17-20
What Does drawnow Do?	17-20
How to Use drawnow	17-21
Achieve a Specific Frame Rate	17-21
System Requirements for Graphics	17-23
Recommended System Requirements	17-23
Upgrade Your Graphics Drivers	17-23
Features with OpenGL Requirements	17-24
Workarounds for Older Graphics Hardware	17-25

Access Property Values	18-2
Object Properties and Dot Notation	18-2
Graphics Object Variables Are Handles	18-4

Listing Object Properties	18-6
Modify Properties with set and get	18-6
Multi Object/Property Operations	18-7

Using Axes Properties

19

Axes Aspect Ratio	19-2
3-D Views	19-3
Additional Commands for Setting Aspect Ratio	19-5
Display Text Outside Axes	19-6
Overlay Axes with Different Sizes	19-9
Graph with Multiple x-Axes and y-Axes	19-12
Automatically Calculated Properties	19-16
Line Styles Used for Plotting — LineStyleOrder	19-20

Plots and Plotting Tools

- “Types of MATLAB Plots” on page 1-2
- “Create Graph Using Plots Tab” on page 1-6
- “Customize Graph Using Plot Tools” on page 1-8
- “Create Subplots Using Plot Tools” on page 1-11

Types of MATLAB Plots

In this section...

[“Two-Dimensional Plotting Functions” on page 1-2](#)

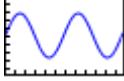
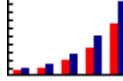
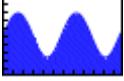
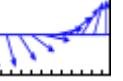
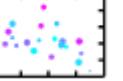
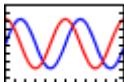
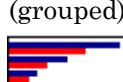
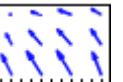
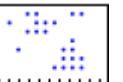
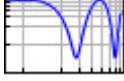
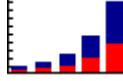
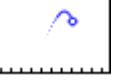
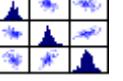
[“Three-Dimensional Plotting Functions” on page 1-3](#)

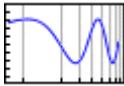
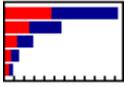
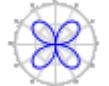
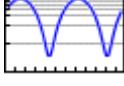
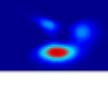
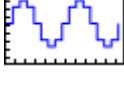
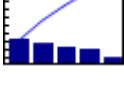
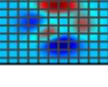
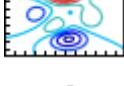
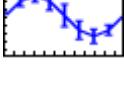
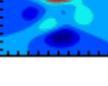
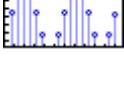
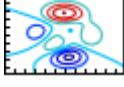
MATLAB® offers a variety of data plotting functions plus a set of GUI tools to create and modify graphic displays. The following two tables classify and illustrate the kinds of plots you can create with MATLAB. They include line, bar, area, direction and vector field, radial, and scatter graphs.

When you execute a plotting function, MATLAB clears and replaces the current graph, if one exists. The function resets axis limits and other properties so that each graph is displayed appropriately.

Two-Dimensional Plotting Functions

This table shows MATLAB 2-D plotting functions. Click any icon to see the documentation for that function.

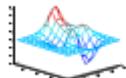
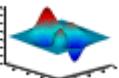
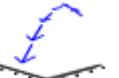
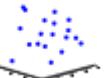
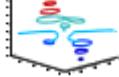
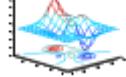
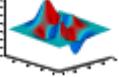
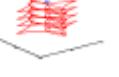
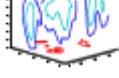
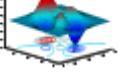
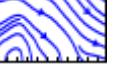
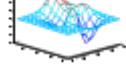
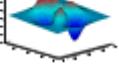
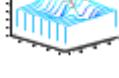
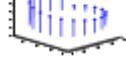
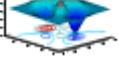
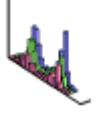
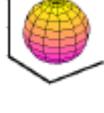
Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
plot 	bar (grouped) 	area 	feather 	polar 	scatter 
plotyy 	barh (grouped) 	pie 	quiver 	rose 	spy 
loglog 	bar (stacked) 	fill 	comet 	compass 	plotmatrix 

Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
<code>semilogx</code>	<code>barr</code> (stacked)	<code>contourf</code>		<code>ezpolar</code>	
					
<code>semilogy</code>	<code>hist</code>	<code>image</code>			
					
<code>stairs</code>	<code>pareto</code>	<code>pcolor</code>			
					
<code>contour</code>	<code>errorbar</code>	<code>ezcontourf</code>			
					
<code>ezplot</code>	<code>stem</code>				
					
<code>ezcontour</code>					
					

Three-Dimensional Plotting Functions

This table shows MATLAB 3-D and volume plotting functions. Some functions generate 3-D data (`cylinder`, `ellipsoid`, `sphere`) that you can use to generate geometric

shapes on which you can superimpose your data. Click any picture in the table to see the documentation for that function.

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
<code>plot3</code>	<code>mesh</code>	<code>pie3</code>	<code>surf</code>	<code>quiver3</code>	<code>scatter3</code>
					
<code>contour3</code>	<code>meshc</code>	<code>fill3</code>	<code>surfl</code>	<code>comet3</code>	<code>coneplot</code>
					
<code>contourslice</code>	<code>meshz</code>	<code>patch</code>	<code>surf c</code>	<code>streamslice</code>	<code>streamline</code>
					
<code>ezplot3</code>	<code>ezmesh</code>	<code>cylinder</code>	<code>ezsurf</code>		<code>streamribbon</code>
					
<code>waterfall</code>	<code>stem3</code>	<code>ellipsoid</code>	<code>ezsurf c</code>		<code>streamtube</code>
					
	<code>bar3</code>	<code>sphere</code>			
					
	<code>bar3h</code>				

Line Graphs

**Mesh Graphs
and Bar
Graphs**

**Area
Graphs and
Constructive
Objects**

**Surface
Graphs**

**Direction
Graphs**

**Volumetric
Graphs**



Create Graph Using Plots Tab

This example shows how to create a 2-D line plot interactively using the **Plots** tab in the MATLAB toolbar. The **Plots** tab shows a gallery of supported plot types based on the variables you select from your workspace.

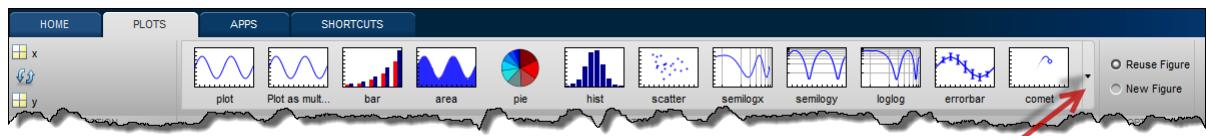
- 1 In the Command Window, define x as a vector of 50 linearly spaced values between 1 and 10. Define y as the sine function.

```
x = linspace(1,10,50);
y = sin(x);
```

- 2 In the Workspace panel in the MATLAB desktop, select the variables to plot. Use **Ctrl** + click to select multiple variables.

Workspace		
Name	Value	Min
x	1x50 double	1
y	1x50 double	-0.9992

- 3 Select the 2-D line plot from the gallery on the **Plots** tab. For additional plot types, click the arrow at the end of the gallery.



MATLAB creates the plot and displays the plotting commands at the command line.

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

See Also

[linspace](#) | [plot](#) | [sin](#)

Related Examples

- “Customize Graph Using Plot Tools” on page 1-8

Customize Graph Using Plot Tools

To customize a graph interactively you can use the plot tools. The plot tools interface consists of three different panels: the Property Editor, the Plot Browser, and the Figure Palette. Use these panels to add different types of customizations to your graph.

In this section...

[“Open Plot Tools” on page 1-8](#)

[“Customize Objects in Graph” on page 1-9](#)

[“Control Visibility of Objects in Graph” on page 1-10](#)

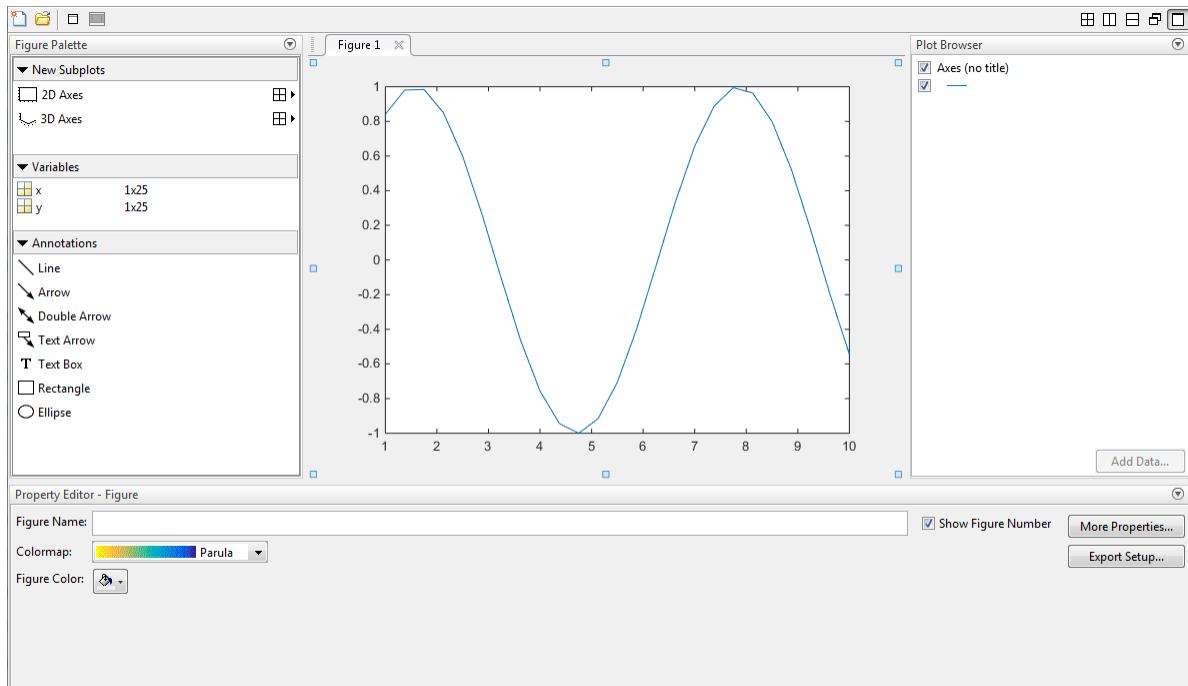
[“Add Annotations to Graph” on page 1-10](#)

[“Close Plot Tools” on page 1-10](#)

Open Plot Tools

To open the plot tools use the `plottools` command or click the Show Plot Tools icon  in the figure window. For example, define variables `x` and `y` in the Command Window, create a line plot and open the plot tools.

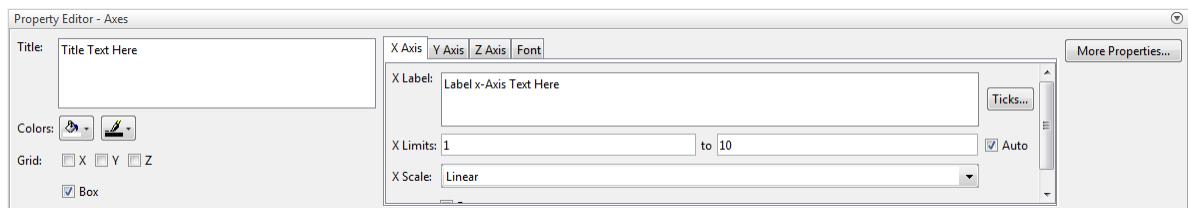
```
x = linspace(1,10,25);  
y = sin(x);  
plot(x,y)  
plottools
```



MATLAB creates a plot of y versus x and opens the plot tools.

Customize Objects in Graph

To customize objects in your graph, you can set their properties using the Property Editor. For example, click the axes to display a subset of common axes properties in the Property Editor. Specify a title and an x -axis label by typing text in the empty fields.



Click other objects in the graph to display and edit a subset of their common properties in the Property Editor. Access and edit more object properties by clicking **More Properties** to open the Property Inspector.

Note: You cannot use the Property Editor to access properties of objects that you cannot click, such as a light or a uicontextmenu. You must store the object handles and use the `inspect` command.

Control Visibility of Objects in Graph

To control the visibility of objects in the graph, you can use the Plot Browser. The Plot Browser lists all the axes and plots in the figure. The check box next to each object controls the object's visibility.

- Hide an object without deleting it by deselecting its box in the Plot Browser.
- Delete an object by right-clicking it and selecting **Delete**.

Add Annotations to Graph

To add annotations to the graph, such as arrows and text, you can use the Annotations panel in the Figure Palette.

Close Plot Tools

To remove the plot tools from the figure, you can use the Hide Plot Tools icon  , or type `plottools('off')` in the Command Window.

Use the **View** menu to show or hide specific plot tools panels. If you change the layout of the plot tools, then the layout persists the next time you open the plot tools.

See Also

`annotation` | `figurepalette` | `inspect` | `plot` | `plotbrowser` | `plottools` | `propertyeditor`

Related Examples

- “Create Subplots Using Plot Tools” on page 1-11
- “Generating a MATLAB File to Recreate a Graph” on page 7-79

Create Subplots Using Plot Tools

This example shows how to create a figure with multiple graphs interactively and add different types of plots to each graph using the plot tools.

In this section...

[“Create Simple Line Plot and Open Plot Tools” on page 1-11](#)

[“Create Upper and Lower Subplots” on page 1-11](#)

[“Add Data to Lower Subplot” on page 1-12](#)

[“Add New Plot Without Overwriting Existing Plot” on page 1-13](#)

Create Simple Line Plot and Open Plot Tools

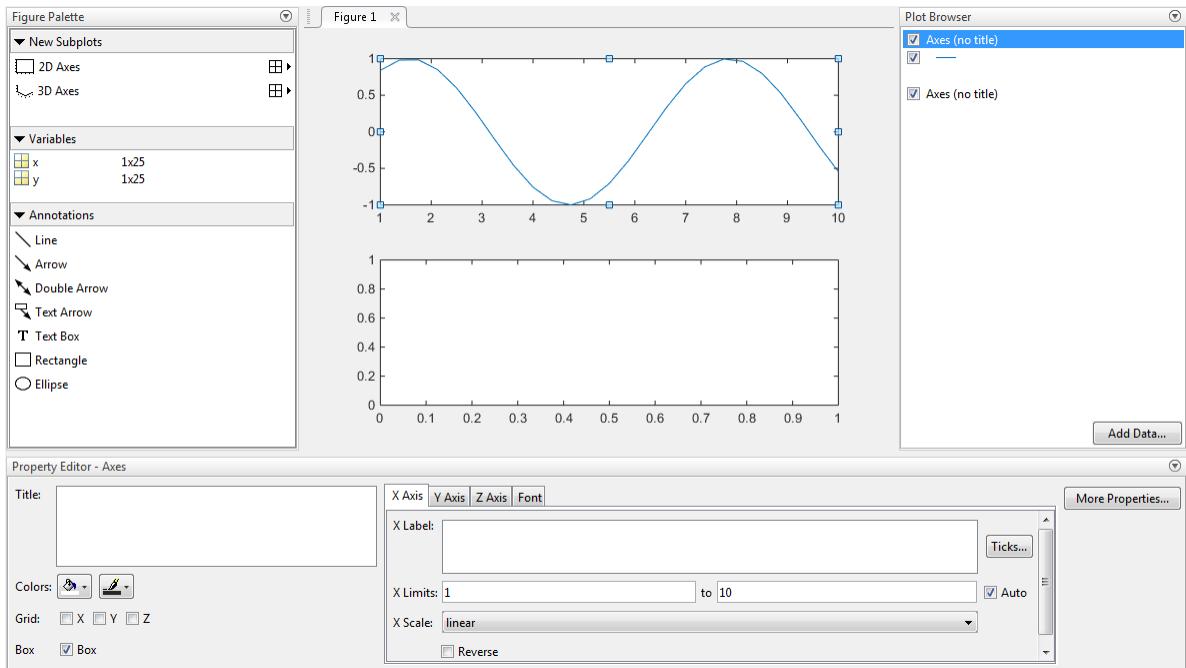
Define variables `x` and `y` in the Command Window and create a line plot using the `plot` function. Open the plot tools using the `plottools` command or by clicking the Show Plot Tools icon  in the figure window.

```
x = linspace(1,10,25);  
y = sin(x);  
plot(x,y)  
plottools
```

MATLAB creates a plot of `y` versus `x` and opens the plot tools.

Create Upper and Lower Subplots

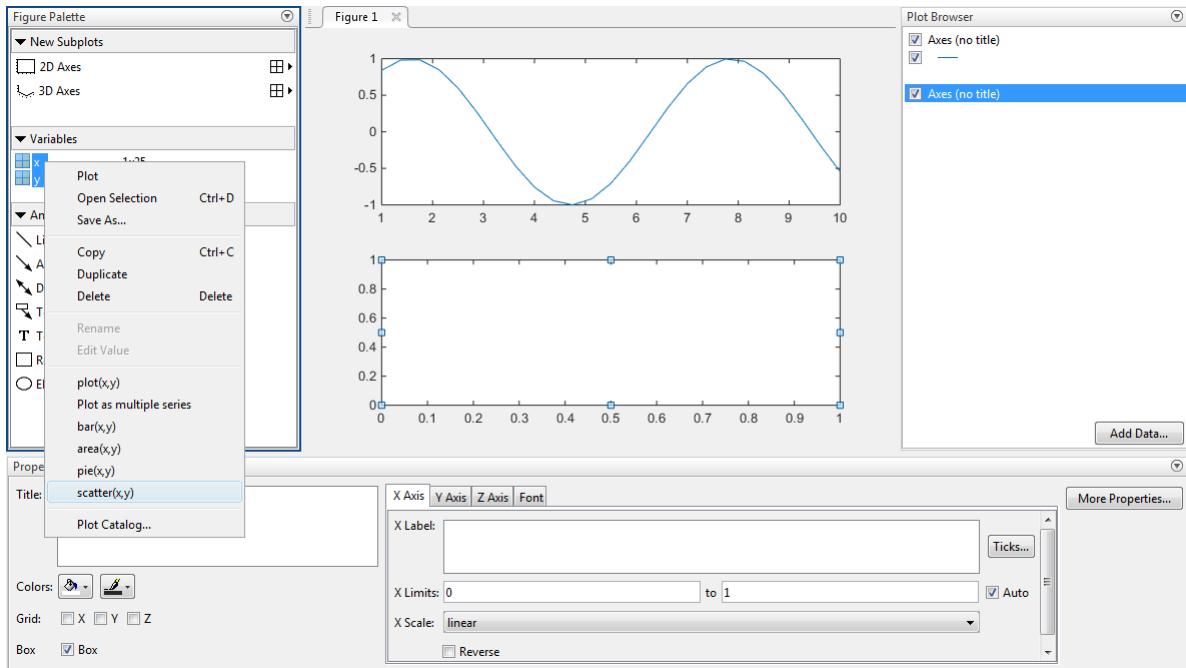
Create upper and lower subplots using the Figure Palette panel in the plot tools. Choose a subplot layout for two horizontal graphs using the 2-D grid icon .



Add Data to Lower Subplot

Create a scatter plot of y versus x in the lower subplot using the Figure Palette.

- 1 Click the lower subplot axes to make it the current axes.
- 2 Select x and y in the Variables panel of the Figure Palette. Select multiple variables using **Ctrl + click**.
- 3 Right-click one of the variables to display a context menu containing a list of possible plot types based on the variables selected.



- 4 Select `scatter(x,y)` from the menu. (The **Plot Catalog** menu option lists additional plot types.)

MATLAB creates a scatter plot in the lower subplot and displays the commands used to create the plot in the Command Window.

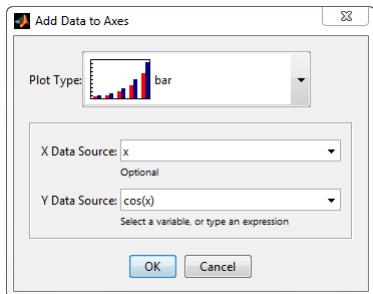
```
scatter(x,y)
```

Note: Adding a plot to an axes using the Variables panel overwrites existing plots in that axes.

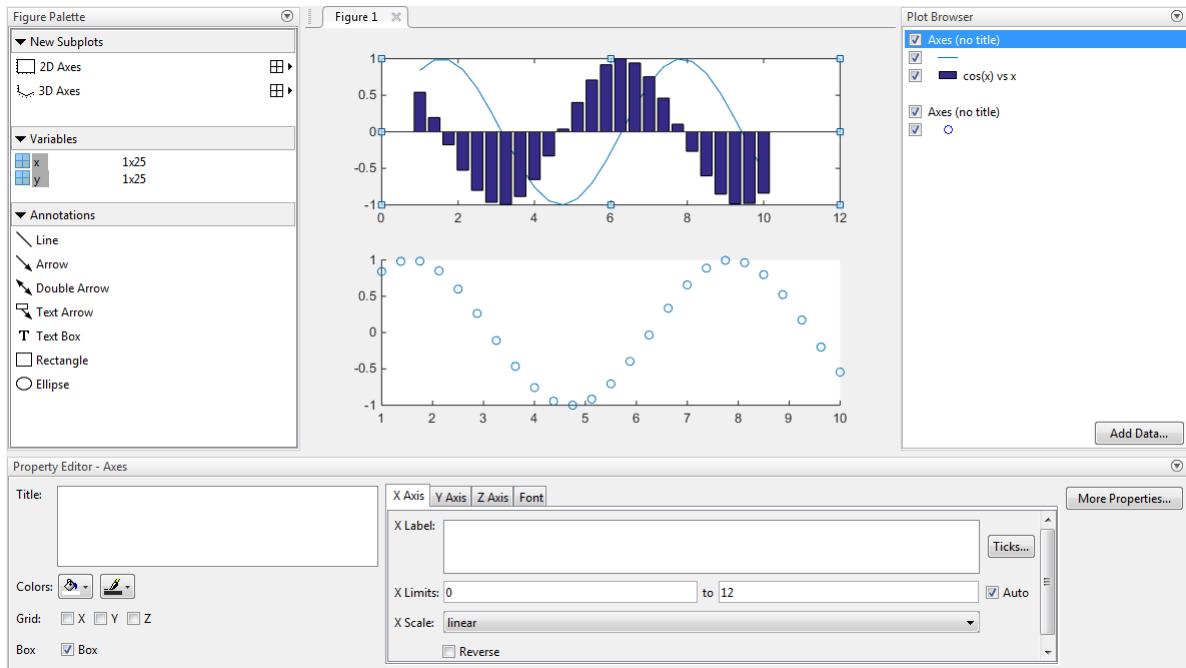
Add New Plot Without Overwriting Existing Plot

Add a bar graph of $\cos(x)$ versus x to the upper subplot without erasing the existing line plot. Use the **Add Data** option in the Plot Browser.

- 1 Open a dialog box by clicking the upper subplot, and then click the **Add Data** button at the bottom of the Plot Browser.
- 2 Use the drop-down menu to select a bar graph as the plot type.
- 3 Specify the variables to plot by setting the **X Data Source** and **Y Data Source** fields. Use the drop-down menu to specify **X Data Source** as the variable x . Since $\cos(x)$ is not defined as a variable, type this expression into the empty field next to **Y Data Source**.



- 4 Click **OK**. MATLAB adds a bar graph to the upper subplot.



See Also

[bar](#) | [figurepalette](#) | [plot](#) | [plottools](#) | [propertyeditor](#) | [scatter](#) | [subplot](#)

Related Examples

- “Add Colorbar to Graph Interactively” on page 4-44
- “Customize Graph Using Plot Tools” on page 1-8
- “Generating a MATLAB File to Recreate a Graph” on page 7-79

Basic Plotting Commands

- “Create 2-D Graph and Customize Lines” on page 2-2
- “Add Title, Axis Labels, and Legend to Graph” on page 2-12
- “Change Axis Limits of Graph” on page 2-18
- “Change Tick Marks and Tick Labels of Graph” on page 2-22
- “Display Grid Lines on 2-D Graph” on page 2-26
- “Add Plot to Existing Graph” on page 2-34
- “Create Figure with Multiple Graphs Using Subplots” on page 2-37
- “Create Graph with Two y-Axes” on page 2-42
- “Display Markers at Specific Data Points on Line Graph” on page 2-47
- “Plot Imaginary and Complex Data” on page 2-49

Create 2-D Graph and Customize Lines

In this section...

- “Create 2-D Line Graph” on page 2-2
- “Create Graph in New Figure Window” on page 2-3
- “Plot Multiple Lines” on page 2-4
- “Colors, Line Styles, and Markers” on page 2-5
- “Specify Line Style” on page 2-6
- “Specify Different Line Styles for Multiple Lines” on page 2-6
- “Specify Line Style and Color” on page 2-7
- “Specify Line Style, Color, and Markers” on page 2-8
- “Plot Only Data Points” on page 2-10

Create 2-D Line Graph

This example shows how to create a simple line graph. Use the `linspace` function to define `x` as a vector of 100 linearly spaced values between 0 and 2π .

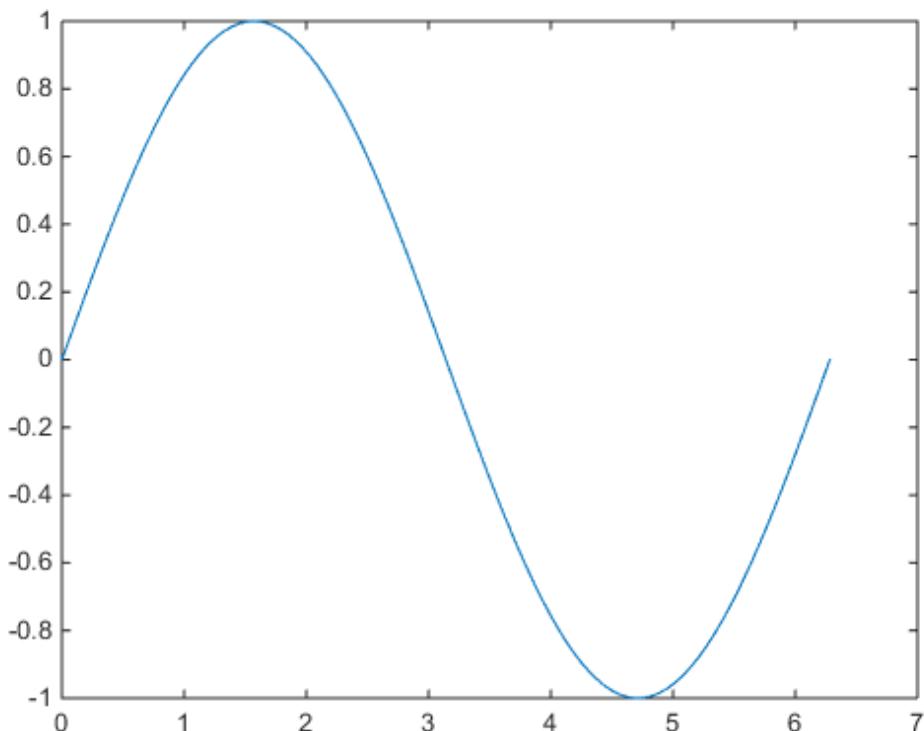
```
x = linspace(0,2*pi,100);
```

Define `y` as the sine function evaluated at the values in `x`.

```
y = sin(x);
```

Plot `y` versus the corresponding values in `x`.

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



Create Graph in New Figure Window

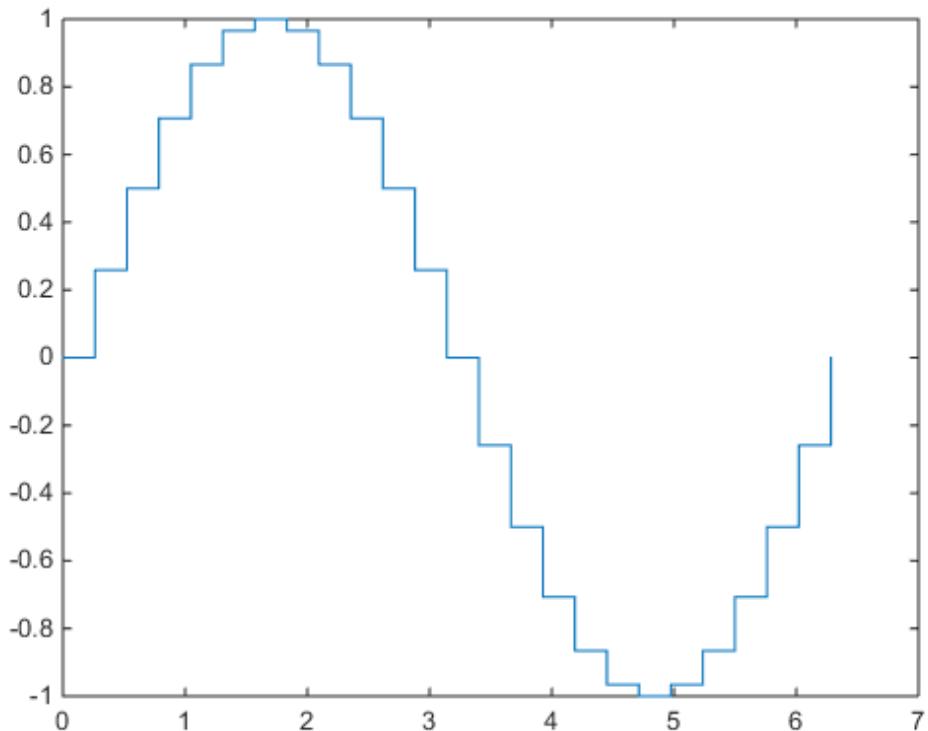
This example shows how to create a graph in a new figure window, instead of plotting into the current figure.

Define x and y .

```
x = linspace(0,2*pi,25);
y = sin(x);
```

Create a staircase plot of y versus x . Open a new figure window using the `figure` command. If you do not open a new figure window, then by default, MATLAB® clears existing graphs and plots into the current figure.

```
figure % new figure window  
stairs(x,y)
```



Plot Multiple Lines

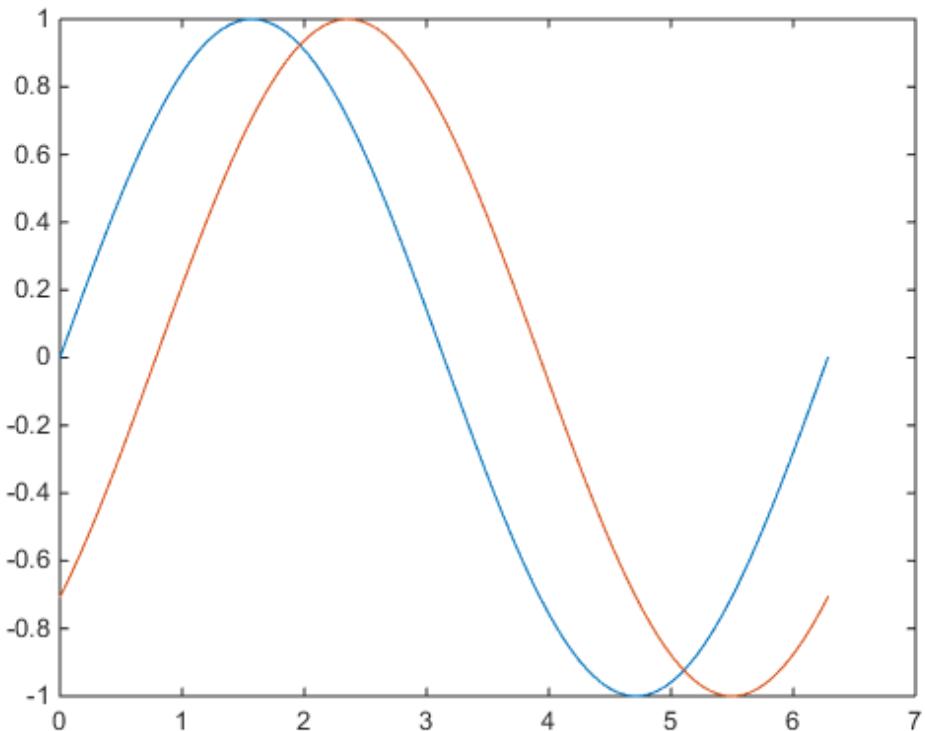
This example shows how to plot more than one line by passing multiple x, y pairs to the `plot` function.

Define y_1 and y_2 as sine waves with a phase shift.

```
x = linspace(0,2*pi,100);  
y1 = sin(x);  
y2 = sin(x-pi/4);
```

Plot the lines.

```
figure  
plot(x,y1,x,y2)
```



`plot` cycles through a predefined list of line colors.

Colors, Line Styles, and Markers

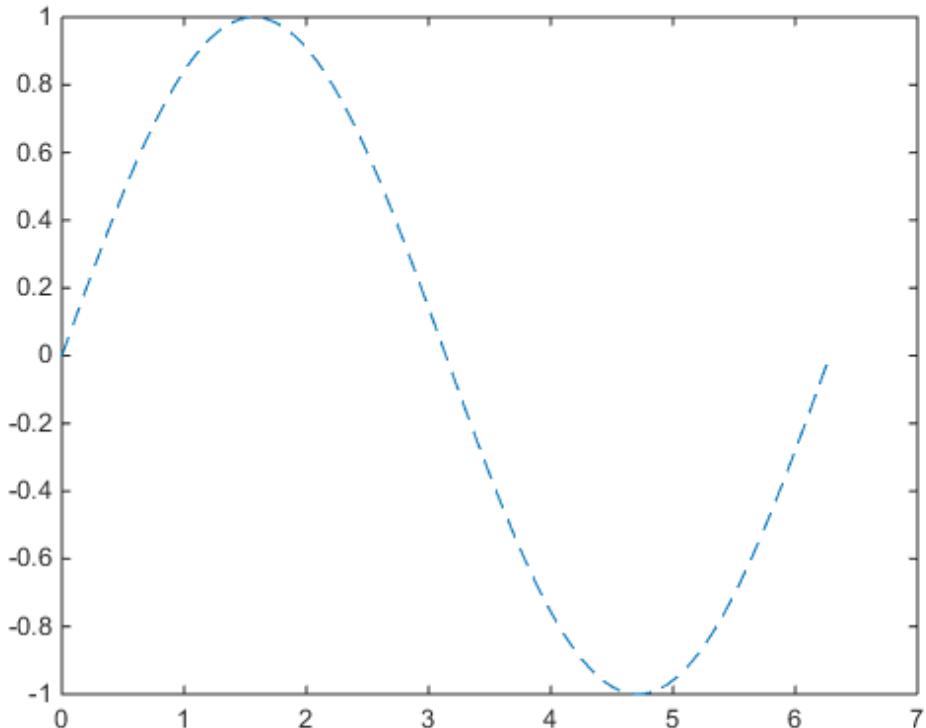
To change the line color, line style, and marker type, add a line specification string to the `x,y` pair. For example, adding the string, '`g:*`', plots a green dotted line with star markers. You can omit one or more options from the line specification, such as '`g:`' for a green dotted line with no markers. To change just the line style, specify only a line style option, such as '`--`' for a dashed line.

For more information, see [LineSpec \(Line Specification\)](#).

Specify Line Style

This example shows how to create a plot using a dashed line. Add the optional line specification string, '`'--'`', to the `x,y` pair.

```
x = linspace(0,2*pi,100);  
y = sin(x);  
  
figure  
plot(x,y, '--')
```



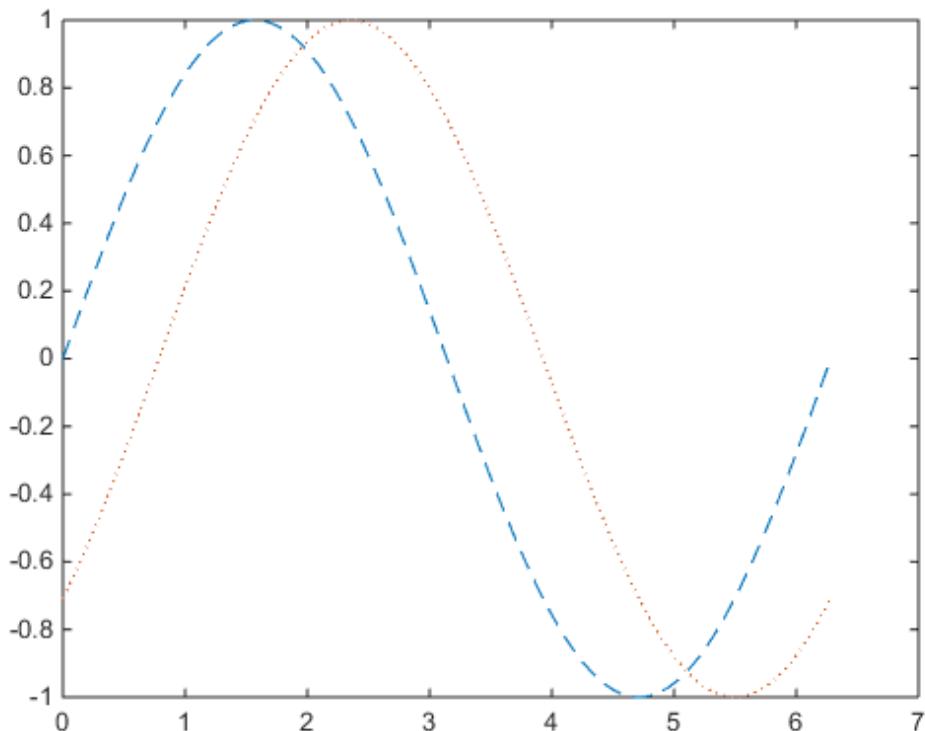
Specify Different Line Styles for Multiple Lines

This example shows how to plot two sine waves with different line styles by adding a line specification string to each x, y pair.

Plot the first sine wave with a dashed line using ' $--$ '. Plot the second sine wave with a dotted line using ' $:$ '.

```
x = linspace(0,2*pi,100);
y1 = sin(x);
y2 = sin(x-pi/4);

figure
plot(x,y1,'--',x,y2,:)
```



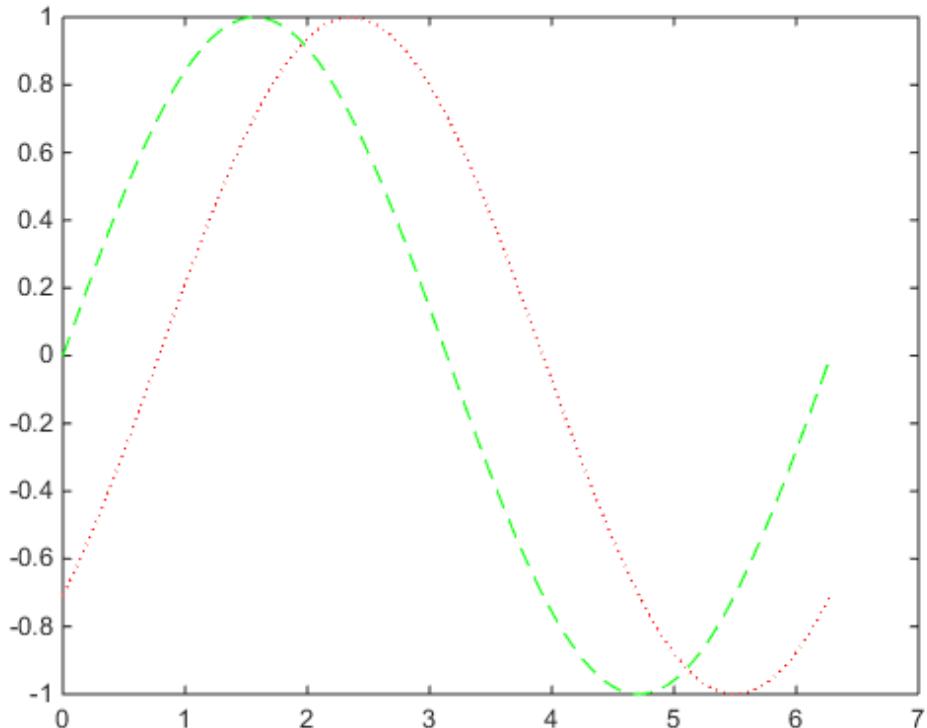
Specify Line Style and Color

This example shows how to specify the line styles and line colors for a plot.

Plot a sine wave with a green dashed line using '`--g`'. Plot a second sine wave with a red dotted line using '`:r`'. The elements of the line specification strings can appear in any order.

```
x = linspace(0,2*pi,100);
y1 = sin(x);
y2 = sin(x-pi/4);

figure
plot(x,y1,'--g',x,y2,:r')
```



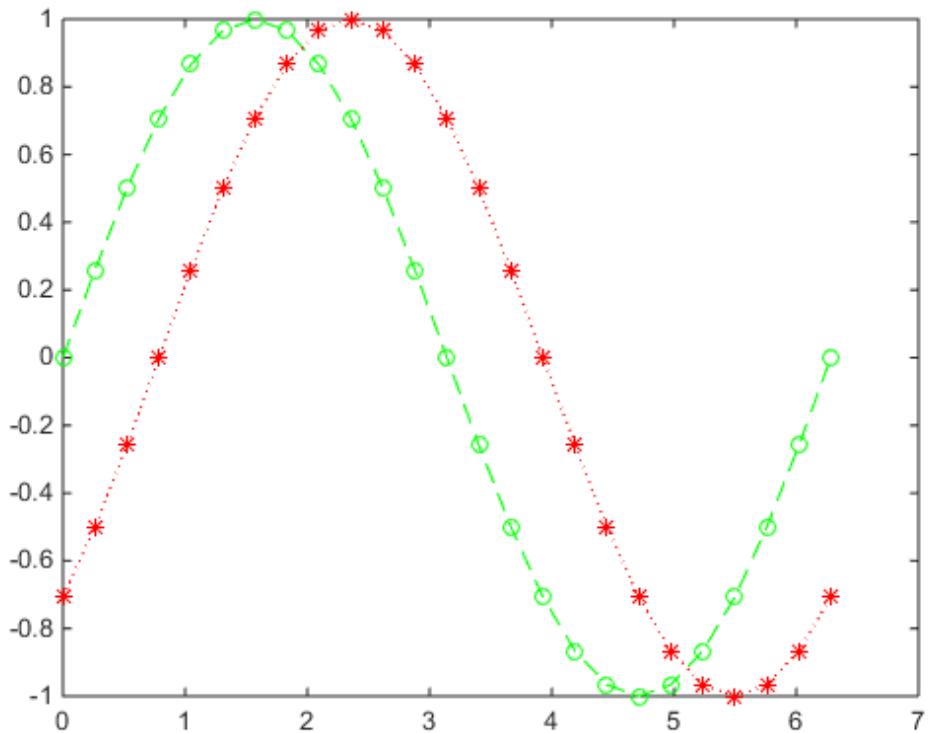
Specify Line Style, Color, and Markers

This example shows how to specify the line style, color, and markers for two sine waves. If you specify a marker type, then `plot` adds a marker to each data point.

Define `x` as 25 linearly spaced values between 0 and 2π . Plot the first sine wave with a green dashed line and circle markers using '`--go`'. Plot the second sine wave with a red dotted line and star markers using '`:r*`'.

```
x = linspace(0,2*pi,25);
y1 = sin(x);
y2 = sin(x-pi/4);

figure
plot(x,y1,'--go',x,y2,:r*)
```



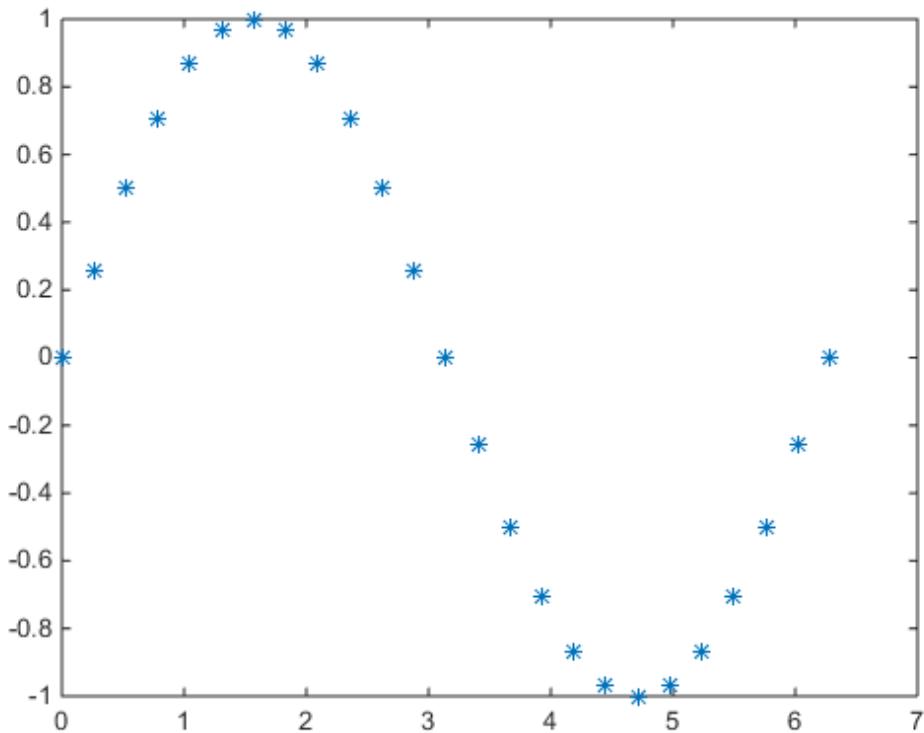
Plot Only Data Points

This example shows how to plot only the data points by omitting the line style option from the line specification string.

Define the data x and y . Plot the data and display a star marker at each data point.

```
x = linspace(0,2*pi,25);
y = sin(x);

figure
plot(x,y, '*')
```



See Also

[contour](#) | [LineSpec \(Line Specification\)](#) | [linspace](#) | [loglog](#) | [plot](#) |
[plotyy](#) | [scatter](#) | [semilogx](#) | [semilogy](#) | [stairs](#) | [stem](#)

Related Examples

- “Add Title, Axis Labels, and Legend to Graph”
- “Change Axis Limits of Graph”
- “Change Tick Marks and Tick Labels of Graph”

Add Title, Axis Labels, and Legend to Graph

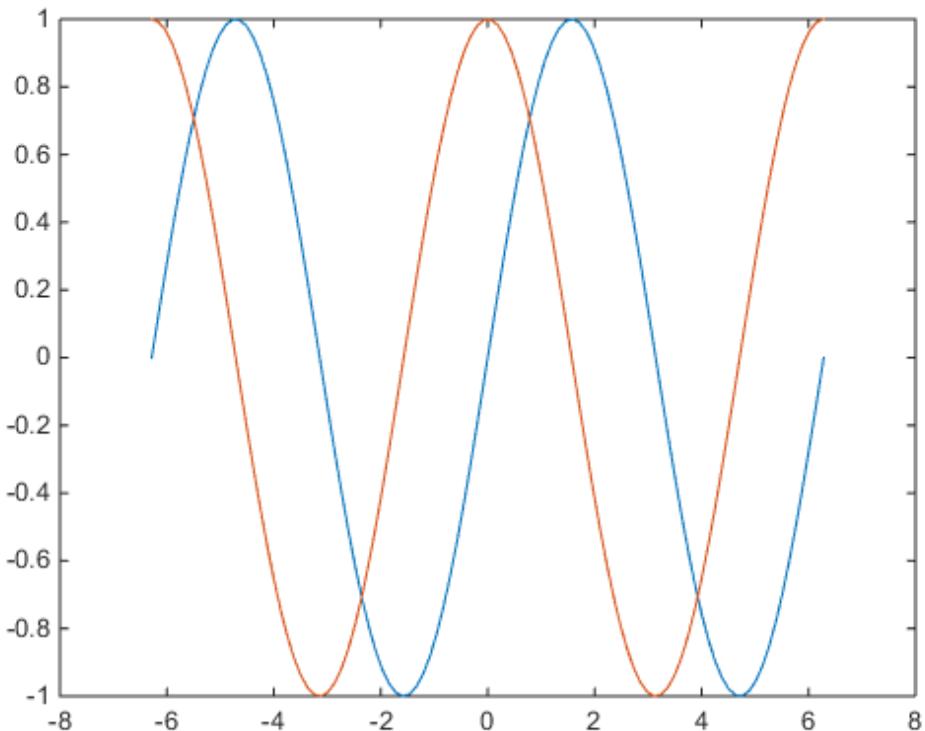
This example shows how to add a title, axis labels and a legend to a graph using the `title`, `xlabel`, `ylabel` and `legend` functions. By default, these functions add the text to the current axes. The current axes is typically the last axes created or the last axes clicked with the mouse.

Create Simple Line Plot

Define `x` as 100 linearly spaced values between -2π and 2π . Define `y1` and `y2` as sine and cosine values of `x`. Create a line plot of both sets of data.

```
x = linspace(-2*pi,2*pi,100);
y1 = sin(x);
y2 = cos(x);

figure
plot(x,y1,x,y2)
```

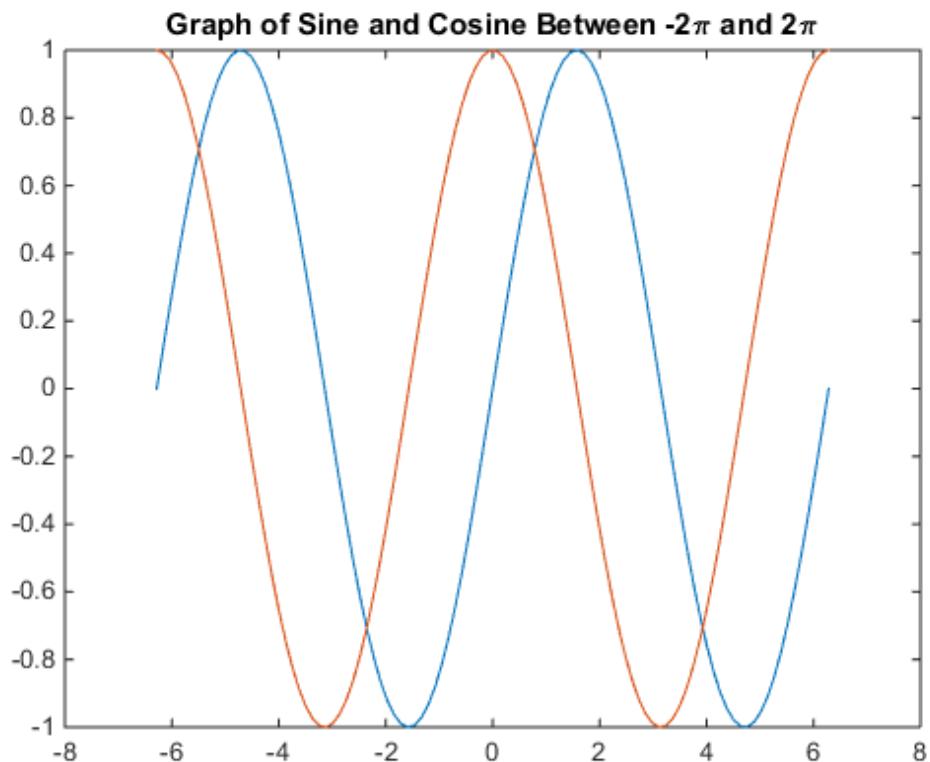


Add Title

Add a title to the graph using the `title` function. Pass the `title` function a text string with the desired title.

To display Greek symbols in a title, use the TeX markup. Use the TeX markup, `\pi`, to display the Greek symbol π .

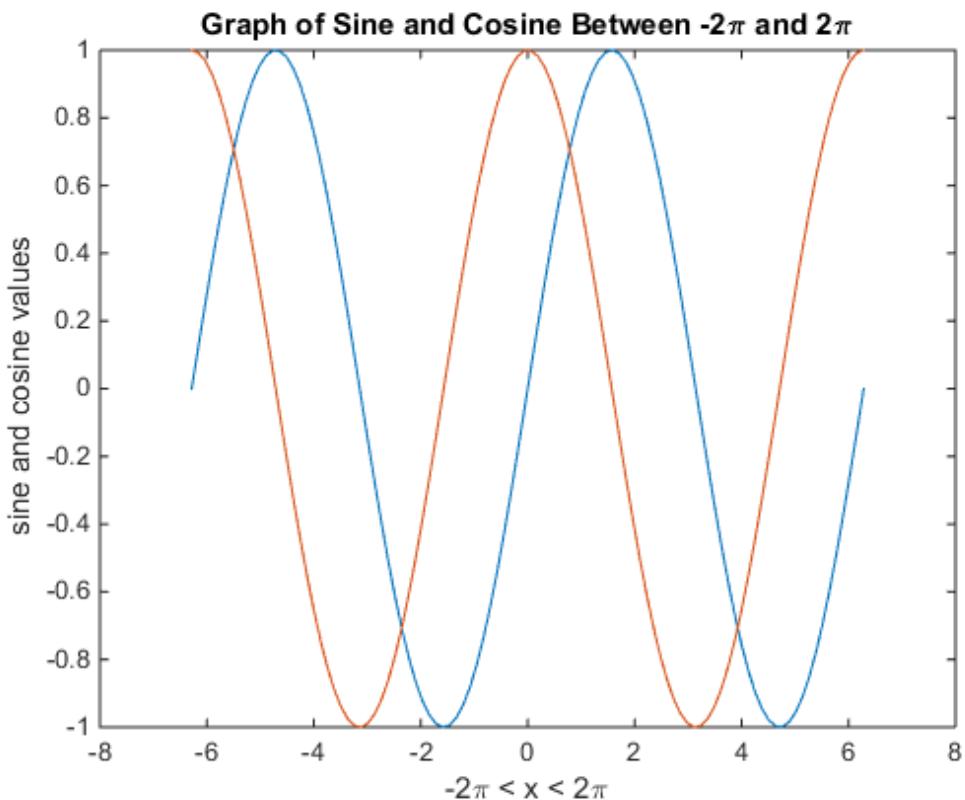
```
title('Graph of Sine and Cosine Between -2\pi and 2\pi')
```



Add Axis Labels

Add axis labels to the graph using the `xlabel` and `ylabel` functions. Pass these functions a text string with the desired label.

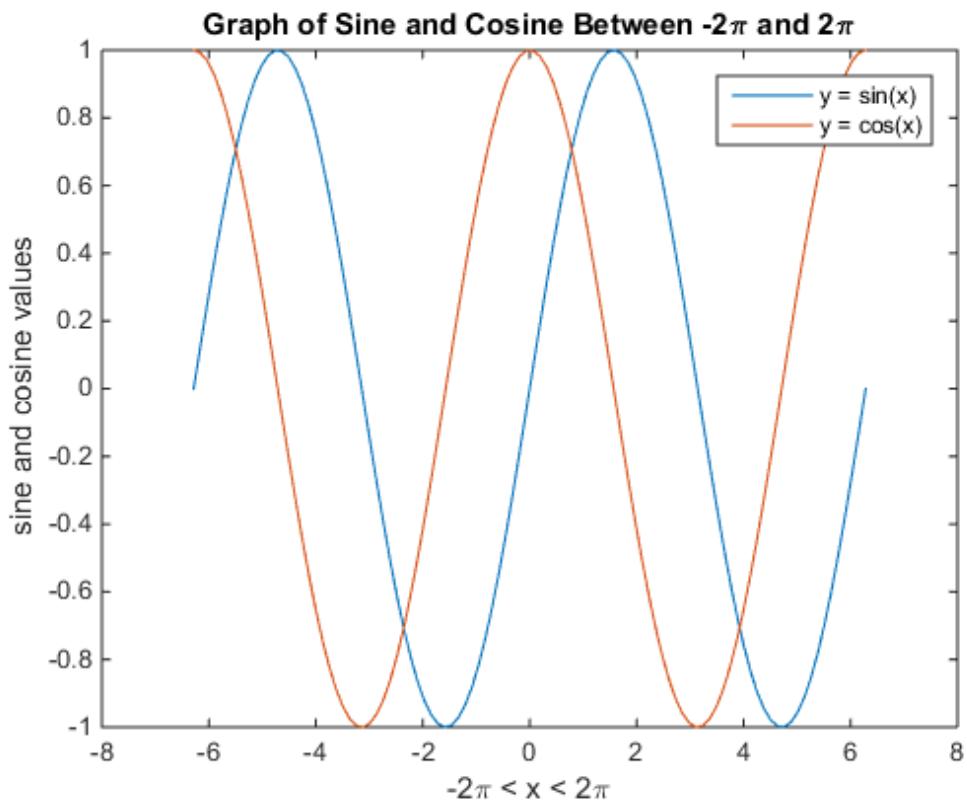
```
xlabel('-2\pi < x < 2\pi') % x-axis label  
ylabel('sine and cosine values') % y-axis label
```



Add Legend

Add a legend to the graph identifying each data set using the `legend` function. Pass the `legend` function a text string description for each line. Specify legend descriptions in the order that you plot the lines.

```
legend('y = sin(x)', 'y = cos(x)')
```

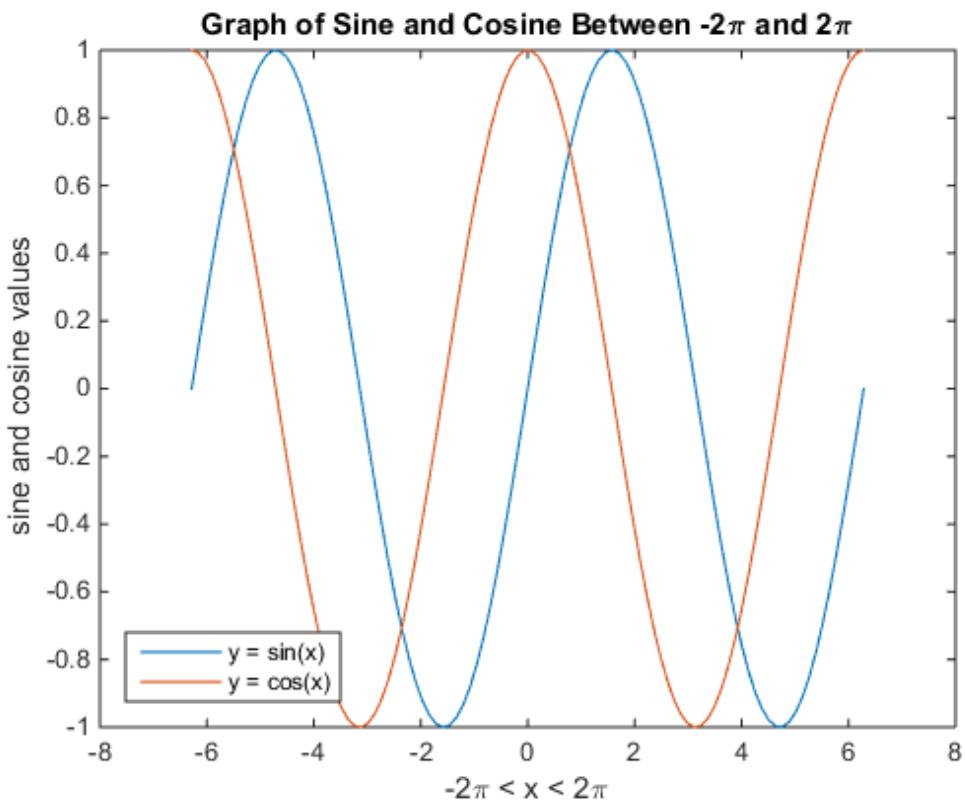


Change Legend Location

Change the location of the legend on the graph by setting its location using one of the eight cardinal or intercardinal directions. Display the legend at the bottom left corner of the axes by specifying its location as 'southwest'.

To display the legend outside the axes, append `outside` to any of the directions, for example, 'southwestoutside'.

```
legend('y = sin(x)', 'y = cos(x)', 'Location', 'southwest')
```



See Also

[legend](#) | [linspace](#) | [title](#) | [xlabel](#) | [ylabel](#)

Related Examples

- “Change Axis Limits of Graph”
- “Change Tick Marks and Tick Labels of Graph”
- “Add Plot to Existing Graph”

Change Axis Limits of Graph

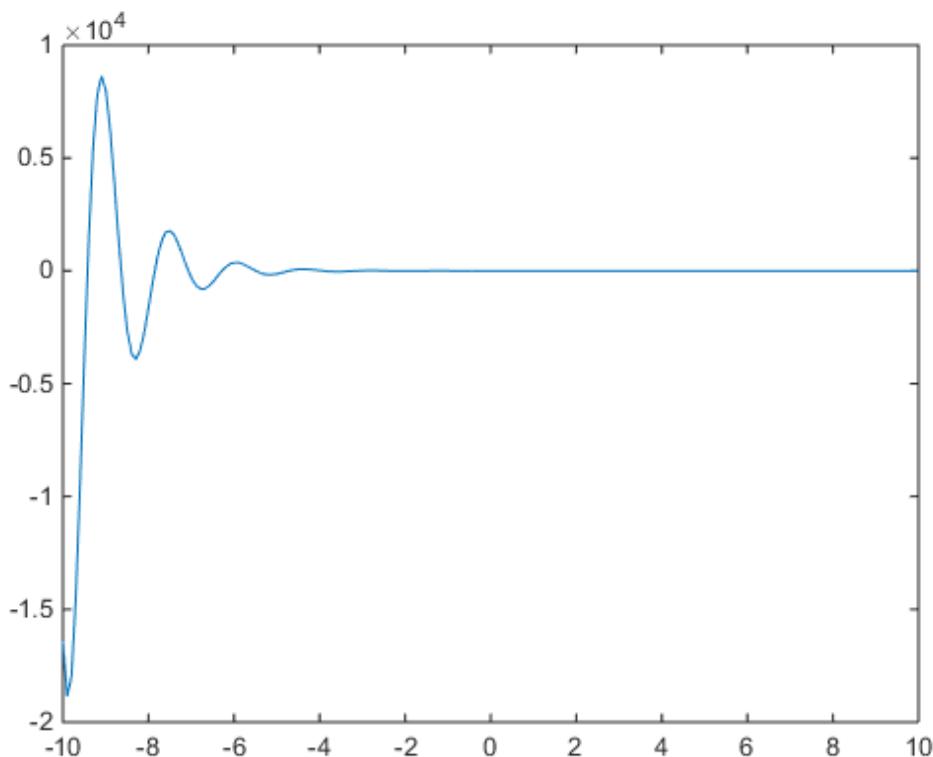
This example shows how to change the axis limits of a graph. By default, MATLAB® chooses axis limits to encompass the data plotted.

Create Simple Line Plot

Define x as 200 linearly spaced values between -10 and 10. Define y as the sine of x with an exponentially decreasing amplitude. Create a line plot of the data.

```
x = linspace(-10,10,200);
y = sin(4*x)./exp(x);

figure
plot(x,y)
```



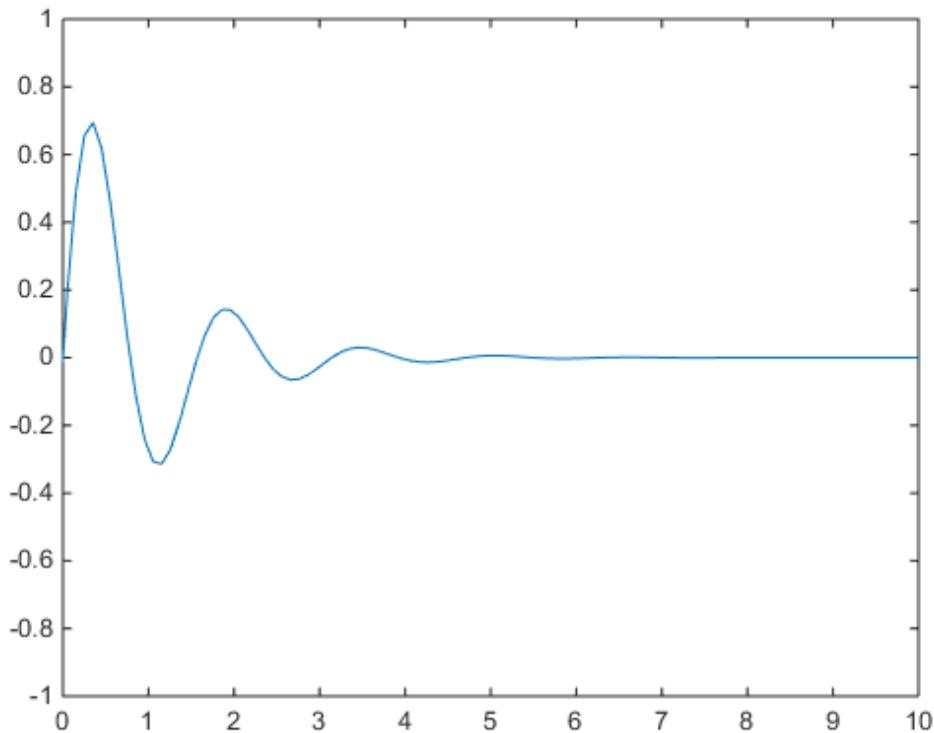
Change Axis Limits

Change the axis limits by passing to the `axis` function a four-element vector of the form `[xmin, xmax, ymin, ymax]`, where `xmin` and `xmax` set the scaling for the `x`-axis, and `ymin` and `ymax` set the scaling for the `y`-axis.

You also can change the axis limits using the `xlim`, `ylim`, and `zlim` functions. The commands `xlim([xmin, xmax])` and `ylim([ymin, ymax])` produce the same result as `axis([xmin, xmax, ymin, ymax])`.

Change the `x`-axis scaling to range from 0 to 10. Change the `y`-axis scaling to range from -1 to 1.

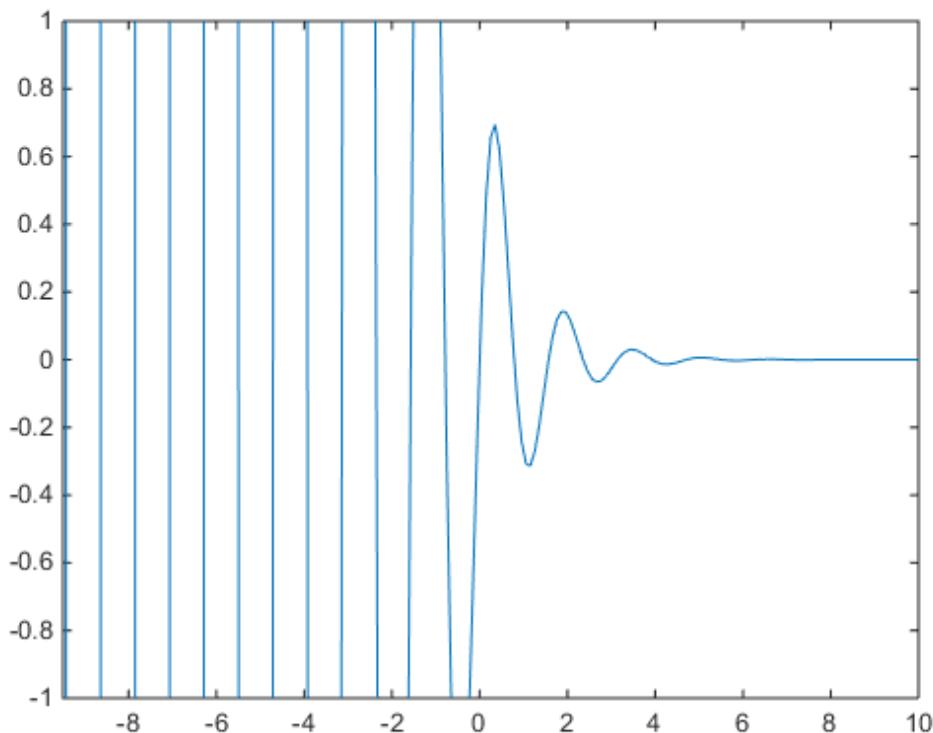
```
axis([0,10,-1,1])
```



Use Semiautomatic Axis Limits

Use an automatically calculated minimum x -axis limit by setting its value to `-inf`. MATLAB® calculates the limit based on the data. Set the maximum x -axis limit to 10, the minimum y -axis limit to -1, and the maximum y -axis limit to 1.

```
axis([-inf,10,-1,1])
```



MATLAB calculates the minimum limit for the *x*-axis based on the data. To use an automatically calculated maximum limit, set the value to `inf`.

See Also

`axis` | `linspace` | `plot` | `xlim` | `ylim`

Related Examples

- “Change Tick Marks and Tick Labels of Graph”
- “Add Plot to Existing Graph”

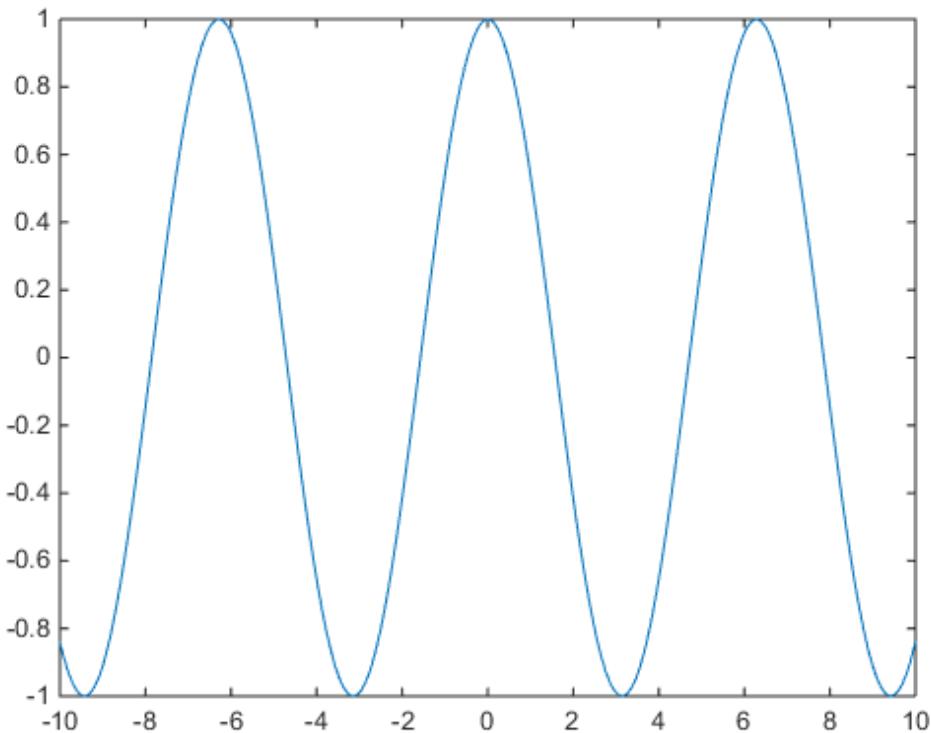
Change Tick Marks and Tick Labels of Graph

This example shows how to change the tick marks and the tick mark labels. MATLAB® chooses tick mark locations based on the range of the data, and automatically uses numeric labels at each tick mark.

Create Simple Line Chart

Define x as 200 linearly spaced values between -10 and 10. Define y as the cosine of x . Plot the data.

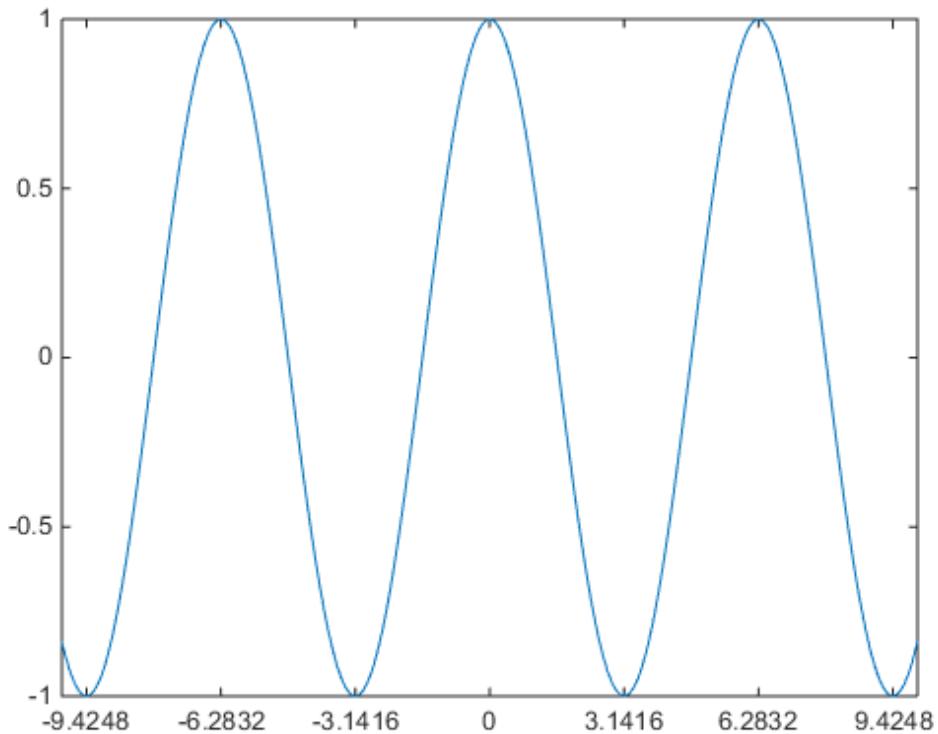
```
x = linspace(-10,10,200);  
y = cos(x);  
  
figure  
plot(x,y)
```



Change Tick Marks

Change the location of the tick marks on the plot by setting the `XTick` and `YTick` properties of the axes. Use `gca` to get the handle for the current axes. Define the tick marks as a vector of increasing values. The values do not need to be equally spaced.

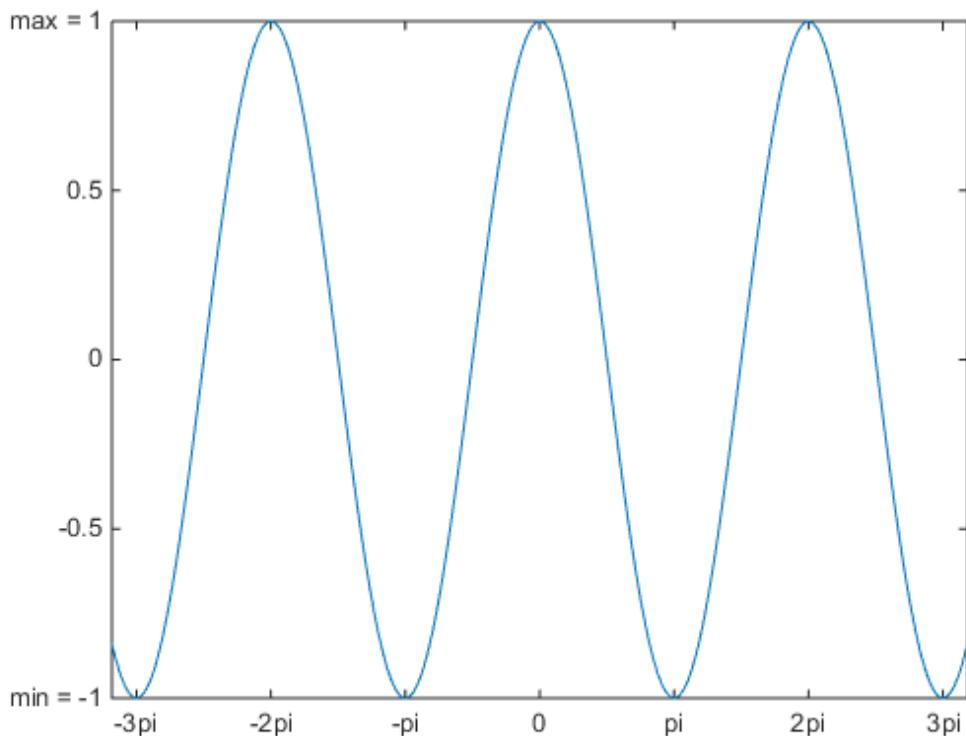
```
h = gca;
h.XTick = [-3*pi,-2*pi,-pi,0,pi,2*pi,3*pi];
h.YTick = [-1,-0.5,0,0.5,1];
```



Change Tick Mark Labels

Specify tick mark labels by setting the `XTickLabel` and `YTickLabel` properties of the axes. Set these properties using a cell array of strings with the desired labels. If you do not specify enough text labels for all the tick marks, then MATLAB cycles through the labels.

```
h.XTickLabel = {'-3pi',' -2pi',' -pi',' 0',' pi',' 2pi',' 3pi'};  
h.YTickLabel = {'min = -1',' -0.5',' 0',' 0.5',' max = 1'};
```



See Also

[gca](#) | [linspace](#) | [plot](#)

Related Examples

- “Change Axis Limits of Graph”
- “Add Plot to Existing Graph”

Display Grid Lines on 2-D Graph

In this section...

[“Display Major and Minor Grid Lines” on page 2-26](#)

[“Display Grid Lines in Single Direction” on page 2-29](#)

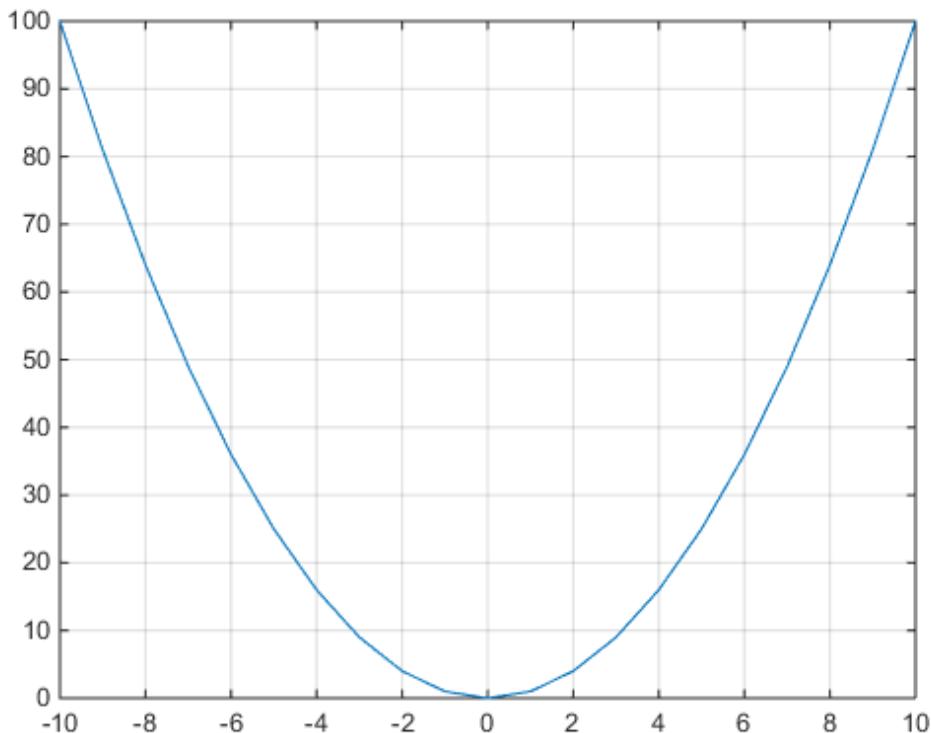
[“Change Grid Line Style” on page 2-31](#)

Display Major and Minor Grid Lines

This example shows how to display major and minor grid lines on a graph. Major grid lines align with the tick mark locations. Minor grid lines appear between the tick marks.

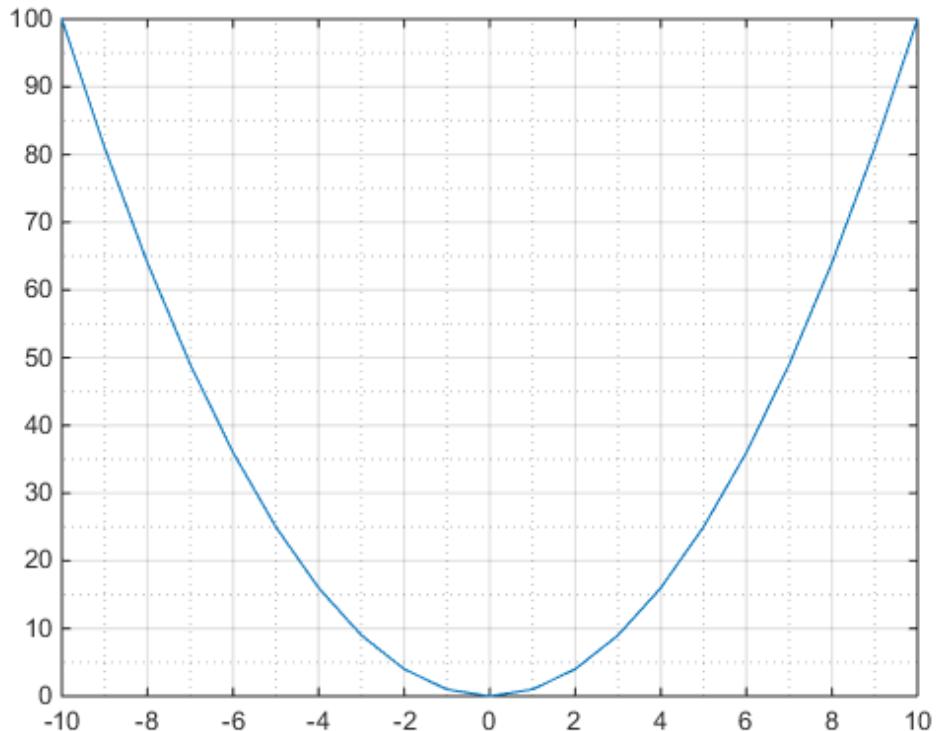
Use `grid on` to display the major grid lines.

```
x = -10:10;  
y = x.^2;  
  
figure  
plot(x,y)  
grid on
```



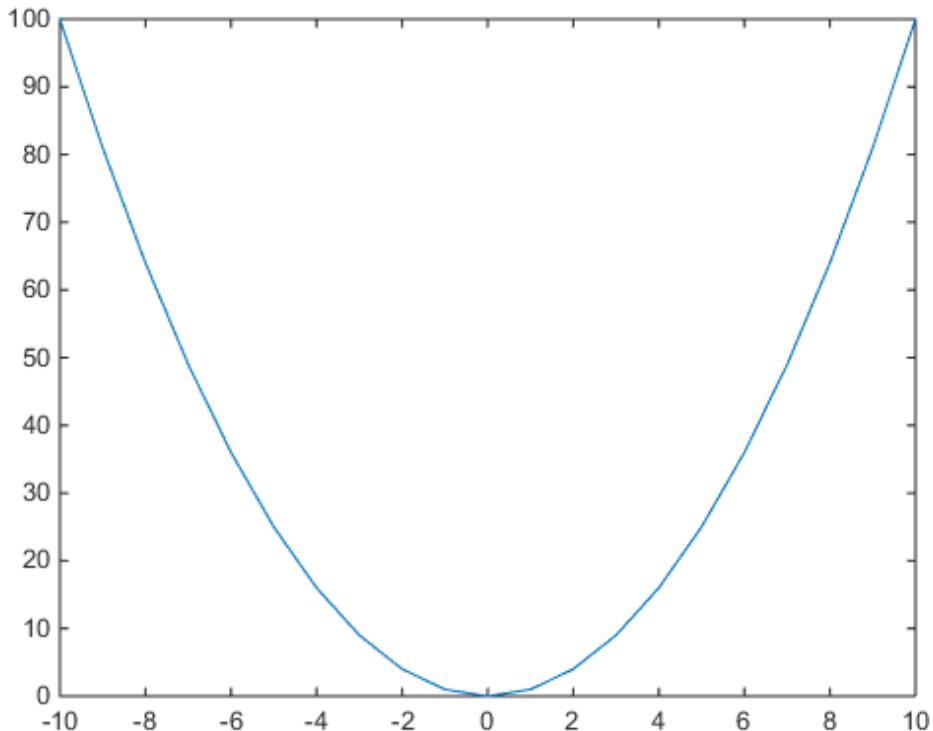
Use `grid minor` to display the minor grid lines.

`grid minor`



Use `grid off` to remove the grid lines.

```
grid off
```



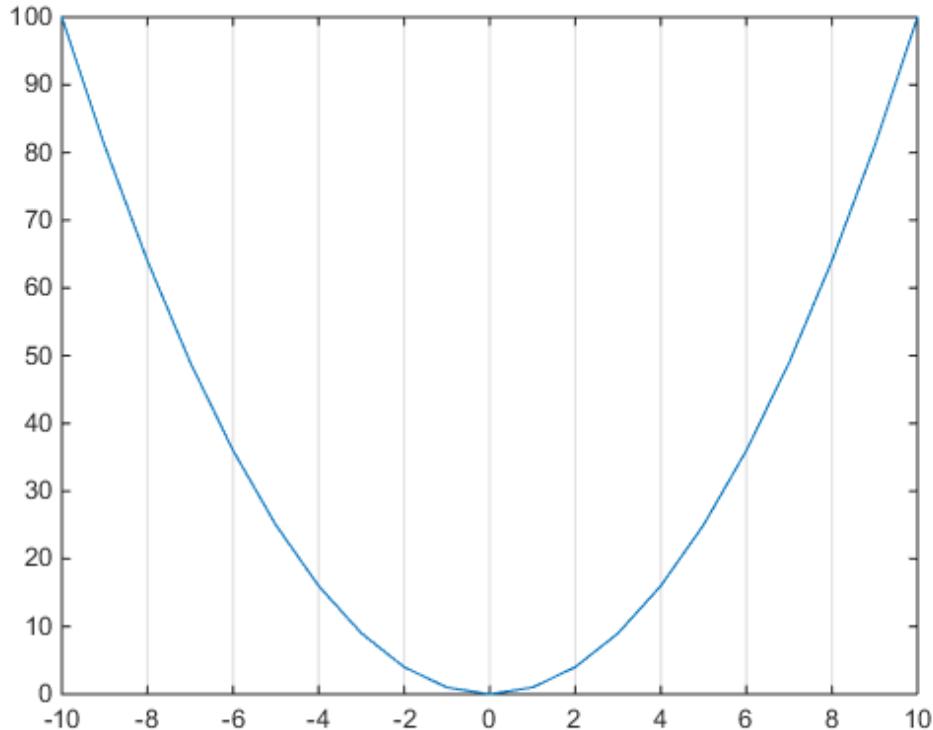
Display Grid Lines in Single Direction

This example shows how to display grid lines in a single direction by setting axes properties. The `grid on` command sets the `XGrid`, `YGrid`, and `ZGrid` axes properties. The `grid minor` command sets the `XMinorGrid`, `YMinorGrid`, and `ZMinorGrid` properties. Control the display of the grid lines for each individual axes by setting these axes properties directly.

Display the grid lines in the x -direction, but not in the y -direction by setting the `XGrid` axes property to '`on`' and the `YGrid` axes property to '`off`'.

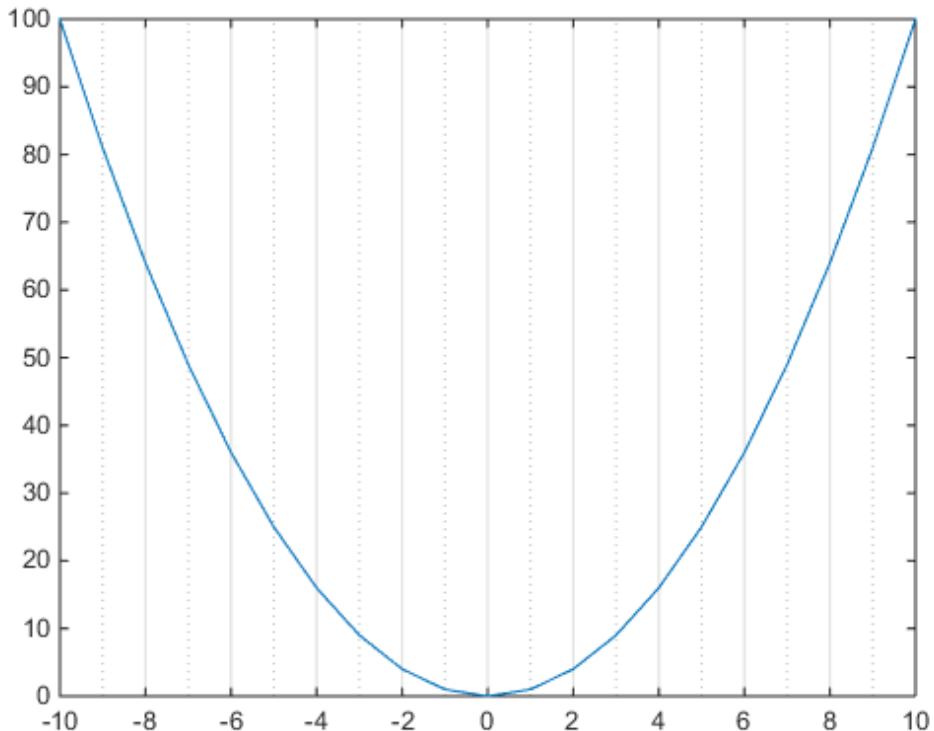
```
x = -10:10;  
y = x.^2;
```

```
figure  
plot(x,y)  
  
ax = gca;  
ax.XGrid = 'on';  
ax.YGrid = 'off';
```



Display the minor grid lines in the x -direction, but not in the y -direction by setting the `XGridMinor` axes property to '`on`' and the `YGridMinor` axes property to '`off`'.

```
ax.XMinorGrid = 'on';  
ax.YMinorGrid = 'off';
```



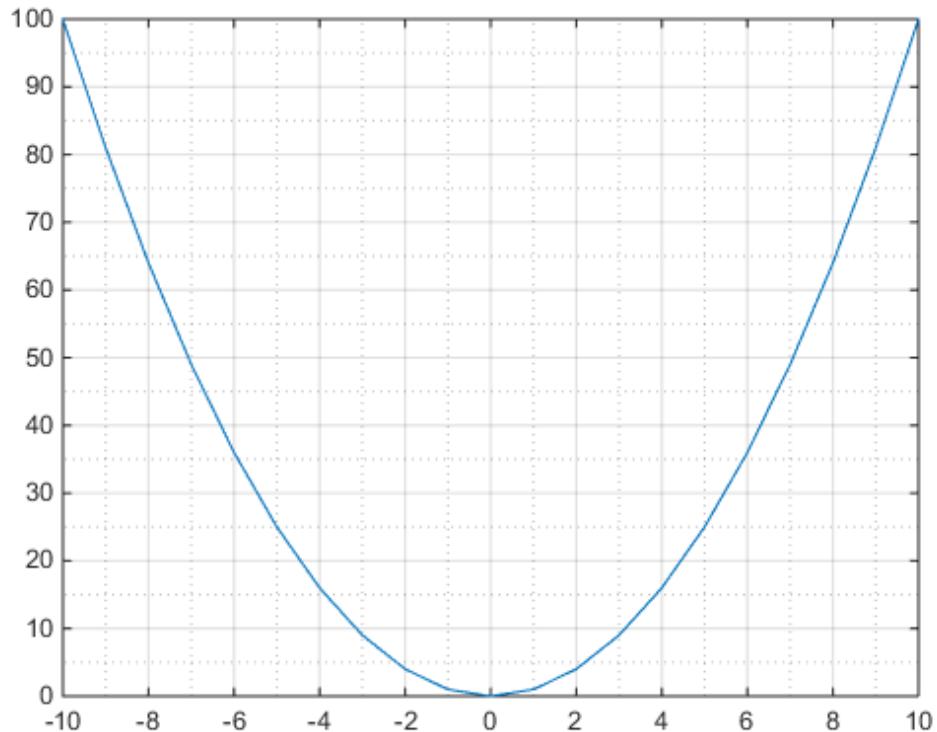
Change Grid Line Style

This example shows how to change the grid line style using the `GridLineStyle` and `MinorGridLineStyle` axes properties. Line style options are a solid line, `' - '`, a dotted line, `' : '`, a dashed line, `' - - '`, a dash-dotted line, `' - . '`.

Create a line plot and display the major and minor grid lines.

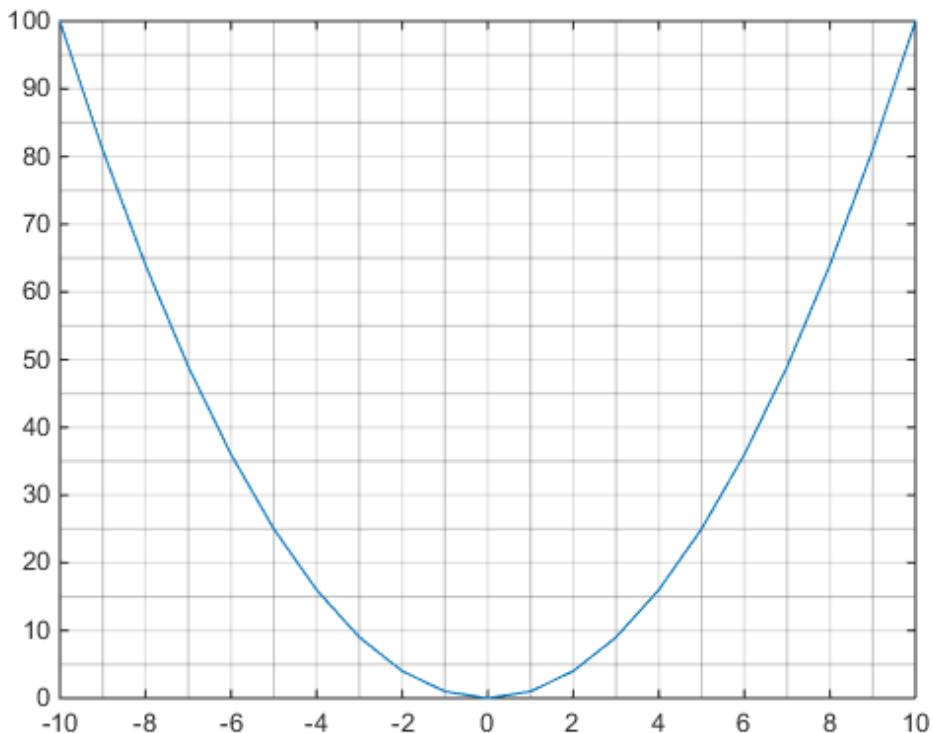
```
x = -10:10;  
y = x.^2;  
  
figure  
plot(x,y)
```

```
grid on  
grid minor
```



Specify a solid line for all the grid lines.

```
ax = gca;  
ax.GridLineStyle = '-';  
ax.MinorGridLineStyle = '-';
```



See Also

[gca](#) | [grid](#) | [plot](#)

Related Examples

- “Add Title, Axis Labels, and Legend to Graph”
- “Change Axis Limits of Graph”
- “Change Tick Marks and Tick Labels of Graph”

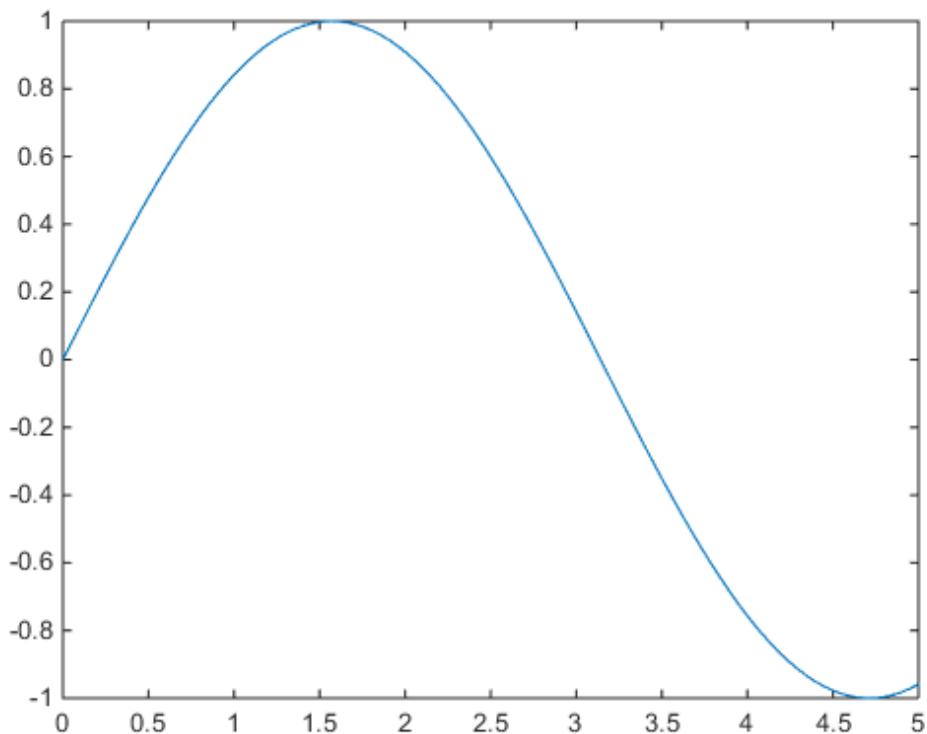
Add Plot to Existing Graph

This example shows how to add a plot to an existing graph.

Plot of a sine wave along the domain [0,5].

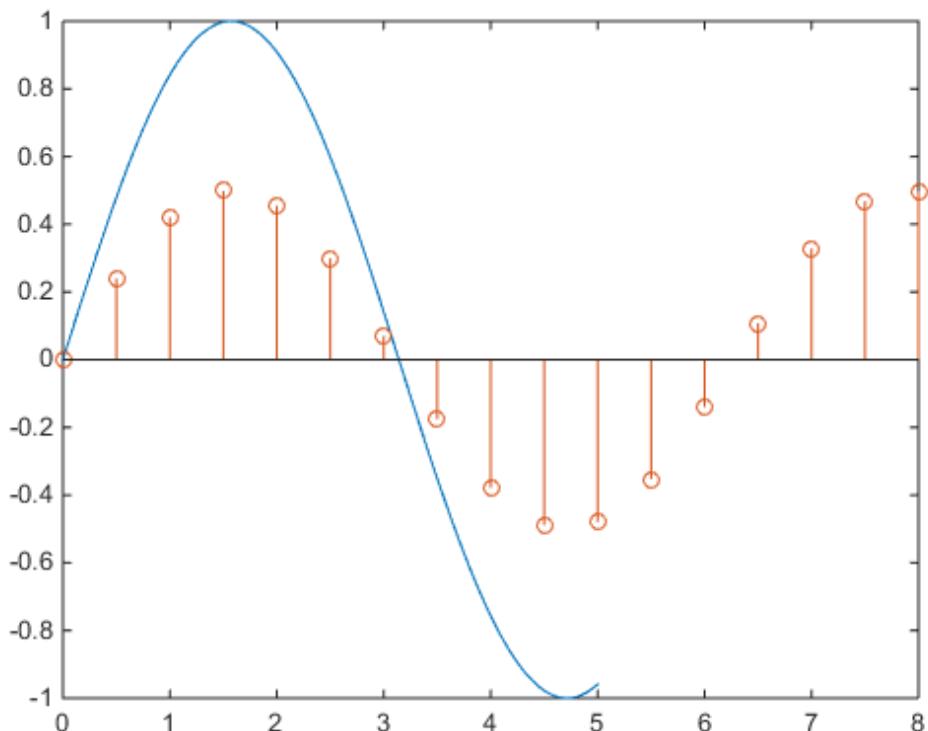
```
x1 = linspace(0,5);
y1 = sin(x1);

figure % new figure window
plot(x1,y1)
```



Use `hold on` to retain the line plot and add a new plot to the graph. Add a stem plot along the domain [0,8]. Then, use `hold off` to reset the hold state so that new plots replace existing plots, which is the default behavior.

```
hold on  
  
x2 = 0:0.5:8;  
y2 = 0.5*sin(x2);  
stem(x2,y2)  
  
hold off % reset hold state
```



MATLAB® rescales the axis limits each time a new plot is added to a graph.

See Also

`hold` | `linspace` | `plot` | `stem`

Related Examples

- “Create Figure with Multiple Graphs Using Subplots”

Create Figure with Multiple Graphs Using Subplots

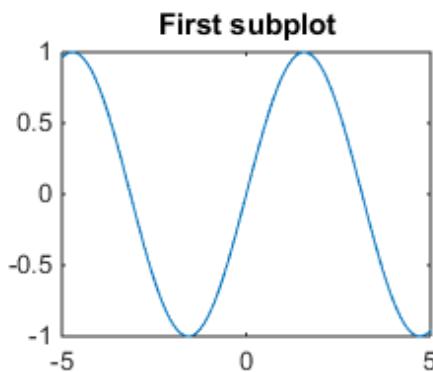
This example shows how to create a figure containing multiple graphs using the `subplot` function. The syntax, `subplot(m,n,p)`, divides the figure into an m-by-n grid with an axes in the pth grid location. The grids are numbered along each row.

Create Subplots and Add Subplot Titles

Use `subplot` to create a figure containing a 2-by-2 grid of graphs. Plot a sine wave in the first subplot.

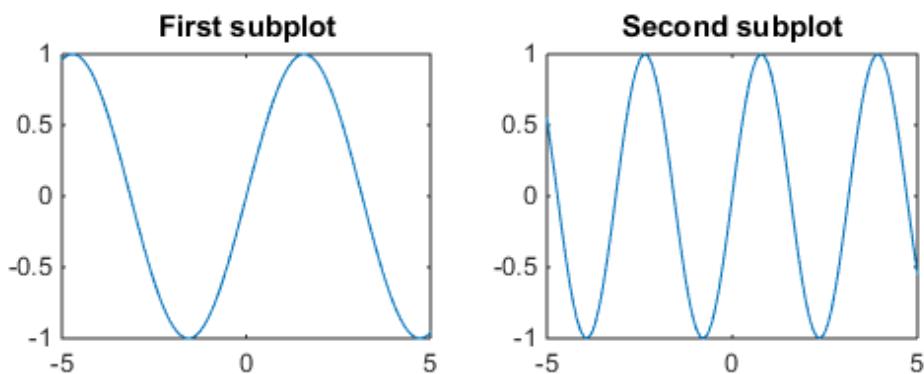
```
x = linspace(-5,5); % define x
y1 = sin(x); % define y1

figure % create new figure
subplot(2,2,1) % first subplot
plot(x,y1)
title('First subplot')
```



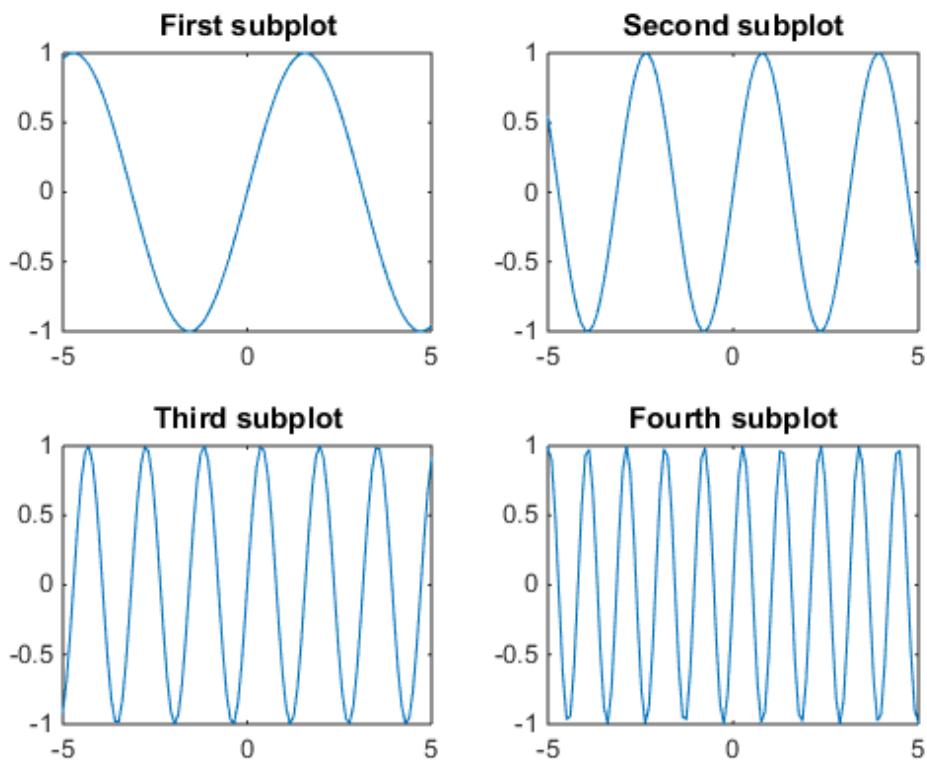
Plot another sine wave in the second subplot.

```
y2 = sin(2*x); % define y2  
subplot(2,2,2) % second subplot  
plot(x,y2)  
title('Second subplot')
```



Plot two more sine waves in the third and fourth subplots.

```
y3 = sin(4*x); % define y3  
y4 = sin(6*x); % define y4  
  
subplot(2,2,3) % third subplot  
plot(x,y3)  
title('Third subplot')  
  
subplot(2,2,4) % fourth subplot  
plot(x,y4)  
title('Fourth subplot')
```

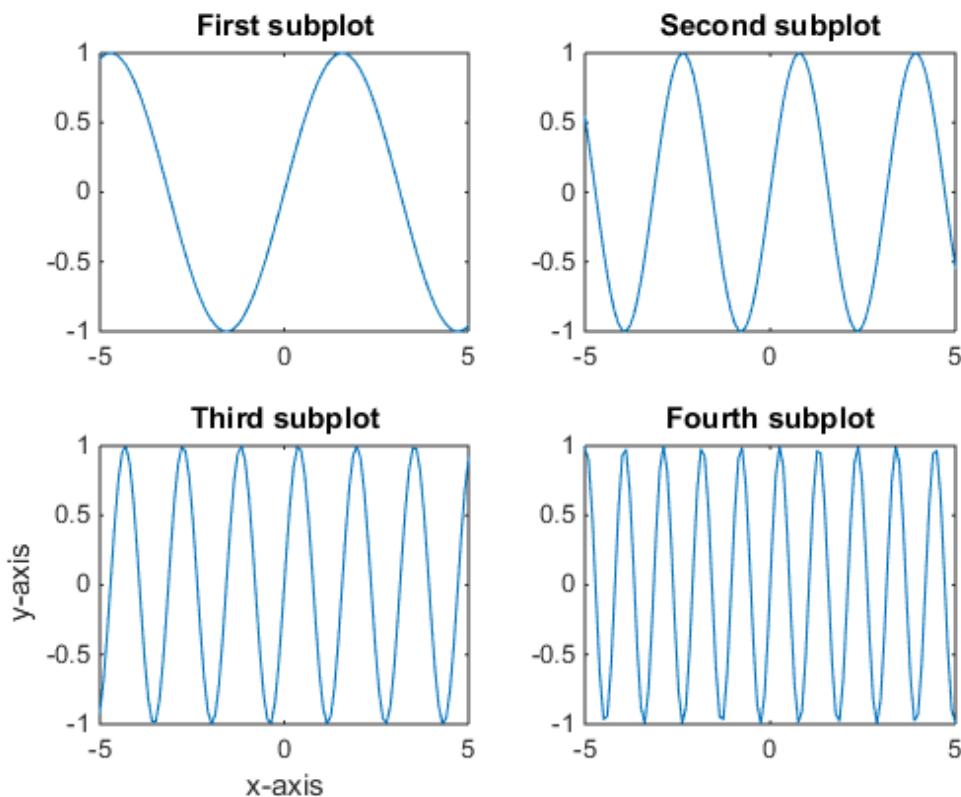


Add Subplot Axis Labels

Add subplot labels using the `xlabel` and `ylabel` functions. By default, `xlabel` and `ylabel` label the current axes. The current axes is typically the last axes created or clicked with the mouse. Reissuing the command, `subplot(m,n,p)`, makes the pth subplot the current axes.

Make the third subplot the current axes. Then, label its *x*-axis and *y*-axis.

```
subplot(2,2,3)
xlabel('x-axis')
ylabel('y-axis')
```



The figure contains four axes with a sine wave plotted in each axes.

See Also

`linspace` | `plot` | `subplot` | `title` | `xlabel` | `ylabel`

Related Examples

- “Add Plot to Existing Graph”

Create Graph with Two y-Axes

This example shows how to create a graph with two *y*-axes, label the axes, and display the grid lines.

Create and Plot Data

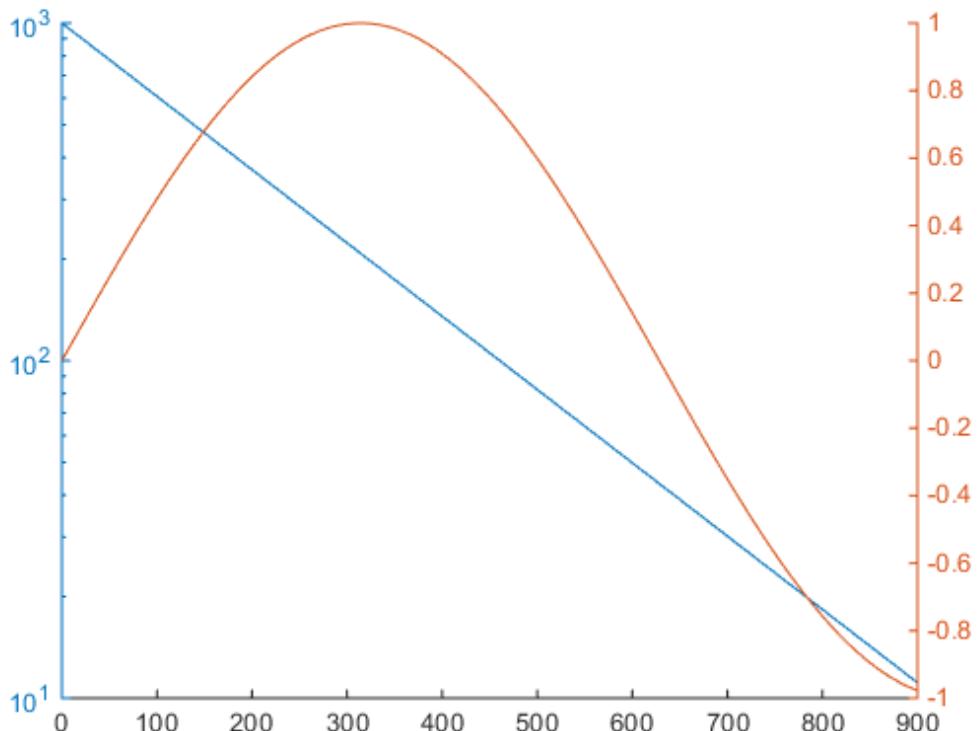
Create the data.

```
A = 1000;
a = 0.005;
b = 0.005;

t = 0:900;
z1 = A*exp(-a*t);
z2 = sin(b*t);
```

Use `plotyy` to create a graph with two *y*-axes. Plot *z1* versus *t* using semilogarithmic scaling. Plot *z2* versus *t* using linear scaling. Return the two axes in array *ax*, and return the two lines as *p1* and *p2*.

```
[ax,p1,p2] = plotyy(t,z1,t,z2,'semilogy','plot');
```



The left y -axis corresponds to the first set of data plotted, which is the semilogarithmic plot for $z1$. The first axes, $ax(1)$, and the line, $p1$, correspond to the first set of data.

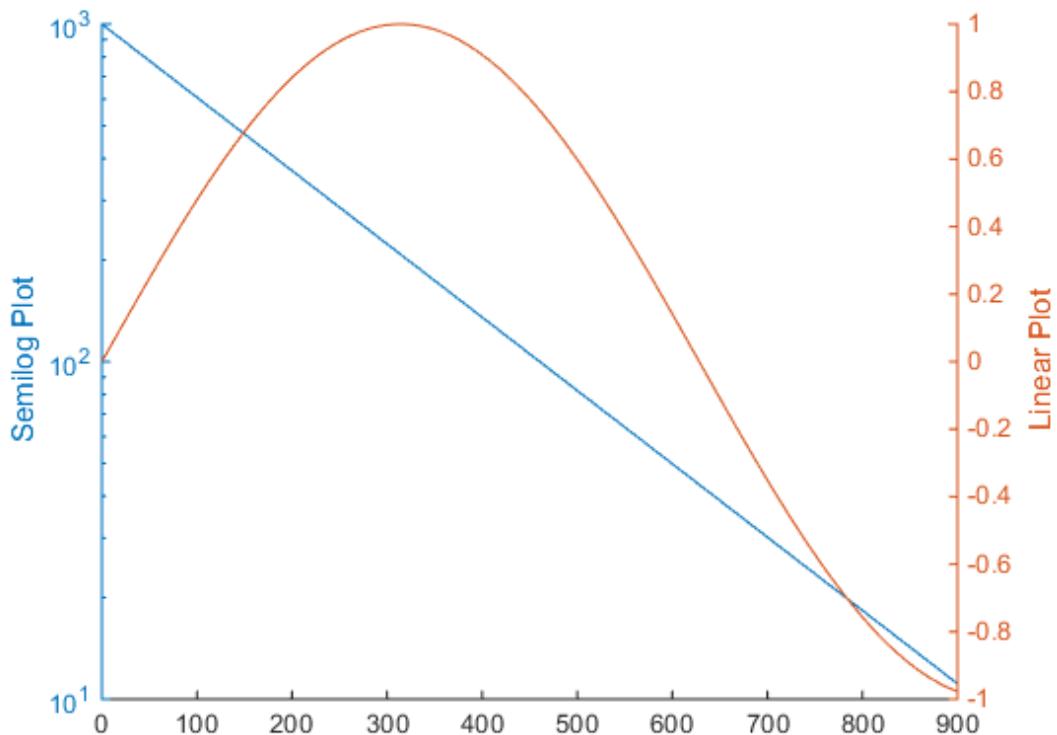
The right y -axis corresponds to the second set of data plotted, which is the line plot for $z2$. The second axes, $ax(2)$, and the line, $p2$, correspond to the second set of data.

Label the Axes

Label the left y -axis by passing the first axes to the `ylabel` function. Then, label the right y -axis by passing the second axes to the `ylabel` function. Label the x -axis using either axes.

```
ylabel(ax(1), 'Semilog Plot') % label left y-axis  
ylabel(ax(2), 'Linear Plot') % label right y-axis
```

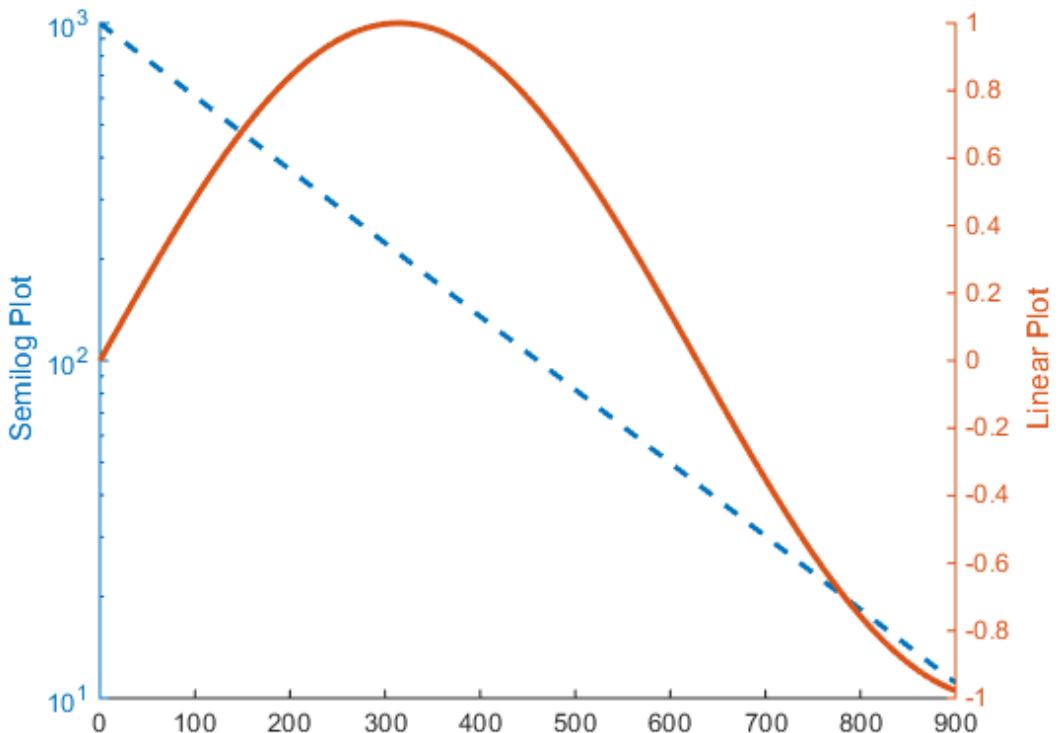
```
xlabel(ax(2), 'Time') % label x-axis
```



Modify Line Appearance

Change the appearance of the lines.

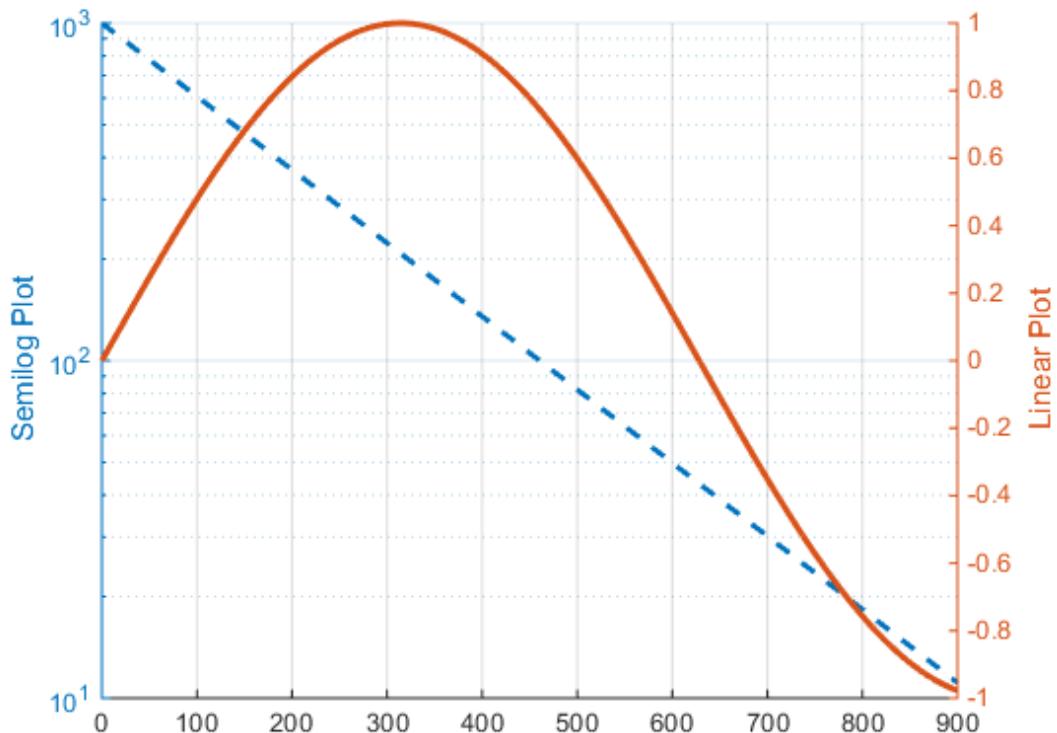
```
p1.LineStyle = '--';  
p1.LineWidth = 2;  
p2.LineWidth = 2;
```



Display Grid Lines

Display the log grid associated with the left y -axis by passing the first axes to the `grid` function.

```
grid(ax(1), 'on')
```



To display the linear grid associated with the right y-axis instead, use
`grid(ax(2), 'on')`.

See Also

`grid` | `plotyy` | `ylabel`

Related Examples

- “Graph with Multiple x-Axes and y-Axes”

Display Markers at Specific Data Points on Line Graph

This example shows how to make a line graph and display markers at particular data points.

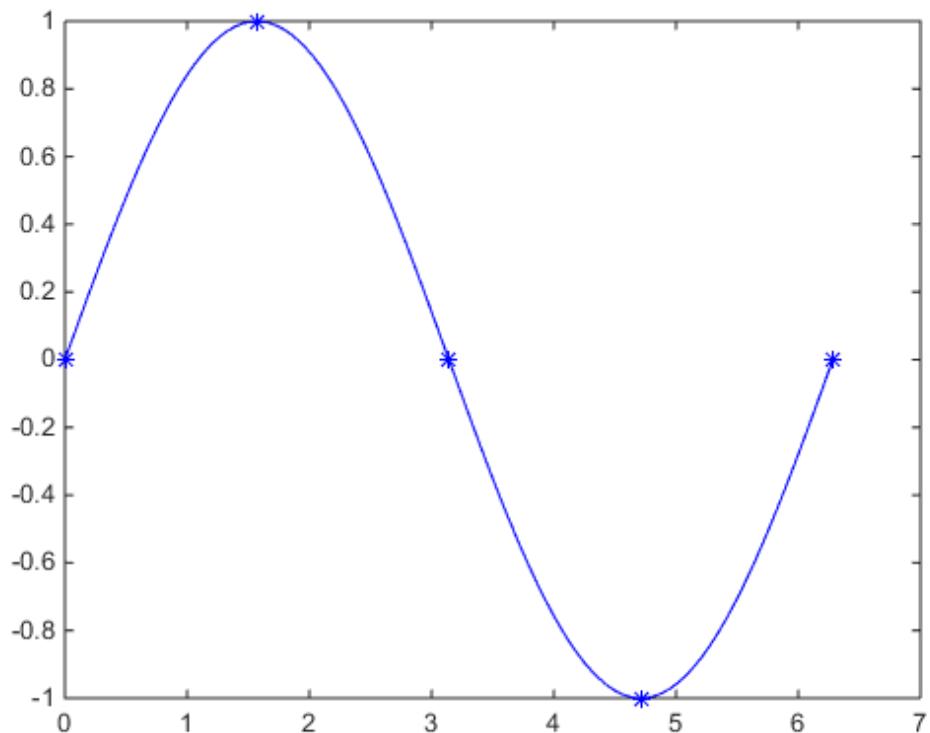
Define x and y as 100-element vectors.

```
x = linspace(0,2*pi,100);
y = sin(x);
```

Plot x versus y . Display markers every $\pi/2$ data points by superimposing a second graph of just markers over the line graph.

```
xmarkers = 0:pi/2:2*pi; % place markers at these x-values
ymarkers = sin(xmarkers);

figure
plot(x,y, 'b', xmarkers, ymarkers, 'b*')
```



See Also

`hold | linspace | plot`

Plot Imaginary and Complex Data

Plot One Complex Input

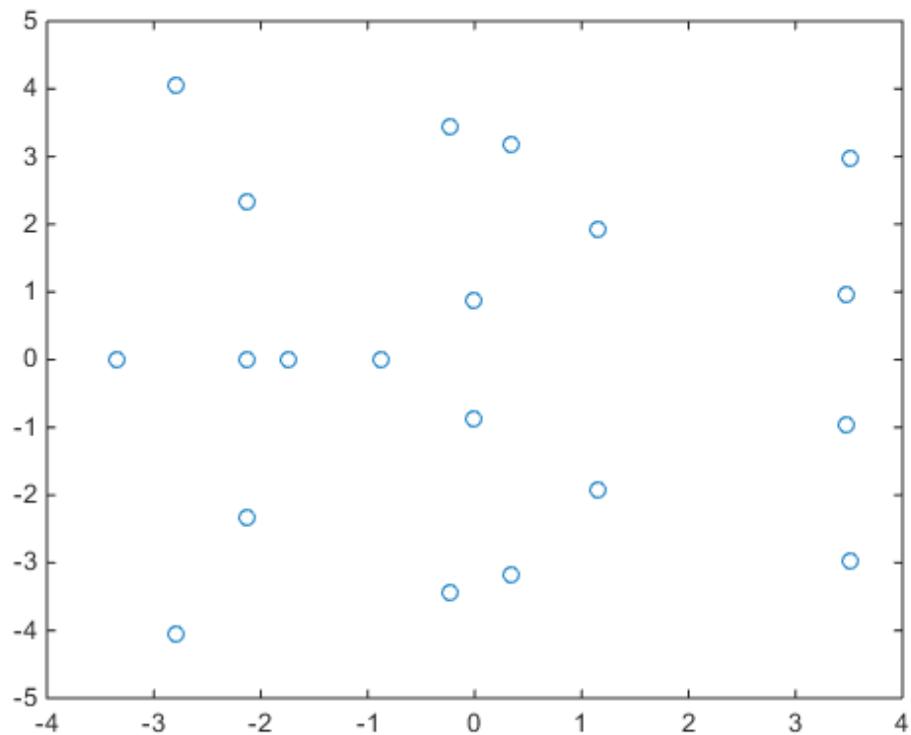
This example shows how to plot the imaginary part versus the real part of a complex vector, z . With complex inputs, `plot(z)` is equivalent to `plot(real(z), imag(z))`, where `real(z)` is the real part of z and `imag(z)` is the imaginary part of z .

Define z as a vector of eigenvalues of a random matrix.

```
z = eig(randn(20));
```

Plot the imaginary part of z versus the real part of z . Display a circle at each data point.

```
figure  
plot(z, 'o')
```



Plot Multiple Complex Inputs

This example shows how to plot the imaginary part versus the real part of two complex vectors, $z1$ and $z2$. If you pass multiple complex arguments to `plot`, such as `plot(z1, z2)`, then MATLAB® ignores the imaginary parts of the inputs and plots the real parts. To plot the real part versus the imaginary part for multiple complex inputs, you must explicitly pass the real parts and the imaginary parts to `plot`.

Define the complex data.

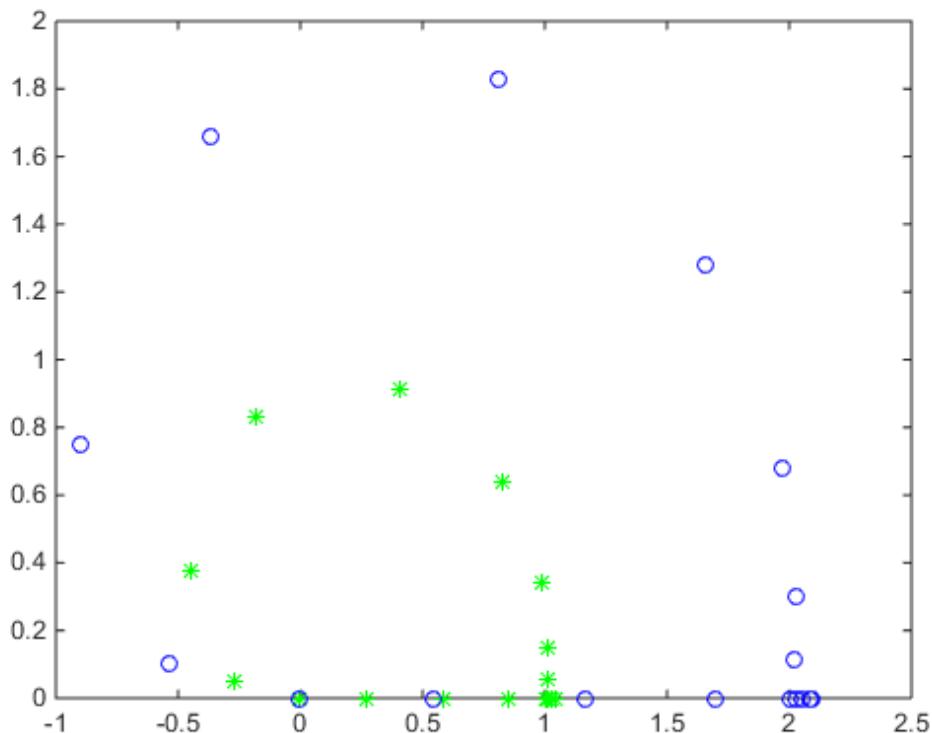
```
x = -2:0.25:2;
z1 = x.^exp(-x.^2);
z2 = 2*x.^exp(-x.^2);
```

Find the real part and imaginary part of each vector using the `real` and `imag` functions. Then, plot the data.

```
real_z1 = real(z1);
imag_z1 = imag(z1);

real_z2 = real(z2);
imag_z2 = imag(z2);

plot(real_z1,imag_z1,'g*',real_z2,imag_z2,'bo')
```



See Also

`imag` | `plot` | `real`

Data Exploration Tools

- “Ways to Explore Graphical Data” on page 3-2
- “Display Data Values Interactively” on page 3-4
- “Zooming in Graphs” on page 3-15
- “Panning — Shifting Your View of the Graph” on page 3-17
- “Rotate in 3-D” on page 3-18

Ways to Explore Graphical Data

In this section...

[“Introduction” on page 3-2](#)

[“Types of Tools” on page 3-2](#)

Introduction

After determining what type of graph best represents your data, you can further enhance the visual display of information using the tools discussed in this section. These tools enable you to explore data interactively.

Once you have achieved the desired results, you can then generate the MATLAB code necessary to reproduce the graph you created interactively. See “Generating a MATLAB File to Recreate a Graph” on page 7-79 for more information.

Types of Tools

See the following sections for information on specific tools.

- [“Display Data Values Interactively” on page 3-4](#)
- [“Zooming in Graphs” on page 3-15](#)
- [“Panning — Shifting Your View of the Graph” on page 3-17](#)
- [“Rotate in 3-D” on page 3-18](#)
- [“View Control with the Camera Toolbar”](#)

You can also explore graphs visually with data brushing and linking:

- Data brushing lets you “paint” observations on a graph to select them for special treatment, such as
 - Extracting them into new variables
 - Replacing them with constant or NaN values
 - Deleting them
- Data linking connects graphs with the workspace variables they display, updating graphs when variables change

Brushing and linking work together across plots. When multiple graphs or subplots display the same variables, linking the graphs and brushing any of them causes the same data to also highlight on other linked graphs. The highlighting also appears on the selected rows of data when the variables are opened in the Variable Editor. For details, see “Marking Up Graphs with Data Brushing” and “Making Graphs Responsive with Data Linking” in the Data Analysis documentation.

You can perform numerical data analysis directly on graphs with basic curve fitting.

- “Linear Regression”
- “Interactive Fitting”

Display Data Values Interactively

In this section...

- “What Is a Data Cursor?” on page 3-4
- “Enabling Data Cursor Mode” on page 3-4
- “Display Style — Datatip or Cursor Window” on page 3-12
- “Selection Style — Select Data Points or Interpolate Points on Graph” on page 3-13
- “Exporting Data Value to Workspace Variable” on page 3-13

What Is a Data Cursor?

Data cursors enable you to read data directly from a graph by displaying the values of points you select on plotted lines, surfaces, images, and so on. You can place multiple datatips in a plot and move them interactively. If you save the figure, the datatips in it are saved, along with any other annotations present.

When data cursor mode is enabled, you can

- Click on any graphics object defined by data values and display the x , y , and z (if 3-D) values of the nearest data point.
- Interpolate the values of points between data points.
- Display multiple data tips on graphs.
- Display the data values in a cursor window that you can locate anywhere in the figure window or as a data tip (small text box) located next to the data point.
- Export data values as workspace variables.
- Print or export the graph with data tip or cursor window displayed for annotation purposes.
- Edit the data tip display function to customize what information is displayed and how it is presented
- Select a different data tip display function

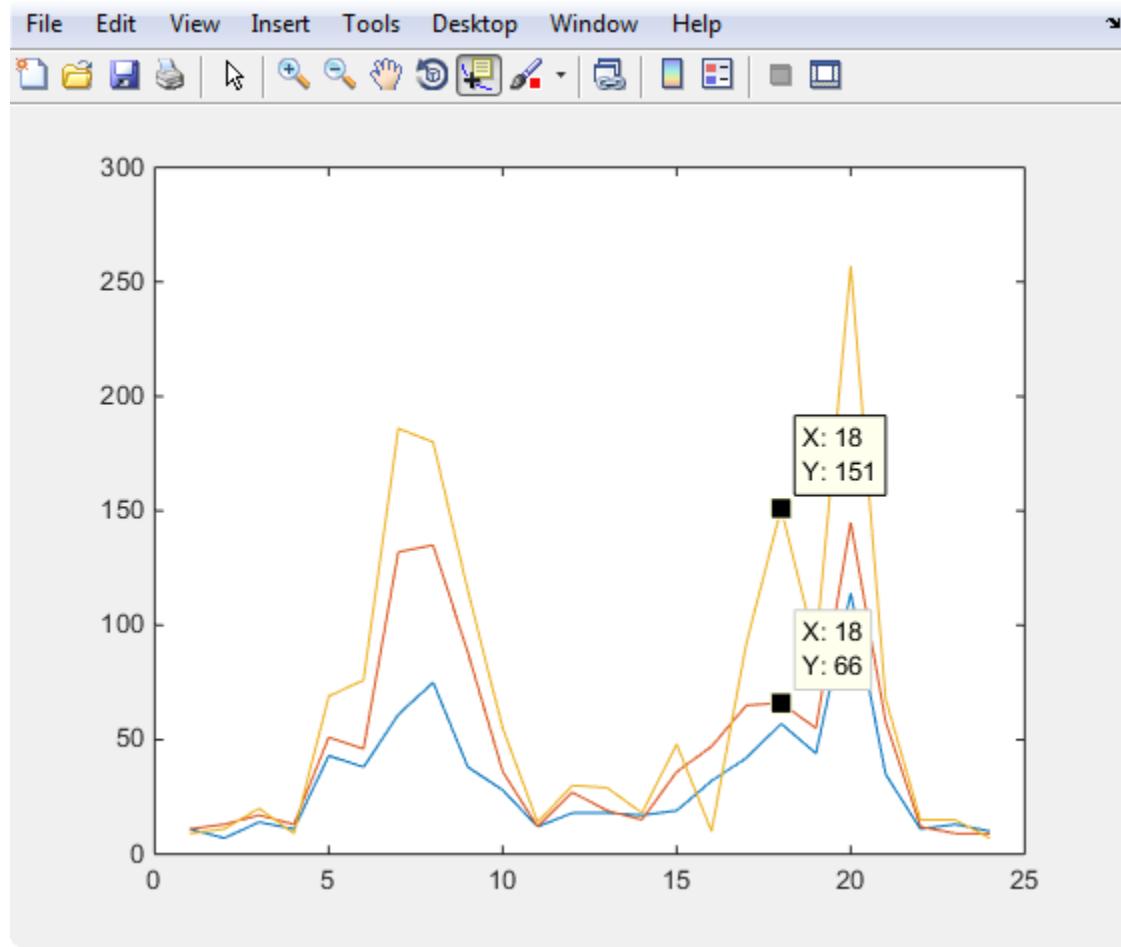
Enabling Data Cursor Mode

Select the data cursor icon in the figure toolbar  or select the **Data Cursor** item in the **Tools** menu.

Once you have enabled data cursor mode, clicking the mouse on a line or other graph object displays data values of the point clicked. Clicking elsewhere does not create or update data tips. To place additional data tips, as the picture below shows, see “Creating Multiple Data Tips” on page 3-10, below. In the picture, the black squares are located at points selected by the Data Cursor tool, and the data tips next to them display the x and y values of those points.

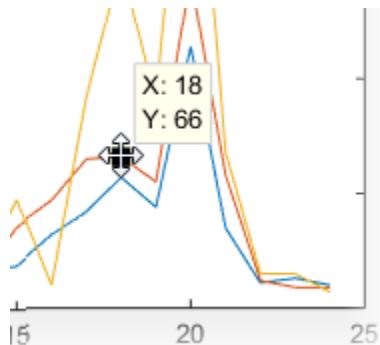
The illustrations below use traffic count data stored in `count.dat`:

```
load count.dat  
plot(count)
```



Moving the Marker

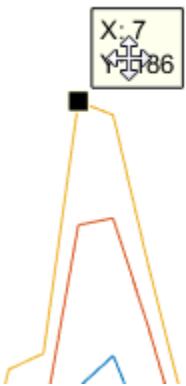
You can move the marker using the arrow keys and the mouse. The up and right arrows move the marker to data points having greater index values in the data arrays. The down and left arrow keys move the marker to data points having lesser index values. When you set **Selection Style** to **Mouse Position** using the tool's context menu, you can drag markers and position them anywhere along a line. However, you cannot drag markers between different line or other series on a plot. The cursor changes to crossed arrows when it comes close enough to a marker for you to drag the datatip, as shown below:



Positioning the Datatip Text Box

You can position the data tip text box in any one of four positions with respect to the data point: upper right (the default), upper left, lower left, and lower right.

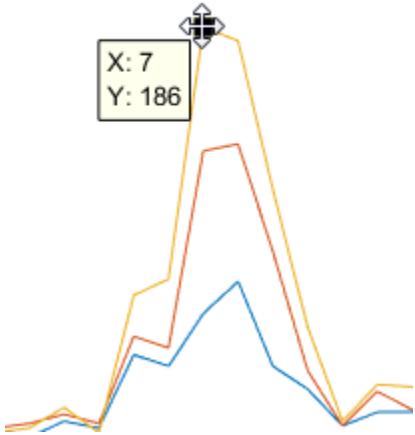
To position the datatip, press, but do not release the mouse button while over the datatip text box and drag it to one of the four positions, as shown below:



You can reposition a datatip, but not its text box, using the arrow keys as well.

Dragging the Datatip to Different Locations

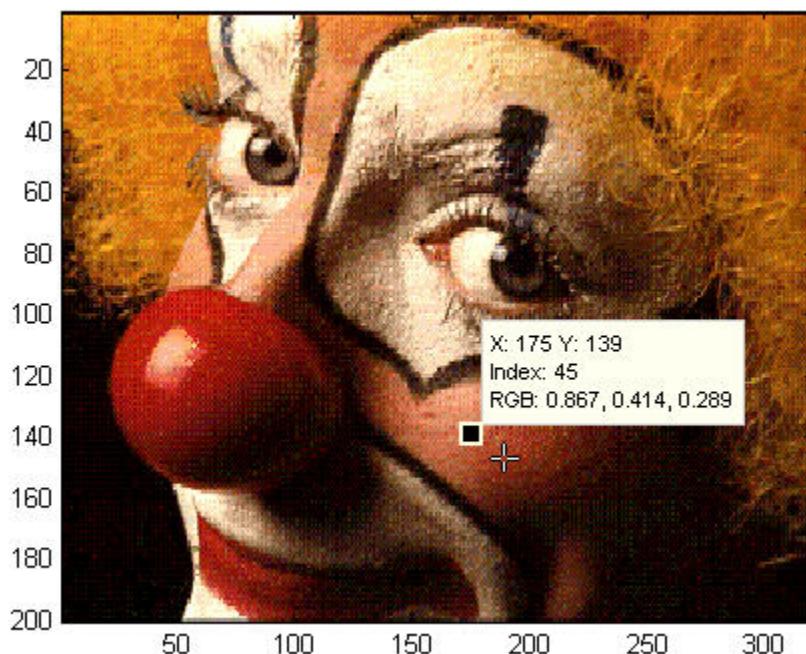
You can drag the datatip to different locations on the graph object by clicking down on the datatip and dragging the mouse. You can also use the arrow keys to move the datatip.



Note: Surface plots and 3-D bar graphs can contain NaN values. If you drag a datatip to a location coded as NaN, the datatip will disappear (because its coordinates become (NaN, NaN, NaN)). You can continue to drag it invisibly, however, and it will reappear when it is over a non-NaN location. However, if you create a new datatip while the previous current one is invisible, the previous one cannot be retrieved.

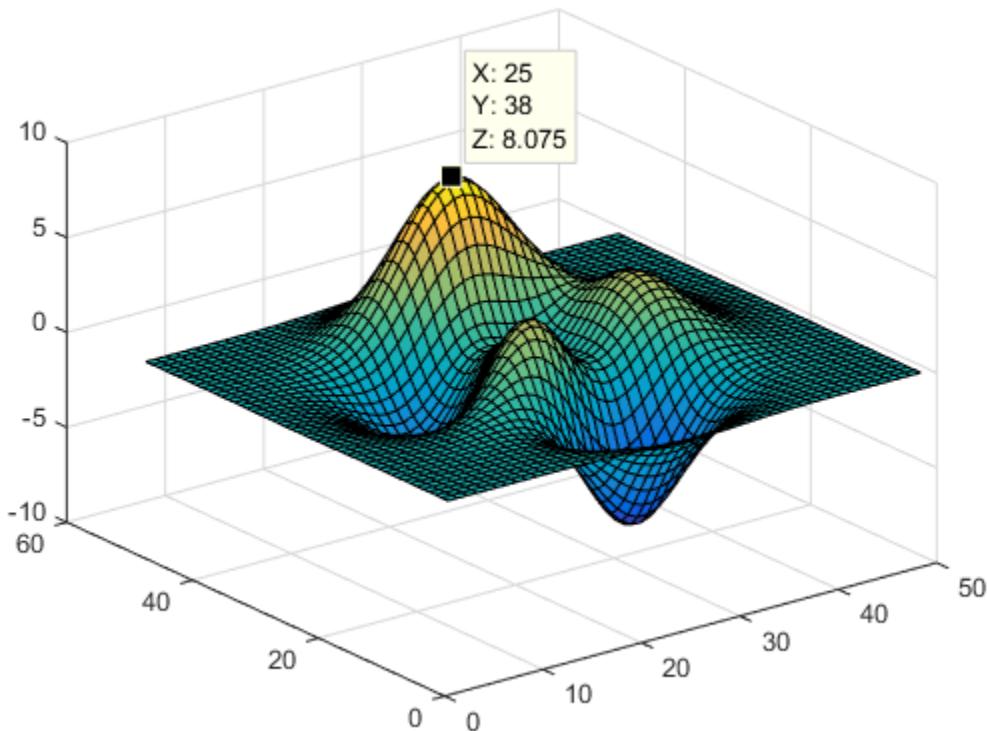
Datatips on Image Objects

Datatips on images display the x - and y -coordinates as well as the RGB values and a color index (for indexed images), as show below:



Datatips on 3-D Objects

You can use datatips to read data points on 3-D graphs as well. In 3-D views, data tips display the x -, y - and z -coordinates.



Creating Multiple Data Tips

Normally, there is only one datatip displayed at one time. However, you can display multiple datatips simultaneously on a graph. This is a simple way to annotate a number of points on a graph.

Use the following procedure to create multiple datatips.

- 1 Enable data cursor mode from the figure toolbar. The cursor changes to a cross.
- 2 Click on the graph to insert a datatip.
- 3 Right-click to display the context menu. Select **Create New Datatip**.
- 4 Click on the graph to place the second datatip.

Deleting Datatips

You can remove the most recently added datatip or all datatips. When in data cursor mode, right-click to display the context menu.

- Select **Delete Current Datatip** or press the **Delete** key to remove the last datatip that you added.
- Select **Delete All Datatips** to remove all datatips.

Customizing Data Cursor Text

You can customize the text displayed by the data cursor using the **datacursormode** function. Use the last two items in the Data Cursor context menu to for this purpose:

- **Edit Text Update Function** — Opens an editor window to let you modify the function currently being used to place text in datatips
- **Select Text Update Function** — Opens an input file dialog for you to navigate to and select a MATLAB file to use to format text in datatips you subsequently create

When you select **Edit Text Update Function** for the first time, an editor window opens with the default text update callback, which consists of the following code:

```
function output_txt = myfunction(obj,event_obj)
% Display the position of the data cursor
% obj          Currently not used (empty)
% event_obj    Handle to event object
% output_txt   Data cursor text string (string or cell array of strings).

pos = get(event_obj,'Position');
output_txt = {[ 'X: ',num2str(pos(1),4)],...
             [ 'Y: ',num2str(pos(2),4)]};

% If there is a Z-coordinate in the position, display it as well
if length(pos) > 2
    output_txt{end+1} = [ 'Z: ',num2str(pos(3),4)];
end
```

You can modify this code to display properties of the graphics object other than position. If you want to do so, you should first save this code to a MATLAB file before changing it, and select that file if you want to revert to default datatip displays during the same session.

If for example you save it as **def_datatip_cb.m**, and then modify the code and save it to another file, you can then choose between the default behavior and customized behavior by choosing **Select Text Update Function** from the context menu and selecting one of the callbacks you saved.

See the Examples section of the **datacursormode** reference page for more information on using data cursor objects and update functions. Also see the example of customizing datatip text in “Using Data Tips to Explore Graphs” in the MATLAB Data Analysis documentation.

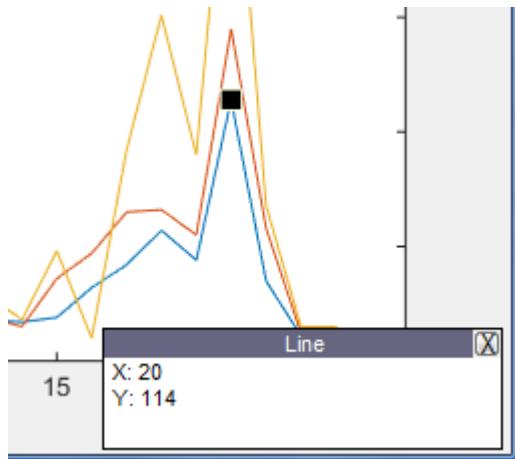
Display Style — Datatip or Cursor Window

By default, the data cursor displays values as a datatip (small text box located next to the data point). You can also display a single data value in a cursor window that is anchored within the figure window. You can place multiple datatips on a graph, which makes this display style useful for annotations.

The cursor window style is particularly useful when you want to drag the data cursor to explore image and surface data; numeric information in the window updates without obscuring the any of the figure's symbology.

To use the cursor window, change the display style as follows:

- 1 While in data cursor mode, right-click to display the context menu.
- 2 Mouse over the **Display Style** item.
- 3 Select **Window Inside Figure**.



Note: If you change the data cursor **Display Style** from **Datatip** to **Window Inside Figure** with the context menu, only the most recent data tip is displayed; all other

existing data tips are removed because the window can display only one datatip at a time.

Selection Style — Select Data Points or Interpolate Points on Graph

By default, the data cursor displays the values of the data point nearest to the point you click with the mouse, and the data marker snaps to this point. The data cursor can also determine the values of points that lie in between the data defining the graph, by linearly interpolating between the two data points closest to the location you click the mouse.

Enabling Interpolation Mode

If you want to be able to select any point along a graph and display its value, use the following procedure:

- 1 While in data cursor mode, right-click to display the context menu.
- 2 Mouse over the **Selection Style** item.
- 3 Select **Mouse Position**.

MATLAB does not honor interpolation mode when you use the arrow keys to move a datatip to a new location.

Exporting Data Value to Workspace Variable

You can export the values displayed with the data cursor to MATLAB workspace variables. To do this, display the right-click context menu while in data cursor mode and select **Export Cursor Data to Workspace**.

The Export Cursor Data to Workspace dialog then displays so that you can name the workspace variable.

Clicking **OK** creates a MATLAB structure with the specified name in your base workspace, containing the following fields:

- **Target** — Handle of the graphics object containing the data point
- **Position** — x - and y - (and z -) coordinates of the data cursor location in axes data units

Line and lineseries objects have an additional field:

- **DataIndex** — A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

For example, if you saved the workspace variable as **cursor_info**, then you would access the position data by referencing the **Position** field.

```
cursor_info.Position  
ans =  
    0.4189  0.1746  0
```

Zooming in Graphs

In this section...

- “Zooming in 2-D and 3-D” on page 3-15
- “Zooming in 2-D Views” on page 3-15

Zooming in 2-D and 3-D

Zooming changes the magnification of a graph without changing the size of the figure or axes. Zooming is useful to see greater detail in a small area. As explained below, zooming behaves differently depending on whether it is applied to a 2-D or 3-D view.

Enable zooming by clicking one of the zoom icons . Select + to zoom in and – to zoom out.

Tip When in zoom in mode, you can use **Shift**+click to zoom out (i.e., press and hold down the **Shift** key while clicking the mouse). You can also right-click and zoom out or restore the plot to its original view using the context menu.

Zooming in 2-D Views

In 2-D views, click the area of the axes where you want to zoom in, or drag the cursor to draw a box around the area you want to zoom in on. MATLAB redraws the axes, changing the limits to display the specified area.

When you right-click in Zoom mode, the context menu enables you to:

- Zoom out
- Reset to the view of the graph when it was plotted (undo one or more changes of view)
- Constrain zooming to expand only the *x*-axis (horizontal zoom)
- Constrain zooming to expand only the *y*-axis (vertical zoom)

Undoing Zoom Actions

If you want to reset the graph to its original view, right-click to display the context menu and select **Reset to Original View**. You can also use the **Undo** item on the **Edit** menu to undo each operation you performed on your graph.

Zoom Constrained to Horizontal or Vertical

In 2-D views, you can constrain zoom to operate in either the horizontal or vertical direction. To do this, right-click to display the context menu while in zoom mode and select the desired constraint from the **Zoom Options** submenu, as illustrated in the previous figure. Horizontal zooming is useful for exploring time series graphs that have dense intervals. Vertical zooming can help you see minor variations in places where the YData range is small compared to the y-axis limits.

Zooming in 3-D Views

In 3-D views, moving the cursor up or to the right zooms in, while moving the cursor down or to the left zooms out. Both toolbar icons enable the same behavior. 3-D zooming does not change the axes limits, as in 2-D zooming. Instead it changes the view (specifically, the axes `CameraViewAngle` property) as if you were looking through a camera with a zoom lens.

Panning — Shifting Your View of the Graph

You can move your view of a graph up and down as well as left and right with the pan tool. Panning is useful when you have zoomed in on a graph and want to translate the plot to view different portions.

Click the hand icon on the figure toolbar to enable panning . In pan mode you can move up, down, left, or right. You can constrain movement to be vertical or horizontal only by right-clicking and selecting one of the **Pan Options** from the pan tool's context menu.

3-D panning moves the axes with the object, because the 3-D view is not aligned to the x -, y -, or z -axis. The axes limits do not change as in 2-D panning.

Rotate in 3-D

In this section...

- “Enabling 3-D Rotation” on page 3-18
- “Selecting Predefined Views” on page 3-18
- “Rotation Style for Complex Graphs” on page 3-19
- “Undo/Redo — Eliminating Mistakes” on page 3-21

Enabling 3-D Rotation

You can easily rotate graphs to any orientation with the mouse. Rotation involves the reorientation of the axes and all the graphics objects it contains. Therefore none of the data defining the graphics objects is affected by rotation; instead the orientation of the x -, y -, and z -axes changes with respect to the viewer.

There are three ways to enable Rotate 3D mode:

- Select **Rotate 3D** from the **Tools** menu.
- Click the Rotate 3D icon in the figure toolbar .
- Execute the `rotate3d` command.

Once the mode is enabled, you press and hold the mouse button while moving the cursor to rotate the graph.

Selecting Predefined Views

When Rotate 3D mode is enabled, you can control various rotation options from the right-click context menu.

You can rotate to predefined views on the right-click context menu:

- **Reset to Original View** — Reset to the default view (azimuth -37.5° , elevation 30°).
- **Go to X-Y View** — View graph along the z -axis (azimuth 0° , elevation 90°).
- **Go to X-Z View** — View graph along the y -axis (azimuth 0° , elevation 0°).
- **Go to Y-Z View** — View graph along the x -axis (azimuth 90° , elevation 0°).

Rotation Style for Complex Graphs

You can select from two rotation styles on the right-click context menu's **Rotation Options** submenu:

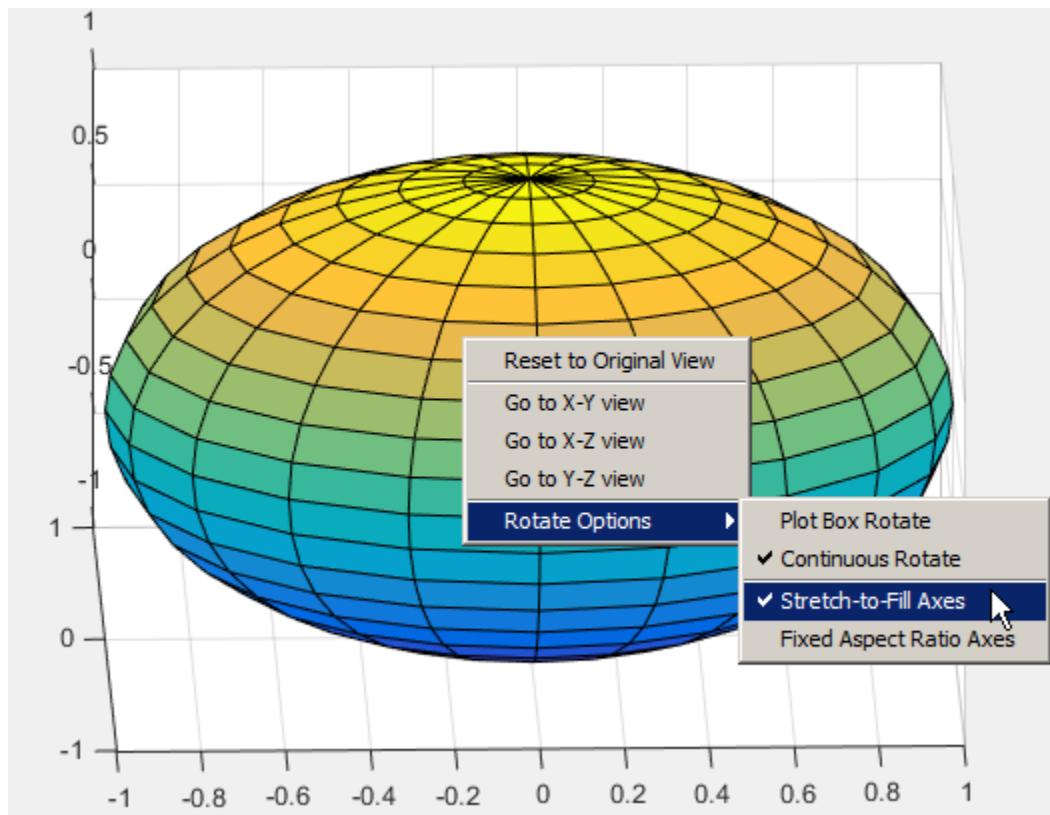
- **Plot Box Rotate** — Display only the axes bounding box for faster rotation of complex objects. Use this option if the default **Continuous Rotate** style is unacceptably slow.
- **Continuous Rotate** — Display all graphics during rotation.

Axes Behavior During Rotation

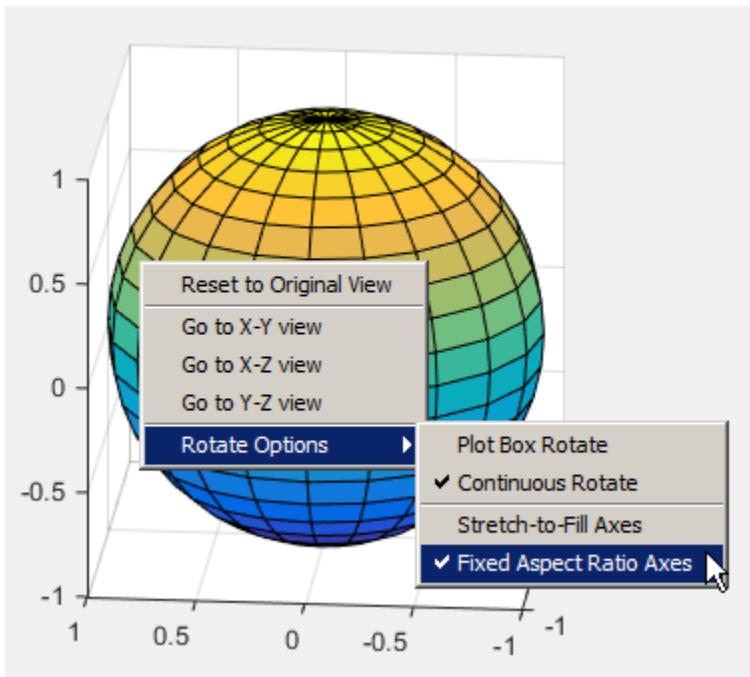
You can select two types of behavior with respect to the aspect ratio of axes during rotation:

- **Stretch-to-Fill Axes** – Default axes behavior is optimized for 2-D plots. Graphs fit the rectangular shape of the figure.
- **Fixed Aspect Ratio Axes** – Maintains a fixed shape of objects in the axes as they are rotated. Use this setting when rotating 3-D plots.

The following pictures illustrate a sphere as it is rotated with **Stretch-to-Fill Axes** selected. Notice that the sphere is not round due to the selected aspect ratio.



The next picture shows how the **Fixed Aspect Ratio Axes** option results in a sphere that maintains its proper shape as it is rotated.



Undo/Redo — Eliminating Mistakes

The figure **Edit** menu contains two items that enable you to undo any zoom, pan, or rotate operation.

Undo — Remove the effect of the last operation.

Redo — Perform again the last operation that you removed by selecting **Undo**.

Annotating Graphs

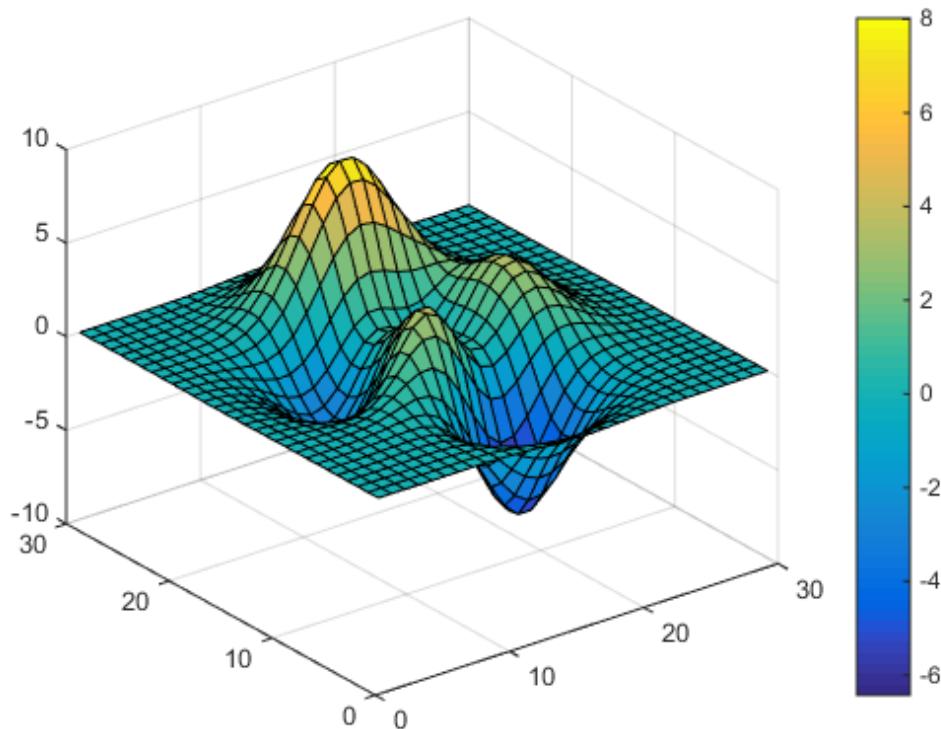
- “Change Mapping of Data Values into the Colormap” on page 4-2
- “Change Colorbar Width” on page 4-5
- “Include Subset of Objects in Graph Legend” on page 4-8
- “Display One Legend Entry for Group of Objects” on page 4-11
- “Specify Legend Descriptions During Line Creation” on page 4-14
- “Add Text to Specific Points on Graph” on page 4-17
- “Include Variable Values in Graph Text” on page 4-23
- “Text with Mathematical Expression Using LaTeX” on page 4-26
- “Text with Greek Letters and Special Characters” on page 4-31
- “Add Annotations to Graph Interactively” on page 4-35
- “Add Text to Graph Interactively” on page 4-38
- “Add Colorbar to Graph Interactively” on page 4-44
- “Align Objects in Graph Using Alignment Tools” on page 4-48

Change Mapping of Data Values into the Colormap

This example shows how to control the mapping of data values into the colormap so that positive data values map to different colors and negative data values map to black.

Create a surface plot of the `peaks` function and add a colorbar.

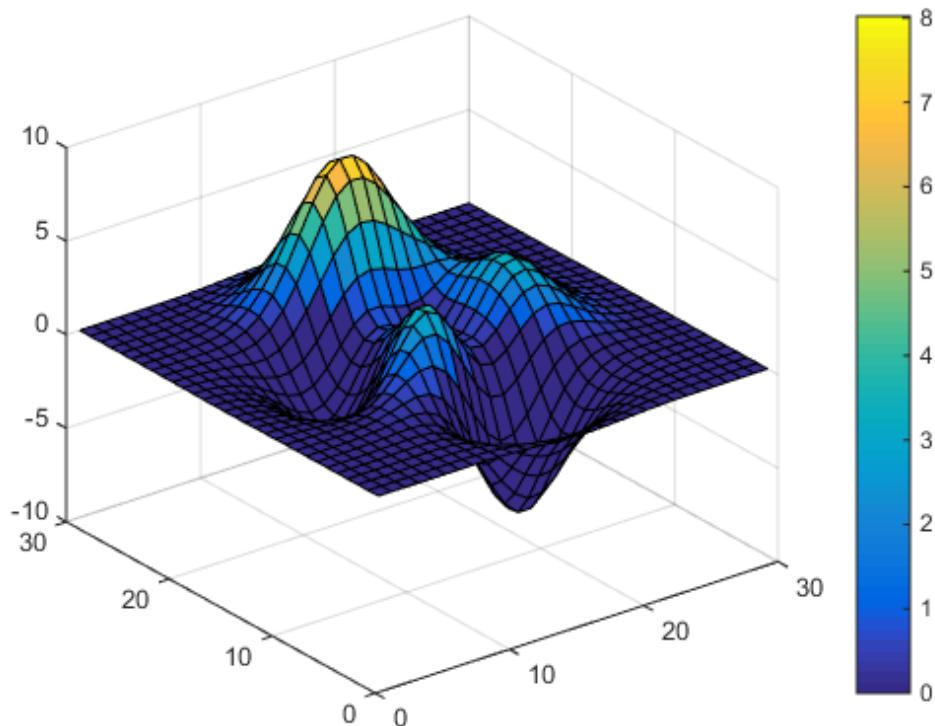
```
figure  
surf(peaks(30))  
colorbar
```



Use `caxis` to return `cmin` and `cmax`, the current data values that map to the minimum and maximum values of the colormap, respectively. Then, use `caxis` again to change

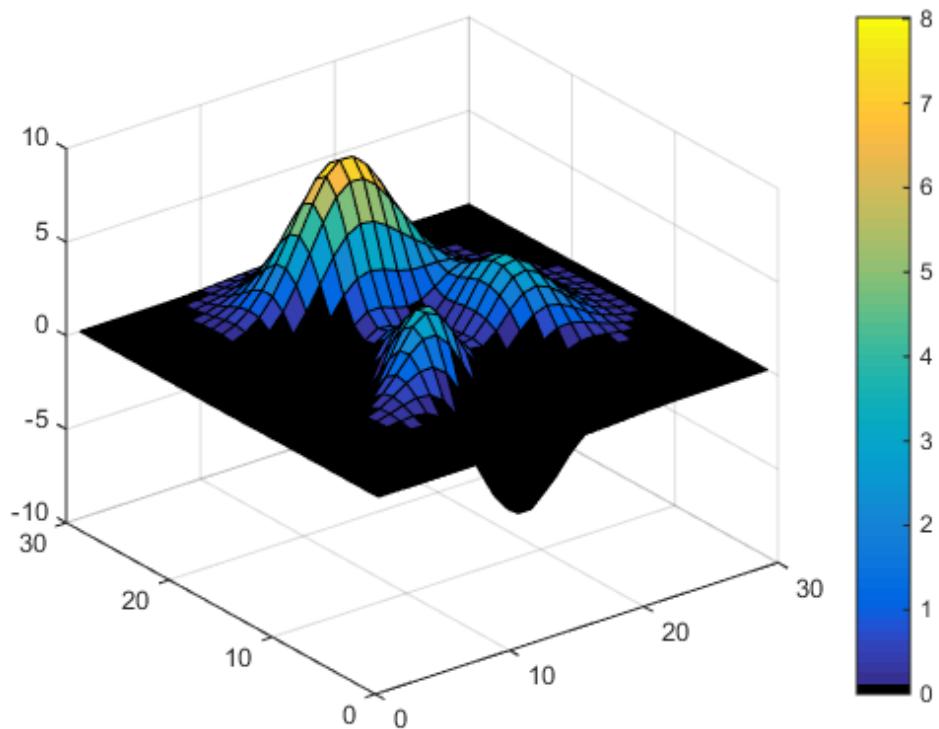
the scaling of data values into the colormap. Set the minimum color limit to zero to map negative data values to the first color in the colormap. Keep the same maximum color limit, `cmax`.

```
[cmin,cmax] = caxis;  
caxis([0,cmax])
```



Set the first color in the colormap to black by setting the first row of the current colormap, `map`, to `[0,0,0]`. Apply the updated colormap to the figure.

```
map = colormap; % current colormap  
map(1,:) = [0,0,0];  
colormap(map)
```



All data values less than or equal to zero map to black.

See Also

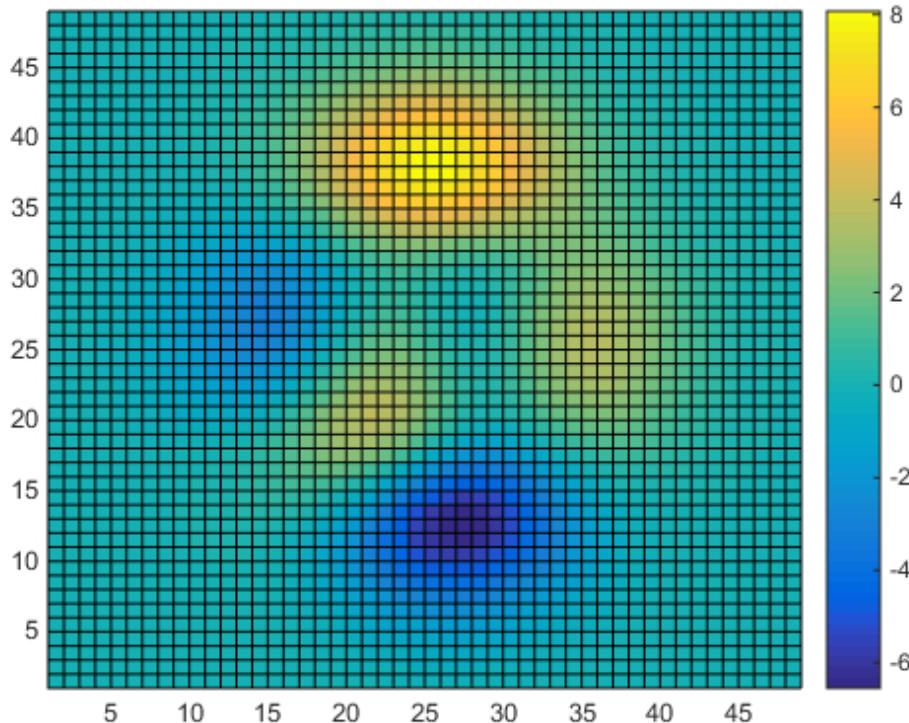
[caxis](#) | [colorbar](#) | [colormap](#) | [surf](#)

Change Colorbar Width

This example shows how to change the width of the colorbar by setting its **Position** property. The **Position** property sets the location and size of the colorbar. Specify the **Position** property value as a four-element vector of the form **[left, bottom, width, height]**, where the first two elements set the colorbar position relative to the figure, and the last two elements set its size.

Create a checkerboard plot of the **peaks** function and add a colorbar.

```
figure  
pcolor(peaks)  
c = colorbar;
```

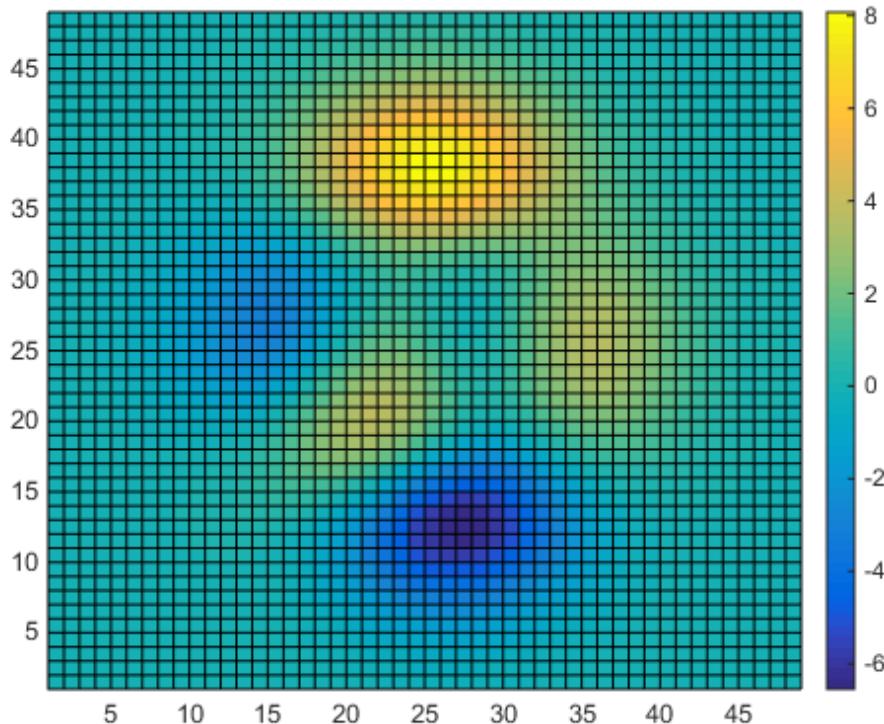


Store the current positions of the axes and the colorbar.

```
ax = gca;
axpos = ax.Position;
cpos = c.Position;
```

Change the colorbar width to half of its original width by adjusting the third element in `cpos`. Set the colorbar `Position` property to the updated position vector. Then, reset the axes to its original position so that it does not overlap the colorbar.

```
cpos(3) = 0.5*cpos(3);
c.Position = cpos;
ax.Position = axpos;
```



The graph displays a colorbar that has a narrow width.

See Also

[colorbar](#) | [pcolor](#)

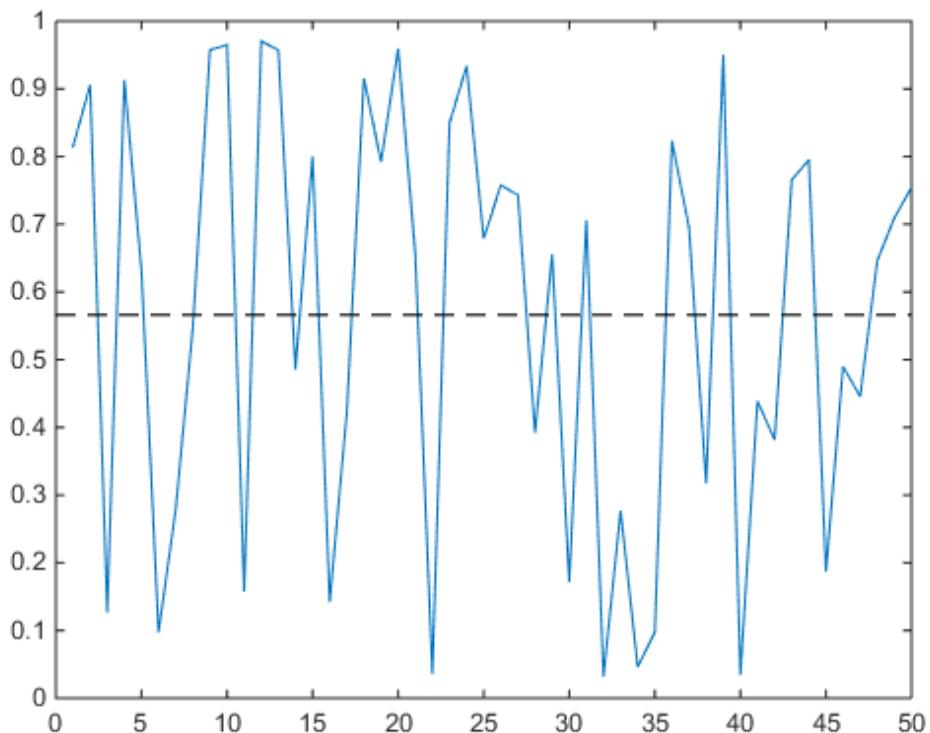
Include Subset of Objects in Graph Legend

This example shows how to add a legend to a graph that includes only a subset of objects in the graph.

Plot a vector of random data. Then, calculate the mean of the data. Draw a black, dashed, horizontal line at the value of the calculated mean.

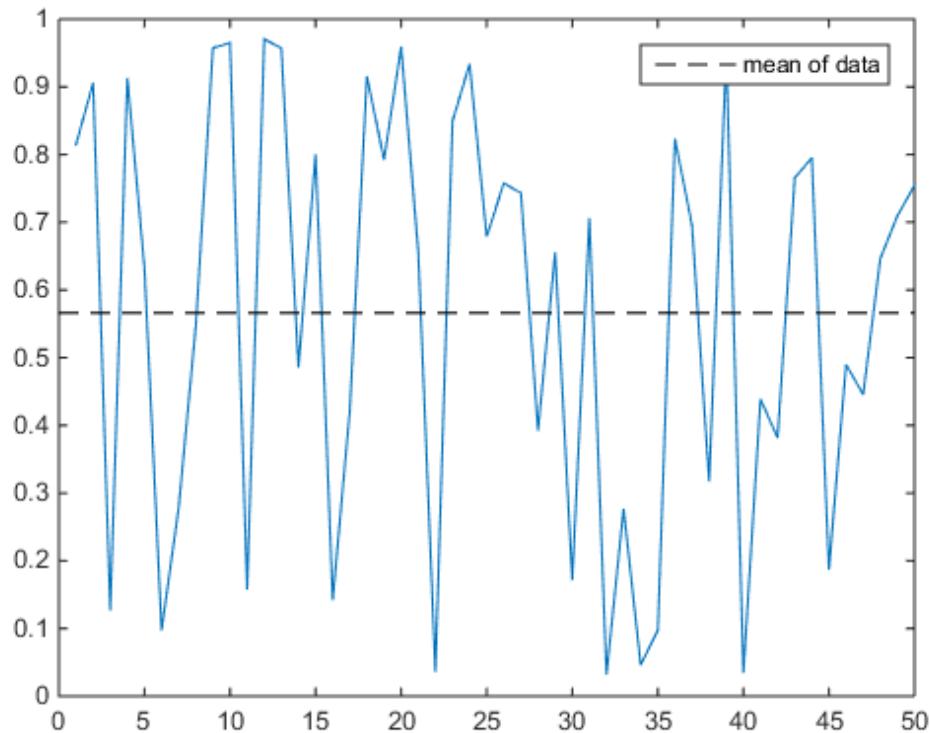
```
dat = rand(50,1);
plot(dat)

m = mean(dat);
ax = gca;
xlimits = ax.XLim;
h = line([xlimits(1),xlimits(2)],[m,m], 'Color', 'k', 'LineStyle', '--');
```



Add a legend for the horizontal line by passing the line object to the `legend` function.

```
legend(h, 'mean of data')
```



The legend contains a description for the horizontal line, but not the line plot of random data.

See Also

[legend](#) | [line](#) | [mean](#) | [plot](#)

Display One Legend Entry for Group of Objects

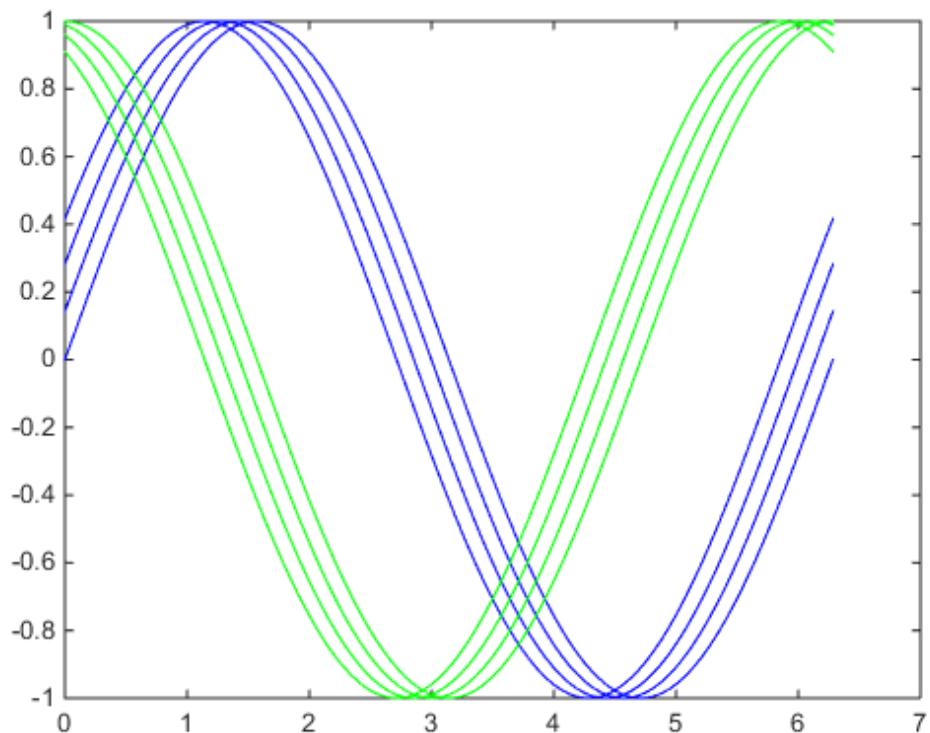
This example shows how to group sets of lines together and add a legend to the graph that contains one entry for each group.

Create two groups, g1 and g2. Plot four sine waves and four cosine waves. Group all the sine plot lines together by setting their **Parent** property to g1. Group all the cosine plot lines together by setting their **Parent** property to g2.

```
g1 = hggroup;
g2 = hggroup;

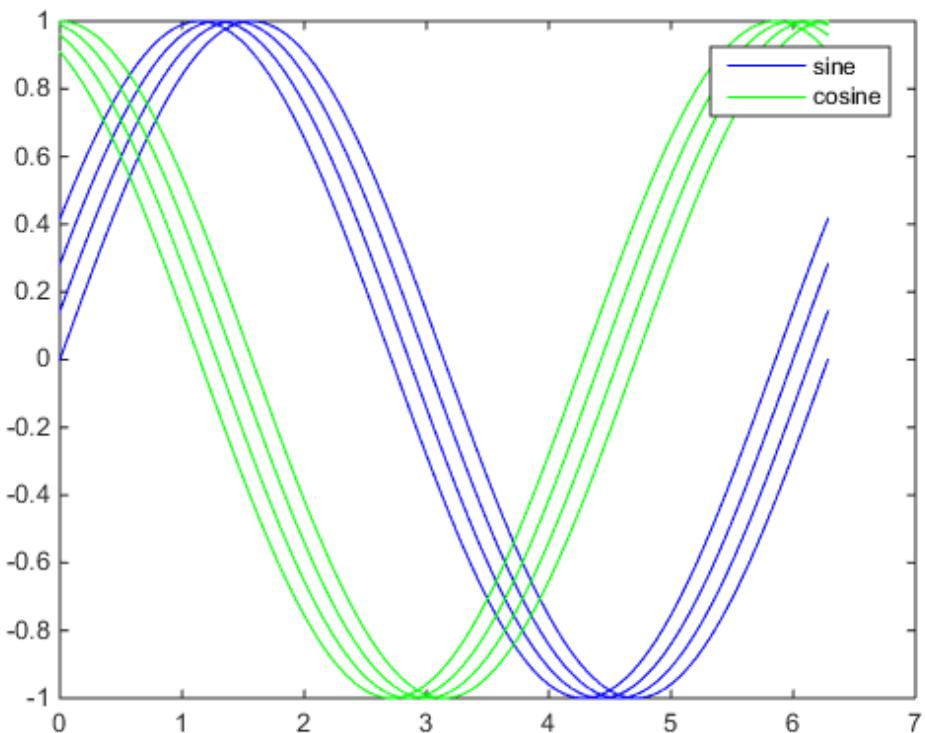
t = linspace(0,2*pi,100);
plot(t,sin(t), 'b', 'Parent', g1)
hold on
plot(t,sin(t+1/7), 'b', 'Parent', g1)
plot(t,sin(t+2/7), 'b', 'Parent', g1)
plot(t,sin(t+3/7), 'b', 'Parent', g1)

plot(t,cos(t), 'g', 'Parent', g2)
plot(t,cos(t+1/7), 'g', 'Parent', g2)
plot(t,cos(t+2/7), 'g', 'Parent', g2)
plot(t,cos(t+3/7), 'g', 'Parent', g2)
hold off % reset hold state to off
```



Add a legend with one description for each group of lines.

```
legend([g1,g2], 'sine', 'cosine')
```



The legend contains two entries, one for each group of lines.

See Also

[hggroup](#) | [hold](#) | [legend](#) | [plot](#)

Specify Legend Descriptions During Line Creation

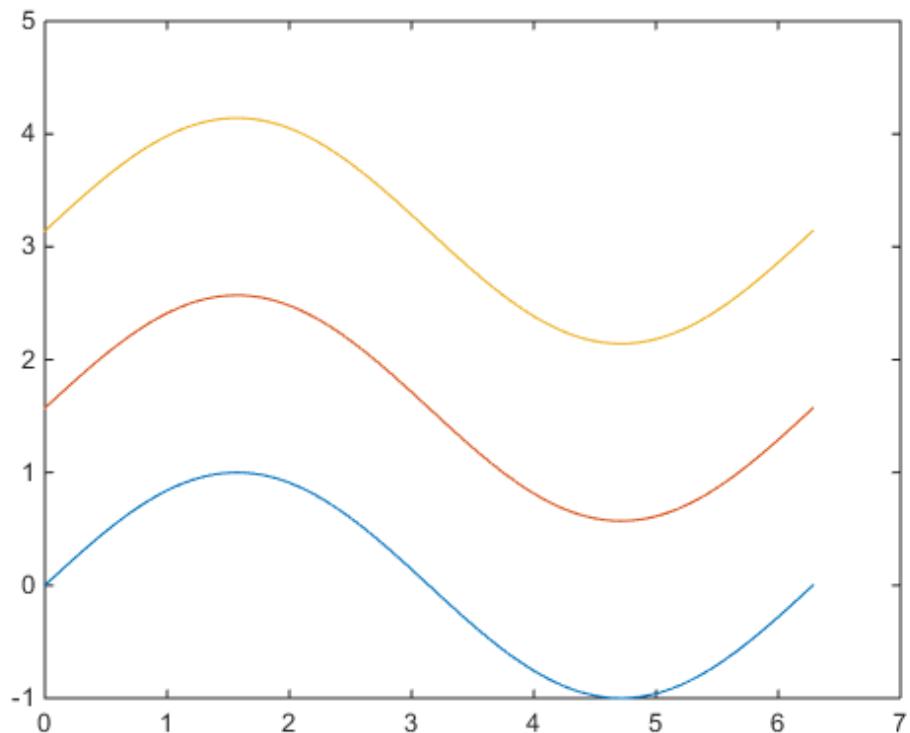
This example shows how to plot data and specify its associated legend description during the plotting command.

Plot three sine curves. For each line, set the `DisplayName` property to a descriptive string.

```
x = linspace(0,2*pi,100);
y1 = sin(x);
p1 = plot(x,y1,'DisplayName','sin(x)');
hold on

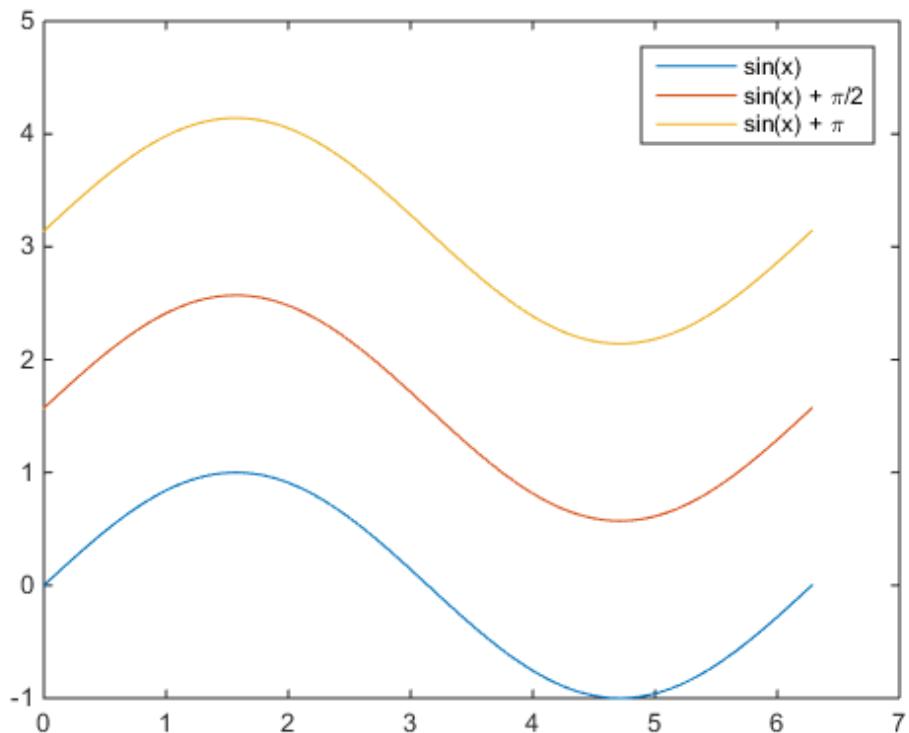
y2 = sin(x) + pi/2;
p2 = plot(x,y2,'DisplayName','sin(x) + \pi/2');

y3 = sin(x) + pi;
p3 = plot(x,y3,'DisplayName','sin(x) + \pi');
hold off
```



The graph does not display the legend until you call the `legend` function. Display the legend for the three lines.

```
legend([p1 p2 p3])
```



If you do not pass strings to the `legend` function, then `legend` uses the `DisplayName` properties as descriptions. If the `DisplayName` property does not have a value, then `legend` uses a default string of the form '`'data1'`', '`'data2'`', and so on.

See Also

`hold` | `legend` | `plot`

Add Text to Specific Points on Graph

In this section...

[“Add Text to Three Data Points on Graph” on page 4-17](#)

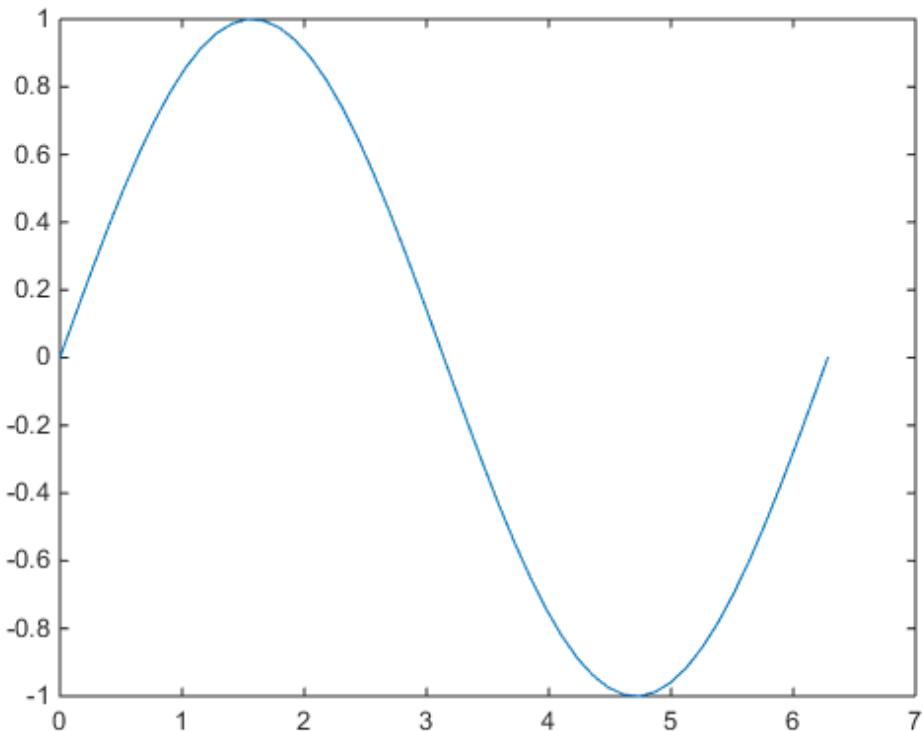
[“Determine Minimum and Maximum Points and Add Text” on page 4-20](#)

Add Text to Three Data Points on Graph

This example shows how to add text descriptions with arrows that point to three data points on a graph.

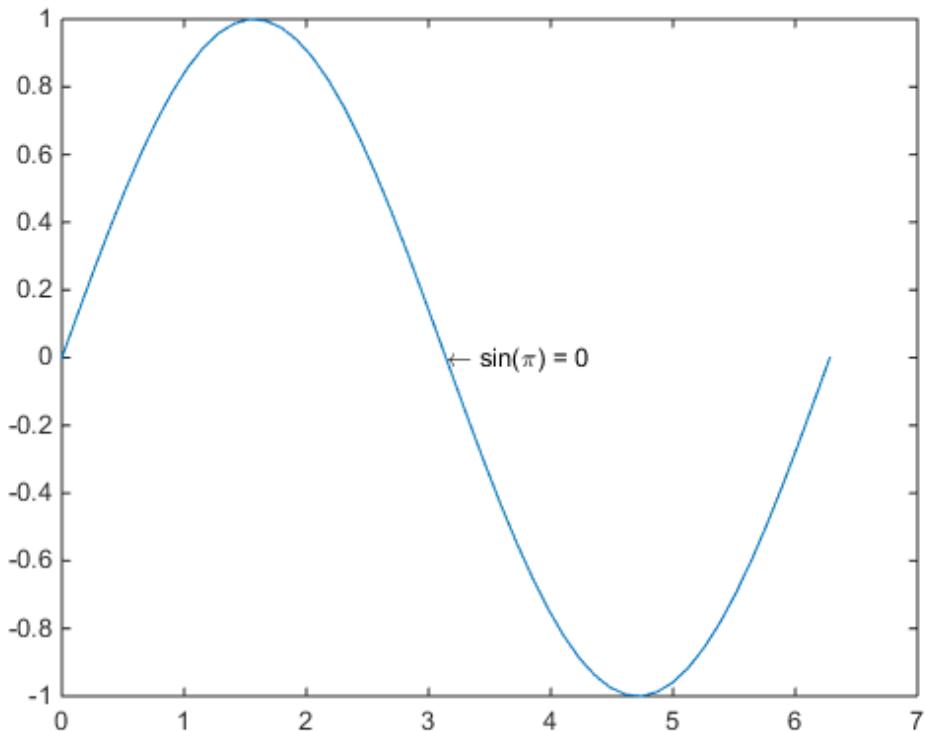
Use the `linspace` function to create `t` as a vector of 50 values between 0 and 2π . Create `y` as sine values. Plot the data.

```
t = linspace(0,2*pi,50);
y = sin(t);
plot(t,y)
```



Use the `text` function to add a text description to the graph at the point $(\pi, \sin(\pi))$. The first two input arguments to this function define the text position. The third argument defines the text string. Display an arrow pointing to the left by including the TeX markup `\leftarrow` in the string. Use the TeX markup `\pi` for the Greek letter π .

```
x1 = pi;  
y1 = sin(pi);  
str1 = '\leftarrow sin(\pi) = 0';  
text(x1,y1,str1)
```

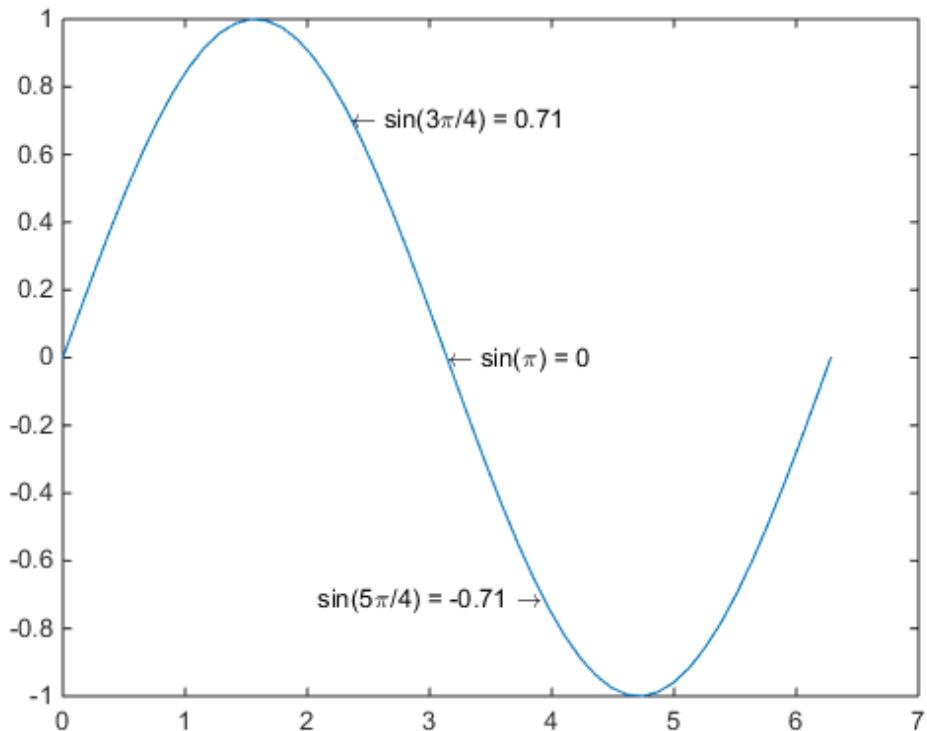


Add text descriptions to two more data points on the graph. By default, the text aligns so that the specified data point is to the left of the string. Specify the **HorizontalAlignment** property for the last description as 'right' so that the data point is to the right of the string. Use the TeX markup \rightarrow to display an arrow pointing to the right.

```
x2 = 3*pi/4;
y2 = sin(3*pi/4);
str2 = '\leftarrow sin(3\pi/4) = 0.71';
text(x2,y2,str2)

x3 = 5*pi/4;
y3 = sin(5*pi/4);
str3 = 'sin(5\pi/4) = -0.71 \rightarrow';
```

```
text(x3,y3,str3,'HorizontalAlignment','right')
```

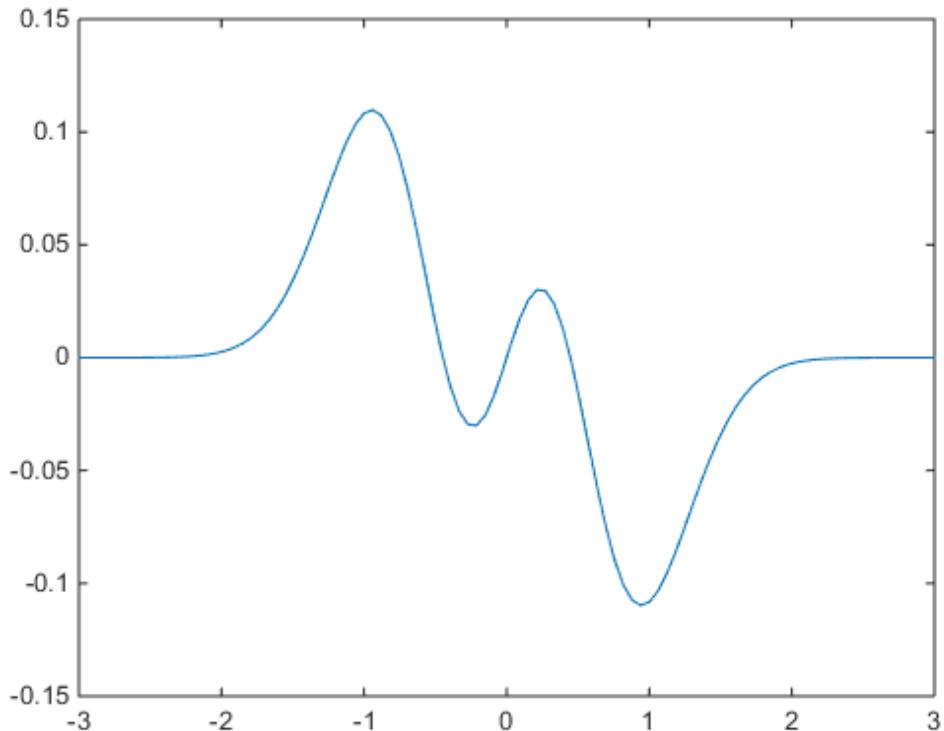


Determine Minimum and Maximum Points and Add Text

This example shows how to determine the minimum and maximum data points on a graph and add text descriptions next to these values.

Create a plot.

```
x = linspace(-3,3);
y = (x/5-x.^3).*exp(-2*x.^2);
plot(x,y)
```

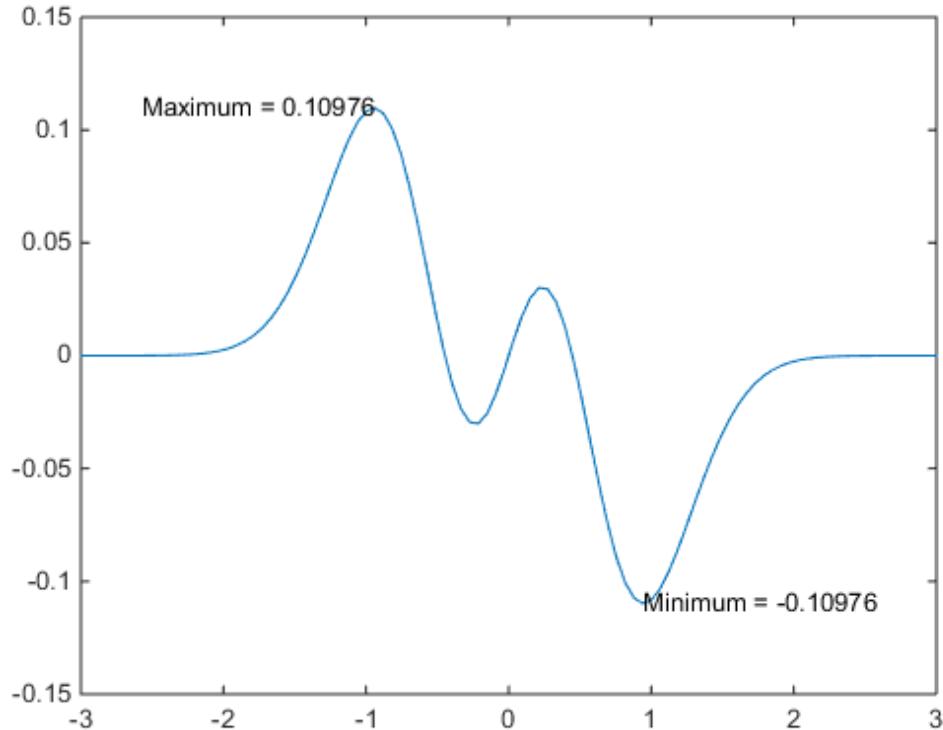


Find the indices of the minimum and maximum values in y . Use the indices to determine the (x,y) values at the minimum and maximum points.

```
indexmin = find(min(y) == y);  
xmin = x(indexmin);  
ymin = y(indexmin);  
  
indexmax = find(max(y) == y);  
xmax = x(indexmax);  
ymax = y(indexmax);
```

Add text to the graph at these points. Use `num2str` to convert the y values to strings. Specify the text alignment in relation to the point using the `HorizontalAlignment` property.

```
strmin = ['Minimum = ',num2str(ymin)];  
text(xmin,ymin,strmin,'HorizontalAlignment','left');  
  
strmax = ['Maximum = ',num2str(ymax)];  
text(xmax,ymax,strmax,'HorizontalAlignment','right');
```



See Also

`linspace` | `plot` | `text` | `title` | `xlabel` | `ylabel`

Related Examples

- “Include Variable Values in Graph Text” on page 4-23
- “Text with Greek Letters and Special Characters” on page 4-31

Include Variable Values in Graph Text

These examples show how to convert variable values to strings to include in text on a graph.

In this section...

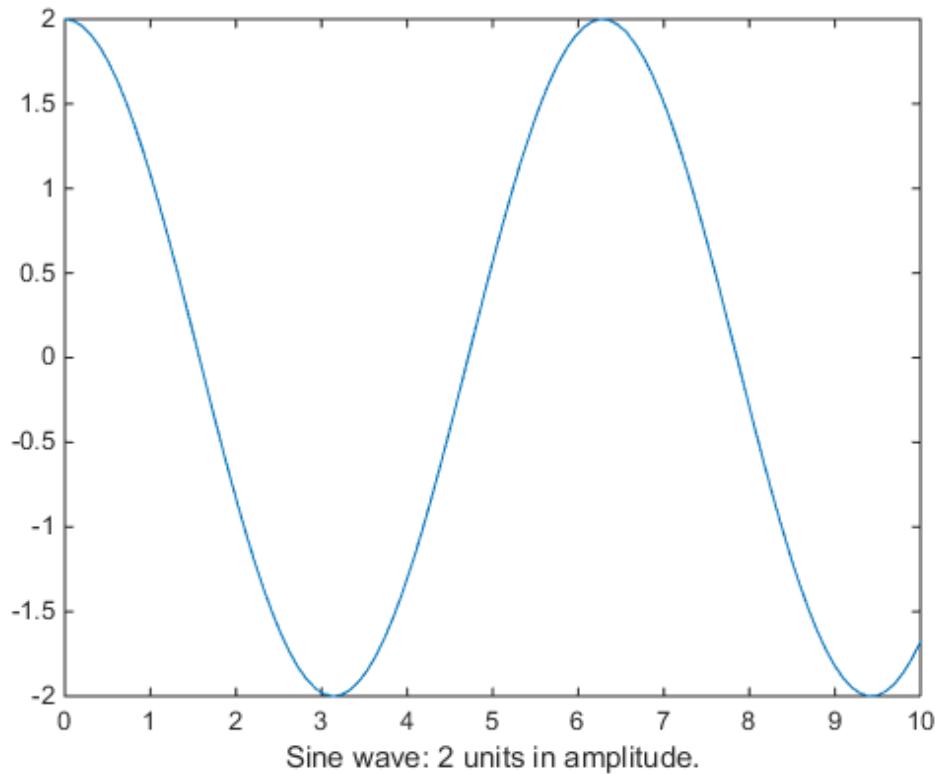
[“Include Variable Value in Axis Label” on page 4-23](#)

[“Include Loop Variable Value in Graph Title” on page 4-24](#)

Include Variable Value in Axis Label

Include a variable value in the *x*-axis label. Use the `num2str` function to convert the number to a string.

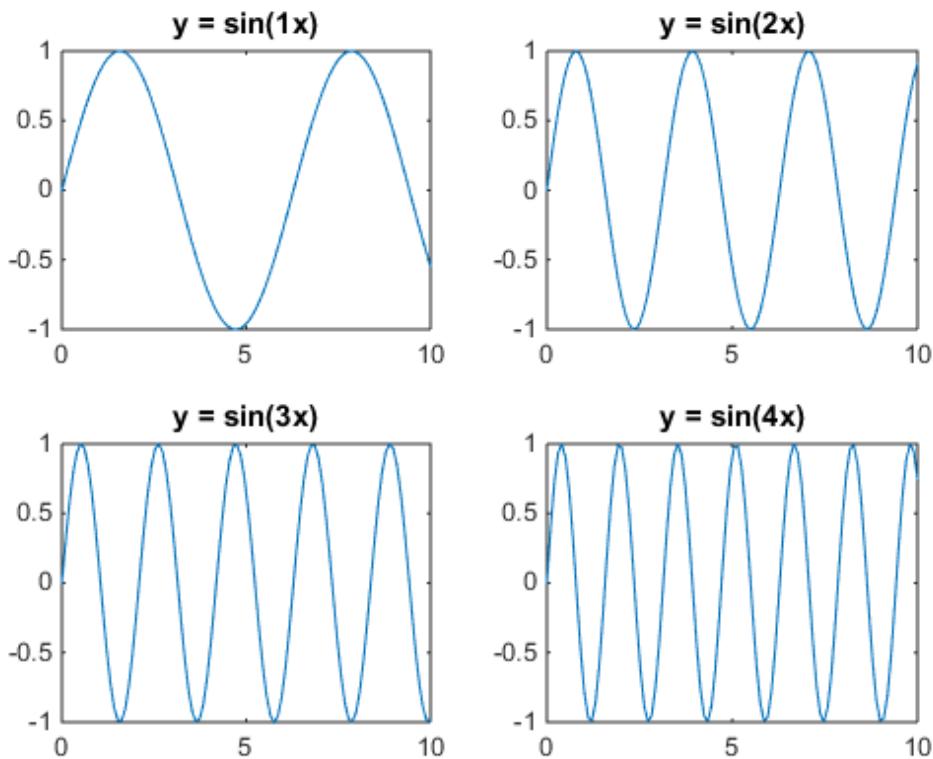
```
x = linspace(0,10);
amp = 2;
y = amp*cos(x);
plot(x,y)
xlabel(['Sine wave: ' num2str(amp) ' units in amplitude.'])
```



Include Loop Variable Value in Graph Title

Use a loop to create a figure containing four subplots. In each subplot, plot a sine wave with different frequencies based on the loop variable k . Add a title to each subplot that includes the value of k .

```
x = linspace(0,10,100);
for k = 1:4
    subplot(2,2,k);
    yk = sin(k*x);
    plot(x,yk)
    title(['y = sin(' num2str(k) 'x)'])
end
```



See Also

`figure` | `linspace` | `num2str` | `plot` | `subplot` | `title`

Related Examples

- “Add Text to Specific Points on Graph” on page 4-17
- “Text with Greek Letters and Special Characters” on page 4-31

Text with Mathematical Expression Using LaTeX

These examples show how add text to a graph that includes mathematical expressions using LaTeX.

By default, text objects in MATLAB support a subset of TeX markup. For a list of supported TeX markup, see the `text Interpreter` property description. To use additional special characters, such as integral and summation symbols, use LaTeX markup. To use LaTeX markup, you must set the `Interpreter` property of the text object to '`latex`'. For more information on LaTeX, see The LaTeX Project website at <http://www.latex-project.org/>.

In this section...

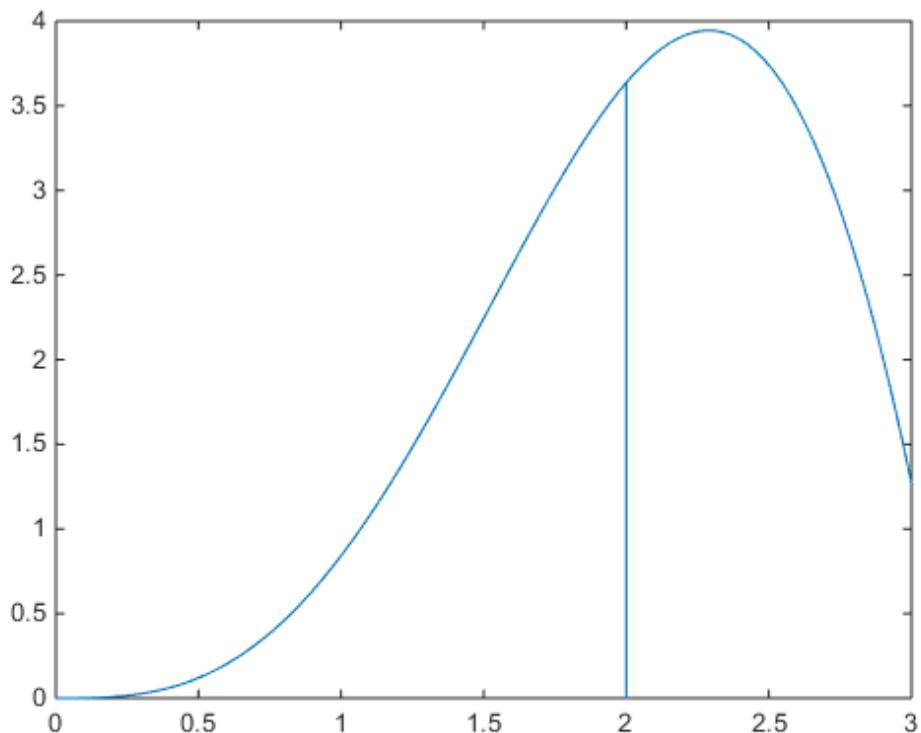
"Add Text with Integral Expression to Graph" on page 4-26

"Add Text with Summation Symbol to Graph" on page 4-28

Add Text with Integral Expression to Graph

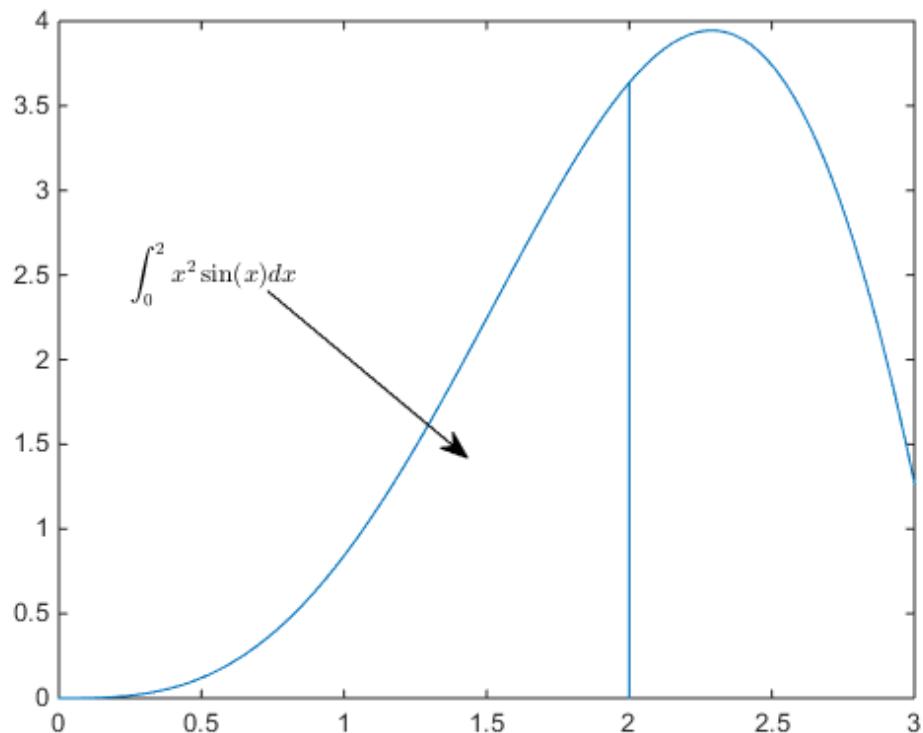
Plot $y = x^2 \sin(x)$. Draw a vertical line at $x = 2$ from the x -axis to the plotted line.

```
x = linspace(0,3);
y = x.^2.*sin(x);
plot(x,y)
line([2,2],[0,2^2*sin(2)])
```



Add text to the graph that contains an integral expression using LaTeX markup and add an arrow annotation to the graph. To use LaTeX markup, set the `Interpreter` property for the text object to '`latex`'.

```
str = '$$ \int_0^2 x^2 \sin(x) dx $$';
text(0.25,2.5,str,'Interpreter','latex')
annotation('arrow','X',[0.32,0.5],'Y',[0.6,0.4])
```



Add Text with Summation Symbol to Graph

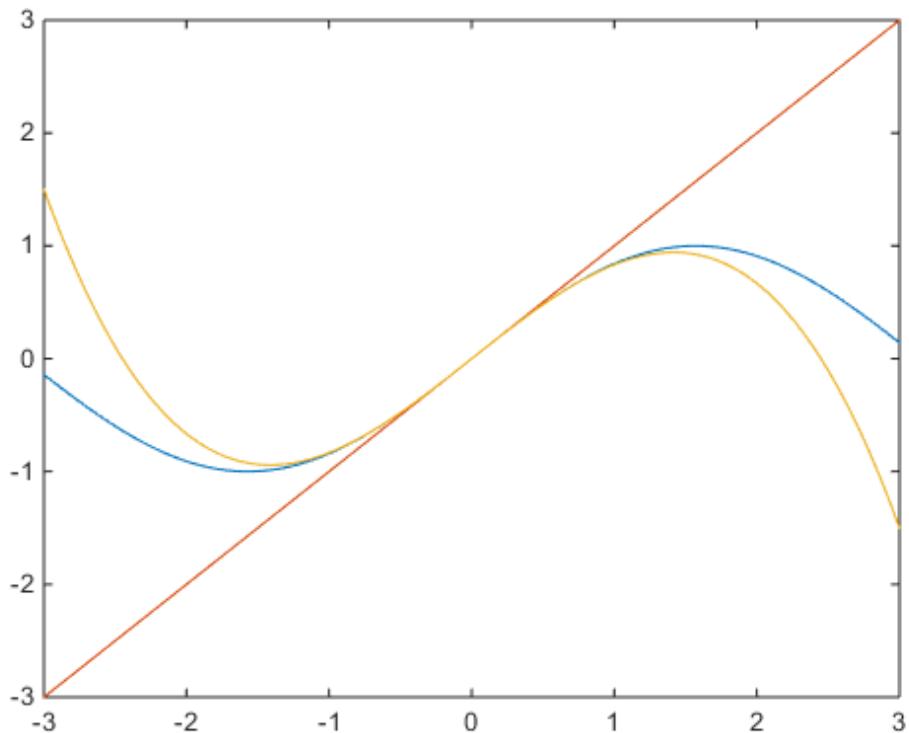
Plot the sine function and plot two polynomials.

```
x = linspace(-3,3);
y = sin(x);
plot(x,y)

y0 = x;
hold on
plot(x,y0)

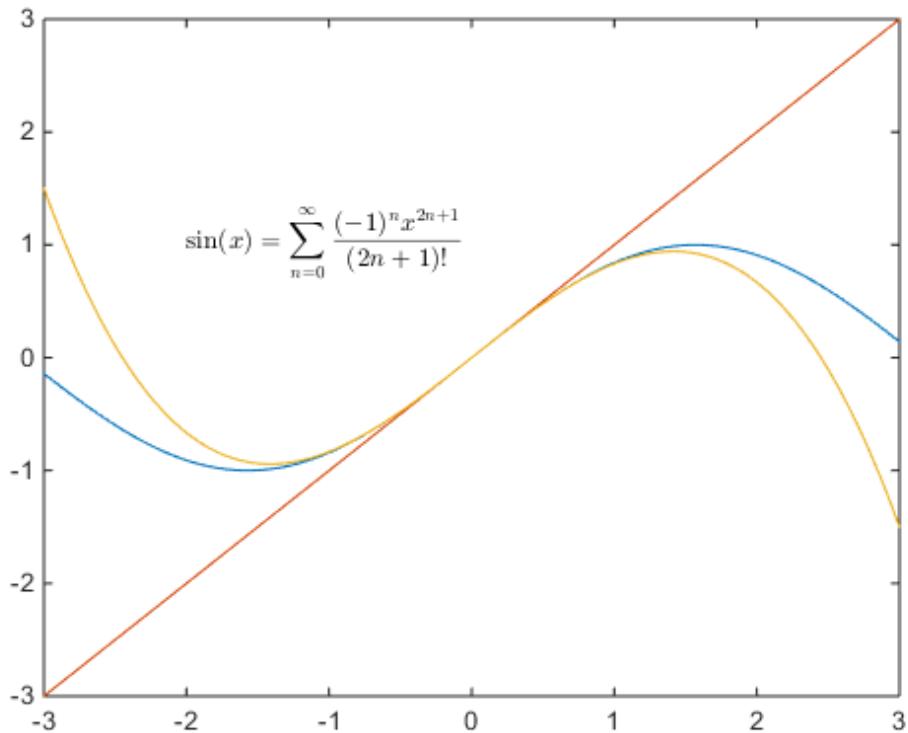
y1 = x - x.^3/6;
plot(x,y1)
```

```
hold off
```



Add a text description to the graph that includes a summation symbol using LaTeX markup. To use LaTeX, set the `Interpreter` property for the text object to '`latex`'.

```
str = '$$\sin(x) = \sum_{n=0}^{\infty}\frac{(-1)^n x^{2n+1}}{(2n+1)!}$$';  
text(-2,1,str,'Interpreter','latex')
```



See Also

[annotation](#) | [Text Properties](#) | `text` | `title` | `xlabel` | `ylabel`

Related Examples

- “Add Text to Specific Points on Graph” on page 4-17
- “Text with Greek Letters and Special Characters” on page 4-31

Text with Greek Letters and Special Characters

These examples show how to add text to a graph that includes Greek letters and other special characters. To define these characters, use TeX markup. For example, use `^` to display superscripts and `_` to display subscripts. For a list of supported TeX markup, see the text `Interpreter` property description.

In this section...

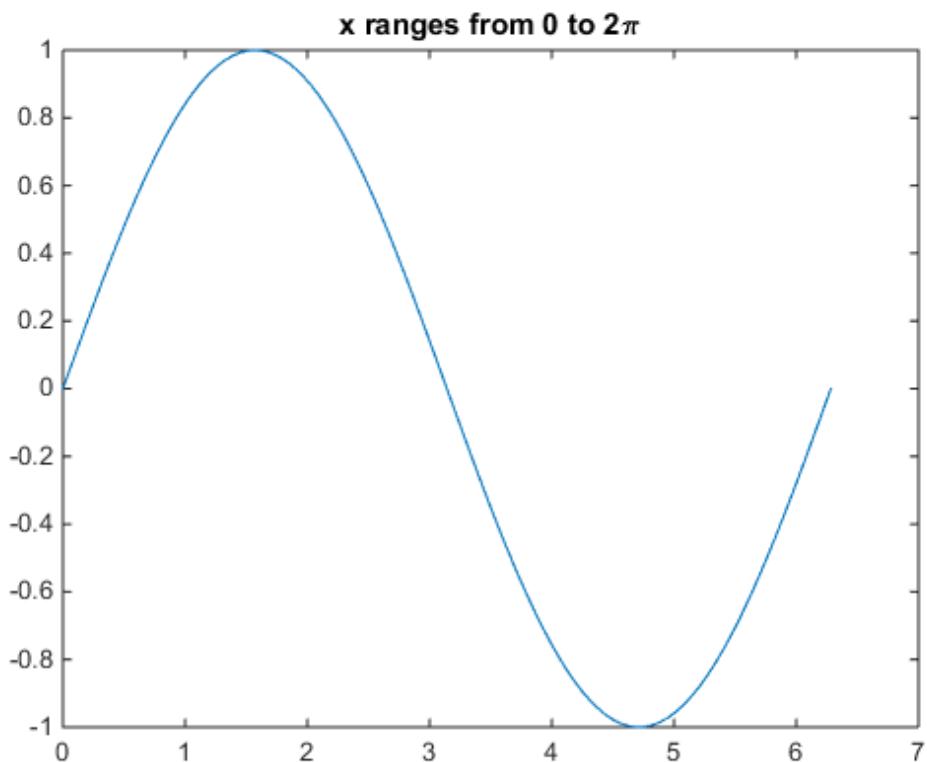
[“Include Greek Letters in Graph Text” on page 4-31](#)

[“Include Superscripts and Annotations in Graph Text” on page 4-32](#)

Include Greek Letters in Graph Text

Create a simple line plot and add a title to the graph. Include the Greek letter π in the title by using the TeX markup `\pi`.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)
title('x ranges from 0 to 2\pi')
```

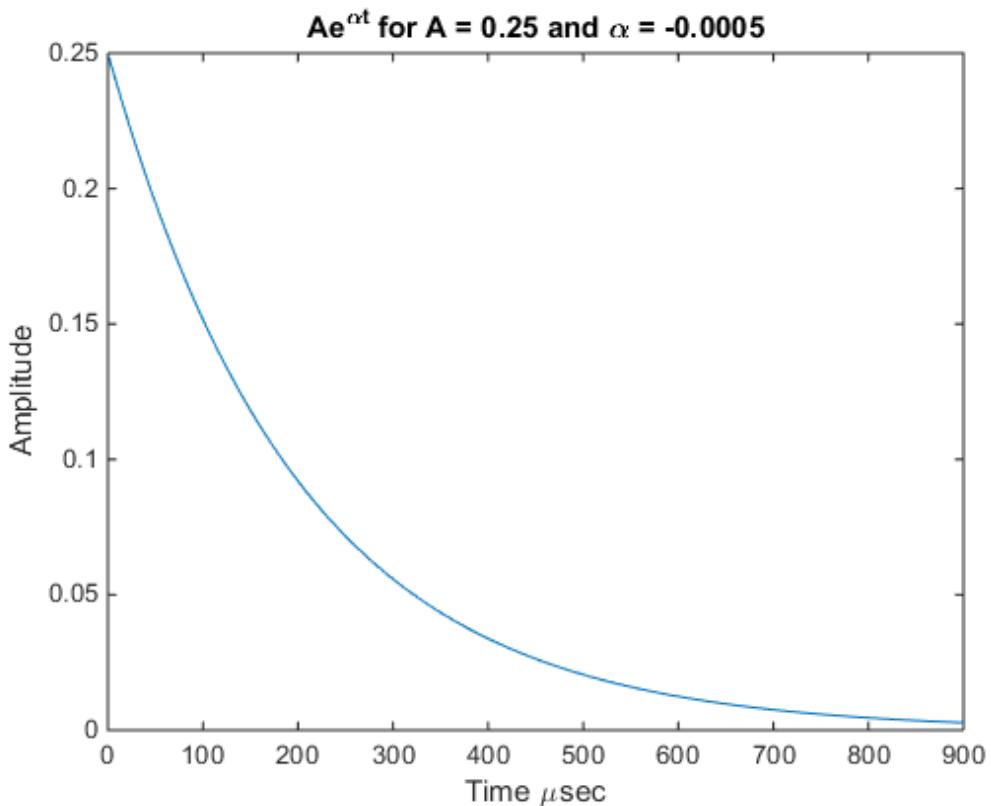


Include Superscripts and Annotations in Graph Text

Create a line plot and add a title and axis labels to the graph. Display a superscript in the title using the `^` character. The `^` character modifies the character immediately following it. Include multiple characters in the superscript by enclosing them in curly braces `{}`. Include the Greek letters `\alpha` and `\mu` in the text using the TeX markups `\alpha` and `\mu`, respectively.

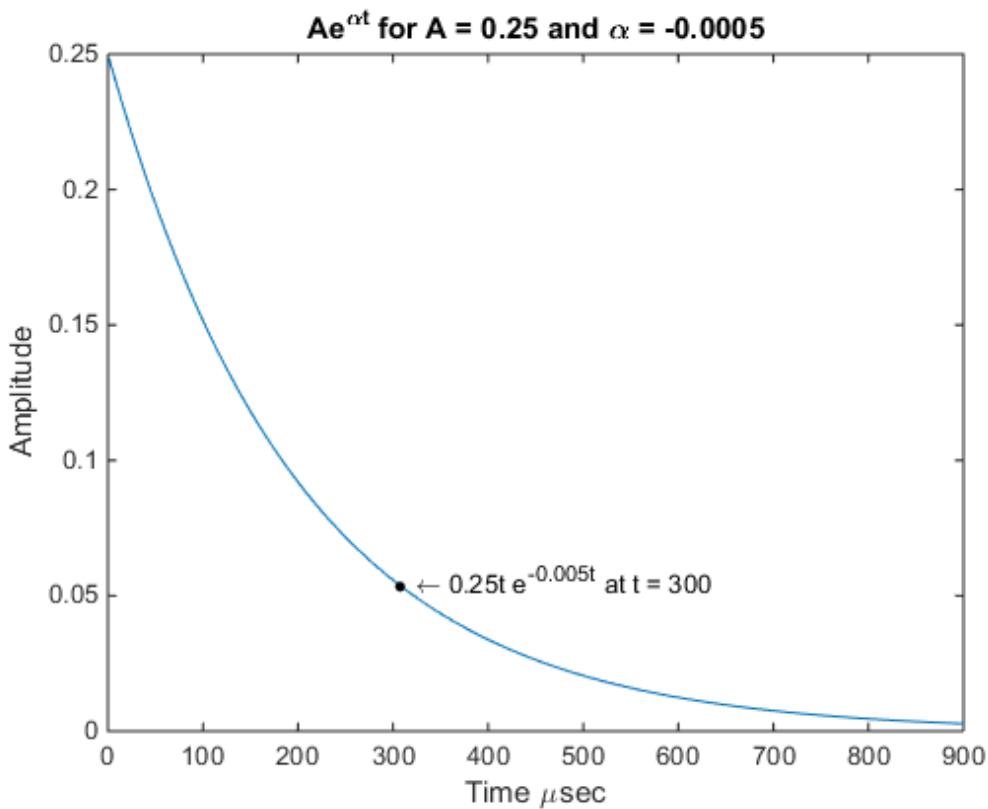
```
t = 1:900;
y = 0.25*exp(-0.005*t);
plot(t,y)
title('Ae^{\\alpha t} for A = 0.25 and \\alpha = -0.0005')
xlabel('Time \\musec')
```

```
ylabel('Amplitude')
```



Add text at the data point where $t = 300$. Use the TeX markup \bullet to add a marker to the specified point and use \leftarrow to include an arrow pointing to the left. By default, the text aligns so that the specified data point is to the left of the string.

```
str = '\bullet \leftarrow 0.25t e^{-0.005t} at t = 300';
text(t(300),y(300),str)
```



See Also

[plot](#) | [text](#) | [title](#) | [xlabel](#) | [ylabel](#)

More About

- “Add Title, Axis Labels, and Legend to Graph”
- “Include Variable Values in Graph Text” on page 4-23
- “Add Text to Specific Points on Graph” on page 4-17

Add Annotations to Graph Interactively

These examples show how to interactively add annotations to a graph and pin them to the axes.

In this section...

[“Add Annotations” on page 4-35](#)

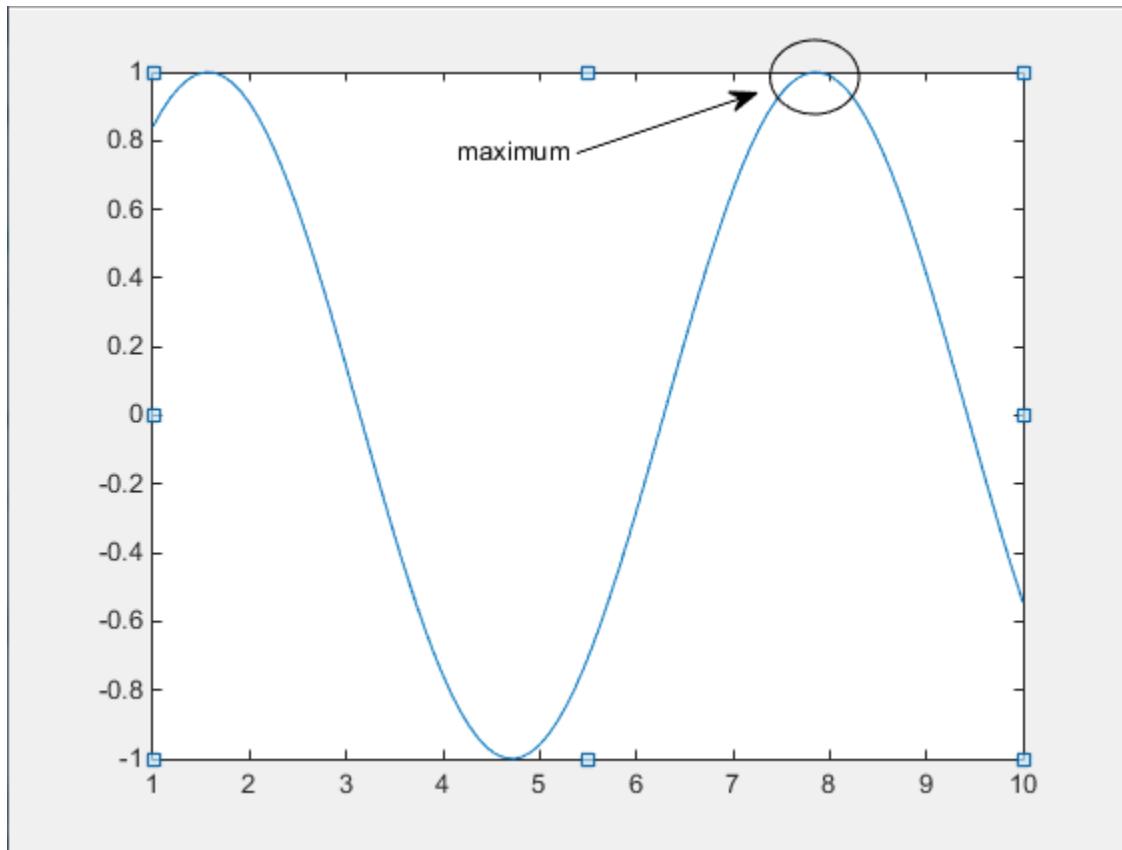
[“Pin Annotations to Points in Graph” on page 4-36](#)

Add Annotations

Create a simple line plot.

```
x = linspace(1,10);  
plot(x,sin(x))
```

Interactively add a text arrow and an ellipse to the graph using the figure **Insert** menu. Position the text arrow by drawing an arrow from tail to head and typing the text at the text cursor next to the tail. Click outside the text entry box to apply the text. Position the ellipse using the mouse to draw.

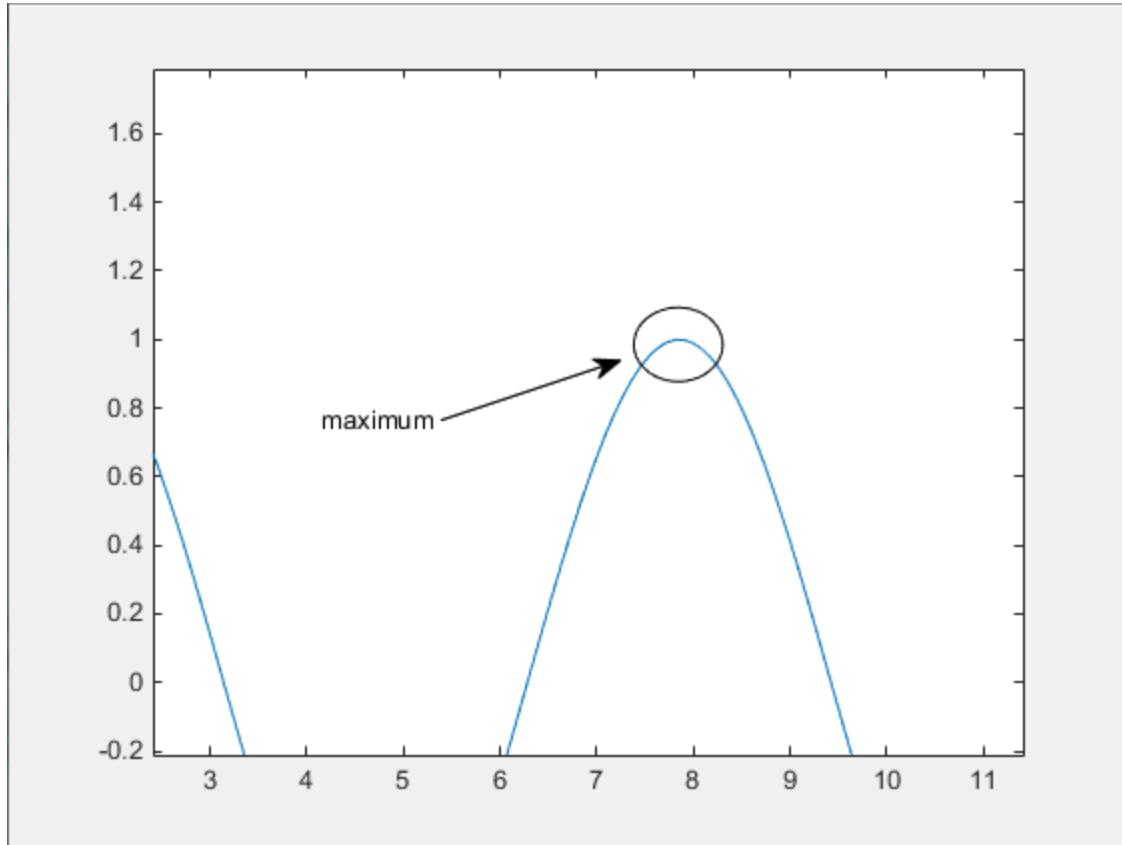


To change the location of an annotation, drag it. To modify the appearance of an annotation, right-click it and use the context menu. To view additional properties, open the Property Editor select **Show Property Editor** from the context menu.

Pin Annotations to Points in Graph

Pin the text arrow and ellipse to the axes so that they stay associated with the same coordinates in the axes, even when you pan the axes or resize the figure. Right-click it and select **Pin to Axes**. Pin both ends of the text arrow.

Click the pan icon  in the figure toolbar and pan the axes by dragging it. The text arrow and ellipse stay associated with the same points in the axes. To unpin an object, right-click it and select **Unpin**.



See Also

annotation

Related Examples

- “Add Text to Graph Interactively” on page 4-38

Add Text to Graph Interactively

In this section...

[“Add Title and Axis Labels” on page 4-38](#)

[“Add Legend” on page 4-40](#)

[“Add Annotations to Graph” on page 4-42](#)

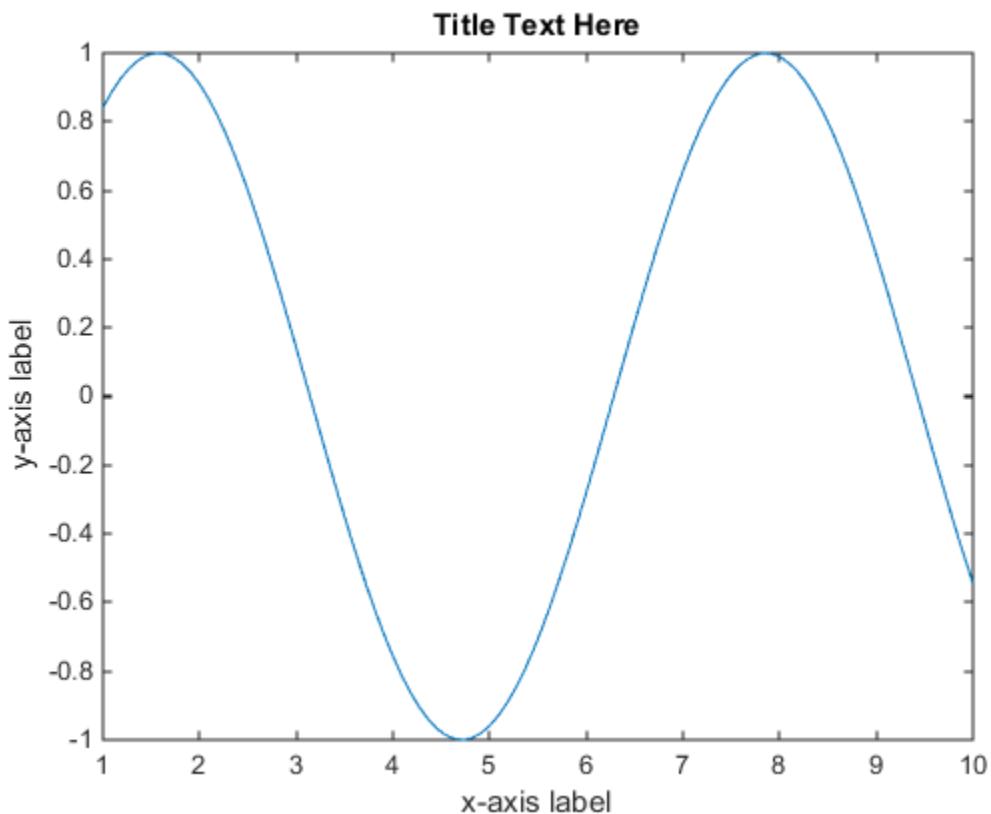
This example shows how to interactively add a title, legend, axis labels, and other text to a graph using the figure menus and plot tools.

Add Title and Axis Labels

Create a simple line plot.

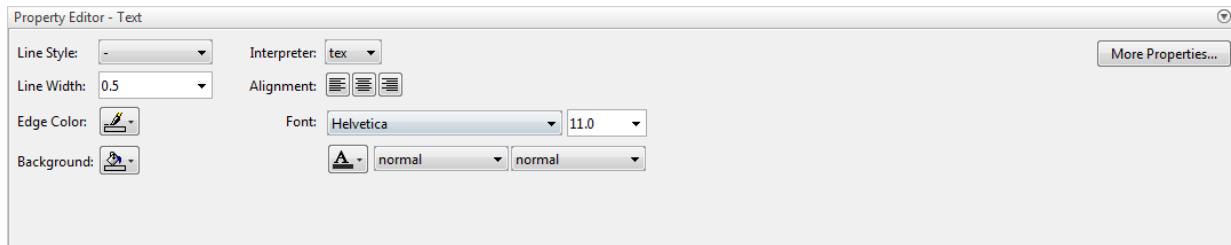
```
x = linspace(1,10);  
plot(x,sin(x))
```

Use the figure **Insert** menu to add a title and axis labels to the graph. After typing the text, click anywhere outside the text entry box to apply the text.



To modify the title and axis labels, first enable plot edit mode by clicking the **Edit Plot** button  on the figure toolbar.

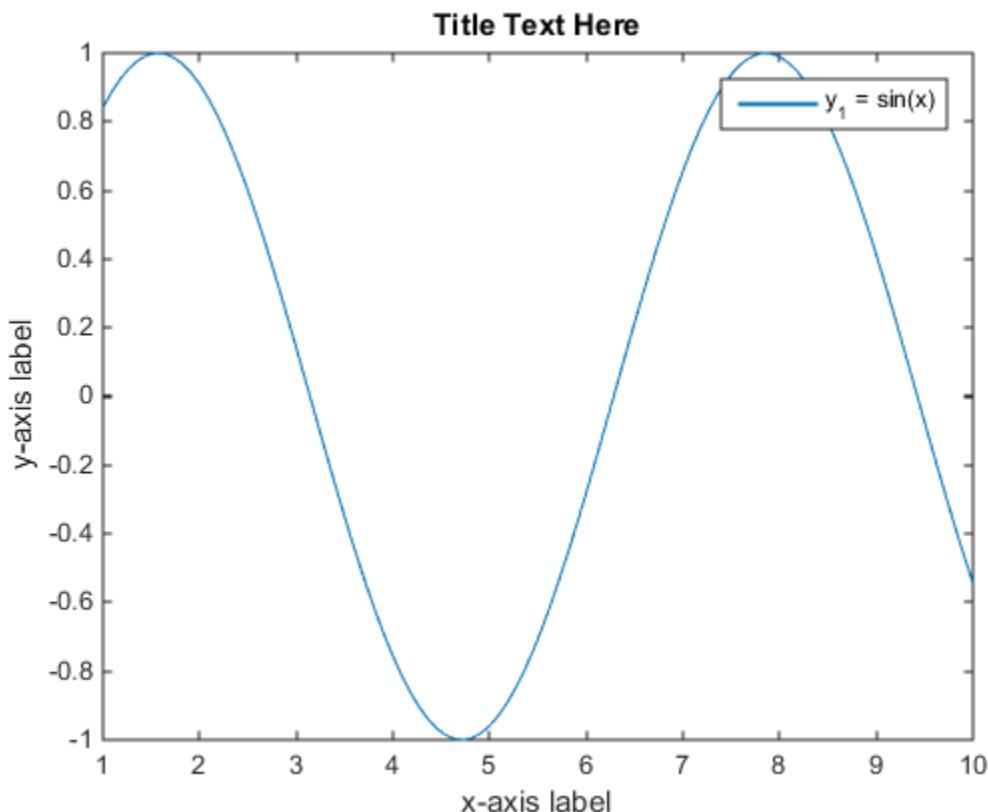
- To change the text, double-click it and type new text.
- To move the text, drag it to a new position.
- To set text properties, such as the color and font style, right-click the text and use the context menu.
- To set additional properties, use the Property Editor. Select **Show Property Editor** from the context menu.



Add Legend

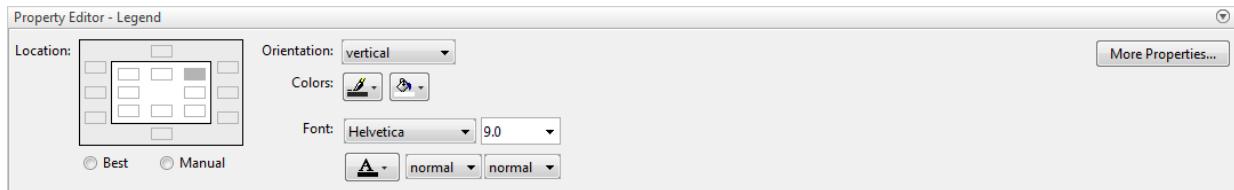
Add a legend to the graph. In the figure, select **Insert > Legend**.

By default, the legend labels each plotted object with the strings `data1`, `data2`, and so on. Change the legend text by double-clicking the text and retyping new text. Display special characters and symbols using TeX markup. For example, use the `_` character to display a subscript. For a list of supported TeX markup, see the text **Interpreter** property.



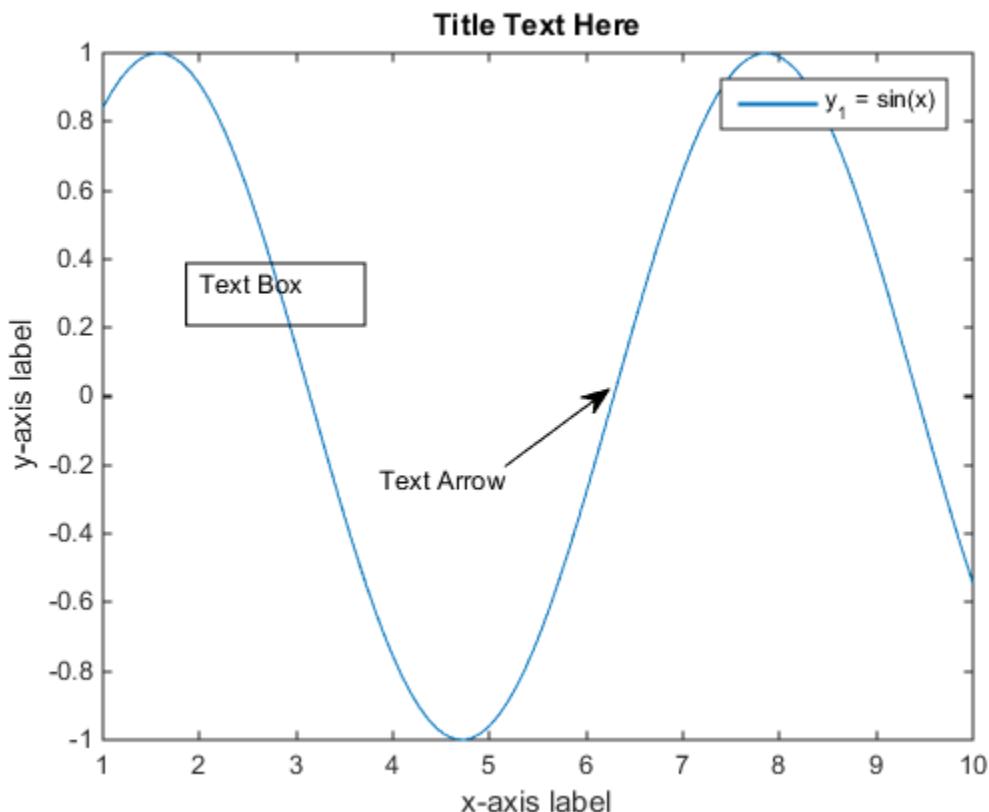
Note: To display a legend with more than 50 entries, you must use the `legend` function.

To change the legend location, right-click the legend and set the **Location** option from the context menu. For additional location options, or to modify other legend properties, use the Property Editor. Select **View > Property Editor** to open the Property Editor. Then, click the legend to access its properties.



Add Annotations to Graph

Add a text box and a text arrow to the graph using the **TextBox** and **Textarrow** options from the **Insert** menu. To add a text box, draw a rectangle and then type the text at the text cursor. To add a text arrow, draw an arrow from tail to head and type the text at the text cursor next to the tail.



See Also

[legend](#) | [title](#) | [xlabel](#) | [ylabel](#) | [zlabel](#)

Related Examples

- “Add Annotations to Graph Interactively” on page 4-35
- “Add Text to Specific Points on Graph” on page 4-17
- “Add Title, Axis Labels, and Legend to Graph”

Add Colorbar to Graph Interactively

This example shows how to interactively add a colorbar to a graph. Colorbars show the current colormap and indicate the mapping of data values to colors in the colormap.

In this section...

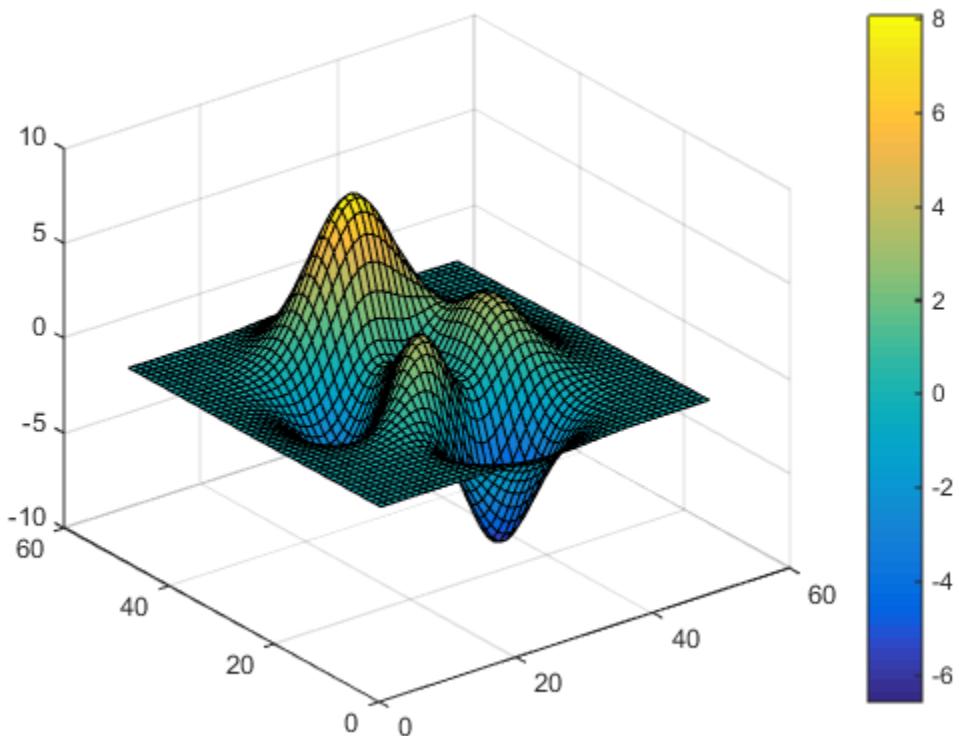
- “Add Colorbar” on page 4-44
- “Change Colorbar Location” on page 4-45
- “Change Colormap” on page 4-46

Add Colorbar

Create a simple surface plot.

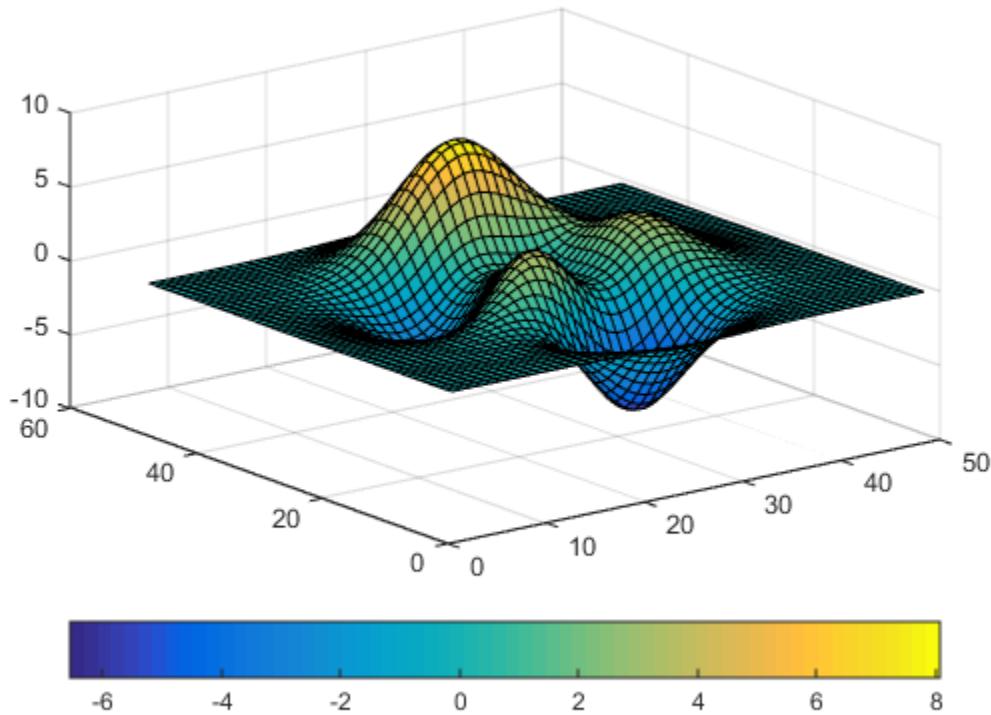
```
surf(peaks)
```

Add a colorbar to a graph. On the toolbar, click the **Colorbar** button .



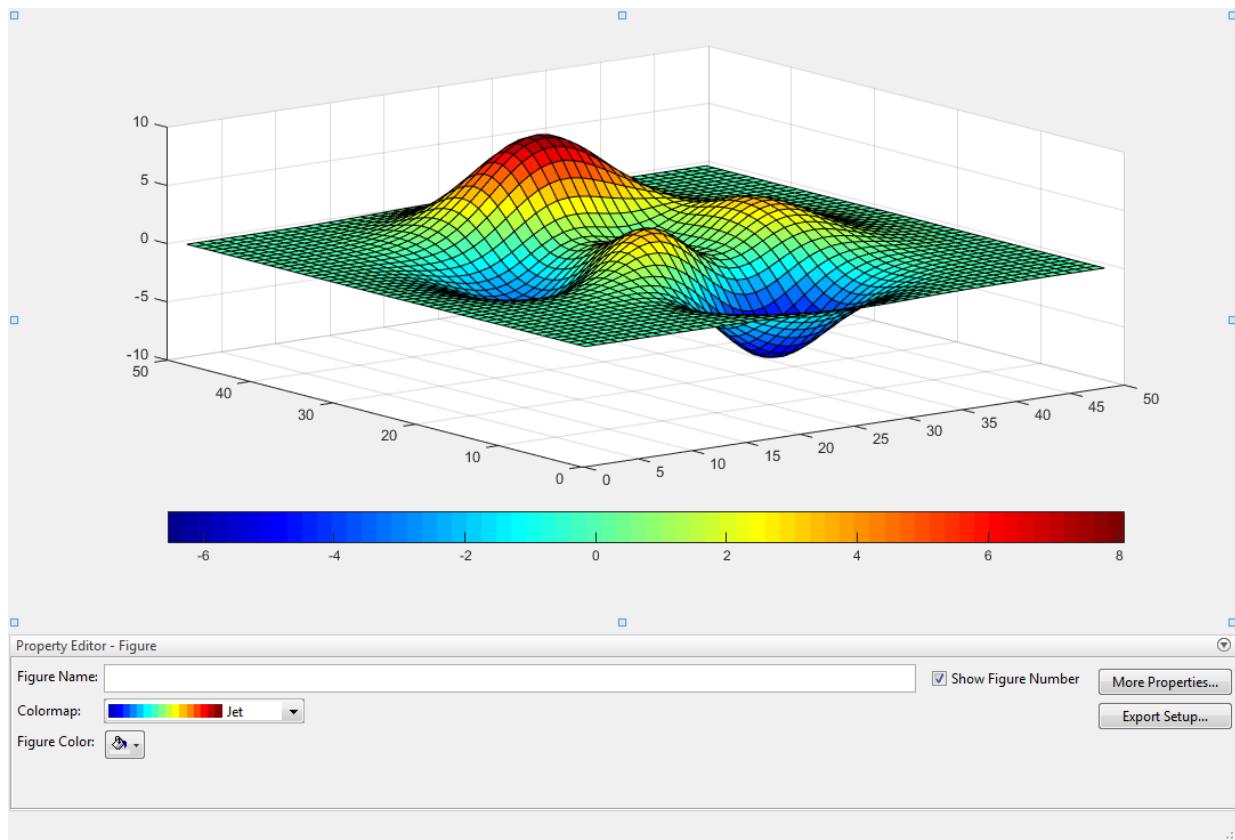
Change Colorbar Location

Change the location of the colorbar. Right-click the colorbar and select **Location >Outside South**.



Change Colormap

Change the colormap using the Property Editor. Right-click the colorbar and select **Show Property Editor**. Depending on where you click the figure, different sets of properties appear. Click the gray background to access the Colormap property. Select the Jet colormap from the list. The graph updates and displays the Jet colormap.



To shift the mapping of data values into the current colormap, right-click the colorbar and select **Interactive Colormap Shift**. Then, click a color in the colorbar and drag. As you drag, the mapping of data values into the colormap shifts.

To perform additional operations on the colormap, open the colormap editor by selecting **Open Colormap Editor** from the colorbar's context menu.

See Also

[colorbar](#) | [colormap](#) | [colormapeditor](#)

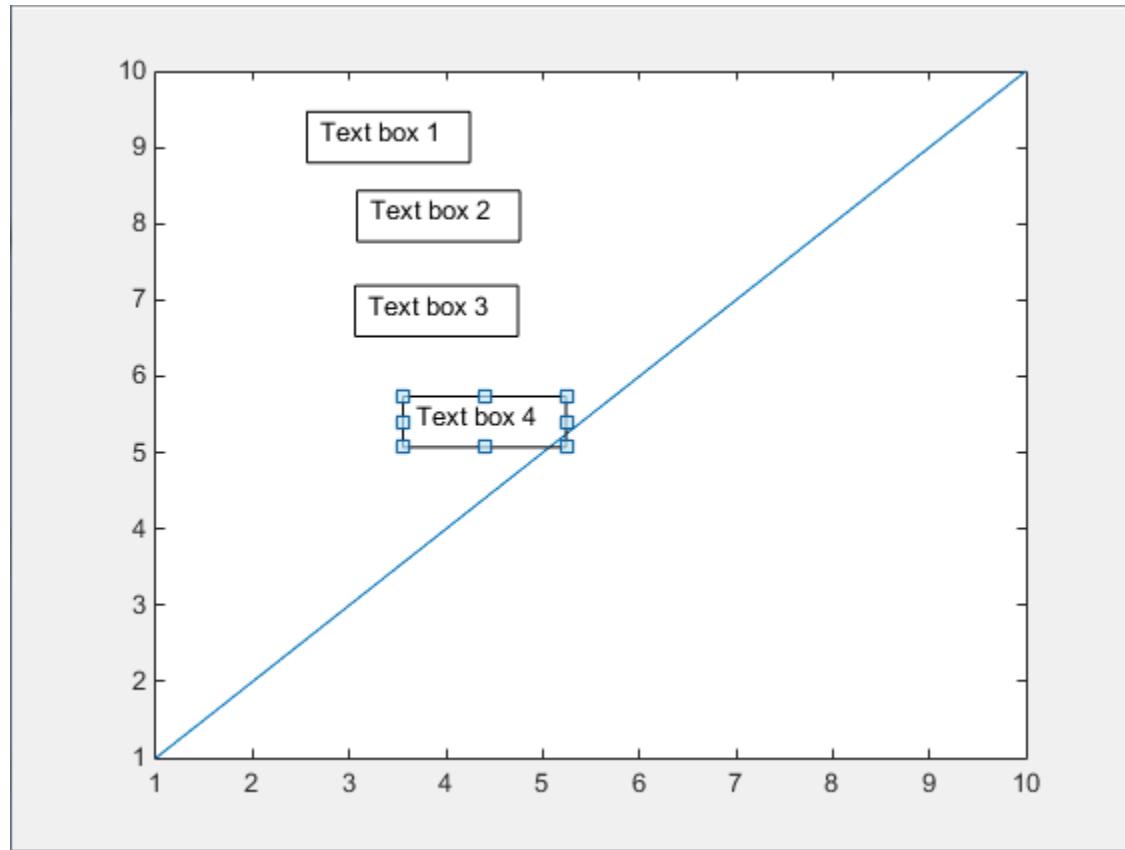
Align Objects in Graph Using Alignment Tools

This example shows how to align text boxes in a graph using alignment tools.

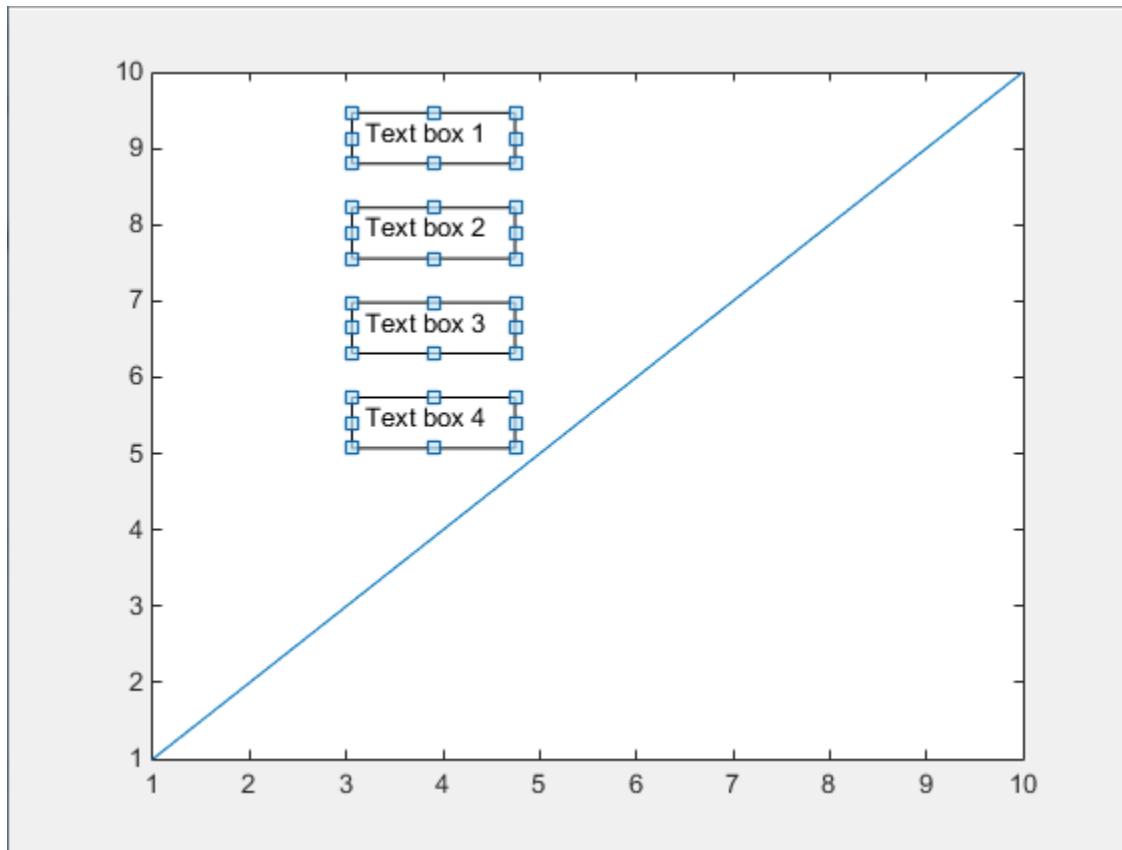
Plot a line.

```
plot(1:10)
```

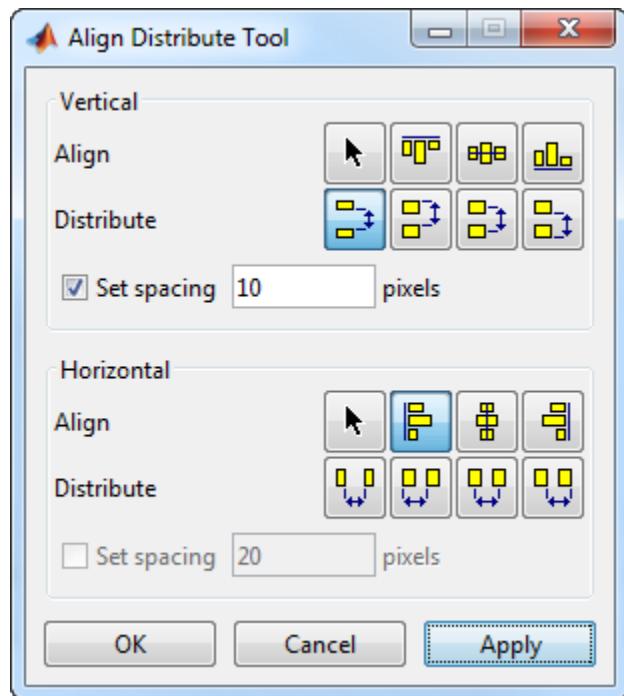
Add four text box annotations to the graph. In the figure, select **Insert > TextBox**. Approximately align the text boxes in a vertical column.



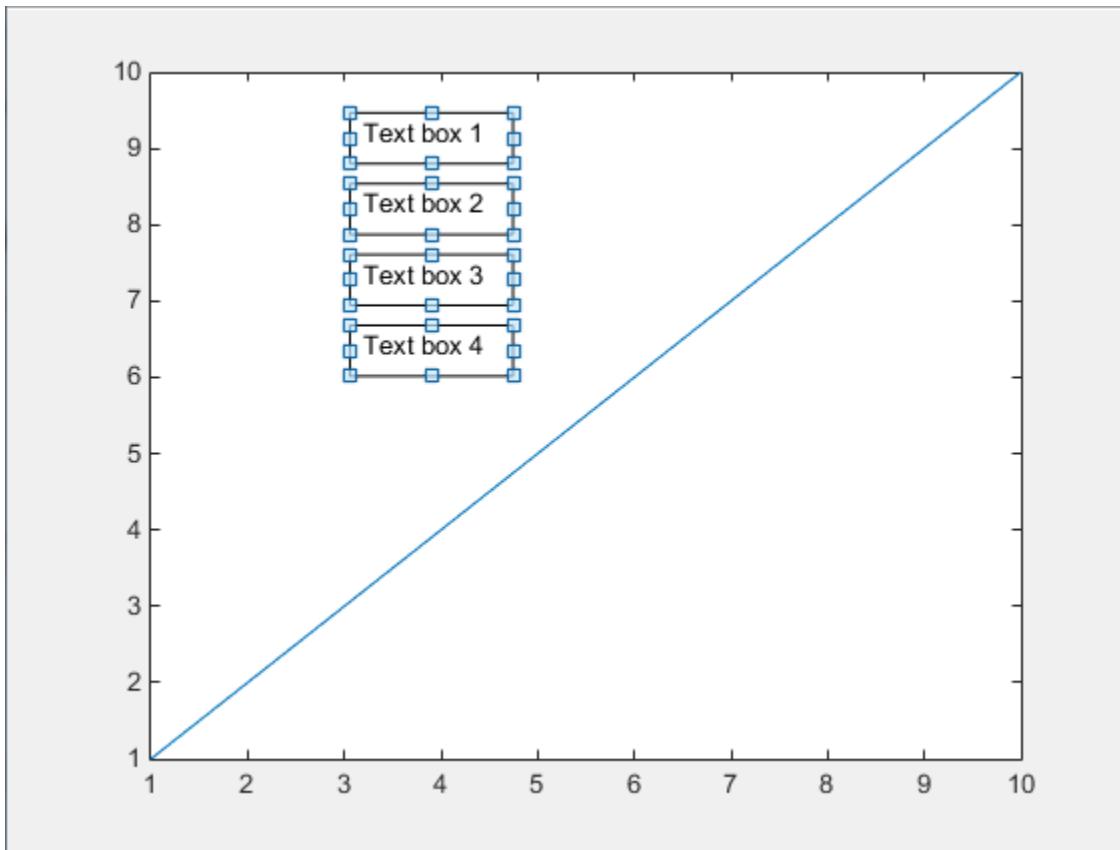
Use **Shift + click** to select all four text boxes. Align the text boxes into one column. In the figure, select **Tools > Smart Align and Distribute**.



For more control over the alignment, use the Align Distribute Tool. Select all four text boxes again and select **Tools > Align Distribute Tool**. Set the vertical distribution between the text boxes to 10 pixels and set the horizontal alignment to left-aligned, and then click OK.



The text boxes align according to your alignment settings.



See Also

annotation

More About

- “Add Text to Graph Interactively” on page 4-38
- “Add Annotations to Graph Interactively” on page 4-35

Creating Specialized Plots

- “Types of Bar Graphs” on page 5-2
- “Modify Baseline of Bar Graph” on page 5-9
- “Overlay Bar Graphs” on page 5-13
- “Overlay Line Plot on Bar Graph Using Different Y-Axes” on page 5-16
- “Color 3-D Bars by Height” on page 5-20
- “Compare Data Sets Using Overlayed Area Graphs” on page 5-23
- “Offset Pie Slice with Greatest Contribution” on page 5-28
- “Add Legend to Pie Chart” on page 5-30
- “Label Pie Chart With Text and Percent Values” on page 5-33
- “Data Cursors with Histograms” on page 5-39
- “Combine Stem Plot and Line Plot” on page 5-41
- “Overlay Stairstep Plot and Line Plot” on page 5-46
- “Display Quiver Plot Over Contour Plot” on page 5-49
- “Projectile Path Over Time” on page 5-51
- “Label Contour Plot Levels” on page 5-53
- “Change Fill Colors for Contour Plot” on page 5-55
- “Highlight Specific Contour Levels” on page 5-57
- “Contour Plot in Polar Coordinates” on page 5-60
- “Animation Techniques” on page 5-66
- “Trace Marker Along Line” on page 5-68
- “Move Group of Objects Along Line” on page 5-71
- “Animate Graphics Object” on page 5-75
- “Line Animations” on page 5-79
- “Record Animation for Playback” on page 5-82

Types of Bar Graphs

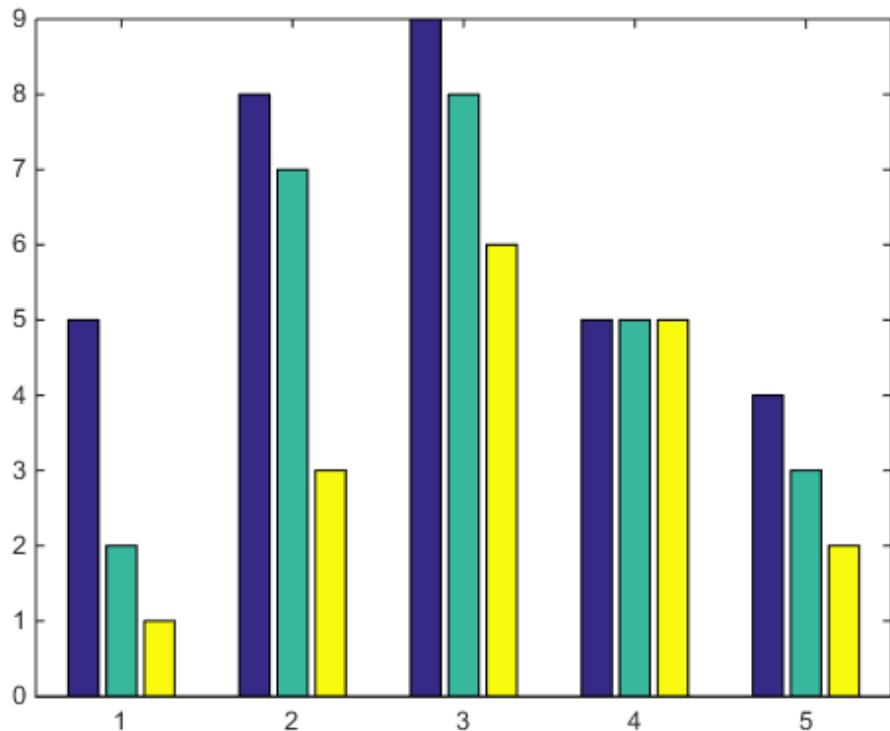
Bar graphs are useful for viewing results over a period of time, comparing results from different data sets, and showing how individual elements contribute to an aggregate amount.

By default, bar graphs represents each element in a vector or matrix as one bar, such that the bar height is proportional to the element value.

2-D Bar Graph

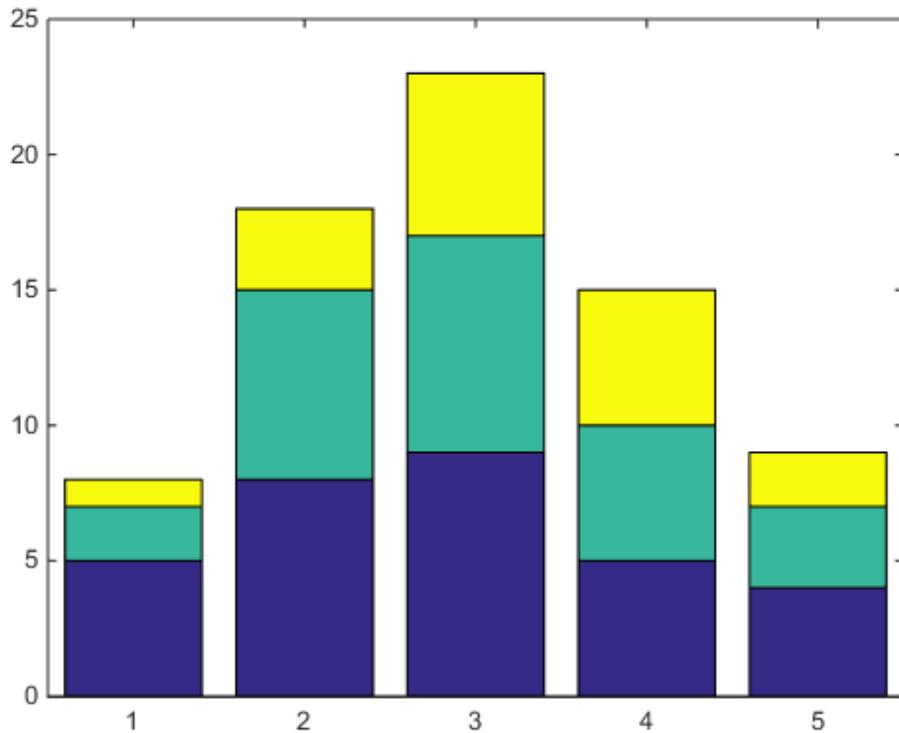
The `bar` function distributes bars along the x -axis. Elements in the same row of a matrix are grouped together. For example, if a matrix has five rows and three columns, then `bar` displays five groups of three bars along the x -axis. The first cluster of bars represents the elements in the first row of Y.

```
Y = [5,2,1  
     8,7,3  
     9,8,6  
     5,5,5  
     4,3,2];  
figure  
bar(Y)
```



To stack the elements in a row, specify the **stacked** option for the **bar** function.

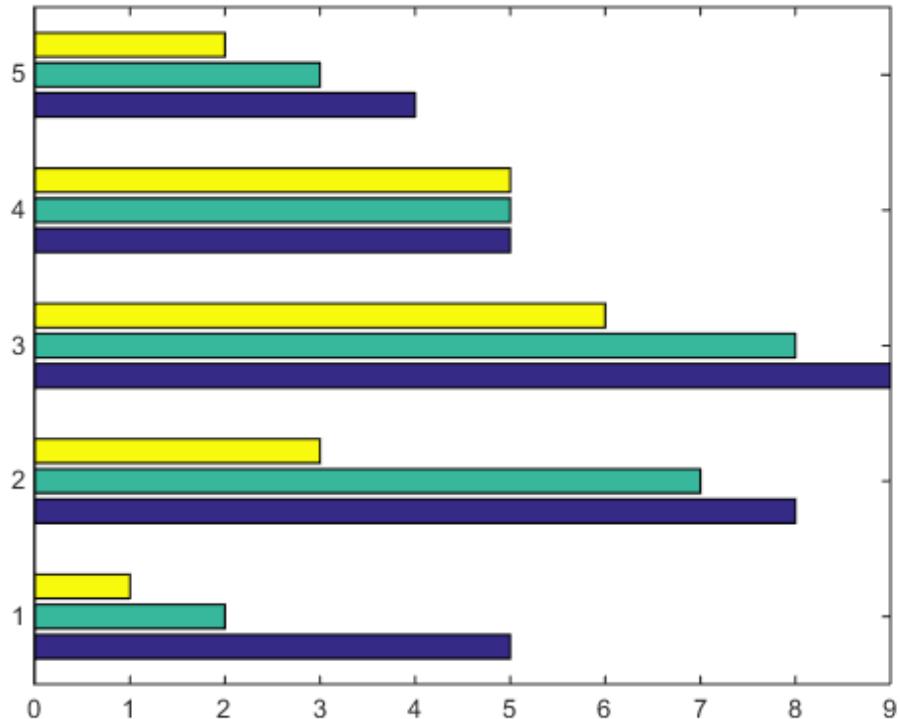
```
figure  
bar(Y, 'stacked')
```



2-D Horizontal Bar Graph

The `barh` function distributes bars along the y -axis. Elements in the same row of a matrix are grouped together.

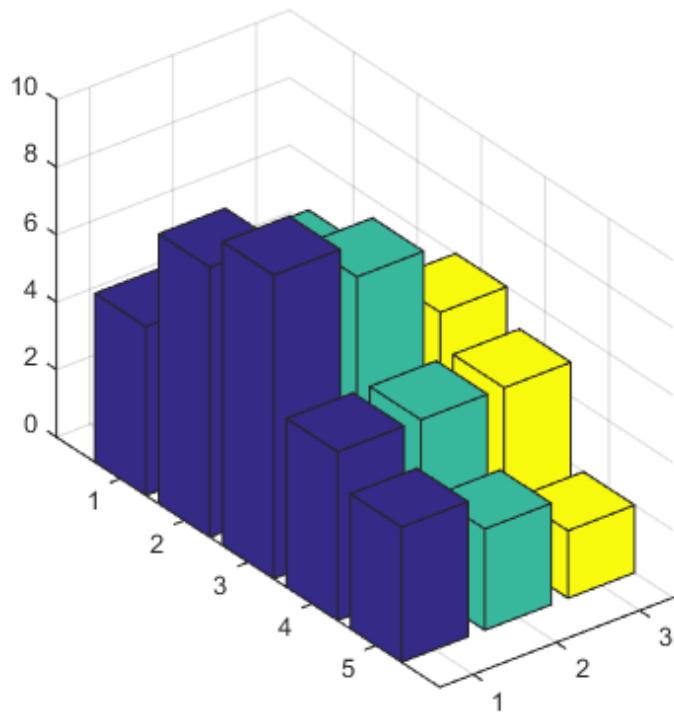
```
Y = [5,2,1  
     8,7,3  
     9,8,6  
     5,5,5  
     4,3,2];  
figure  
barh(Y)
```



3-D Bar Graph

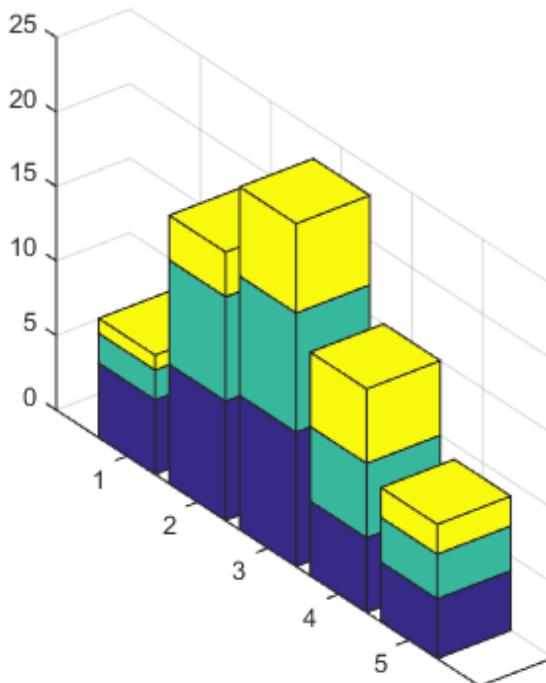
The `bar3` function draws each element as a separate 3-D block and distributes the elements of each column along the *y*-axis.

```
Y = [5,2,1  
     8,7,3  
     9,8,6  
     5,5,5  
     4,3,2];  
figure  
bar3(Y)
```



To stack the elements in a row, specify the **stacked** option for the **bar3** function.

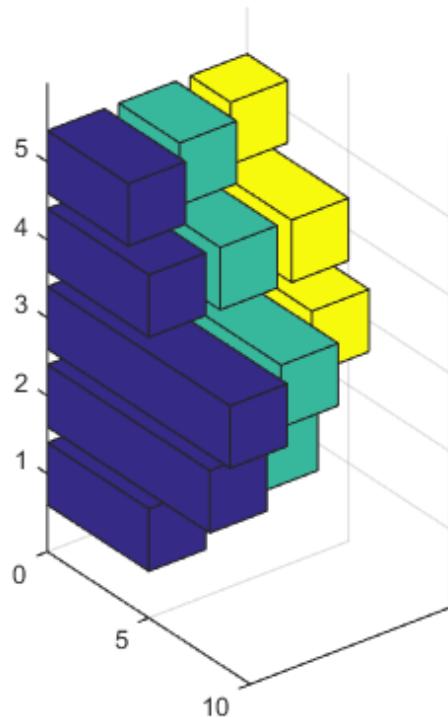
```
figure  
bar3(Y, 'stacked')
```



3-D Horizontal Bar Graph

The `bar3h` function draws each element as a separate 3-D block and distributes the elements of each column along the z -axis.

```
Y = [5,2,1  
     8,7,3  
     9,8,6  
     5,5,5  
     4,3,2];  
figure  
bar3h(Y)
```



See Also

[bar](#) | [bar3](#) | [bar3h](#) | [barh](#)

Modify Baseline of Bar Graph

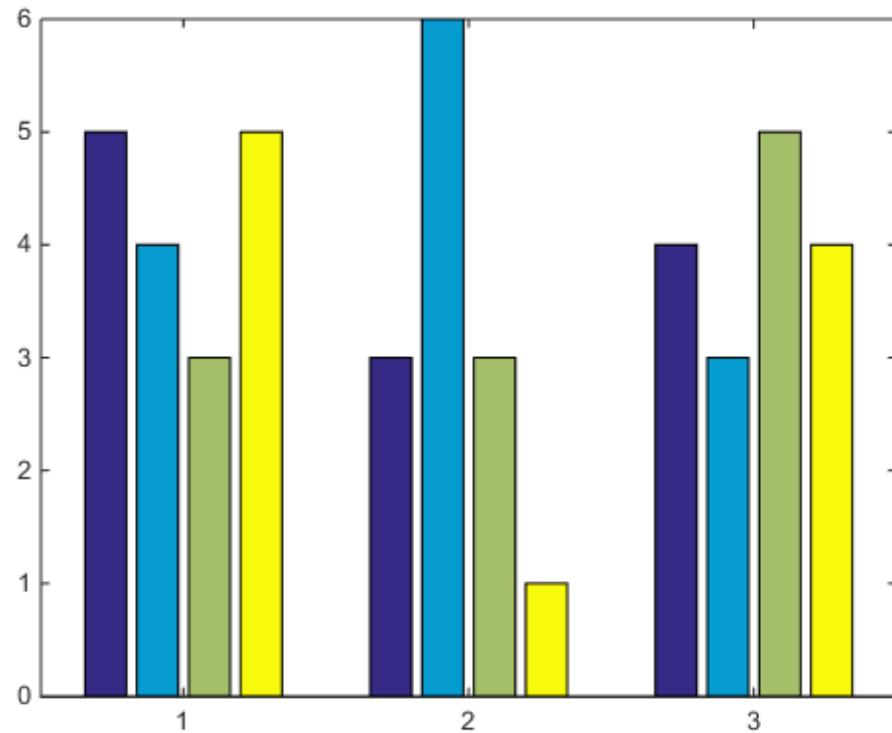
This example shows how to modify properties of the baseline of a bar graph.

The **bar** and **barm** functions create a bar series for each column in a matrix. Each bar series comprises a set of bars that have the same color. All bar series in a graph share the same baseline.

To change the value of the baseline, set the **BaseValue** property for any of the bar series. To change other properties of the baseline, such as the line style or color, you must use the baseline handle.

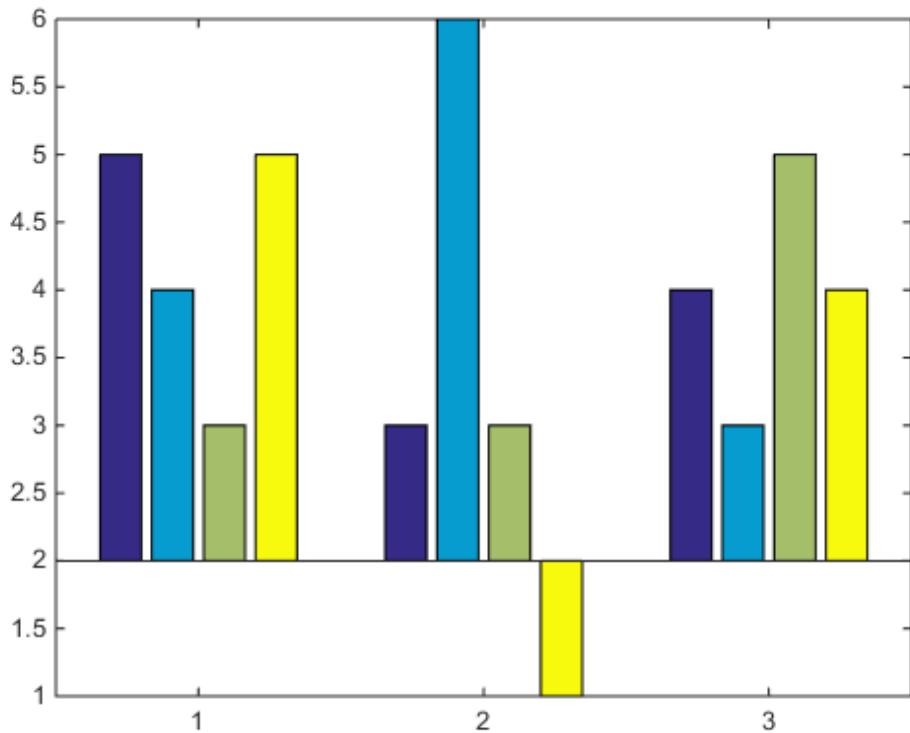
For example, create a bar graph of a four-column matrix. Return the four bar series handles as **hBars**.

```
Y = [5, 4, 3, 5;
      3, 6, 3, 1;
      4, 3, 5, 4];
figure
hBars = bar(Y);
```



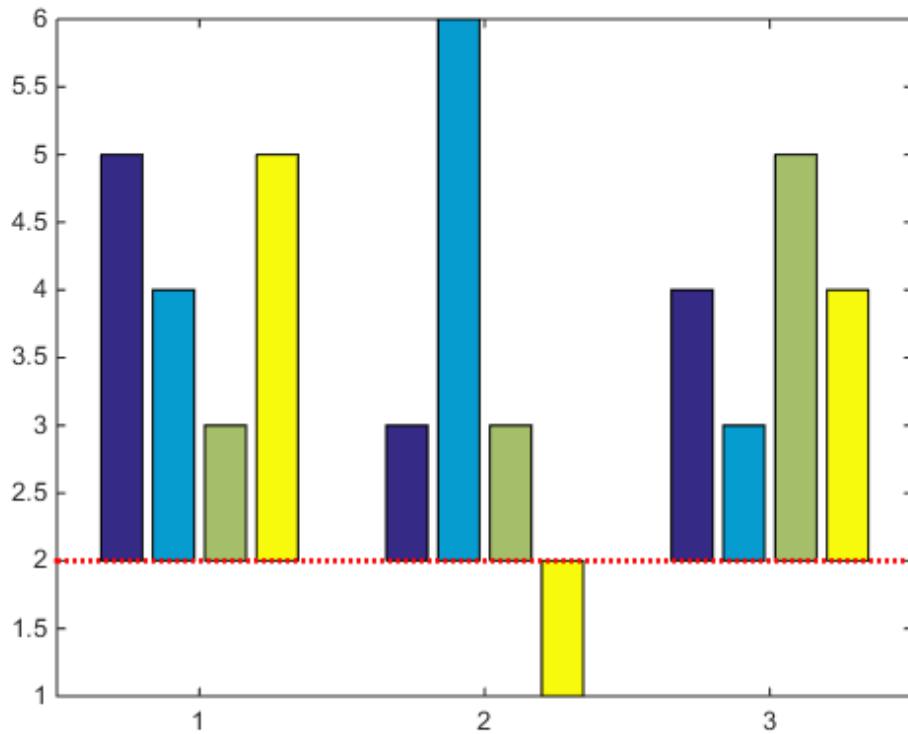
To change the value of the baseline to 2, set the **BaseValue** property for one of the bar objects to 2.

```
hBars(1).BaseValue = 2;
```



Get the handle of the baseline from the **BaseLine** property of one of the bar series. Use the handle to change the baseline to a thick, red dotted line.

```
hBaseline = hBars(1).BaseLine;  
hBaseline.LineStyle = ':';  
hBaseline.Color = 'red';  
hBaseline.LineWidth = 2;
```



See Also

[bar](#) | [barh](#)

Overlay Bar Graphs

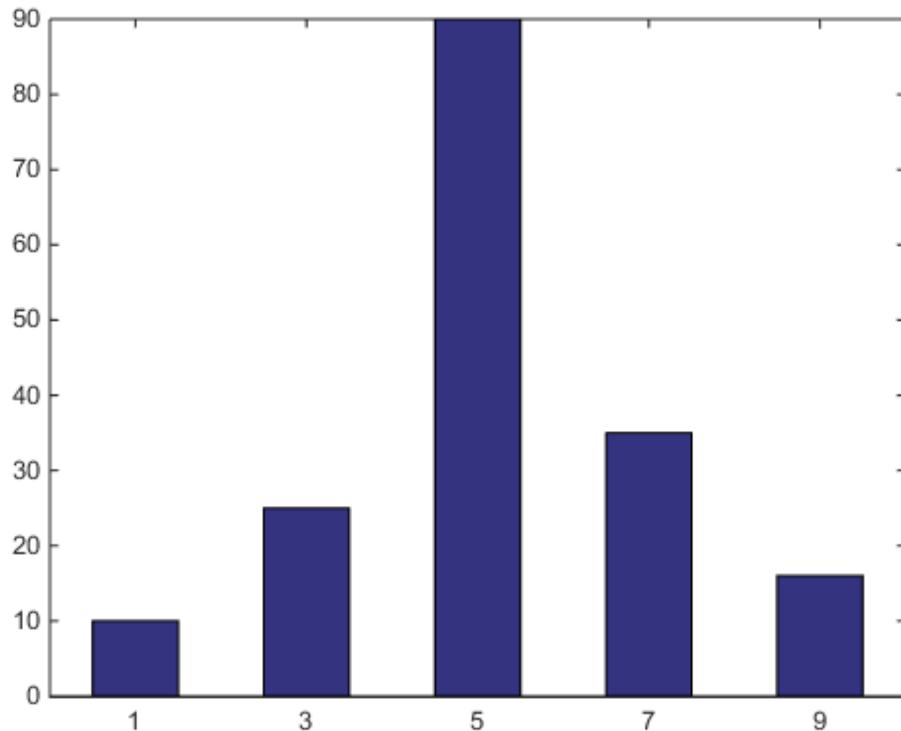
This example shows how to overlay two bar graphs.

Define `series1` and `series2`.

```
x = [1,3,5,7,9]; % place bars at these points along x-axis
series1 = [10,25,90,35,16];
series2 = [7,38,31,50,41];
```

Create a bar graph of the data in `series1`. Set the bar width to 0.5. Set the bar color to dark blue by setting the `FaceColor` property to an RGB color value.

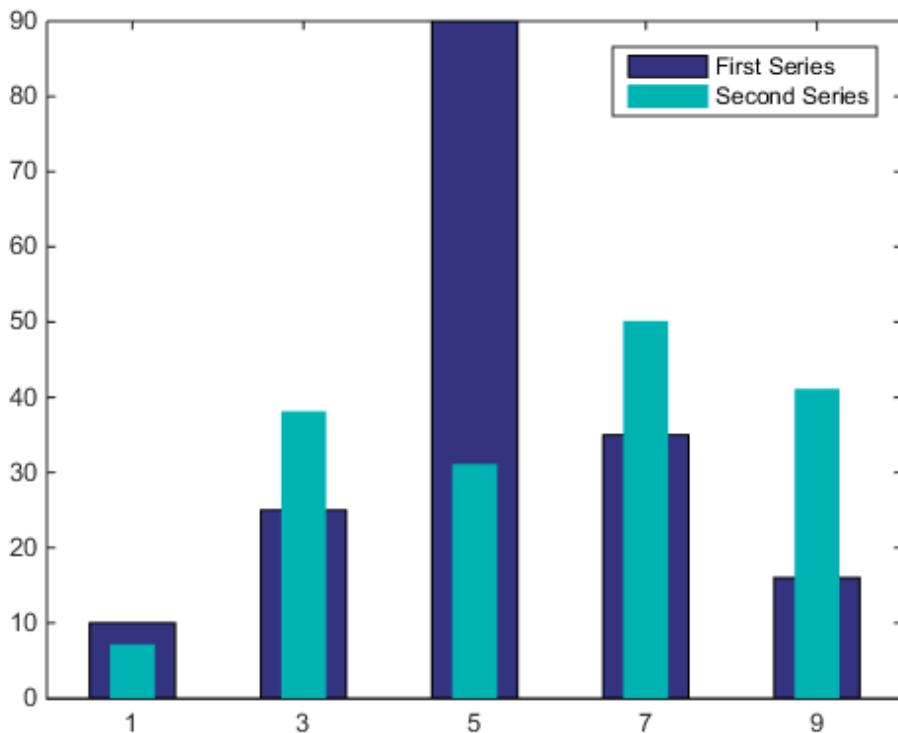
```
figure
width1 = 0.5;
bar(x,series1,width1,'FaceColor',[0.2,0.2,0.5])
```



Use the `hold` function to retain the first graph. Plot the second bar graph over the first bar graph using a smaller bar width. Specify a different RGB color value for the `FaceColor` and `EdgeColor` properties of the second bar graph.

```
hold on
width2 = width1/2;
bar(x,series2,width2,'FaceColor',[0,0.7,0.7],...
      'EdgeColor',[0,0.7,0.7])
hold off

legend('First Series','Second Series') % add legend
```



The figure contains two bar graphs. MATLAB® plots the dark blue bars behind the light blue bars since the dark blue bars are plotted first. The order of the plotting commands determines the stacking order of the bars.

See Also

[bar](#) | [barh](#) | [hold](#)

Overlay Line Plot on Bar Graph Using Different Y-Axes

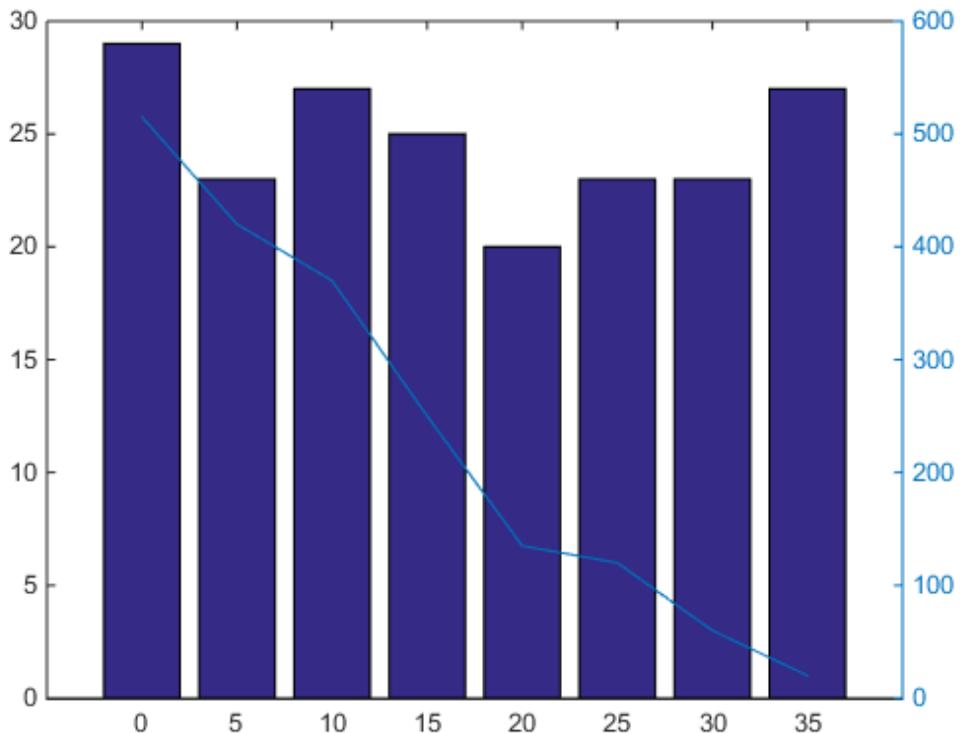
This example shows how to combine a line plot and a bar graph using two different *y*-axes.

Define the concentration and temperature data collected every 5 days for a 35 day period.

```
days = 0:5:35;
conc = [515,420,370,250,135,120,60,20];
temp = [29,23,27,25,20,23,23,27];
```

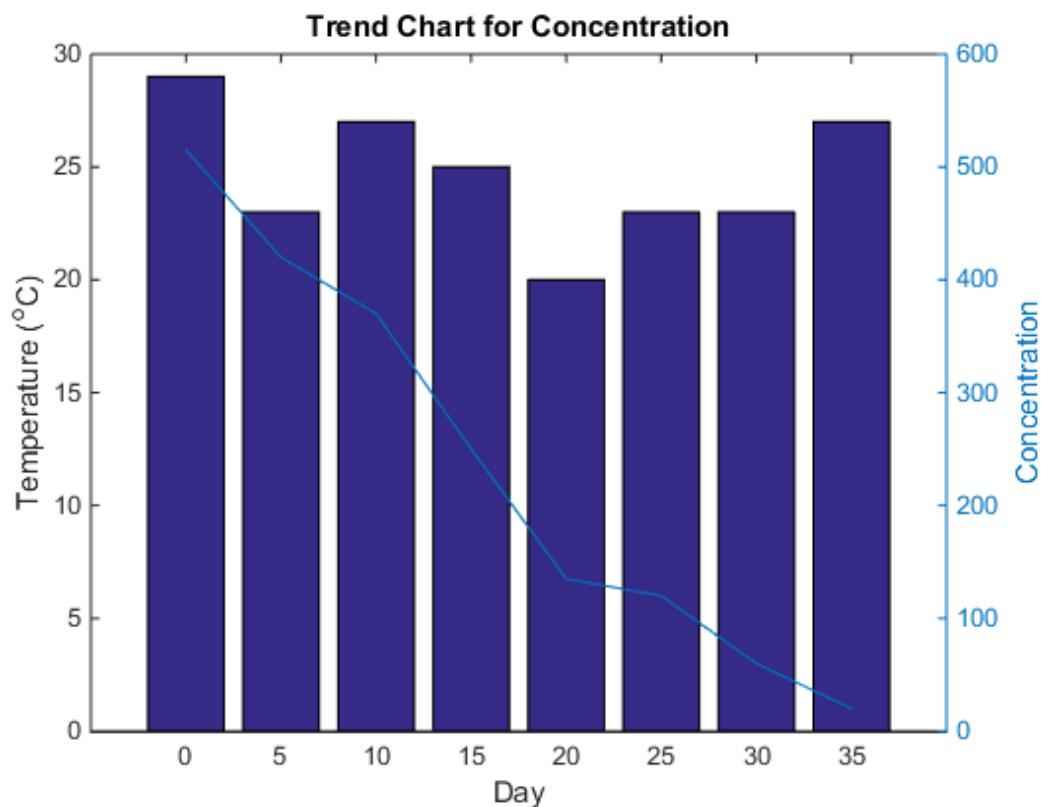
Use `plotyy` to display a bar graph of the temperature data and a line graph of the concentration data. Return the two axes handles as `ax`, the bar graph handle as `hBar`, and the line plot handle as `hLine`.

```
figure
[ax,hBar,hLine] = plotyy(days,temp,days,conc,'bar','plot');
```



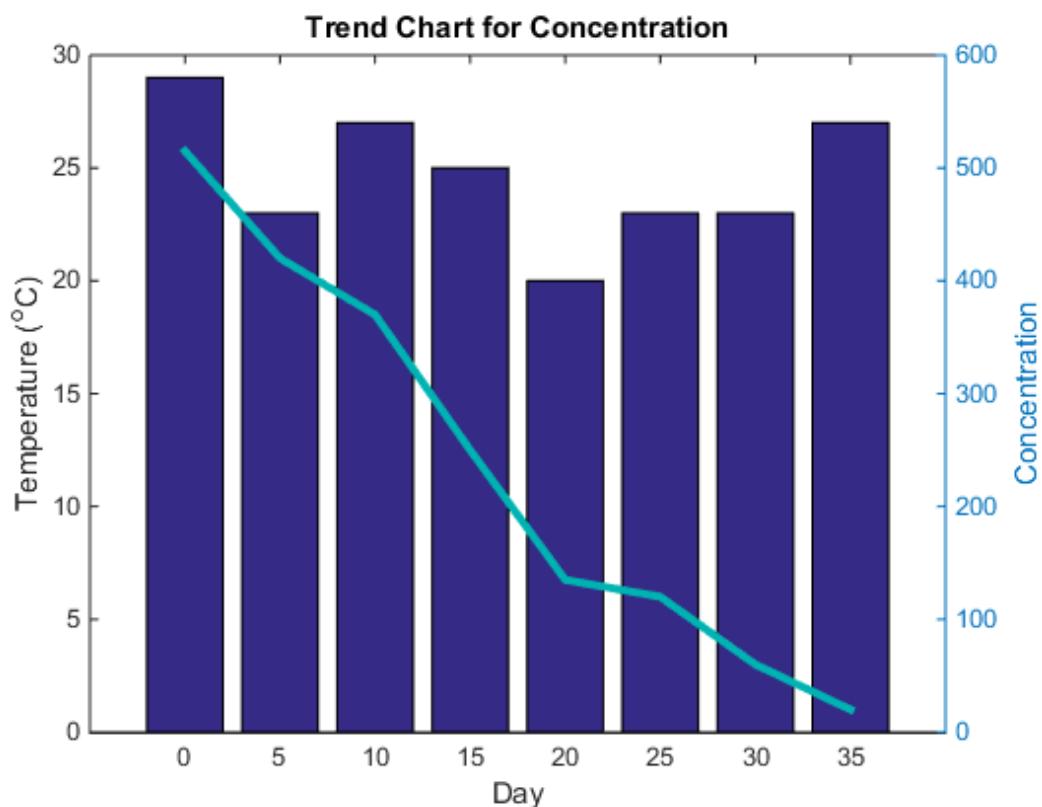
Add a title and axis labels to the graph. Use the axes handles to label the left and right *y*-axis appropriately.

```
title('Trend Chart for Concentration')
xlabel('Day')
ylabel(ax(1), 'Temperature (^{o}C)')
ylabel(ax(2), 'Concentration')
```



Change the line width and color. To change properties of the line, use its handle.

```
hLine.LineWidth = 3;  
hLine.Color = [0,0.7,0.7];
```



The graph uses two different *y*-axes. The left *y*-axis corresponds to the bar graph. The right *y*-axis corresponds to the line plot.

See Also

[bar](#) | [gca](#) | [plot](#) | [text](#)

Color 3-D Bars by Height

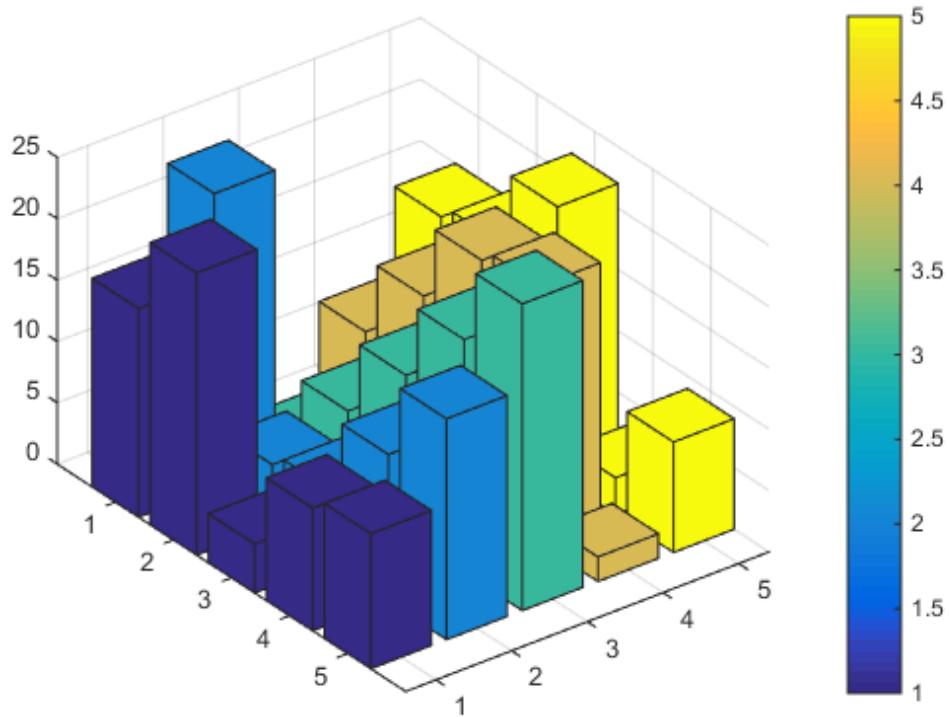
This example shows how to modify a 3-D bar plot by coloring each bar according to its height.

Generate data for this example using the `magic` function.

```
Z = magic(5);
```

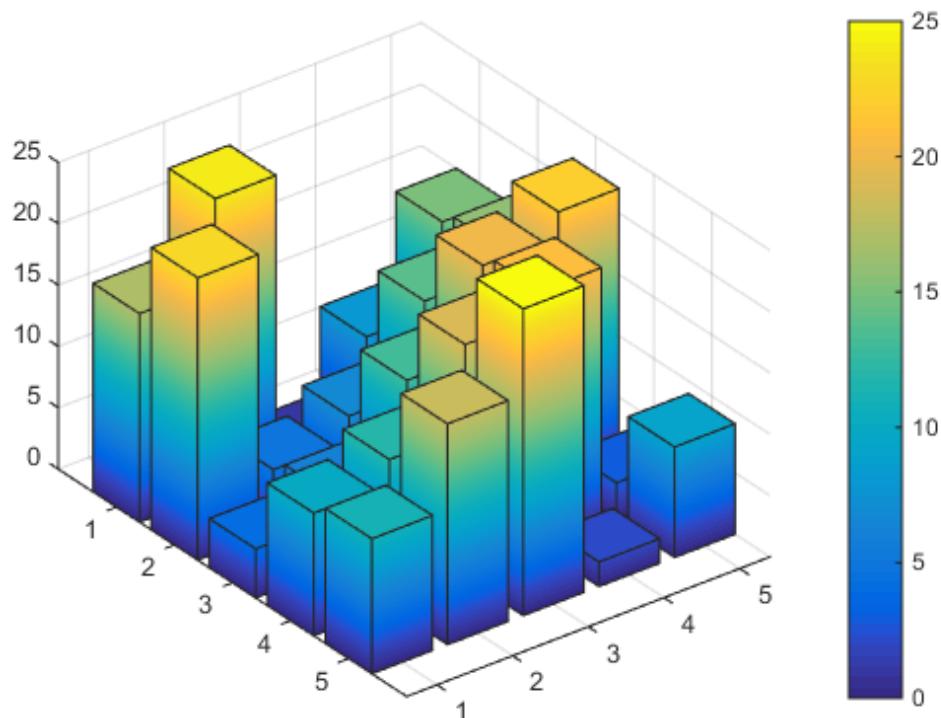
Create a 3-D bar graph of Z. Store the handles to the surface objects used to create the bar graph in array h. Add a colorbar to the graph.

```
figure  
h = bar3(Z);  
colorbar
```



For each surface object, get the array of z-coordinates from the **ZData** property. Use the array to set the **CData** property, which defines the vertex colors. Interpolate the face colors by setting the **FaceColor** properties of the surface objects to **interp**.

```
for k = 1:length(h)
    zdata = h(k).ZData;
    h(k).CData = zdata;
    h(k).FaceColor = 'interp';
end
```



The height of each bar determines its color. You can estimate the bar heights by comparing the bar colors to the colorbar.

See Also

`bar3 | colorbar`

Compare Data Sets Using Overlayed Area Graphs

This example shows how to compare two data sets by overlaying their area graphs.

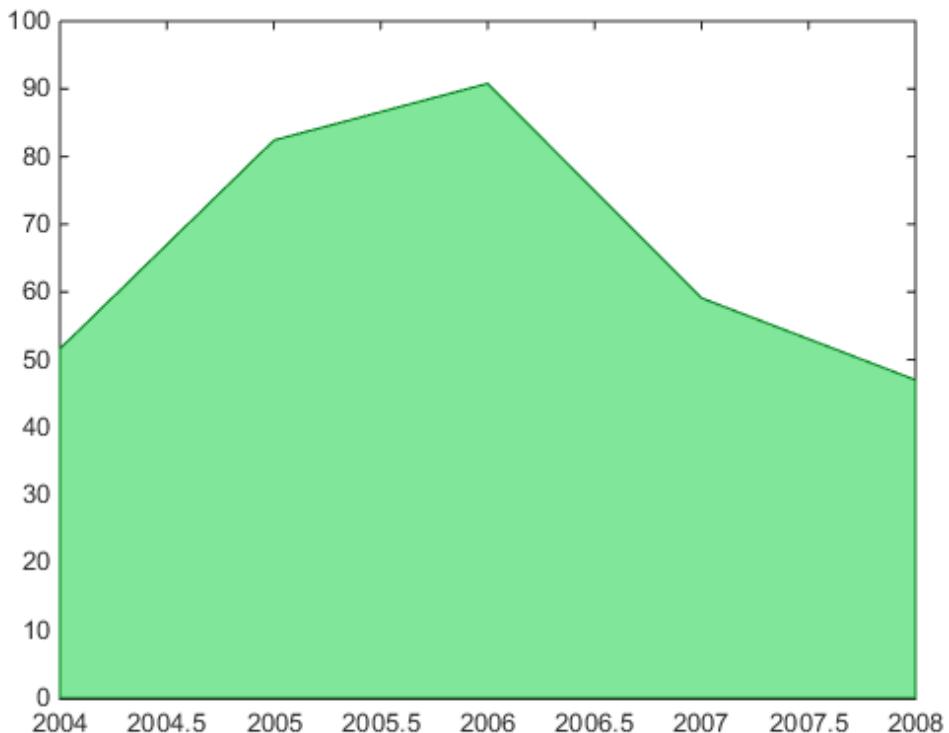
Overlay Two Area Graphs

Define the sales and expenses data from the years 2004 to 2008.

```
years = 2004:2008;
sales = [51.6, 82.4, 90.8, 59.1, 47.0];
expenses = [19.3, 34.2, 61.4, 50.5, 29.4];
```

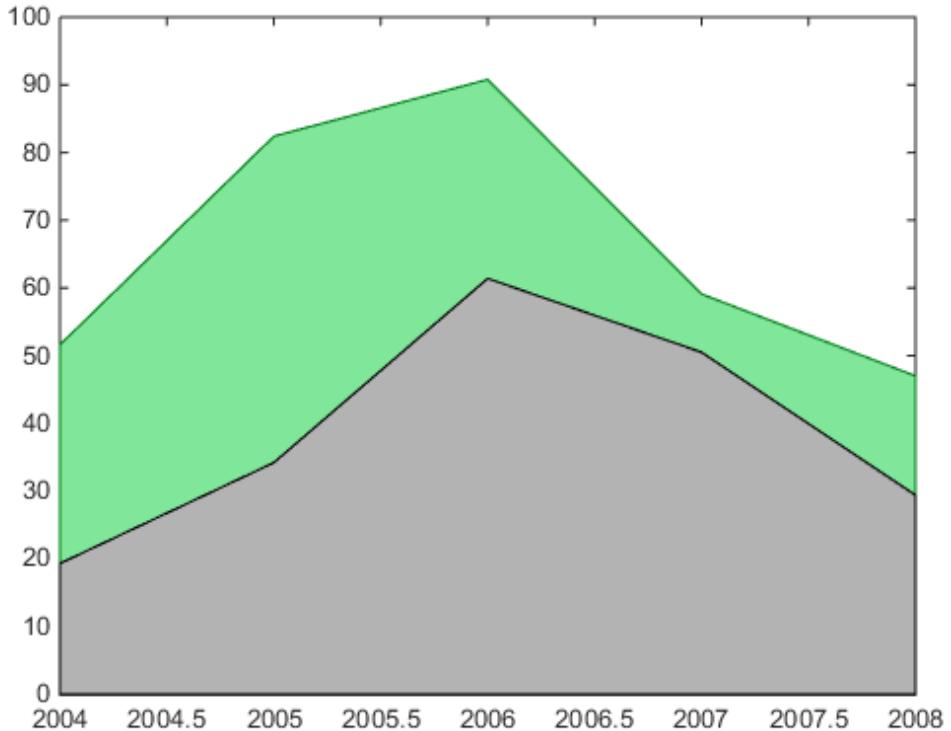
Use the `area` function to display sales and expenses as two separate area graphs in the same axes. First, create an area graph of `sales`. Change the color of the area graph by setting the `FaceColor` and `EdgeColor` properties using RGB color values.

```
figure
area(years,sales,'FaceColor',[0.5,0.9,0.6],...
      'EdgeColor',[0,0.5,0.1])
```



Use the `hold` command to prevent a new graph from replacing the area graph. Create a second area graph of `expenses` and change the color of this graph by specifying the `FaceColor` and `EdgeColor` properties. Set the `hold` state to `off`.

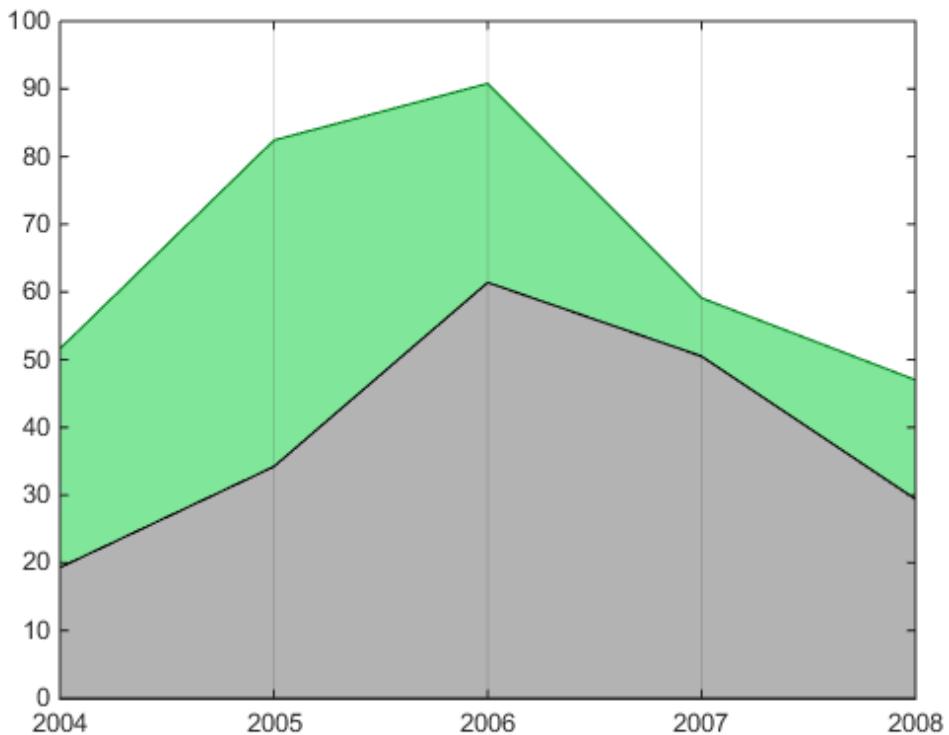
```
hold on
area(years,expenses,'FaceColor',[0.7,0.7,0.7],...
      'EdgeColor','k')
hold off
```



Add Grid Lines

Set the tick marks along the *x*-axis to correspond to whole years. Draw a grid line for each tick mark by setting the `XGrid` property to `on`. Display the grid lines on top of the area graphs by setting the `Layer` property to `top`.

```
hAx = gca; % handle to current axes  
hAx.XTick = years;  
hAx.XGrid = 'on';  
hAx.Layer = 'top';
```

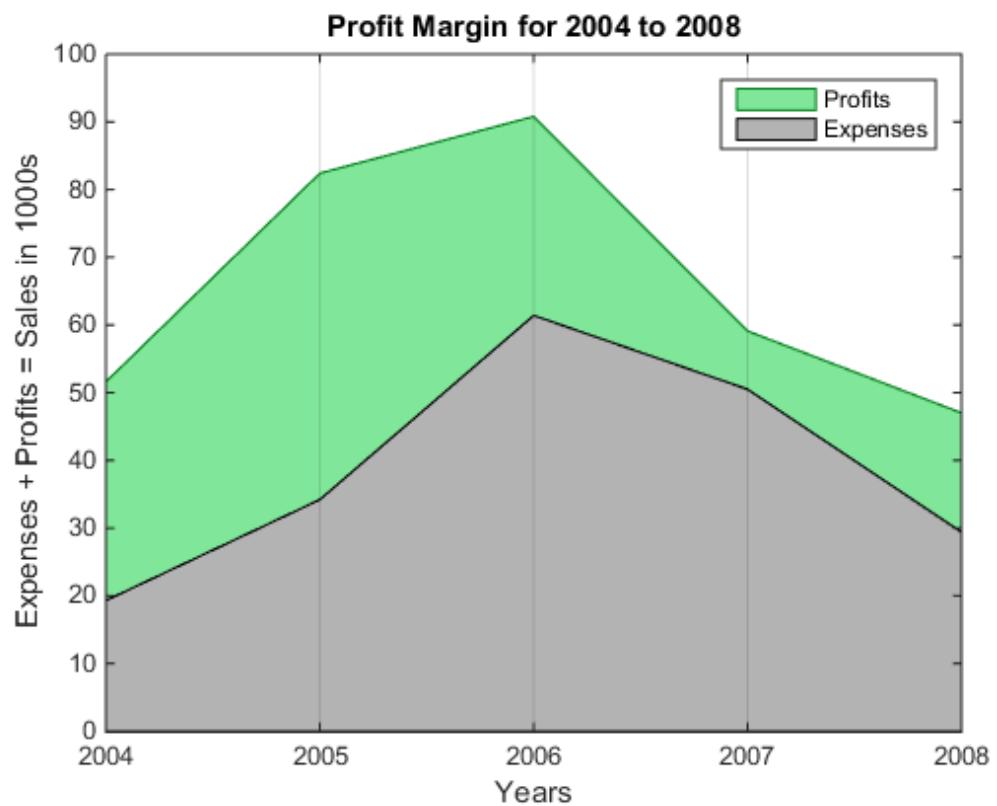


Add Title, Axis Labels, and Legend

Give the figure the title, "Profit Margin for 2004 to 2008". Add axis labels for sales and years. Add a legend to the graph to indicate the areas of profits and expenses.

```
title('Profit Margin for 2004 to 2008')
xlabel('Years')
ylabel('Expenses + Profits = Sales in 1000s')

legend('Profits','Expenses')
```



See Also

[area](#) | [gca](#) | [hold](#) | [legend](#)

Offset Pie Slice with Greatest Contribution

This example shows how to create a pie graph and automatically offset the pie slice with the greatest contribution.

Set up a three-column array, `X`, so that each column contains yearly sales data for a specific product over a 5-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];
```

Calculate the total sales for each product over the 5-year period by taking the sum of each column. Store the results in `product_totals`.

```
product_totals = sum(X);
```

Use the `max` function to find the largest element in `product_totals` and return the index of this element, `ind`.

```
[c,ind] = max(product_totals);
```

Use the `pie` function input argument, `explode`, to offset a pie slice. The `explode` argument is a vector of zero and nonzero values where the nonzero values indicate the slices to offset. Initialize `explode` as a three-element vector of zeros.

```
explode = zeros(1,3);
```

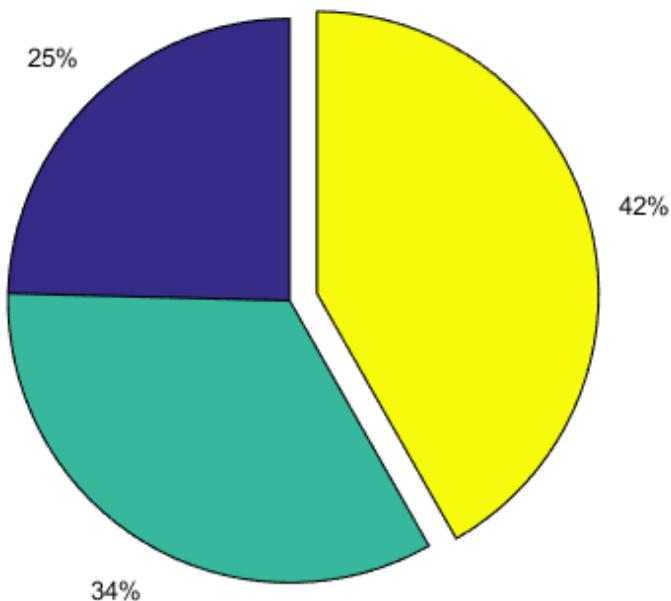
Use the index of the maximum element in `product_totals` to set the corresponding `explode` element to 1.

```
explode(ind) = 1;
```

Create a pie chart of the sales totals for each product and offset the pie slice for the product with the largest total sales.

```
figure  
pie(product_totals,explode)  
title('Sales Contributions of Three Products')
```

Sales Contributions of Three Products



See Also

[max](#) | [pie](#) | [zeros](#)

Related Examples

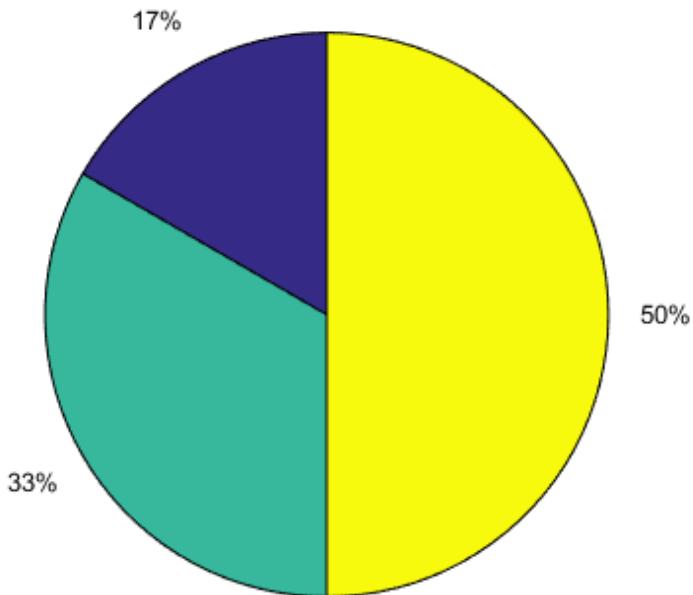
- “Add Legend to Pie Chart”

Add Legend to Pie Chart

This example shows how to add a legend to a pie chart that displays a description for each slice.

Define *x* and create a pie chart.

```
x = [1,2,3];
figure
pie(x)
```

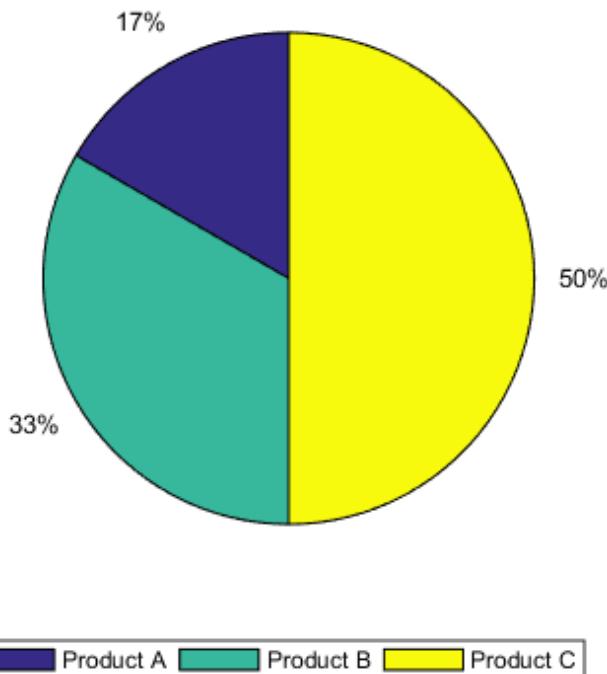


Specify the description for each pie slice in the cell array *labels*.

```
labels = {'Product A','Product B','Product C'};
```

Display a horizontal legend below the pie chart. Pass the descriptions contained in `labels` to the `legend` function. Set the legend's `Location` property to '`southoutside`' and its `Orientation` property to '`horizontal`'.

```
legend(labels,'Location','southoutside','Orientation','horizontal')
```



The graph contains a pie chart and a horizontal legend with descriptions for each pie slice.

See Also

[legend](#) | [pie](#)

Related Examples

- “Offset Pie Slice with Greatest Contribution”

Label Pie Chart With Text and Percent Values

This example shows how to label slices on a pie chart so that the labels contain custom text and the precalculated percent values for each slice.

In this section...

[“Create Pie Chart” on page 5-33](#)

[“Store Precalculated Percent Values” on page 5-34](#)

[“Combine Percent Values and Additional Text” on page 5-35](#)

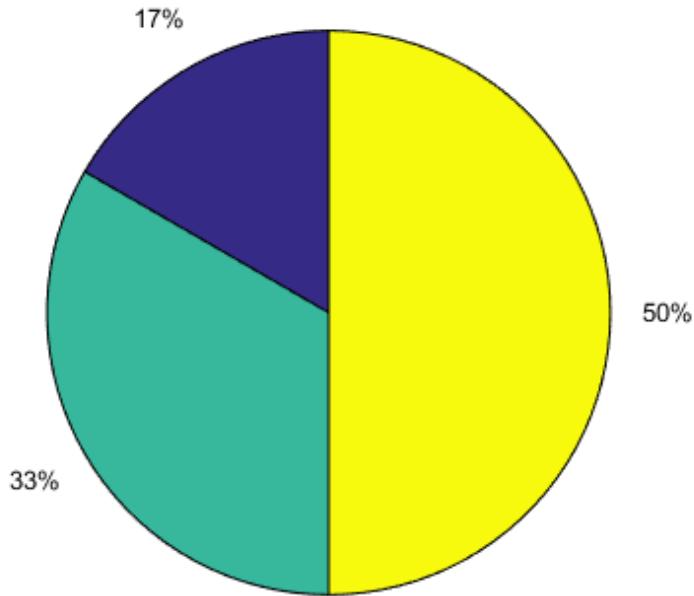
[“Determine Horizontal Distance to Move Each Label” on page 5-36](#)

[“Position New Label” on page 5-37](#)

Create Pie Chart

Define `x` and create a pie chart. Specify an output argument, `h`, to contain the text and patch handles created by the `pie` function.

```
x = [1,2,3];  
  
figure  
h = pie(x);
```



The `pie` function creates one text object and one patch object for each pie slice. By default, MATLAB labels each pie slice with the percentage of the whole that slice represents.

Note: To specify simple text labels, pass the strings directly to the `pie` function. For example, `pie(x, {'Item A', 'Item B', 'Item C'})`.

Store Precalculated Percent Values

Extract the three text handles from `h` and store them in array `hText`. Get the percent contributions for each pie slice from the `String` properties of the text objects.

```
hText = findobj(h, 'Type', 'text'); % text object handles  
percentValues = get(hText, 'String'); % percent values
```

Combine Percent Values and Additional Text

Specify the desired strings for the labels in the cell array `str`. Then, concatenate the strings with the associated percent values in the cell array `combinedstrings`.

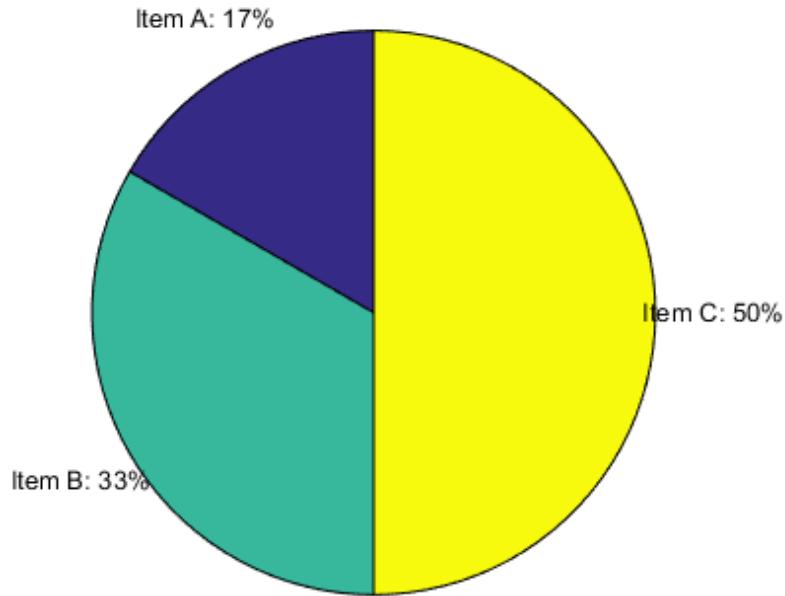
```
str = {'Item A: ' ; 'Item B: ' ; 'Item C: '}; % strings  
combinedstrings = strcat(str, percentValues); % strings and percent values
```

Before changing the labels to the new combined strings, store the text `Extent` property values for the current labels. The extent values gives the width and height of the rectangle that encloses the current labels. You use these values to adjust the position of the new labels.

```
oldExtents_cell = get(hText, 'Extent'); % cell array  
oldExtents = cell2mat(oldExtents_cell); % numeric array
```

Change the labels by setting the `String` properties of the text objects to `combinedstrings`.

```
hText(1).String = combinedstrings(1);  
hText(2).String = combinedstrings(2);  
hText(3).String = combinedstrings(3);
```



Determine Horizontal Distance to Move Each Label

Move each label so that it does not overlap the pie chart. First, get the updated extent values for the new labels from the `Extent` properties. Use the new and old extent values to find the change in width for each label.

```
newExtents_cell = get(hText, 'Extent'); % cell array  
newExtents = cell2mat(newExtents_cell); % numeric array  
width_change = newExtents(:,3)-oldExtents(:,3);
```

Use the change in width to calculate the horizontal distance to move each label. Store the calculated offsets in `offset`.

```
signValues = sign(oldExtents(:,1));
```

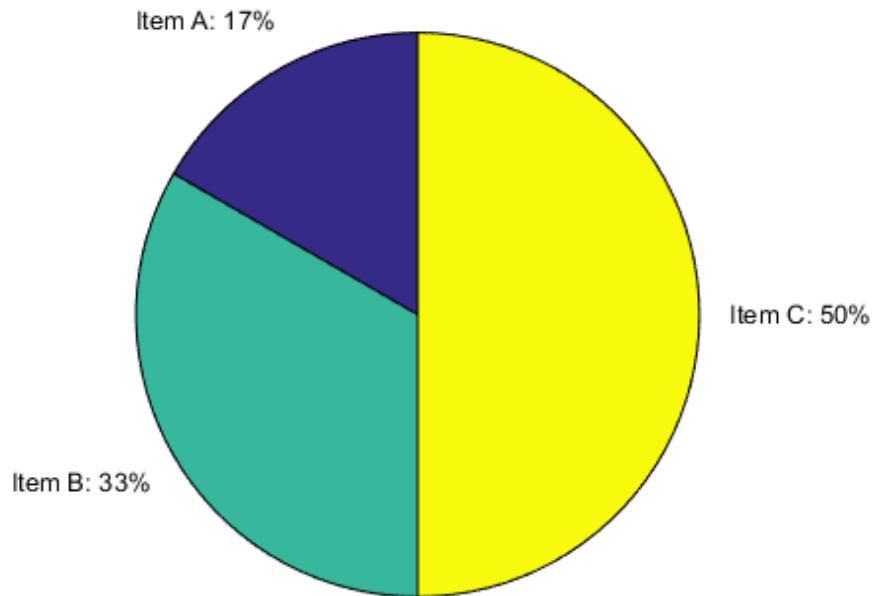
```
offset = signValues.* (width_change/2);
```

Position New Label

The **Position** property of each text object contains a three-element vector, [x, y, z], that specifies the location of the label in three-dimensions. Get the current label positions and move each label to the left or the right by adding the calculated offset to its horizontal position. Then, set the **Position** properties of the text objects to the new values.

```
textPositions_cell = get(hText,{'Position'}); % cell array
textPositions = cell2mat(textPositions_cell); % numeric array
textPositions(:,1) = textPositions(:,1) + offset; % add offset

hText(1).Position = textPositions(1,:);
hText(2).Position = textPositions(2,:);
hText(3).Position = textPositions(3,:);
```



The labels for each pie slice contain custom text with the calculated percentages and do not overlap the pie chart.

See Also

`cell2mat` | `findobj` | `pie`

Related Examples

- “Add Legend to Pie Chart”

Data Cursors with Histograms



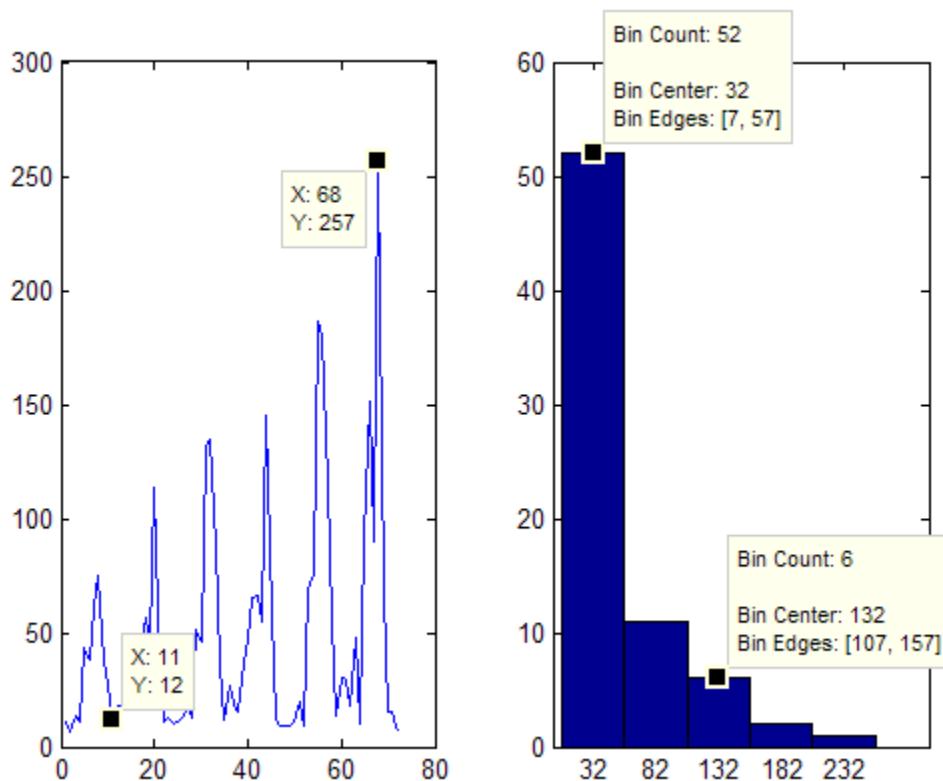
When you use the Data Cursor tool on a histogram plot, it customizes the data tips it displays in an appropriate way. Instead of providing x -, y -, z - coordinates, the datatips display the following information:

- Number of observations falling into the selected bin
- The x value of the bin's center
- The lower and upper x values for the bin

For example, The following figures show a line plot and a histogram of `count.dat`, a data set that contains three columns, giving hourly traffic counts at three different locations. The plots depict the sum the values over the locations. Each graph displays two datatips, but the datatips in the right-hand plot give information specific to histograms.

```
load count.dat
figure;
subplot(1,2,1); plot(count(:))
subplot(1,2,2); hist(count(:,5)
datacursormode on
```

Click to place a datatip or drag an existing one to a new location. You can add new datatips to a plot by right-clicking, selecting **Create new datatip**, and clicking the graph where you want to put it.



When you add datatips to histograms or bar graphs showing groups of data, you can move a datatip to any other bar by clicking inside that bar. If you use the cursor keys to shift a datatip back or forth across the graph, the datatip moves to the preceding or succeeding bar of the same color.

Combine Stem Plot and Line Plot

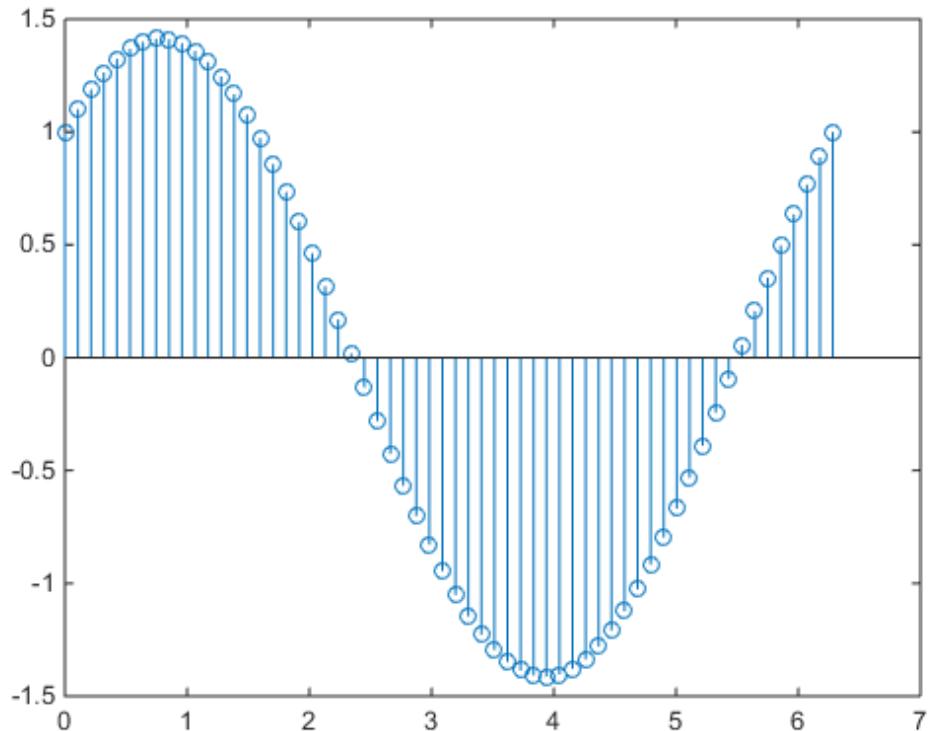
This example shows how to create a graph with a stem plot and a line plot.

Define **x** as a vector with 60 linearly spaced elements. Define **a** and **b** as sine and cosine values.

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

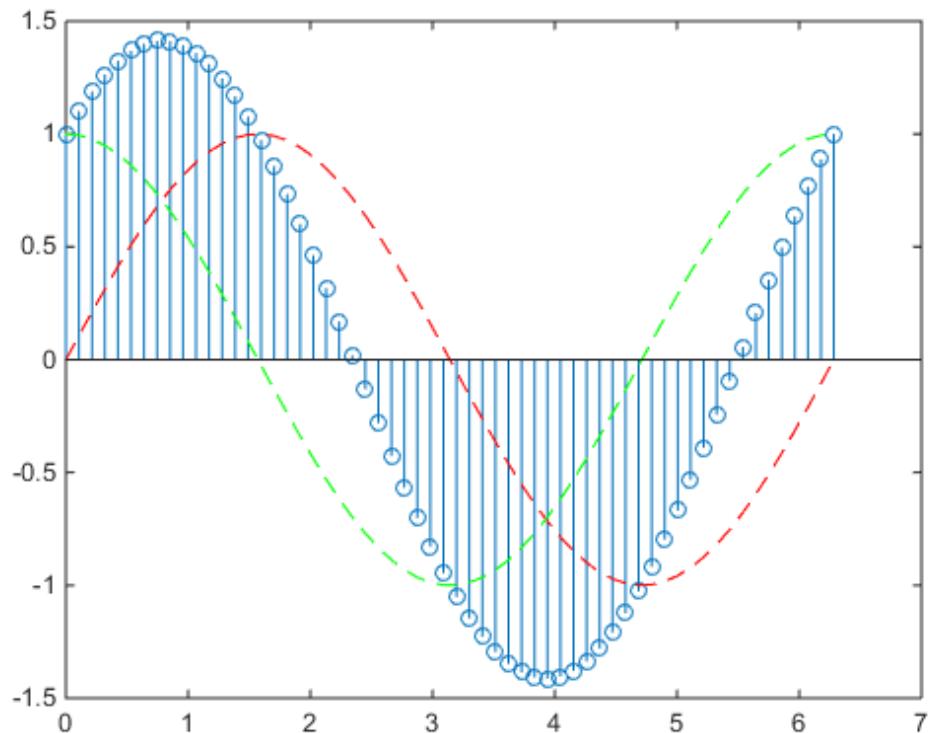
Create a stem plot of the linear combination of **a** and **b**. Return the handles to the **stemseries** object created by the **stem** function.

```
figure
hStem = stem(x,a+b);
```



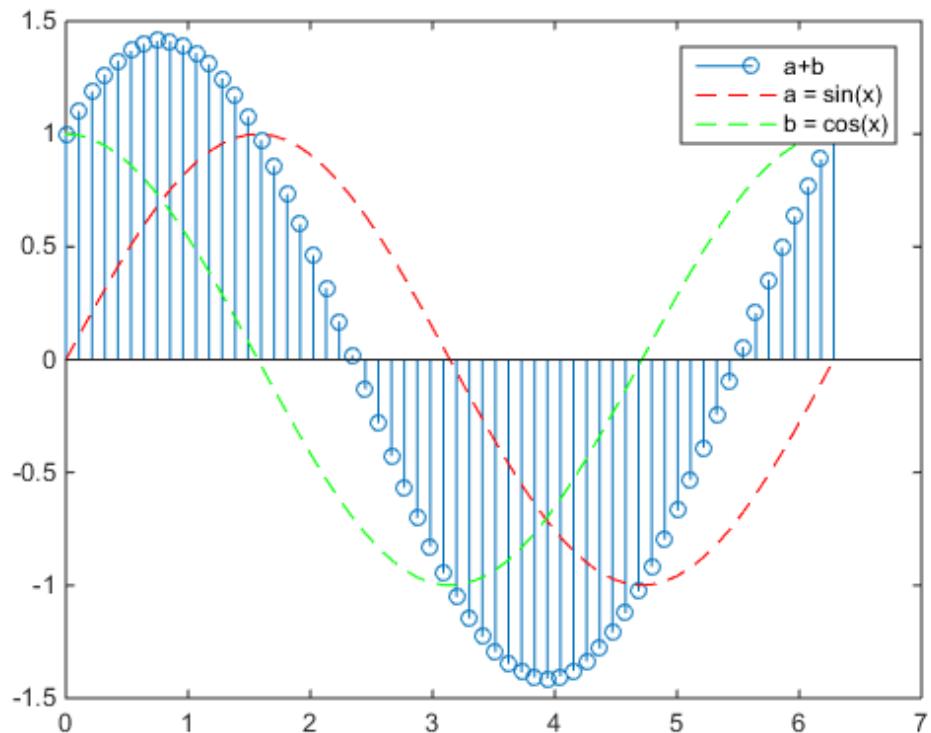
To retain the stem plot so that the **plot** command does not replace it, set the **hold** state to **on**. Overlay the stem plot with line plots of **a** and **b**. Specify a different color for each line and use a dashed line style. Return the handles to the line objects created.

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



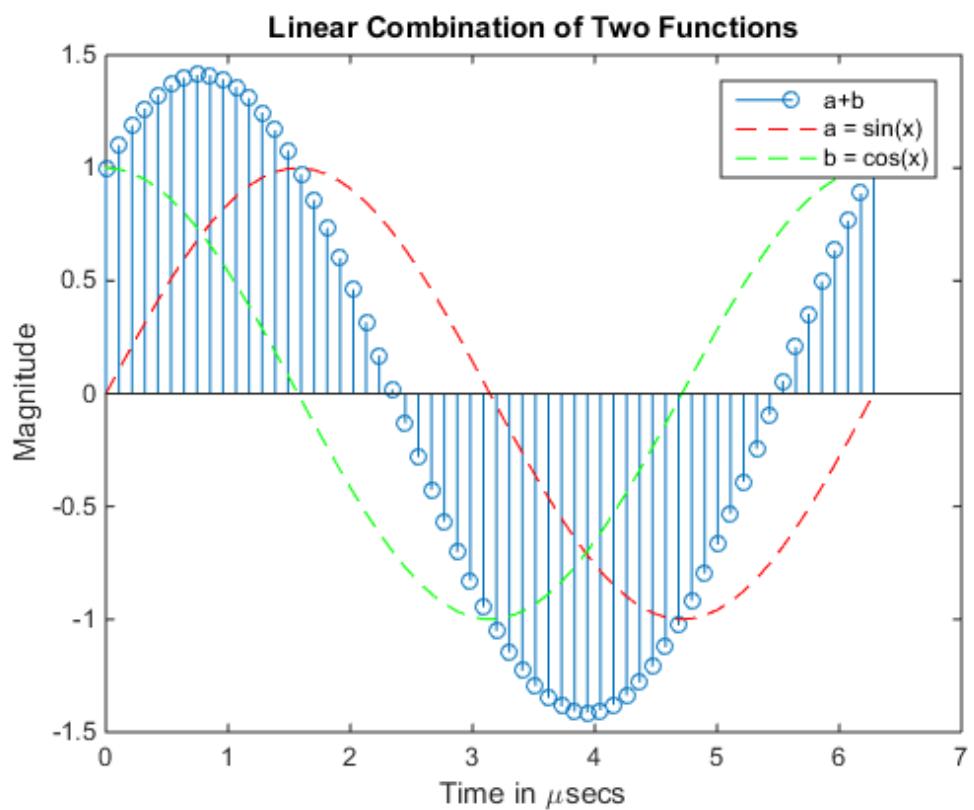
To add a legend to the graph, use the `legend` function. Pass the `stemseries` handle and `line` object handles to `legend`.

```
h = [hStem; hLine];
legend(h, 'a+b', 'a = sin(x)', 'b = cos(x)')
```



Label the axis and add a title to the graph.

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



Overlay Stairstep Plot and Line Plot

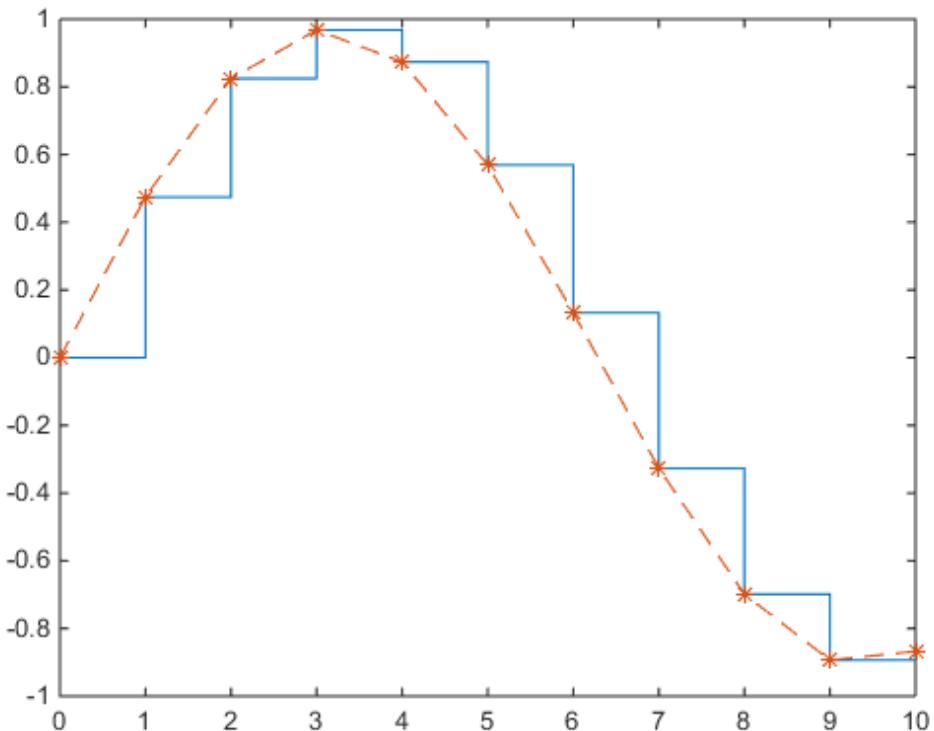
This example shows how to overlay a line plot on a stairstep plot.

Define the data to plot.

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

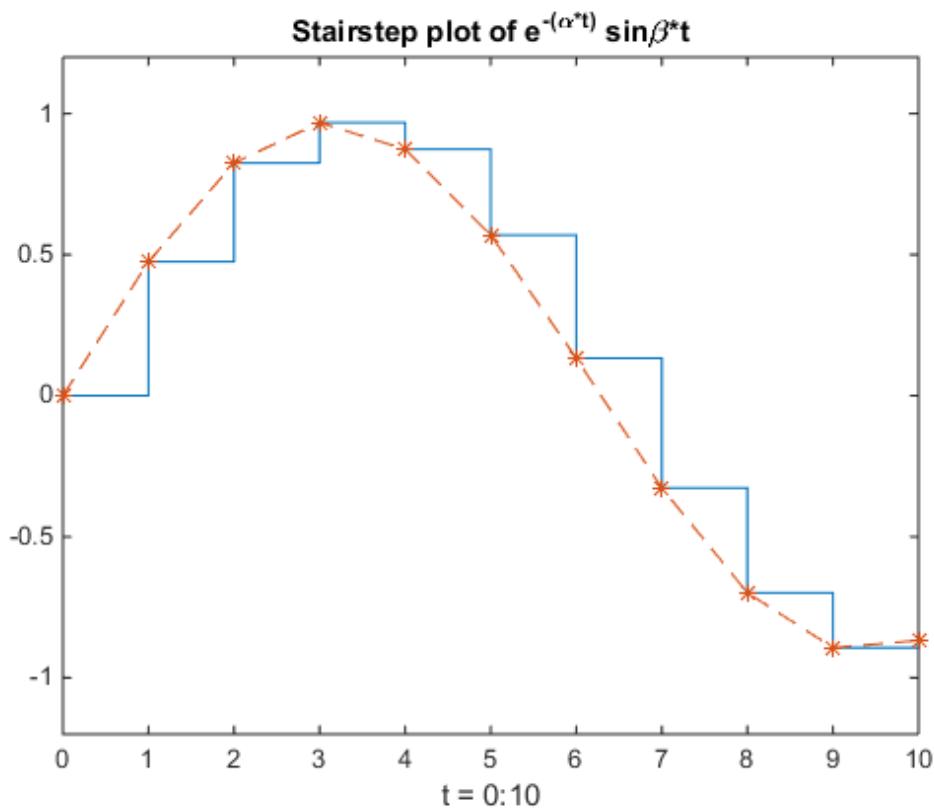
Display f as a stairstep plot. Use the `hold` function to retain the stairstep plot. Add a line plot of f using a dashed line with star markers.

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



Use the `axis` function to set the axis limits. Label the x -axis and add a title to the graph.

```
axis([0,10,-1.2,1.2])
xlabel('t = 0:10')
title('Stairstep plot of e^{-(\alpha*t)} sin\beta*t')
```



See Also

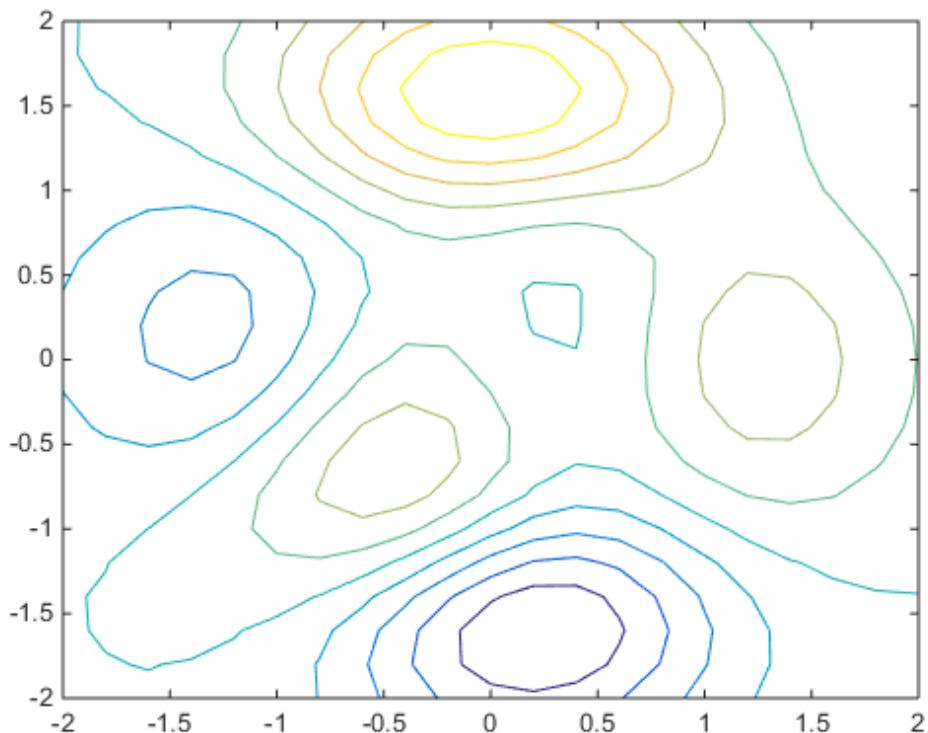
[axis](#) | [plot](#) | [stairs](#)

Display Quiver Plot Over Contour Plot

This example shows how to add a quiver plot over a contour plot.

Create 10 contours of the **peaks** function.

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

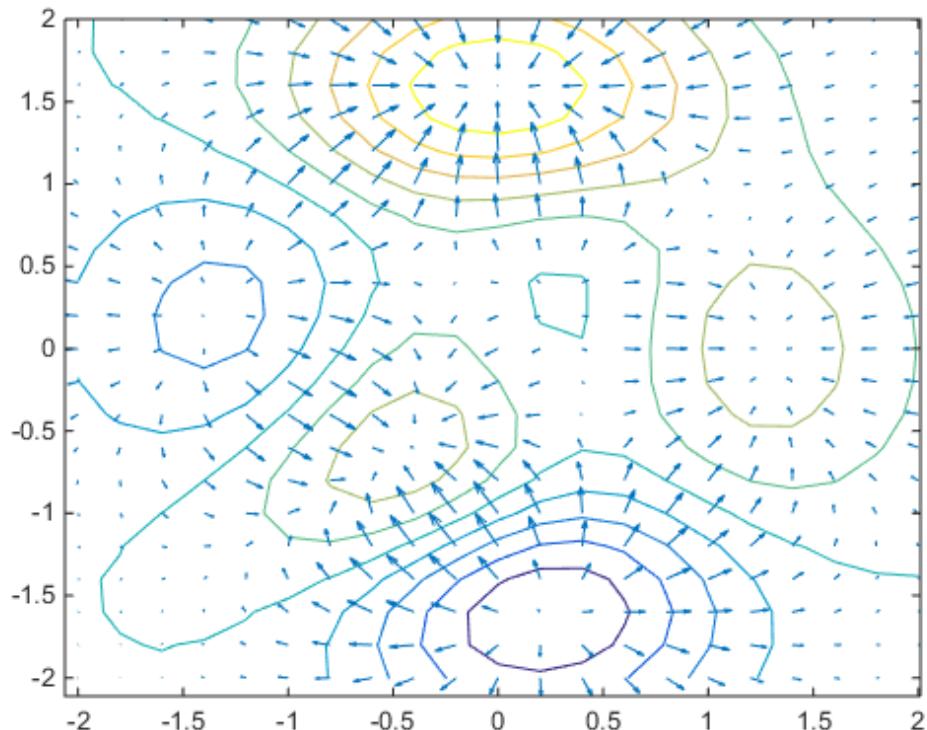


Use the **gradient** function to find the numerical gradient of matrix Z. Store U as the gradient in the x-direction and V as the gradient in the y-direction. Use a spacing of 0.2 between points in each direction.

```
[U,V] = gradient(Z,0.2);
```

Use the `quiver` function to display arrows over the contour plot indicating the gradient values.

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



Projectile Path Over Time

This example shows how to display the path of a projectile as a function of time using a three-dimensional quiver plot.

Show the path of the following projectile using constants for velocity and acceleration, v_z and a .

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

```
vz = 10; % velocity constant
a = -32; % acceleration constant
```

Calculate z as the height as time varies from 0 to 1.

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

Calculate the position in the x -direction and y -direction.

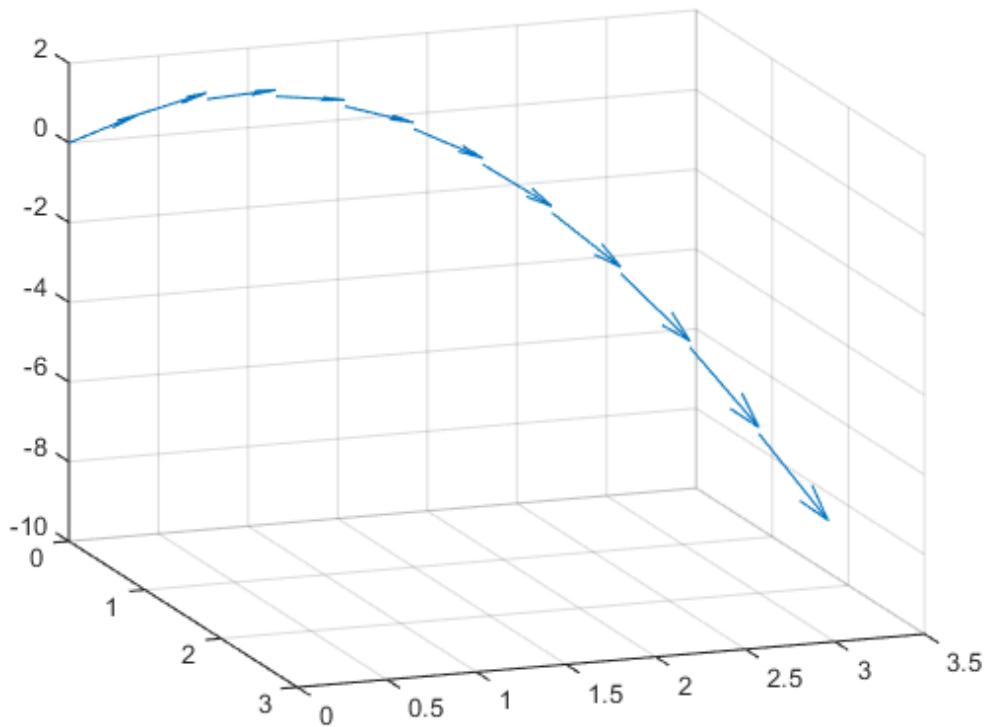
```
vx = 2;
x = vx*t;

vy = 3;
y = vy*t;
```

Compute the components of the velocity vectors and display the vectors using a 3-D quiver plot. Change the viewpoint of the axes to [70,18].

```
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;

figure
quiver3(x,y,z,u,v,w,scale)
view([70,18])
```



Label Contour Plot Levels

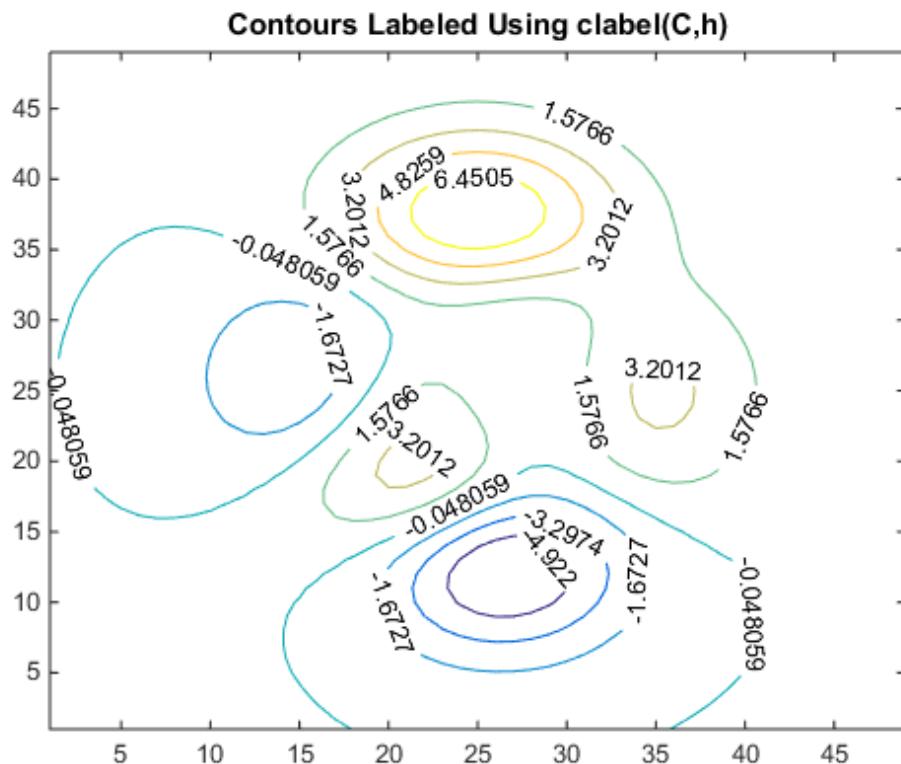
This example shows how to label each contour line with its associated value.

The contour matrix, `C`, is an optional output argument returned by `contour`, `contour3`, and `contourf`. The `clabel` function uses values from `C` to display labels for 2-D contour lines.

Display eight contour levels of the `peaks` function and label the contours. `clabel` labels only contour lines that are large enough to contain an inline label.

```
Z = peaks;
figure
[C,h] = contour(Z,8);

clabel(C,h)
title('Contours Labeled Using clabel(C,h)')
```



To interactively select the contours to label using the mouse, pass the `manual` option to `clabel`, for example, `clabel(C,h,'manual')`. This command displays a crosshair cursor when the mouse is within the figure. Click the mouse to label the contour line closest to the cursor.

See Also

`clabel` | `contour` | `contour3` | `contourf`

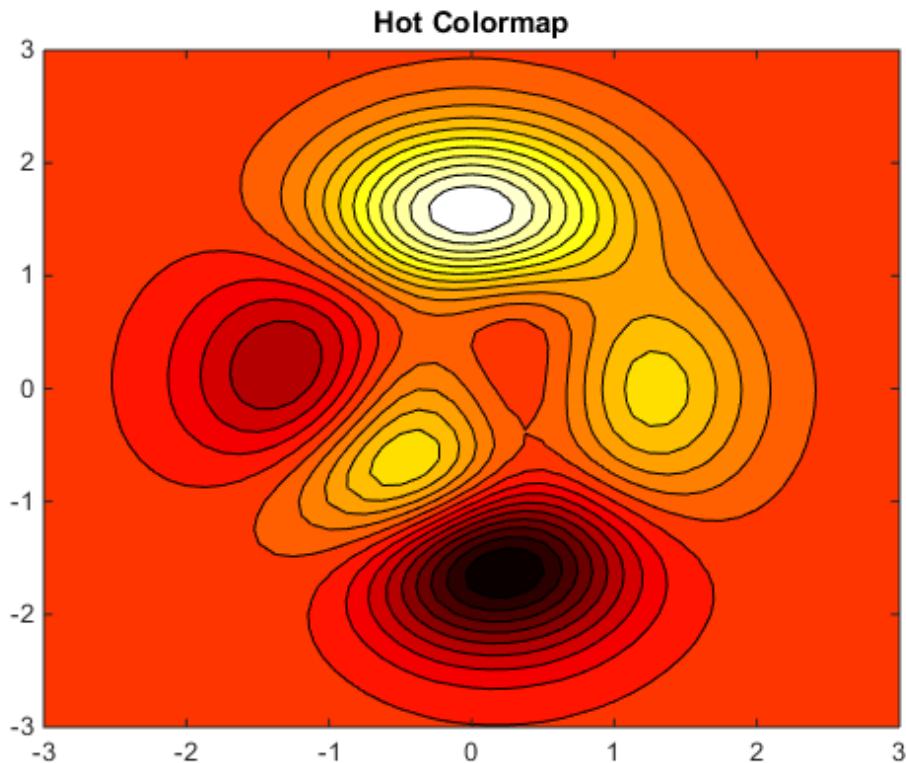
Change Fill Colors for Contour Plot

This example shows how to change the colors used in a filled contour plot.

Change Colormap

Set the colors for the filled contour plot by changing the colormap. Pass the predefined colormap name, `hot`, to the `colormap` function.

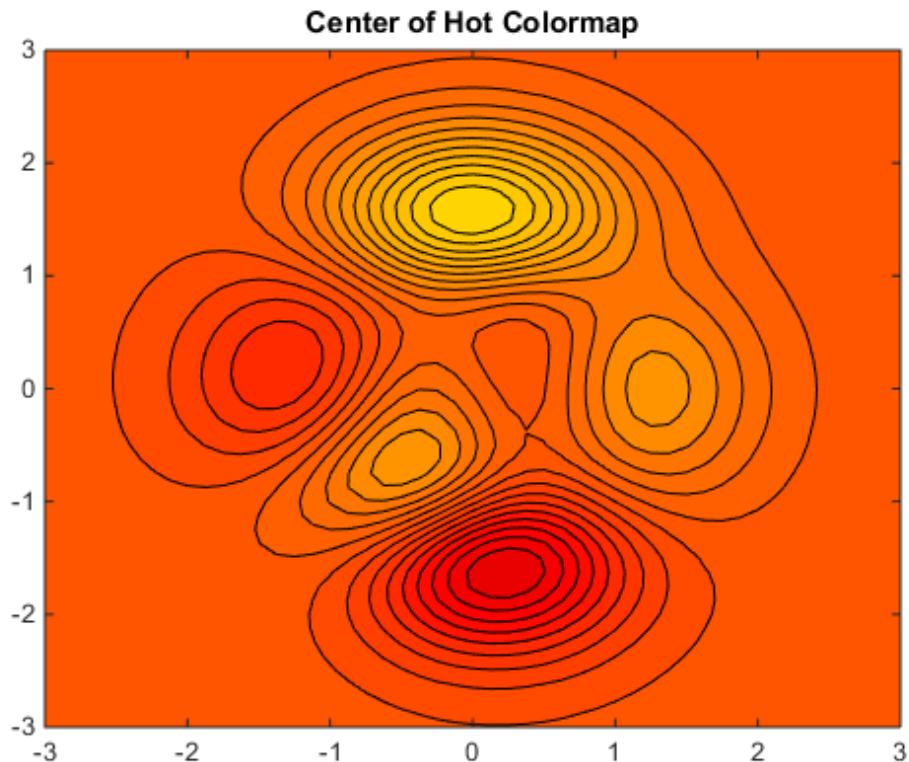
```
[X,Y,Z] = peaks;
figure
contourf(X,Y,Z,20)
colormap(hot)
title('Hot Colormap')
```



Control Mapping of Data Values to Colormap

Use only the colors in the center of the hot colormap by setting the color axis scaling to a range much larger than the range of values in matrix Z. The `caxis` function controls the mapping of data values into the colormap. Use this function to set the color axis scaling.

```
caxis([-20,20])  
title('Center of Hot Colormap')
```



See Also

`caxis` | `colormap` | `contourf`

Highlight Specific Contour Levels

This example shows how to highlight contours at particular levels.

Define `Z` as the matrix returned from the `peaks` function.

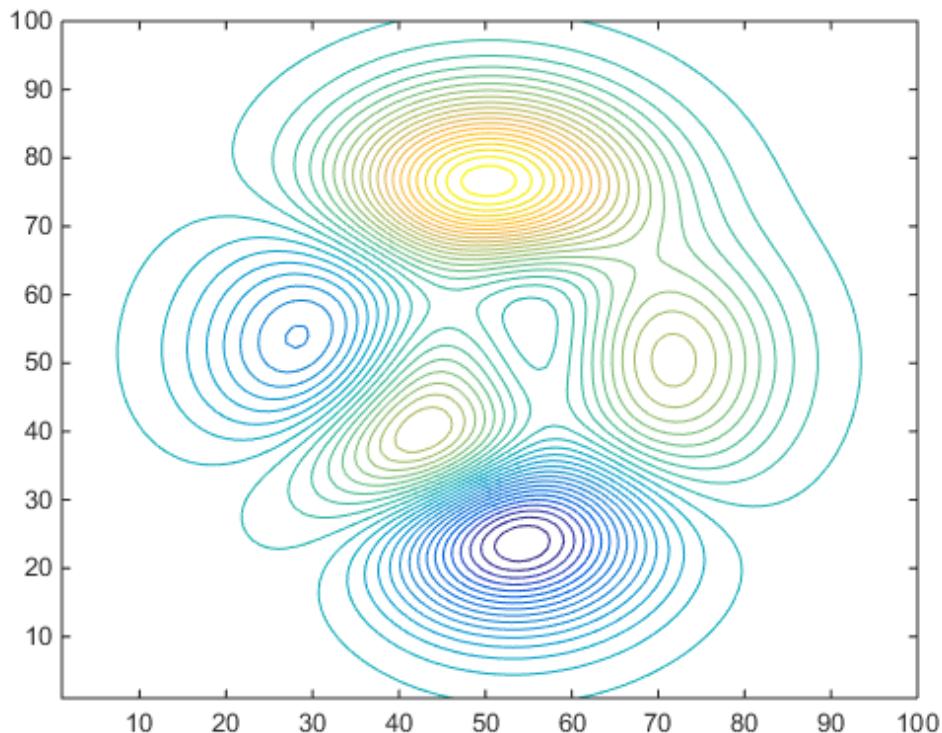
```
Z = peaks(100);
```

Round the minimum and maximum data values in `Z` and store these values in `zmin` and `zmax`, respectively. Define `zlevs` as 40 values between `zmin` and `zmax`.

```
zmin = floor(min(Z(:)));
zmax = ceil(max(Z(:)));
zinc = (zmax - zmin) / 40;
zlevs = zmin:zinc:zmax;
```

Plot the contour lines.

```
figure
contour(Z,zlevs)
```

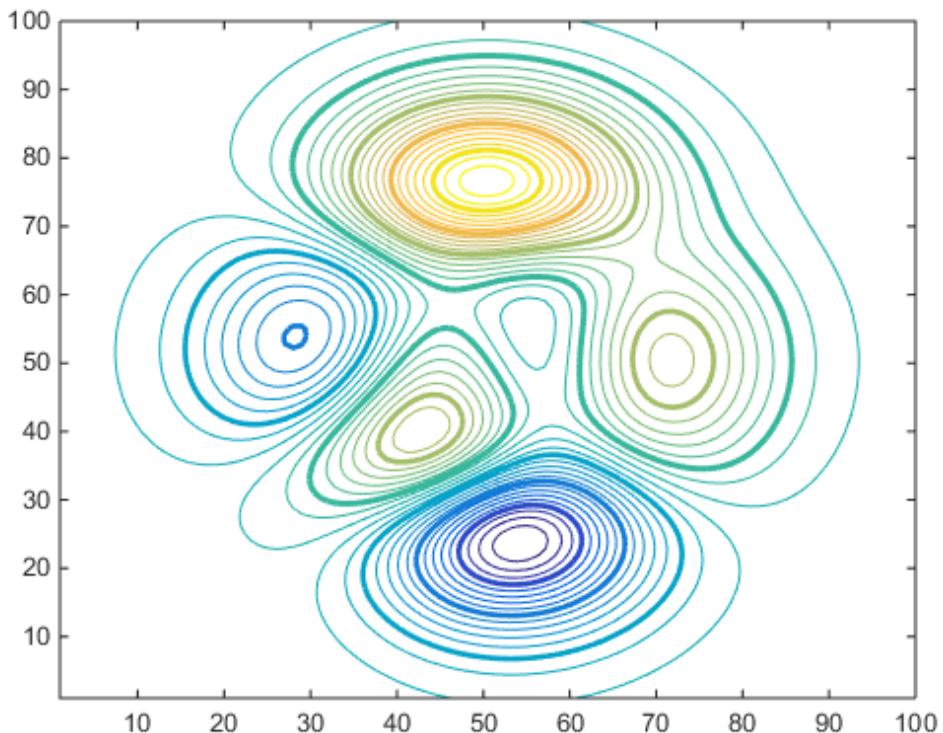


Define `zindex` as a vector of integer values between `zmin` and `zmax` indexed by 2.

```
zindex = zmin:2:zmax;
```

Retain the previous contour plot. Create a second contour plot and use `zindex` to highlight contour lines at every other integer value. Set the line width to 2.

```
hold on  
contour(Z,zindex,'LineWidth',2)  
hold off
```



See Also

[ceil](#) | [contour](#) | [floor](#) | [hold](#) | [max](#) | [min](#)

Contour Plot in Polar Coordinates

This example shows how to create a contour plot for data defined in a polar coordinate system.

Display Surface to Contour

Set up a grid in polar coordinates and convert the coordinates to Cartesian coordinates.

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

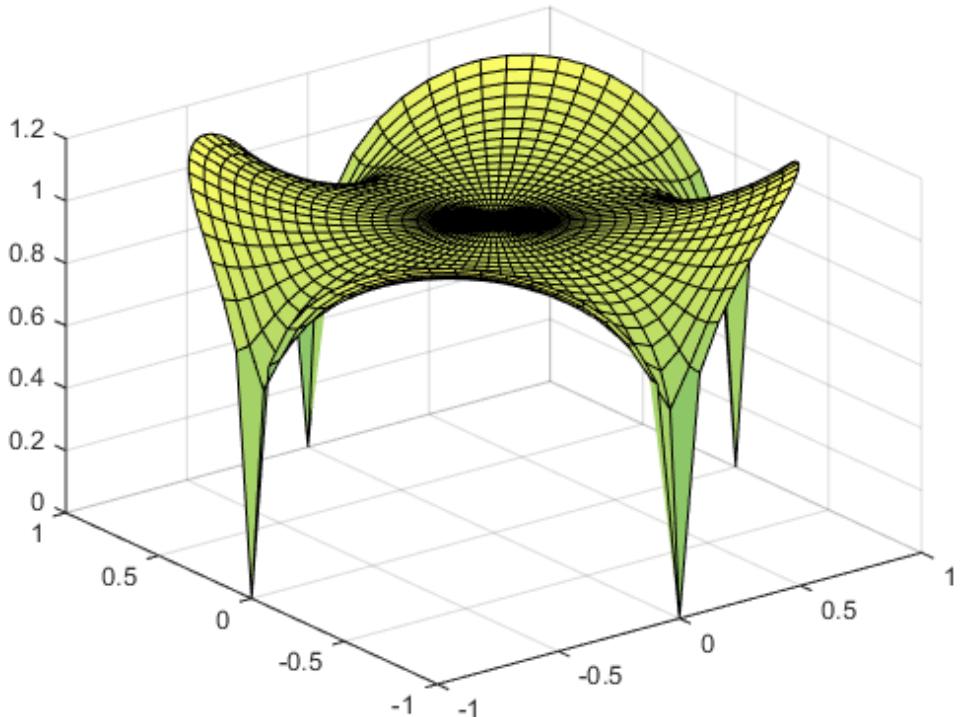
Generate the complex matrix Z on the interior of the unit circle.

```
Z = X + 1i*Y;
```

Display a surface of the mathematical function $\sqrt[4]{z^4 - 1}$. Use the **summer** colormap.

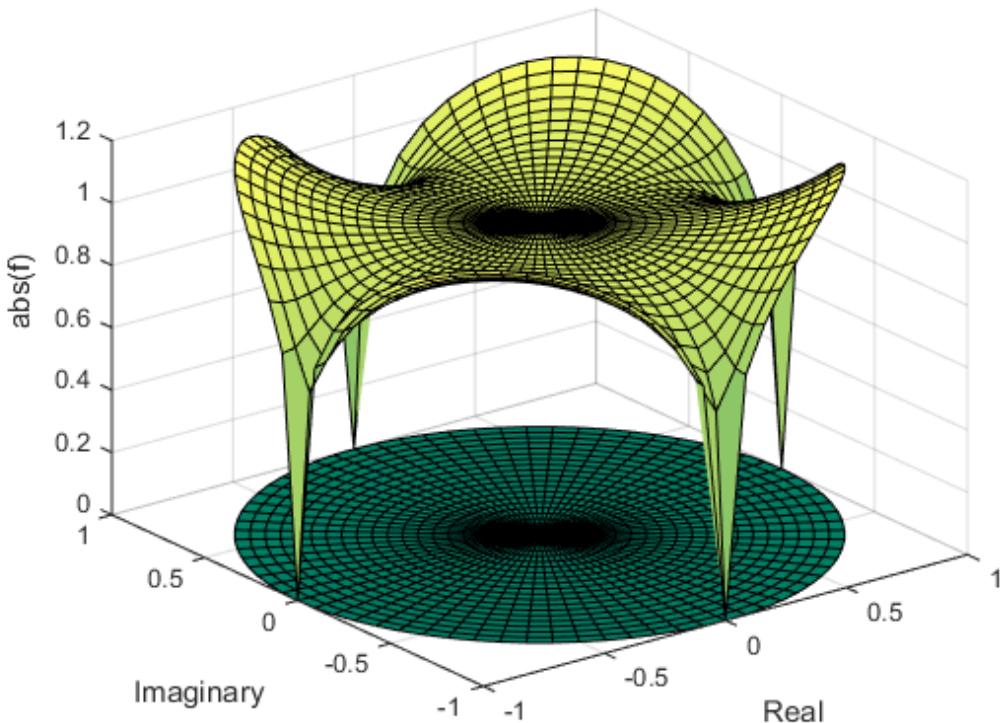
```
f = (Z.^4-1).^(1/4);

figure
surf(X,Y,abs(f))
colormap summer
```



Display the unit circle beneath the surface and add labels to the graph.

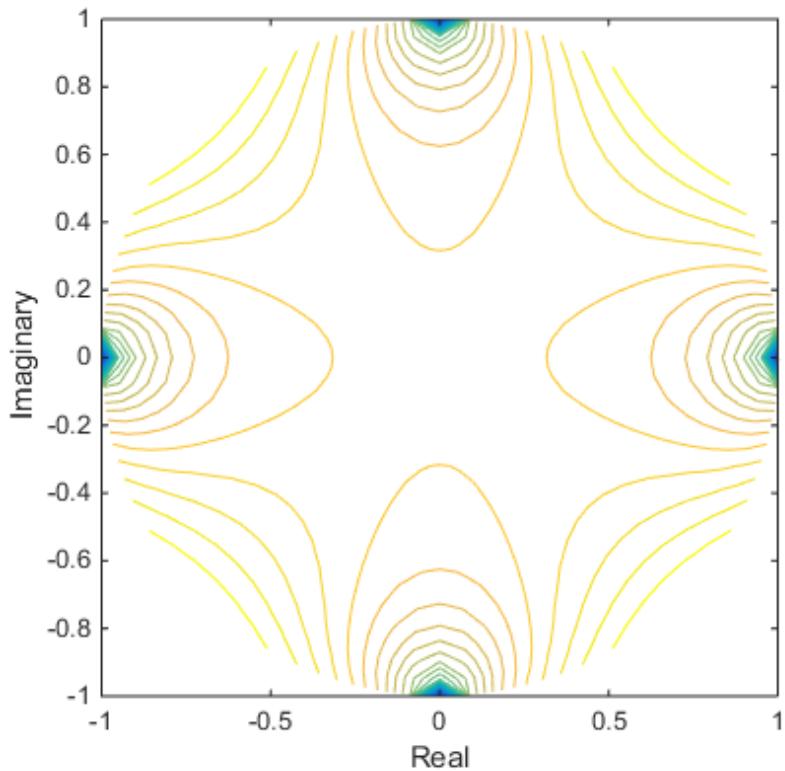
```
hold on
surf(X,Y,zeros(size(X)))
hold off
xlabel('Real')
ylabel('Imaginary')
zlabel('abs(f)')
```



Contour Plot in Cartesian Coordinates

Display a contour plot of the surface in Cartesian coordinates.

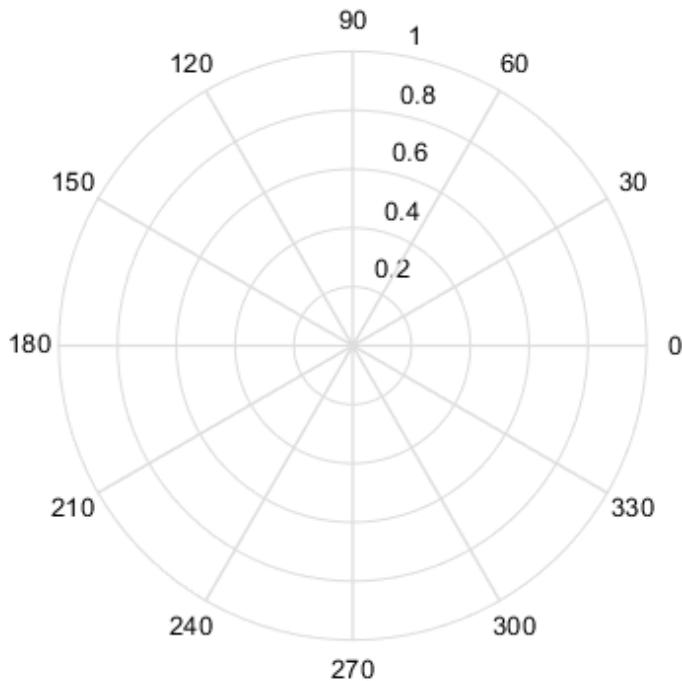
```
figure
contour(X,Y,abs(f),30)
axis equal
xlabel('Real')
ylabel('Imaginary')
```



Contour Plot in Polar Coordinates

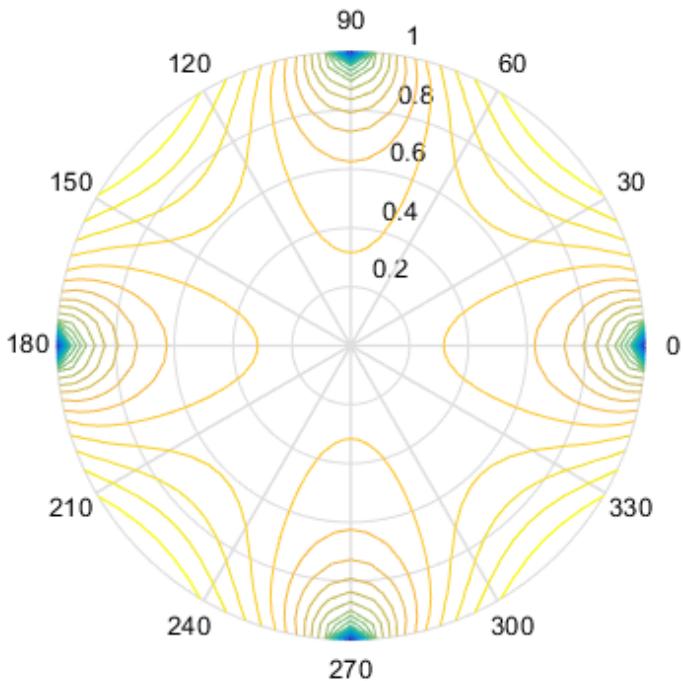
Display a contour plot of the surface in a polar axes. Use the **polar** function to create a polar axes, and then delete the line created with **polar**.

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



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

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



See Also

[colormap](#) | [contour](#) | [meshgrid](#) | [pol2cart](#) | [polar](#) | [surf](#)

Animation Techniques

In this section...

“Updating the Screen” on page 5-66

“Optimizing Performance” on page 5-66

You can use three basic techniques for creating animations in MATLAB:

- Update the properties of a graphics object and display the updates on the screen. This technique is useful for creating animations when most of the graph remains the same. For example, set the `XData` and `YData` properties repeatedly to move an object in the graph.
- Apply transforms to objects. This technique is useful when you want to operate on the position and orientation of a group of objects together. Group the objects as children under a transform object. Create the transform object using `hgtransform`. Setting the `Matrix` property of the transform object adjusts the position of all its children.
- Create a movie. Movies are useful if you have a complex animation that does not draw quickly in real time, or if you want to store an animation to replay it. Use the `getframe` and `movie` functions to create a movie.

Updating the Screen

In some cases, MATLAB does not update the screen until the code finishes executing. Use one of the `drawnow` commands to display the updates on the screen throughout the animation.

Optimizing Performance

To optimize performance, consider these techniques:

- Use the `animatedline` function to create line animations of streaming data.
- Update properties of an existing object instead of creating new graphics objects.
- Set the axis limits (`XLim`, `YLim`, `ZLim`) or change the associated mode properties to manual mode (`XLimMode`, `YLimMode`, `ZLimMode`) so that MATLAB does not recalculate the values each time the screen updates. When you set the axis limits, the associated mode properties change to manual mode.

- Avoid creating a legend or other annotations within a loop. Add the annotation after the loop.

For more information on optimizing performance, see “Graphics Performance”.

Related Examples

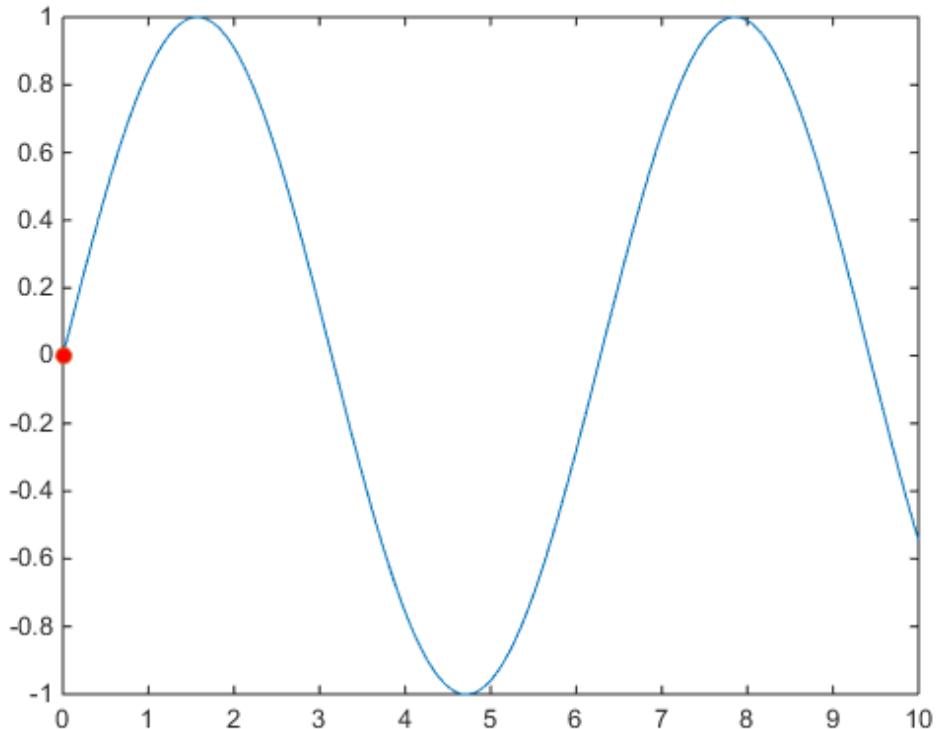
- “Trace Marker Along Line” on page 5-68
- “Move Group of Objects Along Line” on page 5-71
- “Line Animations” on page 5-79
- “Record Animation for Playback” on page 5-82

Trace Marker Along Line

This example shows how to trace a marker along a line by updating the data properties of the marker.

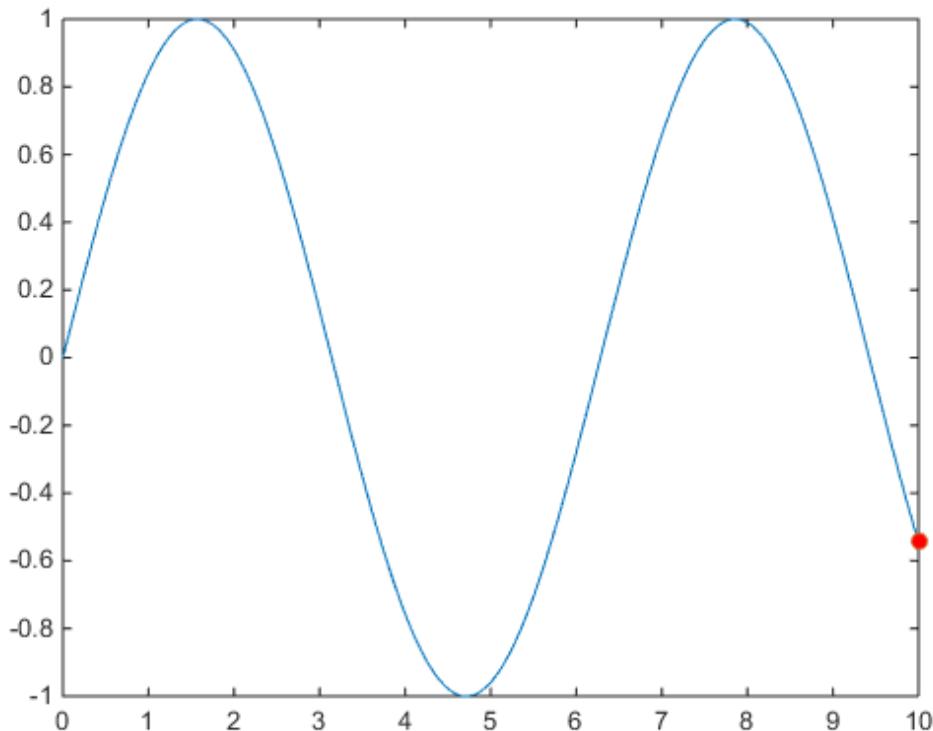
Plot a sine wave and a red marker at the beginning of the line. Set the axis limits mode to manual to avoid recalculating the limits throughout the animation loop.

```
x = linspace(0,10,10000);
y = sin(x);
plot(x,y)
hold on
p = plot(x(1),y(1), 'o', 'MarkerFaceColor', 'red');
hold off
axis manual
```



Move the marker along the line by updating the **XData** and **YData** properties in a loop.
Use a “**drawnow**” or **drawnow update** command to display the updates on the screen.
drawnow update is fastest, but it might not draw every frame on the screen.

```
for k = 2:length(x)
    p.XData = x(k);
    p.YData = y(k);
    drawnow update
end
```



The animation shows the marker moving along the line.

See Also

`drawnow` | `linspace` | `plot`

Related Examples

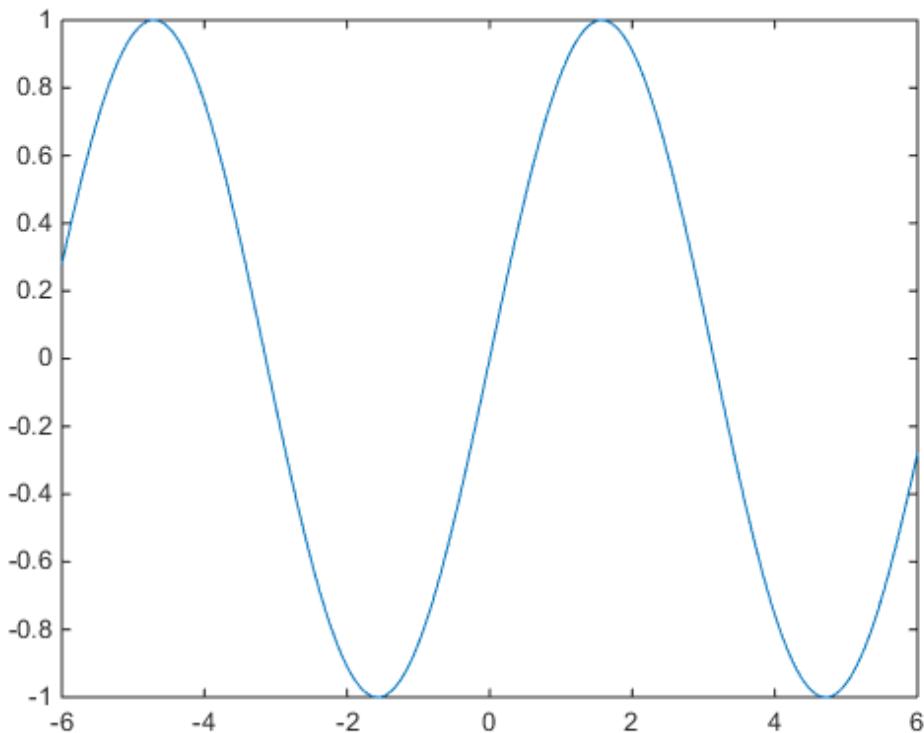
- “Move Group of Objects Along Line” on page 5-71
- “Animate Graphics Object” on page 5-75
- “Record Animation for Playback” on page 5-82
- “Line Animations” on page 5-79

Move Group of Objects Along Line

This example shows how to move a group of objects together along a line using transforms.

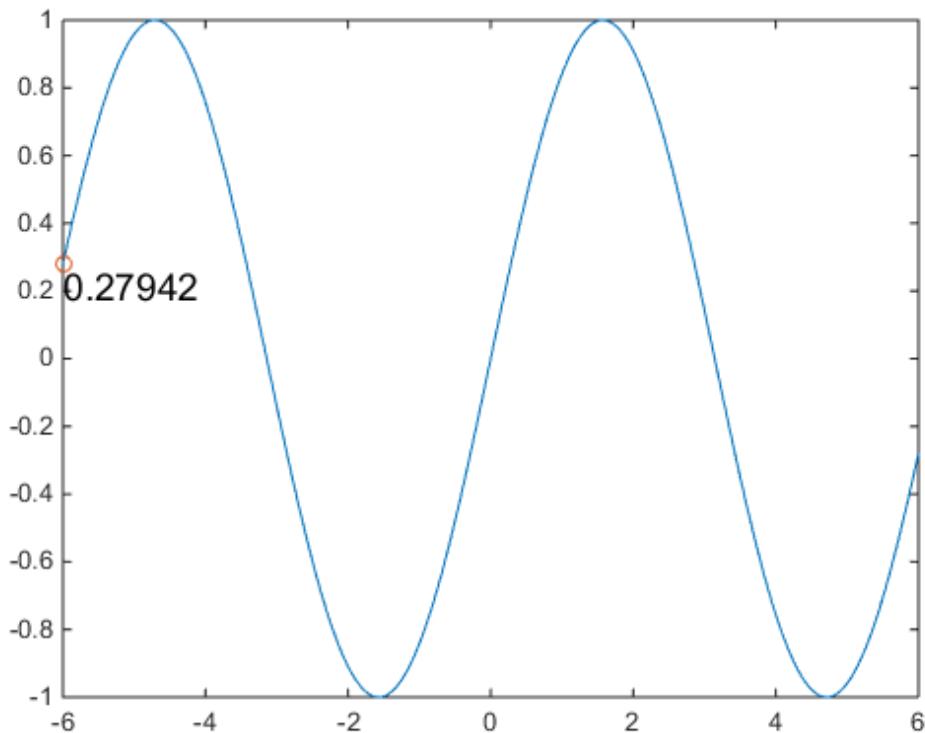
Plot a sine wave and set the axis limits mode to manual to avoid recalculating the limits during the animation loop.

```
x = linspace(-6,6,1000);  
y = sin(x);  
plot(x,y)  
axis manual
```



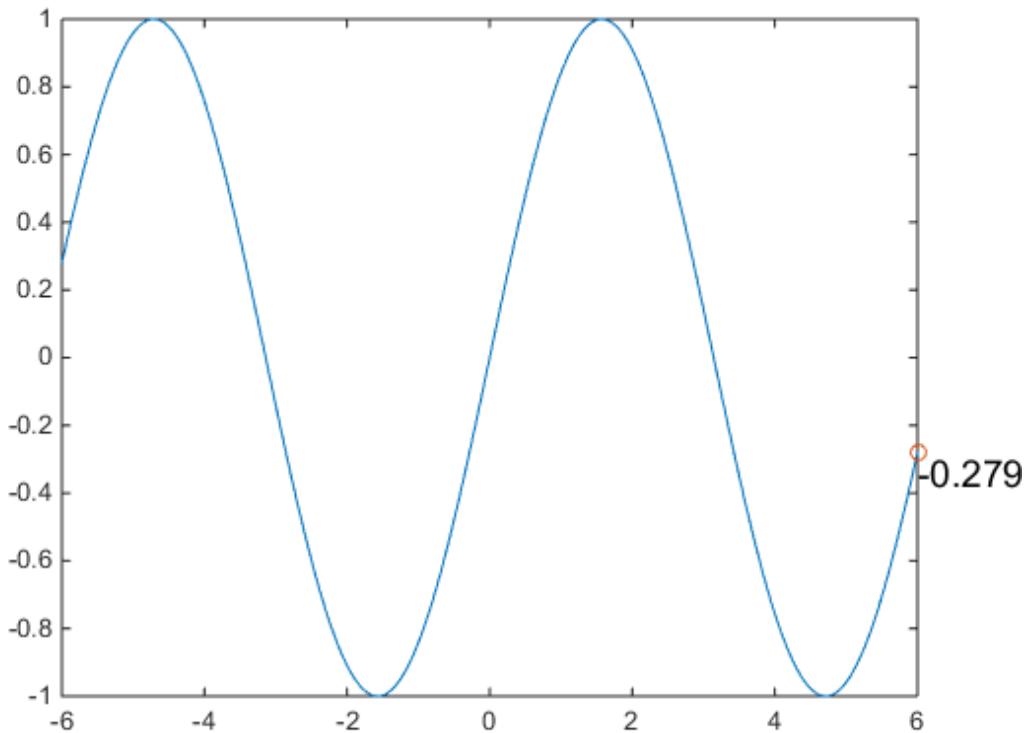
Create a transform object and set its parent to the current axes. Plot a marker and a text annotation at the beginning of the line. Use the `num2str` function to convert the y -value at that point to a text string. Group the two objects by setting their parents to the transform object.

```
ax = gca;
h = hgtransform('Parent',ax);
hold on
plot(x(1),y(1),'o','Parent',h);
hold off
t = text(x(1),y(1),num2str(y(1)), 'Parent',h,...
    'VerticalAlignment','top','FontSize',14);
```



Move the marker and text to each subsequent point along the line by updating the `Matrix` property of the transform object. Use the `x` and `y` values of the next point in the line and the first point in the line to determine the transform matrix. Update the text string to match the `y`-value as it moves along the line. Use `drawnow` to display the updates to the screen after each iteration.

```
for k = 2:length(x)
    m = makehgform('translate',x(k)-x(1),y(k)-y(1),0);
    h.Matrix = m;
    t.String = num2str(y(k));
    drawnow
end
```



The animation shows the marker and text moving together along the line.

If you have a lot of data, you can use `drawnow update` instead of “`drawnow`” for a faster animation. However, `drawnow update` might not draw every frame on the screen if the previous frame is still rendering.

See Also

`axis` | `drawnow` | `hgtransform` | `makehgform` | `plot` | `text`

Related Examples

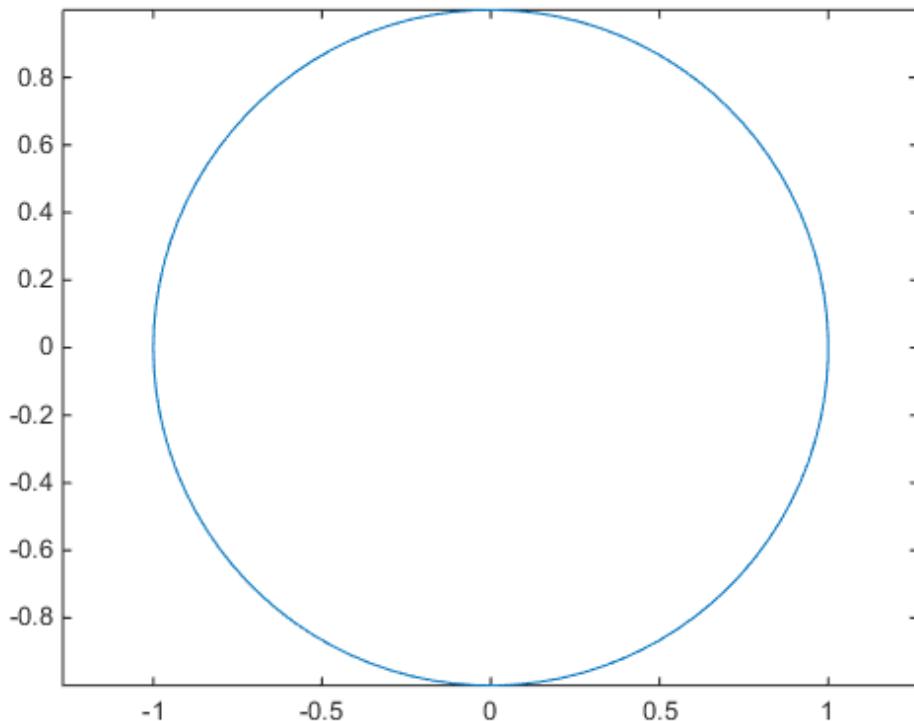
- “Animate Graphics Object” on page 5-75
- “Record Animation for Playback” on page 5-82
- “Line Animations” on page 5-79

Animate Graphics Object

This example shows how to animate a triangle looping around the inside of a circle by updating the data properties of the triangle.

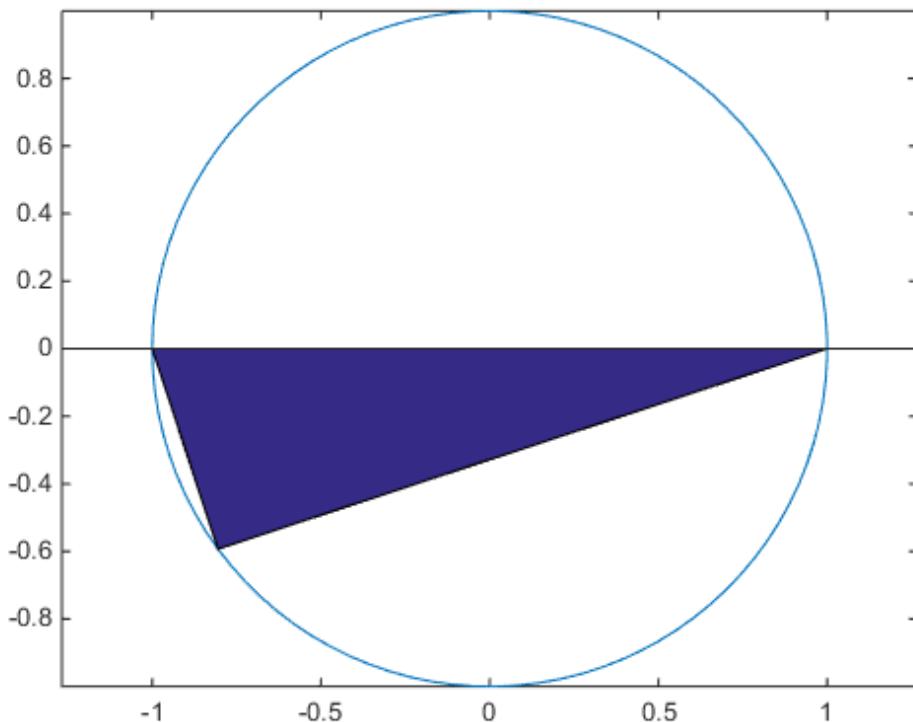
Plot the circle and set the axis limits so that the data units are the same in both directions.

```
theta = linspace(-pi,pi);
xc = cos(theta);
yc = -sin(theta);
plot(xc,yc);
axis equal
```



Use the `area` function to draw a flat triangle. Then, change the value of one of the triangle vertices using the (x,y) coordinates of the circle. Change the value in a loop to create an animation. Use a “`drawnow`” or `drawnow update` command to display the updates after each iteration. `drawnow update` is fastest, but it might not draw every frame on the screen.

```
xt = [-1 0 1 -1];
yt = [0 0 0 0];
hold on
t = area(xt,yt); % initial flat triangle
hold off
for j = 1:length(theta)-10
    xt(2) = xc(j); % determine new vertex value
    yt(2) = yc(j);
    t.XData = xt; % update data properties
    t.YData = yt;
    drawnow update % display updates
end
```



The animation shows the triangle looping around the inside of the circle.

See Also

`area` | `axis` | `drawnow` | `hold` | `plot`

Related Examples

- “Trace Marker Along Line” on page 5-68
- “Line Animations” on page 5-79
- “Record Animation for Playback” on page 5-82

More About

- “Animation Techniques” on page 5-66

Line Animations

This example shows how to create an animation of two growing lines. The **animatedline** function helps you to optimize line animations. It allows you to add new points to a line without redefining existing points.

Create Lines and Add Points

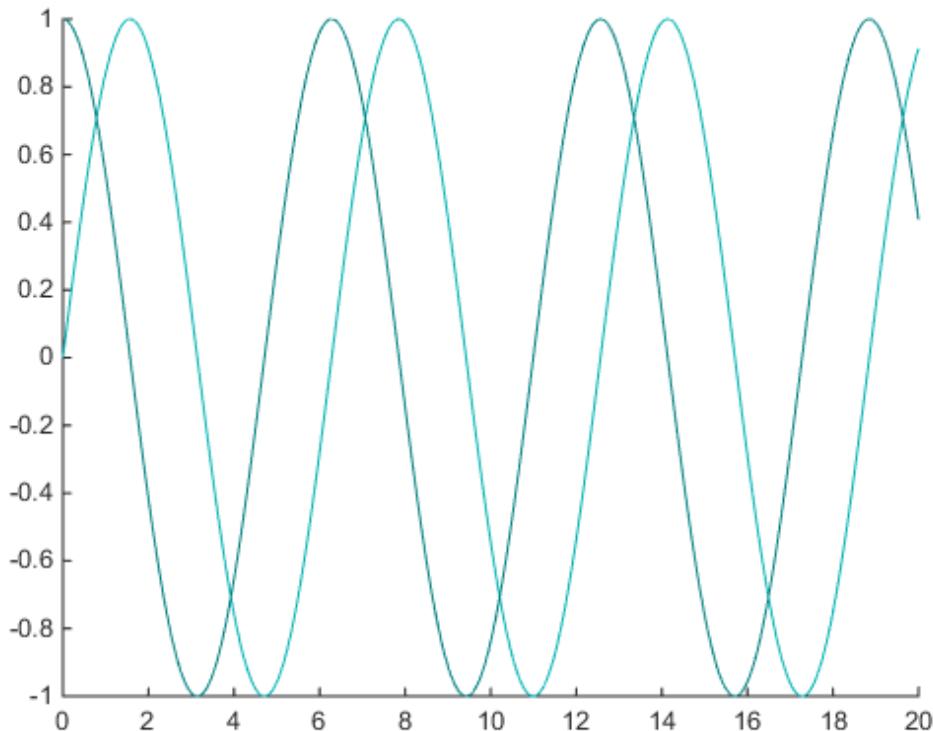
Create two animated lines of different colors. Then, add points to the lines in a loop. Set the axis limits before the loop so that to avoid recalculating the limits each time through the loop. Use a “**drawnow**” or **drawnow update** command to display the updates on the screen after adding the new points.

```
a1 = animatedline('Color',[0 .7 .7]);
a2 = animatedline('Color',[0 .5 .5]);

axis([0 20 -1 1])
x = linspace(0,20,10000);
for k = 1:length(x);
    % first line
    xk = x(k);
    ysin = sin(xk);
    addpoints(a1,xk,ysin);

    % second line
    ycos = cos(xk);
    addpoints(a2,xk,ycos);

    % update screen
    drawnow update
end
```



The animation shows two lines that grow as they accumulate data.

Query Points of Line

Query the points of the first animated line.

```
[x,y] = getpoints(a1);
```

x and y are vectors that contain the values defining the points of the sine wave.

See Also

[addpoints](#) | [animatedline](#) | [clearpoints](#) | [drawnow](#) | [getpoints](#)

Related Examples

- “Trace Marker Along Line” on page 5-68
- “Move Group of Objects Along Line” on page 5-71
- “Record Animation for Playback” on page 5-82

More About

- “Animation Techniques” on page 5-66

Record Animation for Playback

These examples show how to record animations as movies that you can replay.

In this section...

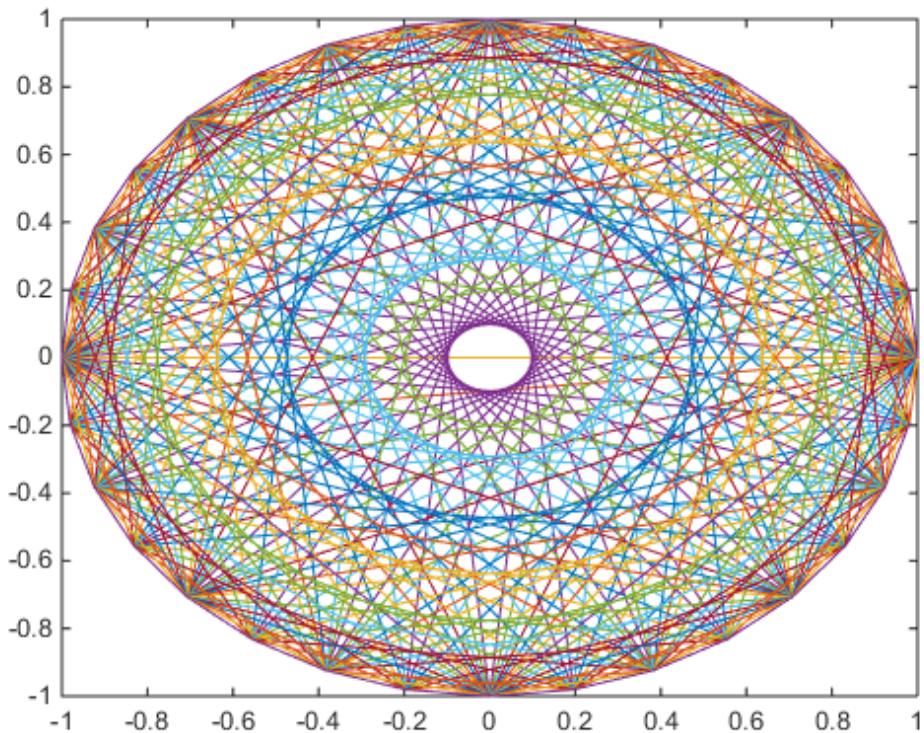
“Record and Play Back Movie” on page 5-82

“Capture Entire Figure for Movie” on page 5-83

Record and Play Back Movie

Create a series of plots within a loop and capture each plot as a frame. Ensure the axis limits stay constant by setting them each time through the loop. Store the frames in M.

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



Play back the movie five times using the `movie` function.

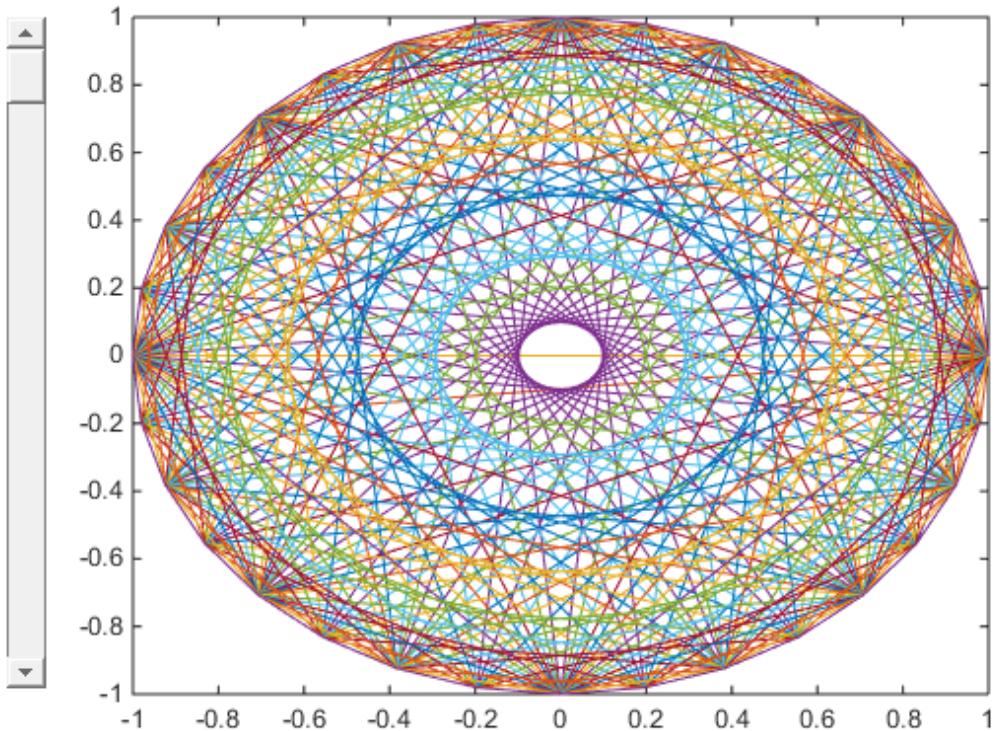
```
figure  
movie(M,5)
```

Capture Entire Figure for Movie

Include a slider on the left side of the figure. Capture the entire figure window by specifying the figure as an input argument to the `getframe` function.

```
figure  
u = uicontrol('Style','slider','Position',[10 50 20 340],...  
'Min',1,'Max',16,'Value',1);
```

```
for k = 1:16
    plot(fft(eye(k+16)))
    axis([-1 1 -1 1])
    u.Value = k;
    M(k) = getframe(gcf);
end
```



Play back the movie five times. Movies play back within the current axes. Create a new figure and an axes to fill the figure window so that the movie looks like the original animation.

```
figure
axes('Position',[0 0 1 1])
movie(M,5)
```

See Also

`axes` | `axis` | `eye` | `fft` | `getframe` | `movie` | `plot`

Related Examples

- “Animate Graphics Object” on page 5-75
- “Line Animations” on page 5-79

More About

- “Animation Techniques” on page 5-66

Displaying Bit-Mapped Images

- “Working with Images in MATLAB Graphics” on page 6-2
- “Image Types” on page 6-5
- “8-Bit and 16-Bit Images” on page 6-10
- “Read, Write, and Query Image Files” on page 6-18
- “Displaying Graphics Images” on page 6-22
- “The Image Object and Its Properties” on page 6-27
- “Printing Images” on page 6-34
- “Convert Image Graphic or Data Type” on page 6-35

Working with Images in MATLAB Graphics

In this section...

[“What Is Image Data?” on page 6-2](#)

[“Supported Image Formats” on page 6-3](#)

[“Functions for Reading, Writing, and Displaying Images” on page 6-4](#)

What Is Image Data?

The basic MATLAB data structure is the *array*, an ordered set of real or complex elements. An array is naturally suited to the representation of *images*, real-valued, ordered sets of color or intensity data. (An array is suited for complex-valued images.)

In the MATLAB workspace, most images are represented as two-dimensional arrays (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 stored as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

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

`I(2,15)`

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

The following sections describe the different data and image types, and give details about how to read, write, work with, and display graphics images; how to alter the display properties and aspect ratio of an image during display; how to print an image; and how to convert the data type or graphics format of an image.

Data Types

MATLAB math supports three different numeric classes for image display:

- double-precision floating-point (`double`)

- 16-bit unsigned integer (`uint16`)
- 8-bit unsigned integer (`uint8`)

The image display commands interpret data values differently depending on the numeric class the data is stored in. “8-Bit and 16-Bit Images” on page 6-10 includes details on the inner workings of the storage for 8- and 16-bit images.

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

For images stored in one of the graphics file formats supported by MATLAB functions, however, this data representation is not always ideal. The number of pixels in such an image can be very large; for example, a 1000-by-1000 image has a million pixels. Since at least one array element represents each pixel , this image requires about 8 megabytes of memory if it is stored as class `double`.

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

Bit Depth

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

Supported Image Formats

MATLAB commands read, write, and display several types of graphics file formats for images. As with MATLAB generated images, once a graphics file format image is displayed, it becomes a Handle Graphics® image object. MATLAB supports the following graphics file formats, along with others:

- BMP (Microsoft® Windows® Bitmap)
- GIF (Graphics Interchange Files)
- HDF (Hierarchical Data Format)
- JPEG (Joint Photographic Experts Group)

- PCX (Paintbrush)
- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)

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

Functions for Reading, Writing, and Displaying Images

Images are essentially two-dimensional matrices, so many MATLAB functions can operate on and display images. The following table lists the most useful ones. The sections that follow describe these functions in more detail.

Function	Purpose	Function Group
<code>axis</code>	Plot axis scaling and appearance.	Display
<code>image</code>	Display image (create image object).	Display
<code>imagesc</code>	Scale data and display as image.	Display
<code>imread</code>	Read image from graphics file.	File I/O
<code>imwrite</code>	Write image to graphics file.	File I/O
<code>imfinfo</code>	Get image information from graphics file.	Utility
<code>ind2rgb</code>	Convert indexed image to RGB image.	Utility

Image Types

In this section...

- “Indexed Images” on page 6-5
- “Intensity Images” on page 6-7
- “RGB (Truecolor) Images” on page 6-8

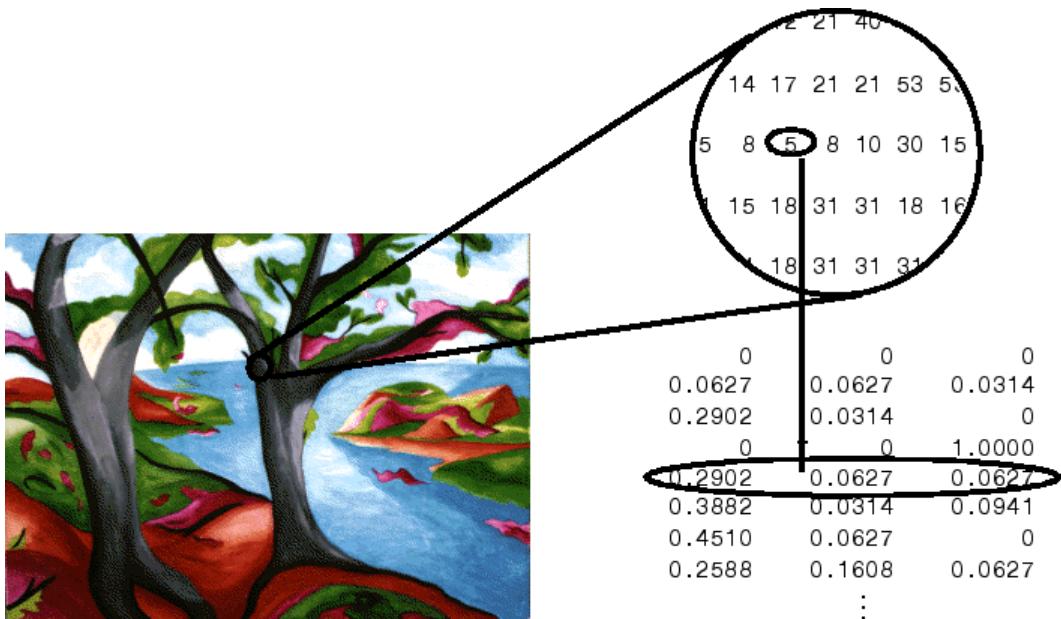
Indexed Images

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



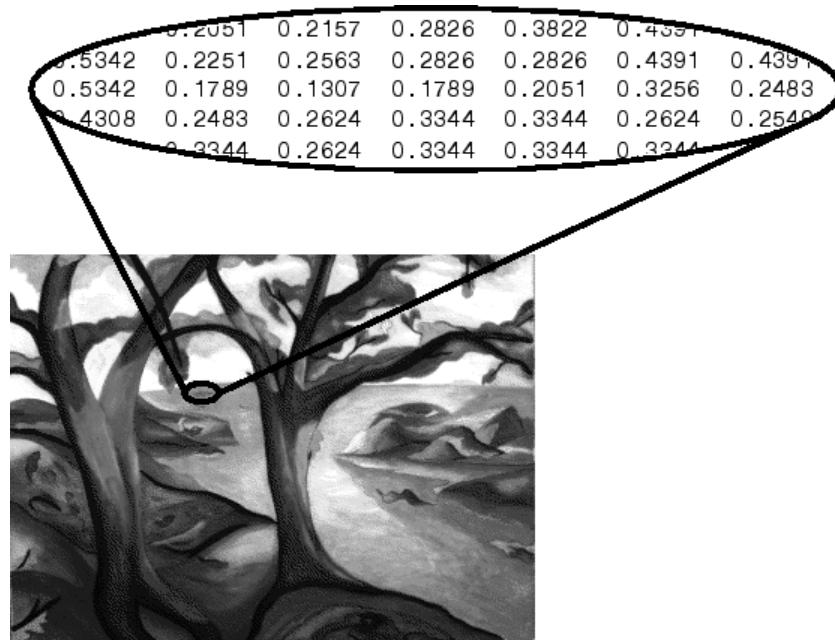
The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `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 preceding image, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Note: When using the painters renderer on the Windows platform, you should only use 256 colors when attempting to display an indexed image. Larger colormaps can lead to unexpected colors because the painters algorithm uses the Windows 256 color palette, which graphics drivers and graphics hardware are known to handle differently. To work around this issue, use the `Zbuffer` or `OpenGL` renderer, as appropriate.

Intensity Images

An intensity image is a data matrix, I , whose values represent intensities within some range. An intensity image is represented 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, a colormap is still used to display them. In essence, handles intensity images are treated 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 `imagesc` function 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

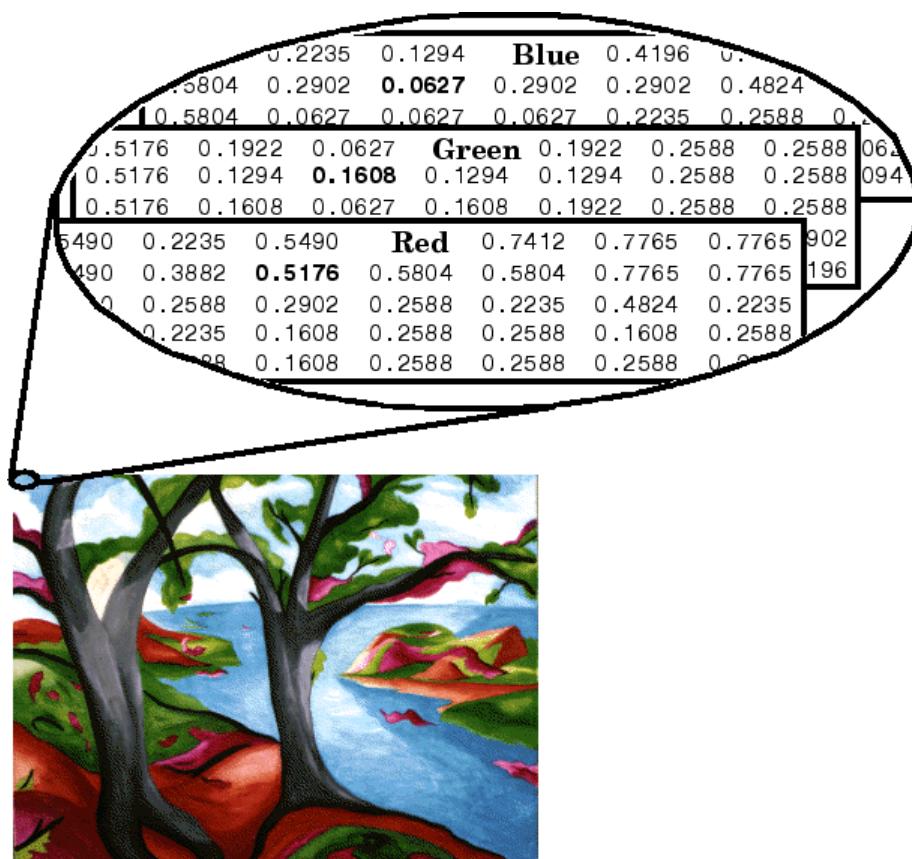
An RGB image, sometimes referred to as a *truecolor* image, is stored as an m -by- n -by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities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

An RGB MATLAB array can be of class `double`, `uint8`, or `uint16`. In an RGB array of class `double`, each color component is a value between 0 and 1. A pixel whose color components are $(0,0,0)$ is displayed as black, and a pixel whose color components are $(1,1,1)$ is displayed as white. The three color components for each pixel are stored along the third dimension of the data array. For example, the red, green, and blue color components of the pixel $(10,5)$ are stored in `RGB(10,5,1)`, `RGB(10,5,2)`, and `RGB(10,5,3)`, respectively.

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

```
image(RGB)
```

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



To determine the color of the pixel at (2,3), 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

8-Bit and 16-Bit Images

In this section...

- “Indexed Images” on page 6-10
- “Intensity Images” on page 6-11
- “RGB Images” on page 6-11
- “Mathematical Operations Support for uint8 and uint16” on page 6-12
- “Other 8-Bit and 16-Bit Array Support” on page 6-12
- “Converting an 8-Bit RGB Image to Grayscale” on page 6-13
- “Summary of Image Types and Numeric Classes” on page 6-16

Indexed Images

Double-precision (64-bit) floating-point numbers are the default MATLAB representation for numeric data. However, to reduce memory requirements for working with images, you can store images as 8-bit or 16-bit unsigned integers using the numeric classes `uint8` or `uint16`, respectively. An image whose data matrix has class `uint8` is called an 8-bit image; an image whose data matrix has class `uint16` is called a 16-bit image.

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

If the class of `X` is `uint8` or `uint16`, its values are offset by 1 before being used as colormap indices. The value 0 points to the first row of the colormap, the value 1 points to the second row, and so on. The `image` command automatically supplies the proper offset, so the display method is the same whether `X` is `double`, `uint8`, or `uint16`:

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

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

Intensity Images

The range of `double` image arrays is usually [0, 1], but the range of 8-bit intensity images is usually [0, 255] and the range of 16-bit intensity images is usually [0, 65535]. Use the following command to display an 8-bit intensity image with a grayscale colormap:

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

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

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

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

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

RGB Images

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

```
image(RGB);
```

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

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

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

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

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

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

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

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

Mathematical Operations Support for `uint8` and `uint16`

To use the following MATLAB functions with `uint8` and `uint16` data, first convert the data to type `double`:

- `conv2`
- `convn`
- `fft2`
- `fftn`

For example, if `X` is a `uint8` image, cast the data to type `double`:

```
fft(double(X))
```

In these cases, the output is always `double`.

The `sum` function returns results in the same type as its input, but provides an option to use double precision for calculations.

MATLAB Integer Mathematics

See “Arithmetic Operations on Integer Classes” for more information on how mathematical functions work with data types that are not doubles.

Most Image Processing Toolbox™ functions accept `uint8` and `uint16` input. If you plan to do sophisticated image processing on `uint8` or `uint16` data, consider including that toolbox in your MATLAB computing environment.

Other 8-Bit and 16-Bit Array Support

You can perform several other operations on `uint8` and `uint16` arrays, including:

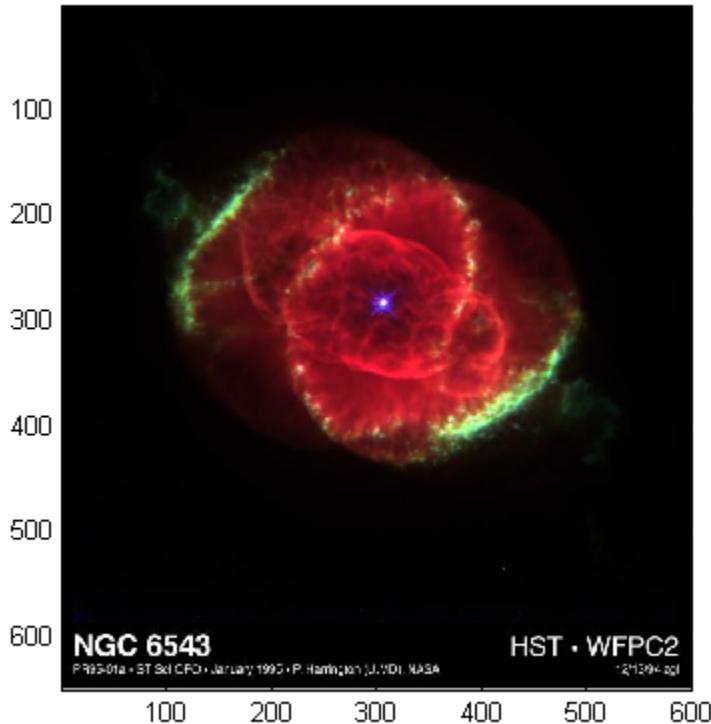
- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading `uint8` and `uint16` arrays in MAT-files using `save` and `load`. (Remember that if you are loading or saving a graphics file format image, you must use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

Converting an 8-Bit RGB Image to Grayscale

You can perform arithmetic operations on integer data, which enables you to convert image types without first converting the numeric class of the image data.

This example reads an 8-bit RGB image into a MATLAB variable and converts it to a grayscale image:

```
rgb_img = imread('ngc6543a.jpg'); % Load the image  
image(rgb_img) % Display the RGB image  
  
axis image;
```



Note: This image was created with the support of the Space Telescope Science Institute, operated by the Association of Universities for Research in Astronomy, Inc., from NASA contract NAS5-26555, and is reproduced with permission from AURA/STScI. Digital renditions of images produced by AURA/STScI are obtainable royalty-free. Credits: J.P. Harrington and K.J. Orkowsky (University of Maryland), and NASA.

Calculate the monochrome luminance by combining the RGB values according to the NTSC standard, which applies coefficients related to the eye's sensitivity to RGB colors:

```
I = .2989*rgb_img(:,:,1)...
+.5870*rgb_img(:,:,2)...
```

```
+.1140*rgb_img(:,:,3);
```

I is an intensity image with integer values ranging from a minimum of zero:

```
min(I(:))
ans =
0
```

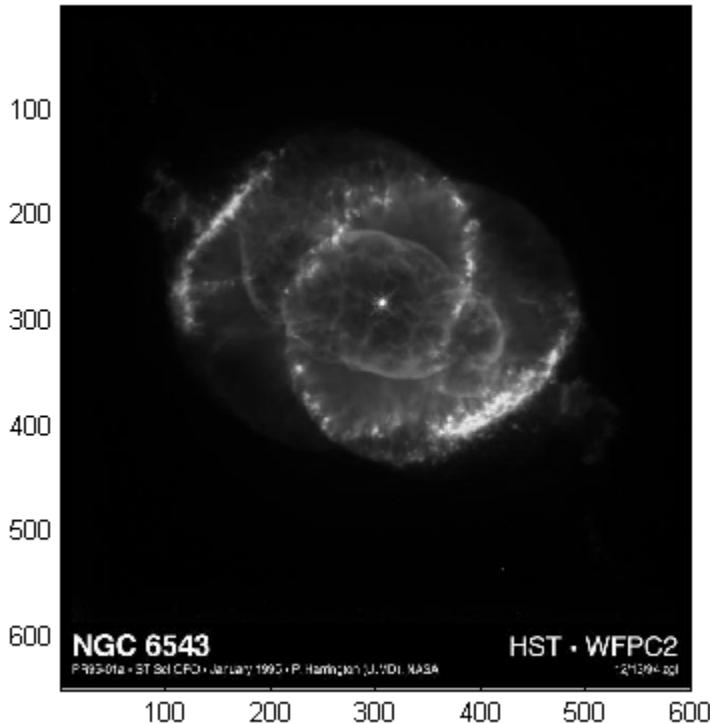
to a maximum of 255:

```
max(I(:))
ans =
255
```

To display the image, use a grayscale colormap with 256 values. This avoids the need to scale the data-to-color mapping, which is required if you use a colormap of a different size. Use the **imagesc** function in cases where the colormap does not contain one entry for each data value.

Now display the image in a new figure using the gray colormap:

```
figure; colormap(gray(256)); image(I);
axis image;
```



Related Information

Other colormaps with a range of colors that vary continuously from dark to light can produce usable images. For example, try `colormap(summer(256))` for a classic oscilloscope look. See `colormap` for more choices.

The `brighten` function enables you to increase or decrease the color intensities in a colormap to compensate for computer display differences or to enhance the visibility of faint or bright regions of the image (at the expense of the opposite end of the range).

Summary of Image Types and Numeric Classes

This table summarizes how data matrix elements are interpreted as pixel colors, depending on the image type and data class.

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

Read, Write, and Query Image Files

In this section...

- “Working with Image Formats” on page 6-18
- “Reading a Graphics Image” on page 6-19
- “Writing a Graphics Image” on page 6-19
- “Subsetting a Graphics Image (Cropping)” on page 6-20
- “Obtaining Information About Graphics Files” on page 6-21

Working with Image Formats

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

Call special MATLAB functions to read and write image data from graphics file formats:

- To read a graphics file format image use `imread`.
- To write a graphics file format image, use `imwrite`.
- To obtain information about the nature of a graphics file format image, use `imfinfo`.

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

Procedure	Functions to Use
Load or save a matrix as a MAT-file.	<code>load</code> <code>save</code>
Load or save graphics file format image, e.g., BMP, TIFF.	<code>imread</code> <code>imwrite</code>
Display any image loaded into the MATLAB workspace.	<code>image</code>

Procedure	Functions to Use
	<code>imagesc</code>
Utilities	<code>imfinfo</code> <code>ind2rgb</code>

Reading a Graphics Image

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

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

The following commands read the image `ngc6543a.jpg` into the workspace variable `RGB` and then displays the image using the `image` function:

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

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

```
load clown % An image that is included with MATLAB
imwrite(X,map,'clown.bmp')
```

create a BMP file containing the clown image.

Writing a Graphics Image

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

default behavior by specifying `uint16` as the data type for `imwrite`. The following example shows writing a 16-bit PNG file using `imwrite`.

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

Subsetting a Graphics Image (Cropping)

Sometimes you want to work with only a portion of an image file or you want to break it up into subsections. Specify the intrinsic coordinates of the rectangular subsection you want to work with and save it to a file from the command line. If you do not know the coordinates of the corner points of the subsection, choose them interactively, as the following example shows:

```
% Read RGB image from graphics file.
im = imread('street2.jpg');

% Display image with true aspect ratio
image(im); axis image

% Use ginput to select corner points of a rectangular
% region by pointing and clicking the mouse twice
p = ginput(2);

% Get the x and y corner coordinates as integers
sp(1) = min(floor(p(1)), floor(p(2))); %xmin
sp(2) = min(floor(p(3)), floor(p(4))); %ymin
sp(3) = max(ceil(p(1)), ceil(p(2))); %xmax
sp(4) = max(ceil(p(3)), ceil(p(4))); %ymax

% Index into the original image to create the new image
MM = im(sp(2):sp(4), sp(1): sp(3), :);

% Display the subsetted image with appropriate axis ratio
figure; image(MM); axis image

% Write image to graphics file.
imwrite(MM,'street2_cropped.tif')
```

If you know what the image corner coordinates should be, you can manually define `sp` in the preceding example rather than using `ginput`.

You can also display a “rubber band box” as you interact with the image to subset it. See the code example for `rbbox` for details. For further information, see the documentation for the `ginput` and `image` functions.

Obtaining Information About Graphics Files

The `imfinfo` function enables you to obtain information about graphics files in any of the standard formats listed earlier. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file, including the folder path if the file is not in the current folder
- File format
- Version number of the file format
- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

Displaying Graphics Images

In this section...

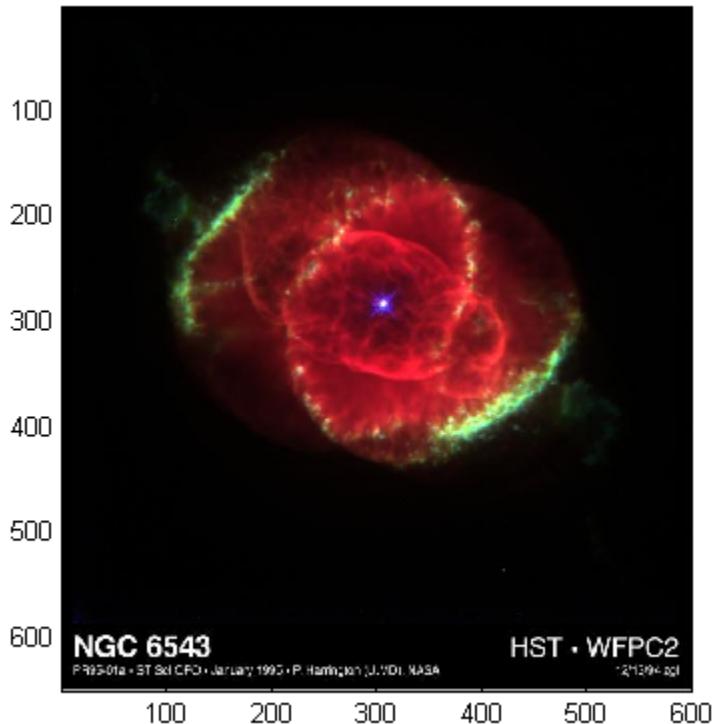
[“Image Types and Display Methods” on page 6-22](#)

[“Controlling Aspect Ratio and Display Size” on page 6-24](#)

Image Types and Display Methods

To display a graphics file image, use either `image` or `imagesc`. For example, read the image `ngc6543a.jpg` to a variable *RGB* and display the image using the `image` function. Change the axes aspect ratio to the true ratio using `axis` command.

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



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

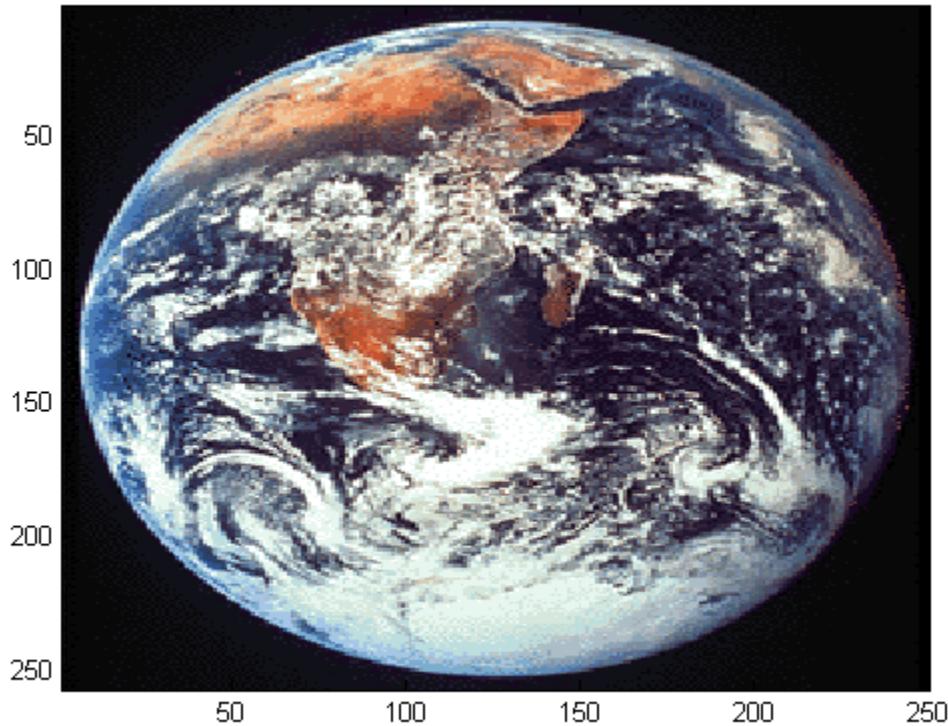
Image Type	Display Commands	Uses Colormap Colors
Indexed	<code>image(X); colormap(map)</code>	Yes
Intensity	<code>imagesc(I,[0 1]); colormap(gray)</code>	Yes
RGB (truecolor)	<code>image(RGB)</code>	No

Controlling Aspect Ratio and Display Size

The `image` function displays the image in a default-sized figure and axes. The image stretches or shrinks 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 `axis image` command.

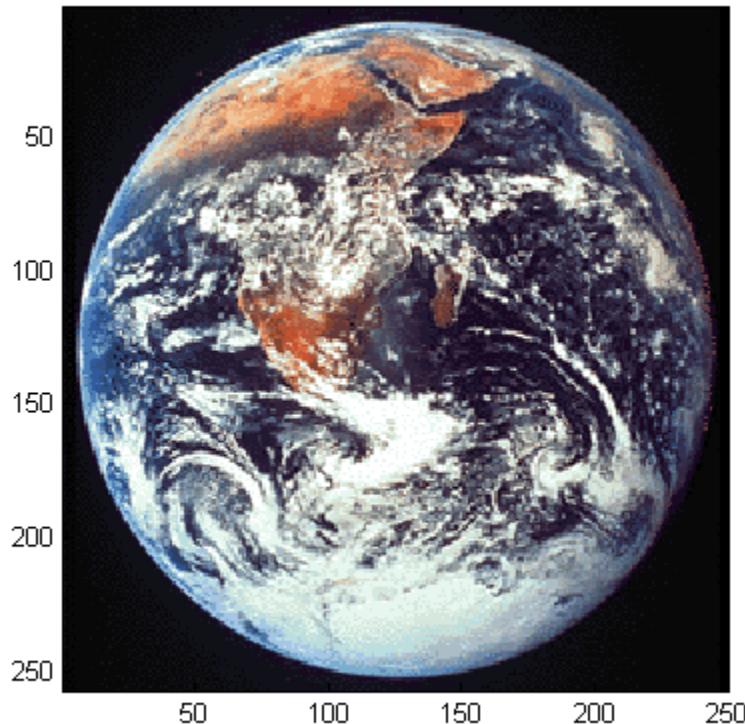
For example, these commands display the `earth` image using the default figure and axes positions:

```
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 `axis image` command 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 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 **figure** command positions the figure so that its lower left corner is at position (100,100) on the screen and so that its width and height match the image width and height. Setting the axes position to [0 0 1 1] in normalized units creates an axes that fills the figure. The resulting picture is shown.



The Image Object and Its Properties

In this section...

- “Image CData” on page 6-27
- “Image CDataMapping” on page 6-27
- “XData and YData” on page 6-28
- “Add Text to Image Data” on page 6-30
- “Additional Techniques for Fast Image Updating” on page 6-32

Image CData

Note: The `image` and `imagesc` commands 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`, and `YData`. These properties are discussed in this and the following sections. For detailed information about these and all the properties of the image object, see `image`.

The `CData` property of an image object contains the data array. In the following commands, `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 the image displays using colormap colors or as an RGB image. If the `CData` array is two-dimensional, the image is either an indexed image or an intensity image; in either case, the image is displayed using colormap colors. If, on the other hand, the `CData` array is m -by- n -by-3, it displays as a truecolor image, ignoring the colormap colors.

Image CDataMapping

The `CDataMapping` property controls whether an image is `indexed` or `intensity`. To display an indexed image set the `CDataMapping` property to '`direct`', so that

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 axes **CLim** property controls the scale factors. The **imagesc** function creates an image object whose **CDataMapping** property is set to 'scaled', and it 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:

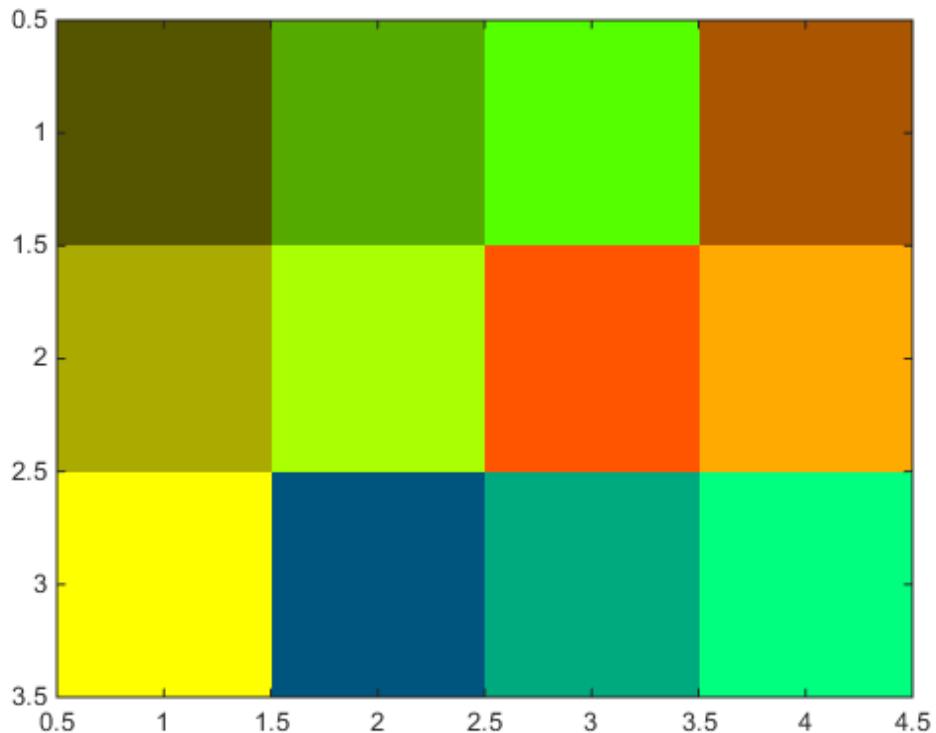
- The left column of the image has an x -coordinate of 1.
- The right column of the image has an x -coordinate of n .
- The top row of the image has a y -coordinate of 1.
- The bottom row of the image has a y -coordinate of m .

Coordinate System for Images

Use Default Coordinate System

Display an image using the default coordinate system. Use colors from the `colormap` map.

```
C = [1 2 3 4; 5 6 7 8; 9 10 11 12];  
im = image(C);  
colormap(colormaps)
```

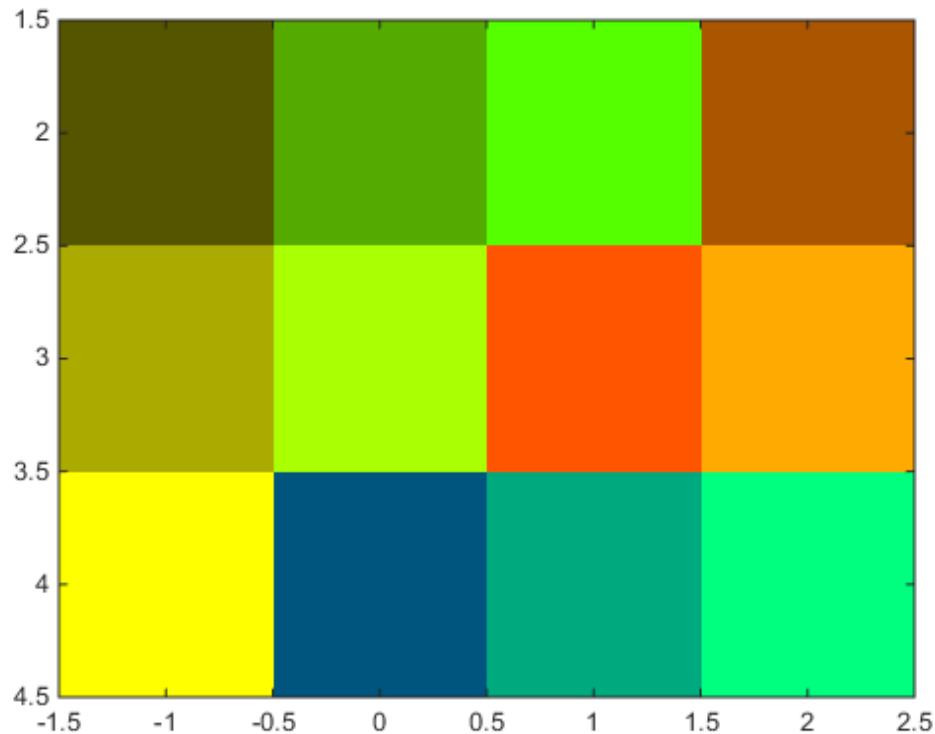


Specify Coordinate System

Display an image and specify the coordinate system. Use colors from the `colormap` map.

```
C = [1 2 3 4; 5 6 7 8; 9 10 11 12];
```

```
x = [-1 2];  
y = [2 4];  
figure  
image(x,y,C)  
colormap(colormcube)
```



Add Text to Image Data

This example shows how to use array indexing to rasterize text strings into an existing image.

Draw the text in an axes using the `text` function. Then, capture the text from the screen using `getframe` and close the figure.

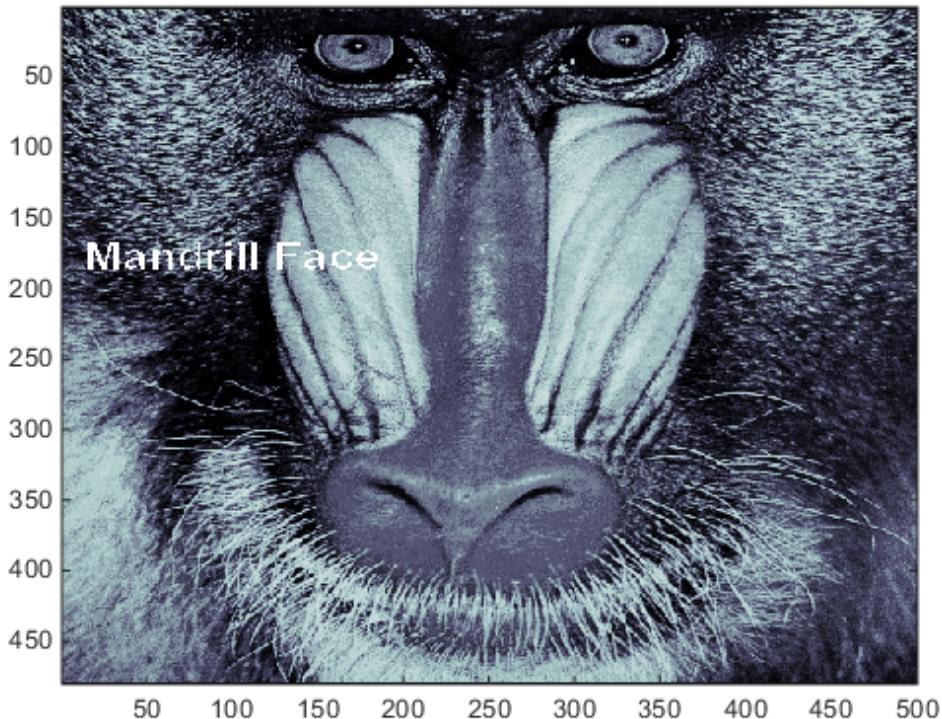
```
fig = figure;
t = text(.05,.1,'Mandrill Face','FontSize',20,'FontWeight','bold');
F = getframe(gca,[10 10 200 200]);
close(fig)
```

Select any plane of the resulting RGB image returned by `getframe`. Find the pixels that are black (black is 0) and convert their subscripts to indexes using `sub2ind`. Use these subscripts to "paint" the text into the image contained in the `mandrill` MAT-file. Use the size of that image, plus the row and column locations of the text to determine the locations in the new image. Index into new image, replacing pixels.

```
c = F.cdata(:,:,1);
[i,j] = find(c==0);
load mandrill
ind = sub2ind(size(X),i,j);
X(ind) = uint8(255);
```

Display the new image using the bone colormap.

```
imagesc(X)
colormap bone
```



Additional Techniques for Fast Image Updating

To increase the rate at which the **CData** property of an image object updates, optimize **CData** and set some related figure and axes properties:

- Use the smallest data type possible. Using a **uint8** data type for your image will be faster than using a **double** data type.

Part of the process of setting the image's **CData** property includes copying the matrix for the image's use. The overall size of the matrix is dependent on the size of its individual elements. Using smaller individual elements (i.e., a smaller data type) decreases matrix size, and reduces the amount of time needed to copy the matrix.

- Use the smallest acceptable matrix.

If the speed at which the image is displayed is your highest priority, you may need to compromise on the size and quality of the image. Again, decreasing the size reduces the time needed to copy the matrix.

- Make the axes exactly the same size (in pixels) as the **CData** matrix.

Maintaining a one-to-one correspondence between the data and the onscreen pixels eliminates the need for interpolation. For example:

```
set(gca,'Units','pixels')
pos = get(gca,'Position')
width = pos(3);
height = pos(4);
```

When the size of your **CData** exactly equals [width height], each element of the array corresponds directly to a pixel. Otherwise, the values in the **CData** array must be interpolated so the image fits the axes at their current size.

- Set the limit mode properties (**XLimMode** and **YLimMode**) of your axes to **manual**.

If they are set to **auto**, then every time an object (such as an image, line, patch, etc.) changes some aspect of its data, the axes must recalculate its related properties. For example, if you specify

```
image(firstimage);
set(gca, 'xlimmode','manual',...
'yliimmode','manual',...
'zliimmode','manual',...
'climmode','manual',...
'alimmode','manual');
```

the axes do not recalculate any of the limit values before redrawing the image.

- Consider using a **movie** object if the main point of your task is to simply display a series of images onscreen.

The MATLAB **movie** object utilizes underlying system graphics resources directly, instead of executing MATLAB object code. This is faster than repeatedly setting an image's **CData** property, as described earlier.

Printing Images

When you set the axes **Position** to [0 0 1 1] so that it fills the entire figure, the aspect ratio is not preserved when you print because MATLAB printing software 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(groot, 'defaultFigurePaperPositionMode', 'auto')
```

Printed images may not always be the same size as they are on your monitor. The size depends on accurately specifying the numbers of pixels per inch that you monitor is displaying.

To specify the pixels-per-inch on your display, do the following (in Microsoft Windows):

- 1** Go into your Display Properties by right-clicking on an empty space on your desktop and choose **Properties**.
- 2** Click the **Settings** pane.
- 3** Click the **Advanced** button and choose the General pane.
- 4** Switch DPI setting to Custom setting and hold a real ruler up to the picture of the ruler on the screen and drag until they match.

Until you do this, neither Windows software nor any other can determine how big images on the screen are, and printed images cannot match the size.

On the Macintosh platform, pixels per inch is hard-coded to 72.

Convert Image Graphic or Data Type

Converting between data types changes the interpretation of the image data. If you want the resulting array to be interpreted properly as image data, rescale or offset the data when you convert it. (See the earlier sections “Image Types” on page 6-5 and “Indexed Images” on page 6-10 for more information about offsets.)

For certain operations, it is helpful to convert an image to a different image type. For example, to filter a color image that is stored as an indexed image, first convert it to RGB format. To do this efficiently, use the `ind2rgb` function. When you apply the filter to the RGB image, the intensity values in the image are filtered, as is appropriate. If you attempt to filter the indexed image, the filter is applied 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, to convert a grayscale image to RGB, 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 appears as shades of gray.

Changing the graphics format of an image, perhaps for compatibility with another software product, is very straightforward. For example, to convert an image from a BMP to a PNG, load the BMP using `imread`, set the data type to `uint8`, `uint16`, or `double`, and then save the image using `imwrite`, with ‘PNG’ specified as your target format. See `imread` and `imwrite` for the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file.

Printing and Saving

- “Overview of Printing and Exporting” on page 7-2
- “Bitmap vs. Vector Formats” on page 7-8
- “How to Print or Export” on page 7-10
- “Printing and Exporting Use Cases” on page 7-30
- “Change Figure Settings” on page 7-37
- “Troubleshooting” on page 7-65
- “Saving Figures” on page 7-75

Overview of Printing and Exporting

In this section...

- “Print and Export Operations” on page 7-2
- “Graphical User Interfaces” on page 7-2
- “Command Line Interface” on page 7-3
- “Specifying Parameters and Options” on page 7-4
- “Default Settings and How to Change Them” on page 7-5

Print and Export Operations

There are four basic operations that you can perform in printing or transferring figures you've created with MATLAB graphics to specific file formats for other applications to use.

Operation	Description
Print	Send a figure from the screen directly to the printer.
Print to File	Write a figure to a PostScript® file to be printed later.
Export to File	Export a figure in graphics format to a file, so that you can import it into an application.
Export to Clipboard	Copy a figure to the Microsoft Windows clipboard, so that you can paste it into an application.

Graphical User Interfaces

In addition to typing MATLAB commands, you can use interactive tools for either Microsoft Windows or UNIX® to print and export graphics. The table below lists the GUIs available for doing this and explains how to open them from figure windows.

Dialog Box	How to Open	Description
Print (Windows and UNIX)	File > Print or printdlg function	Send figure to the printer, select the printer, print to file, and several other options

Dialog Box	How to Open	Description
Print Preview	File > Print Preview or <code>printpreview</code> function	View and adjust the final output
Export	File > Export	Export the figure in graphics format to a file
Copy Options	Edit > Copy Options	Set format, figure size, and background color for Copy to Clipboard
Figure Copy Template	File > Preferences	Change text, line, axes, and UI control properties

You can open the Print and Print Preview dialog boxes from a MATLAB file or from the command line with the `printdlg` and `printpreview` functions.

Command Line Interface

You can print a MATLAB figure from the command line or from a MATLAB file. Use the `print` function to specify the output format and start the print or export operation.

Note: Printed output from MATLAB commands and Print Previews of it are not guaranteed to duplicate the look of figures on your display screen in every detail. Many factors, including the complexity of the figure, available fonts, and whether a native printer driver or a MATLAB built-in driver is used, affect the final output and can cause printed output to differ from what you see on your screen.

Printing and Exporting with `print`

The `print` function performs any of the four actions shown in the table below. You control what action is taken, depending on the presence or absence of certain arguments.

Action	Print Command
Print a figure to a printer	<code>print</code>
Print a figure to a file for later printing	<code>print filename</code>
Copy a figure in graphics format to the clipboard on Microsoft Windows systems	<code>print -dfileformat</code>

Action	Print Command
Export a figure to a graphics format file that you can later import into an application	<code>print -dfileformat filename</code>

You can also include optional arguments with the `print` command. For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, use

```
print -f2 -r600 -depsc spline2d
```

The functional form of this command is

```
print('-f2', '-r600', '-depsc', 'spline2d');
```

Printing on UNIX Platforms without a Display

If you run with the PostScript `-nodisplay` startup option, or run without the `DISPLAY` environment variable set, you can use most `print` options that apply to the UNIX platform, but some restrictions apply. For example, in `nodisplay` mode uicontrols do not print; thus you cannot print a GUI if you run in this mode.

See “Printing and Exporting Without a Display” in the documentation for the `print` function for details.

Specifying Parameters and Options

The table below lists parameters you can modify for the figure to be printed or exported. To change one of these parameters, use the Print Preview or the UNIX Print dialog box, or use the `set` or `print` function.

See “Change Figure Settings” on page 7-37 for more detailed instructions.

Parameter	Description
Figure size	Set size of the figure on printed page
Figure position	Set position of figure on printed page
Paper size	Select printer paper, specified by dimension or type
Paper orientation	Specify way figure is oriented on page

Parameter	Description
Position mode	Specify figure position yourself or let it be determined automatically
Graphics format	Select format for exported data (e.g., EPS, JPEG)
Resolution	Specify how finely your figure is to be sampled
Renderer	Select method (algorithm) for drawing graphics
Renderer mode	Specify the renderer yourself or automatically determine which renderer to use based on the figure's contents
Axes tick marks	Keep axes tick marks and limits as shown or automatically adjust them depending on figure size
Background color	Keep background color as shown on screen or force it to white
Line and text color	Keep line and text objects as shown on screen or print them in black and white
UI controls	Show or hide all user interface controls in figure
Bounding box	Leave space between outermost objects in plot and edges of its background area
CMYK	Automatically convert RGB values to CMYK values
Character set encoding	Select character set for PostScript printers

Default Settings and How to Change Them

If you have not changed the default print and export settings, MATLAB prints or exports the figure as follows:

- 8-by-6 inches with no window frame
- Centered, in portrait format, on 8.5-by-11 inch paper if available
- Using white background color for the figure and axes
- Scaling ticks and limits of the axes to accommodate the printed size

Setting Defaults for a Figure

In general, to change the property settings for a specific figure, follow the instructions given in the section “Change Figure Settings” on page 7-37.

Any settings you change with the Print Preview and Print dialog boxes or with the `set` function are saved with the figure and affect each printing of the figure until you change the settings again.

The settings you change with the **Figure Copy Template Preferences** and **Copy Options Preferences** panels alter the figure as it is displayed on the screen.

Setting Defaults for the Session

You can set the session defaults for figure properties. Set the session default for a property using the syntax

```
set(groot,'defaultFigurepropertyName','value')
```

where `propertyName` is one of the named figure properties. This example sets the paper orientation for all subsequent print operations in the current MATLAB session.

```
set(groot,'defaultFigurePaperOrientation','landscape')
```

The Figure Properties properties page contains a complete list of the properties.

To see what default properties you can set that will be applied to all subsequent figures in the same MATLAB session, type

```
set(groot,'default')
```

To see their current settings, type

```
get(groot,'default')
```

Setting Defaults Across Sessions

You can set the session-to-session defaults for figure properties, the print driver, and the `print` function.

Print Device and Print Command

Set the default print driver and the default print command in your `printopt.m` file. This file contains instructions for changing these settings and for displaying the current defaults. Open `printopt.m` in your editor by typing the command

```
edit printopt
```

Scroll down about 40 lines until you come to this comment line:

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

Add your changes after that line. For example, to change the default driver, first find the line that sets `dev`, and then replace the text string with an appropriate value. So, to set the default driver to HP® LaserJet III, modify the line to read

```
dev = '-dIjet3';
```

For the full list of values for `dev`, see the Drivers section of the `print` reference page.

Note If you set `dev` to be a graphics format, such as `-djpeg`, the figure is exported to that type of file rather than being printed.

Figure Properties

Set the session-to-session default for a property by including commands like the following in your `startup.m` file:

```
set(groot, 'defaultFigurePropertyName', 'value')
```

where `PropertyName` is one of the named figure properties. For example,

```
set(groot, 'defaultFigureInvertHardcopy', 'off')
```

keeps the figure background in the screen color.

This is the same command you use to change a session default, except by adding it to your `startup.m` file, it executes automatically every time you launch MATLAB.

Options you specify in arguments to the `print` command override properties set using MATLAB commands or the Print Preview dialog box, which in turn override any MATLAB default settings specified in `printopt.m` or `startup.m`.

Note: To export to vector graphics formats, ensure the figure `RendererMode` property is set to `auto`.

Bitmap vs. Vector Formats

In this section...

- “Choosing a Format” on page 7-8
- “How Renderer Affects Format” on page 7-9
- “Controlling Graphics Output” on page 7-9

Choosing a Format

There are two kinds of graphics file formats that you can create from MATLAB figures:

- Bitmaps images — Screen pixels captured in a file.
- Vector graphics — Commands to drawn individual objects.

The kind of file you use depends on how you want to present the exported graphics and if you want to modify the graphics in an external program.

Bitmap Format

- Widely used by Web browsers and other applications that display graphics.
- Resolution and output dimensions determine size
- Scaling can reduce quality
- Cannot modify individual graphics objects (such as lines and text) from other graphics applications

Vector Format

- Graphics applications can modify graphics object characteristics (such as color, line style, and text font)
- Scalable size
- Slower to render and can result in larger files
- Might not produce correct 3-D arrangement of objects in certain cases

For a list of supported formats, see [Graphics File Formats in the print command page](#).

How Renderer Affects Format

The *renderer* is the part of MATLAB that determines how to display graphics on the screen, send graphics to a printer, or write graphics to a file. MATLAB uses two rendering methods:

- Painter's — Used for vector format output (PDF, SVG, PostScript, EPS, metafile) and whenever you specify Painter's as the screen renderer by setting the figure **Renderer** property.
- OpenGL® — Produces a bitmap even with vector formats. OpenGL is the default screen renderer and is used for printing except when you specify a vector format.

MATLAB attempts to use the Painter's renderer for exporting to vector formats if the figure **RendererMode** property is set to **auto**. However, MATLAB does not use the Painter's renderer in cases in which you:

- Explicitly specify OpenGL as the screen renderer, which sets the figure **RendererMode** property to **manual**.
- Override the figure **Renderer** property from the **Export Setup** dialog.
- Specify a renderer with the **print** command.

Controlling Graphics Output

In most cases, MATLAB uses the appropriate renderer to produce the output format that you request.

However, if a figure contains a very large number of graphics objects that need to be arranged by 3-D depth, MATLAB might use OpenGL and create a bitmap in the output format. In these cases, a vector format like EPS or PDF actually contains a bitmap, which might limit the extend to which you can edit the image in other applications.

If you want to ensure that your output format is a true vector graphic file, then specify the renderer with the **Export Setup** dialog or the **print** command.

For example, this command always generates a vector format:

```
print -deps -painters myVectorFile
```

In some cases, exporting a file with the **-painters** option can cause longer rendering times and, in rare cases, might produce graphics that are not accurate with regard to 3-D arrangement of the graphics objects.

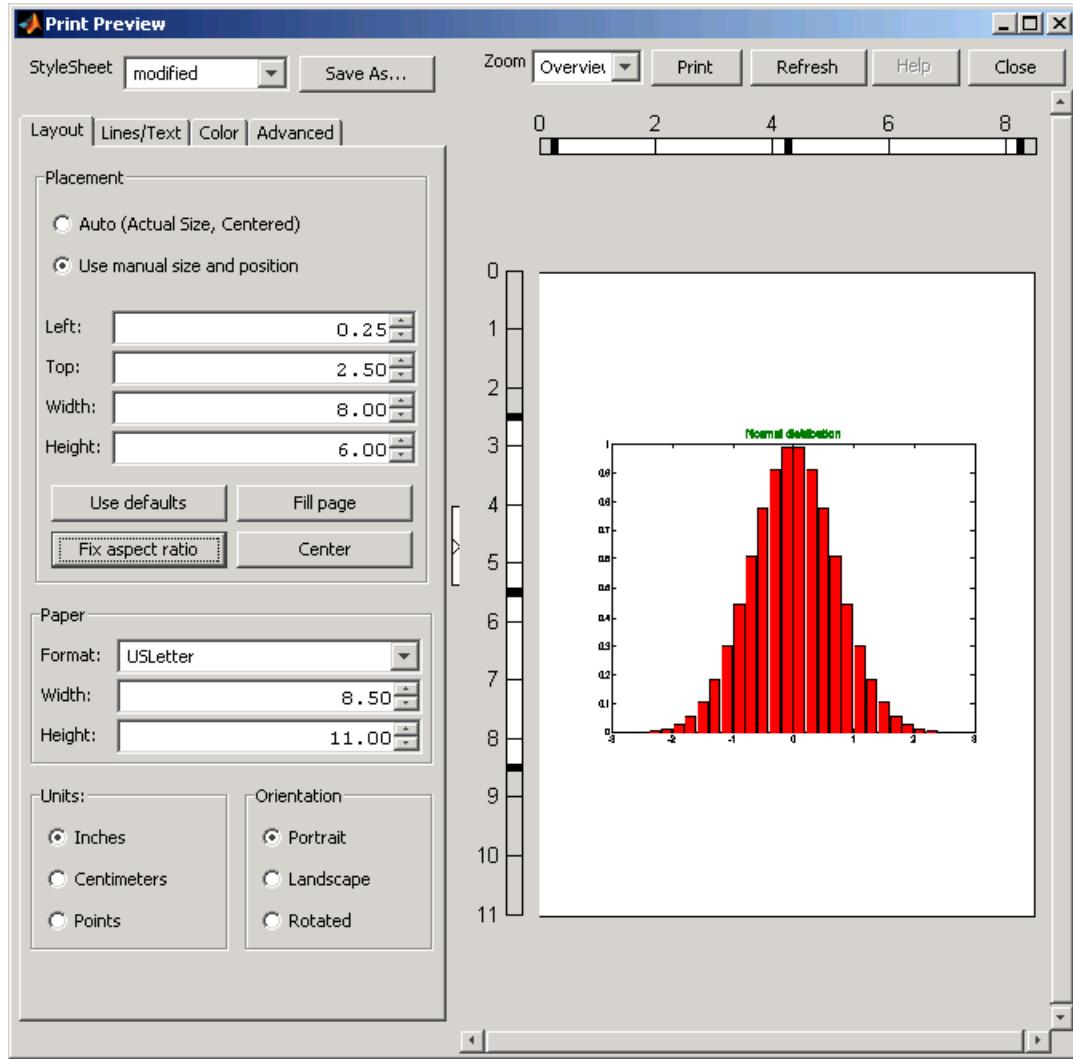
How to Print or Export

In this section...

- “Using Print Preview” on page 7-10
- “Printing a Figure” on page 7-13
- “Printing to a File” on page 7-15
- “Exporting to a File” on page 7-17
- “Exporting to the Windows or Macintosh Clipboard” on page 7-25

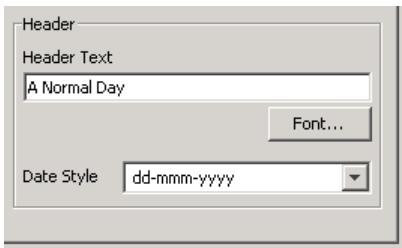
Using Print Preview

Before you print or export a figure, preview the image by selecting **Print Preview** from the figure window's **File** menu. If necessary, you can use the **set** function to adjust specific characteristics of the printed or exported figure. Adjustments that you make in the Print Preview dialog also set figure properties; these changes can affect the output you get should you print the figure later with the **print** command. See “Change Figure Settings” on page 7-37 for details.

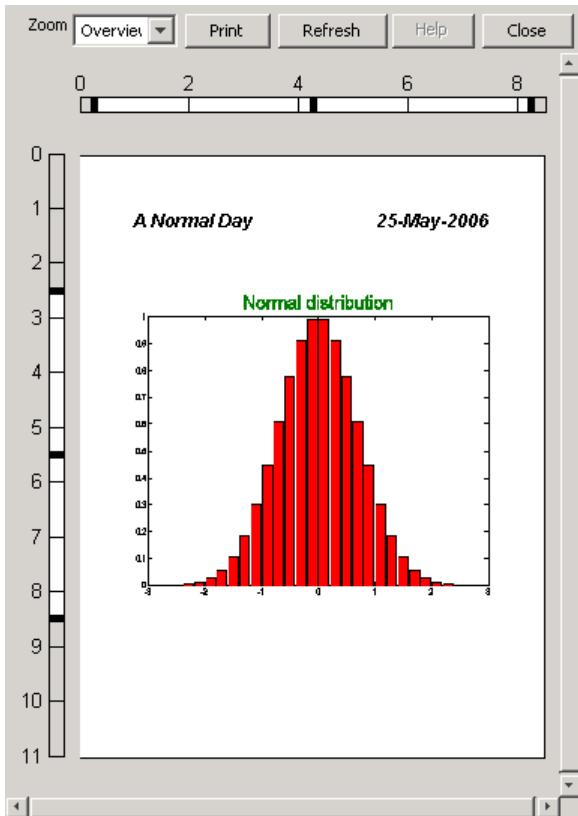


Adding a Header to the Printed Page

You can add a header to the page you are about to print by clicking the **Lines/Text** tab at the top of the Print Preview dialog box. At the bottom of that panel are the **Header** controls, as shown here:



The print header includes any text you want to appear at the top of the printed page. It can also include the current date. In the **Header Text** edit box, enter the text of the header. Under **Date Type**, select from a number of possible formats with which to display the current date and/or time. The default is to include no date. Click the **Font** button to change the font, font style, font size, or script type for the header text and date format. If you don't see the header as you specified it, click the **Refresh** button over the preview pane. A page containing a header plus date in bold italics is shown in the preview below:



Click **Print** to open the standard print dialog box to print the page. Click **Close** to close the dialog box and apply these settings to your figure.

Note: The print preview header appears only when printing directly to a printer, not when printing to a file.

Printing a Figure

This section tells you how to print your figure to a printer:

- “Printing with the Print GUI on Microsoft Windows” on page 7-14
- “Printing with the Print GUI on UNIX Platforms” on page 7-14

Printing with the Print GUI on Microsoft Windows

MATLAB printing on Windows platforms uses the standard Windows Print dialog box, which most Windows software products share. To open the Windows Print dialog box, select **Print** from the figure window's **File** menu or click the **Print** button in the Print Preview dialog box.

- To print a figure, first select a printer from the list box, then click **OK**.
- To save it to a file, click the **Print to file** check box, click **OK**, and when the Print to File window appears, enter the filename you want to save the figure to. The file is written to your current working folder.

Settings you can change in the Windows Print dialog box are as follows:

Properties

To make changes to settings specific to a printer, click the **Properties** button. This opens the Windows Document Properties window.

Print range

You can only select **All** in this panel. The selection does not affect your printed output.

Copies

Enter the number of copies you want to print.

You can also open the Print dialog programmatically via the `printdlg` function.

Printing with the Print GUI on UNIX Platforms

MATLAB printing on UNIX platforms has a Print dialog box containing three tabs. To open the Print dialog box, select **Print** from the figure window's **File** menu. It opens showing the **General** tab's contents.

To print a figure, click the **Name** button under **Print Service** and select a printer from the list box.

Note: Printers accessed from the Print dialog are assumed to be PostScript-enabled. If you want to print to a non-PostScript device, you will need to use **File** > **Save As** and specify the **Save as type** or issue a `print` command specifying the appropriate driver with the `-d` flag.

Set paper characteristics and margins with the **Page Setup** tab on the Print dialog. You might want to use the Print Preview dialog instead, however, as it allows you to do the same things and gives you visual feedback at the same time. For details, see “Using Print Preview” on page 7-10.

The **Appearance** options include duplex and tumble printing, whether a banner page should precede the printed page, whether to print in color, and what quality of printing to use. You can also use **Print Preview** to control color.

Printing in Color

Depending on the capabilities of the printer you are using, you can print in black and white, grayscale, or color by selecting the appropriate button in the **Color Scale** panel of the Print Preview **Color** tab. You can also choose a background color that is the same or different from the figure's color.

Figure Size and Position on Printed Page

If you want the printed plot to have the same size as it does on your screen, select **Auto (Actual Size, Centered)** on the **Layout** tab. If you want the printed output to have a specific size, select **Use manual size and position**.

See “Setting the Figure Size and Position” on page 7-41 for more information.

Axes Limits and Ticks

To force the same number of ticks and the same limit values for the axes as are used on the screen to be printed, select **Keep screen limits and ticks** on the **Advanced** tab of the Print Preview dialog box. To automatically scale the limits and ticks of the axes based on the size of the printed figure, select **Recompute limits and ticks**.

See “Setting the Axes Ticks and Limits” on page 7-52 for more information.

Printing to a File

Instead of sending your figure to the printer right now, you have the option of “printing” it to a file, and then sending the file to the printer later on. You can also append additional figures to the same file using the **print** command.

When you print to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current folder, the filename must have fewer than 126 characters, because MATLAB places ‘./’ or ‘.\’ at the beginning of the filename when referring to it.

This section tells you how to save your figure to a file:

- “Printing to a File with the Print GUI on Windows Platforms” on page 7-16
- “Printing to a File with the Print GUI on UNIX Platforms” on page 7-16
- “Printing to a File Using MATLAB Commands” on page 7-16

Printing to a File with the Print GUI on Windows Platforms

- 1 To open the Print dialog box, select **Print** from the figure window's **File** menu.
- 2 Select the check box labeled **Print to file**, and click the **OK** button.
- 3 The **Print to file** dialog box appears, allowing you to specify the output folder and filename.

Printing to a File with the Print GUI on UNIX Platforms

- 1 To open the Print dialog box, select **Print** from the figure window's **File** menu.
- 2 Select the radio button labeled **File**, and either fill in or browse for the folder and filename.

Printing to a File Using MATLAB Commands

To print the figure to a PostScript file, type

```
print filename
```

If you don't specify the filename extension, MATLAB uses an extension that is appropriate for the print driver being used.

You can also include an **-options** argument when printing to a file. For example, to append the current figure to an existing file, type

```
print -append filename
```

The only way to append to a file is by using the **print** function. There is no dialog box that enables you to do this.

Note If you print a figure to a file, the file can only be printed and cannot be imported into another application. If you want to create a figure file that you can import into an application, see the next section, “Exporting to a File”

Appending Additional Figures to a File

Once you have printed one figure to a PostScript file, you can append other figures to that same file using the `-append` option of the `print` function. You can only append using the `print` function and full PostScript (the `-dps`, `-dpSC`, `-dps2`, and `-dpSC2` drivers).

Exporting to a File

Export a figure in a graphics format to a file if you want to import it into another application, such as a word processor. You can export to a file from the Windows or UNIX Export Setup dialog box or from the command line.

This section tells you how to export your figure to a file:

- “Using the Export Setup GUI” on page 7-17
- “Exporting Using MATLAB Commands” on page 7-23
- “Importing MATLAB Graphics into Other Applications” on page 7-24

Note: When you export to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current folder, the filename must have fewer than 126 characters, because MATLAB places ‘./’ or ‘.\’ at the beginning of the filename when referring to it.

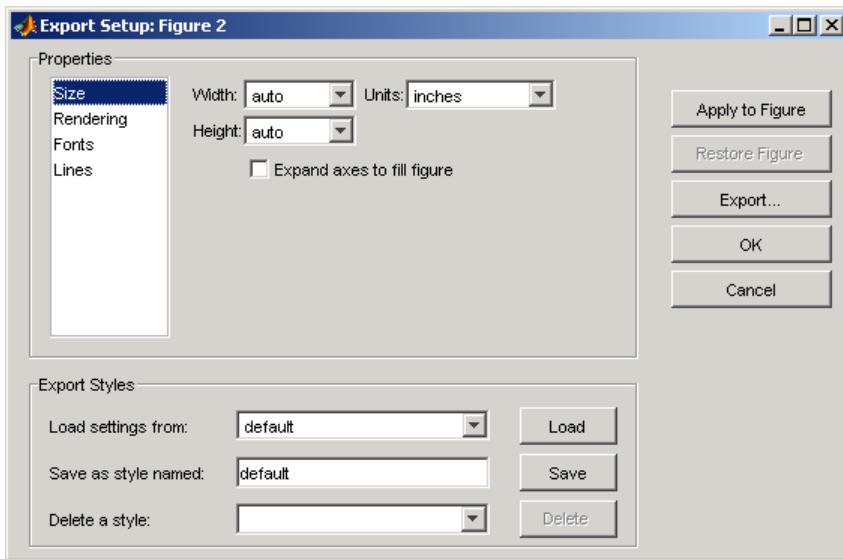
Using the Export Setup GUI

The Export Setup GUI appears when you select **Export Setup** from the **File** menu of a figure window. This GUI has four dialog boxes that enable you to adjust the size, rendering, font, and line appearance of your figure prior to exporting it. You select each of these dialog boxes by clicking **Size**, **Rendering**, **Fonts**, or **Lines** from the **Properties** list. For a description of each dialog box, see

- “Adjusting the Figure Size” on page 7-17
- “Changing the Rendering” on page 7-18
- “Changing Font Characteristics” on page 7-20
- “Changing Line Characteristics” on page 7-22

Adjusting the Figure Size

Click **Size** in the Export Setup dialog box to display this dialog box.

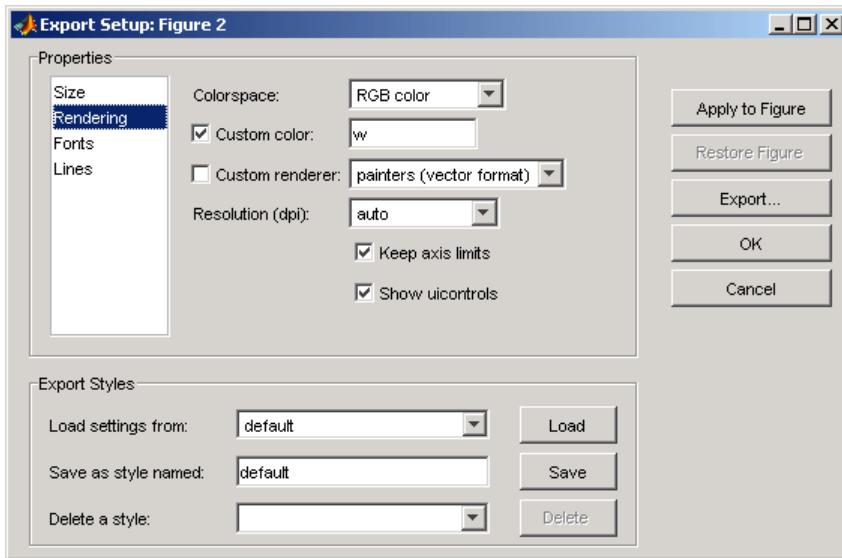


The Size dialog box modifies the size of the figure as it will appear when imported from the export file into your application. If you leave the **Width** and **Height** settings on **auto**, the figure remains the same size as it appears on your screen. You can change the size of the figure by entering new values in the **Width** and **Height** text boxes and then clicking **Apply to Figure**. To go back to the original settings, click **Restore Figure**.

To save any settings that you change, or to load settings that you used earlier, see “Saving and Loading Settings” on page 7-23.

Changing the Rendering

Click **Rendering** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Colorspace

Use the drop-down list to select a colorspace:

- **Black and white** — Lines and text in black, all other objects in grayscale
- **Grayscale** — All objects in grayscale
- **RGB color** — Color using the RGB colorspace
- **CMYK color** — Color using the CMYK colorspace

Custom Color

Click the check box and enter a color to be used for the figure background. Valid entries are

- **white, yellow, magenta, red, cyan, green, blue, or black**
- Abbreviated name for the same colors — **w, y, m, r, c, g, b, k**
- Three-element RGB value — See the help for **colorspec** for valid values. Examples: **[1 0 1]** is magenta. **[0 .5 .4]** is a dark shade of green.

Custom Renderer

Click the check box and select a renderer from the drop-down list:

- **painters (vector format)** — Select **painters** when exporting to a vector format.
- **OpenGL (bitmap format)** — Select **OpenGL** when exporting to a bit-mapped format.

Resolution

You can select one of the following from the drop-down list:

- **Screen** — The same resolution as used on your screen display
- A specific numeric setting — 150, 300, or 600 dpi
- **auto** — UNIX selects a suitable setting

Keep axis limits

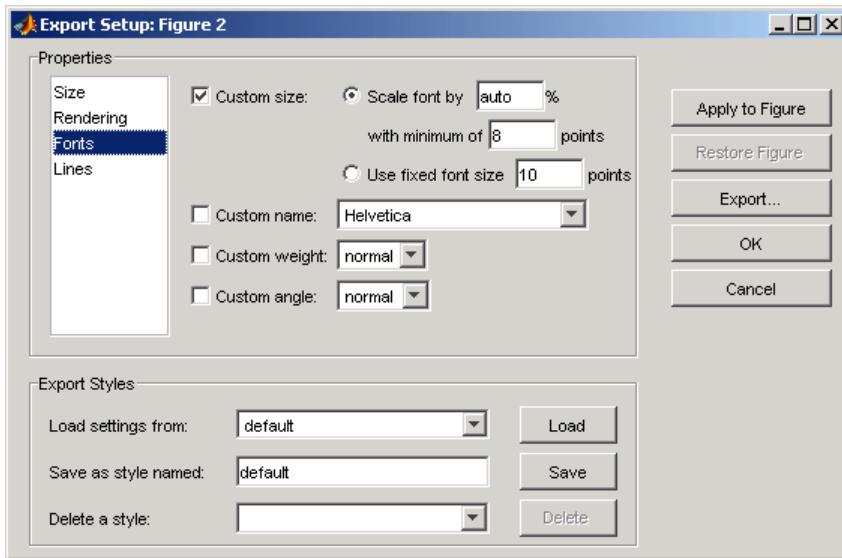
Click the check box to keep axis tick marks and limits as shown. If unchecked, automatically adjust depending on figure size.

Show uicontrols

Click the check box to show all user interface controls in the figure. If unchecked, hide user interface controls.

Changing Font Characteristics

Click **Fonts** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom Size

Click the check box and use the radio buttons to select a relative or absolute font size for text in the figure.

- **Scale font by N %** — Increases or decreases the size of all fonts by a relative amount, N percent. Enter the word **auto** to automatically select the appropriate font size.
- **With minimum of N points** — You can specify a minimum font size when scaling the font by a percentage.
- **Use fixed font size N points** — Sets the size of all fonts to an absolute value, N points.

Custom Name

Click the check box and use the drop-down list to select a font name from those offered in the drop-down list.

Custom Weight

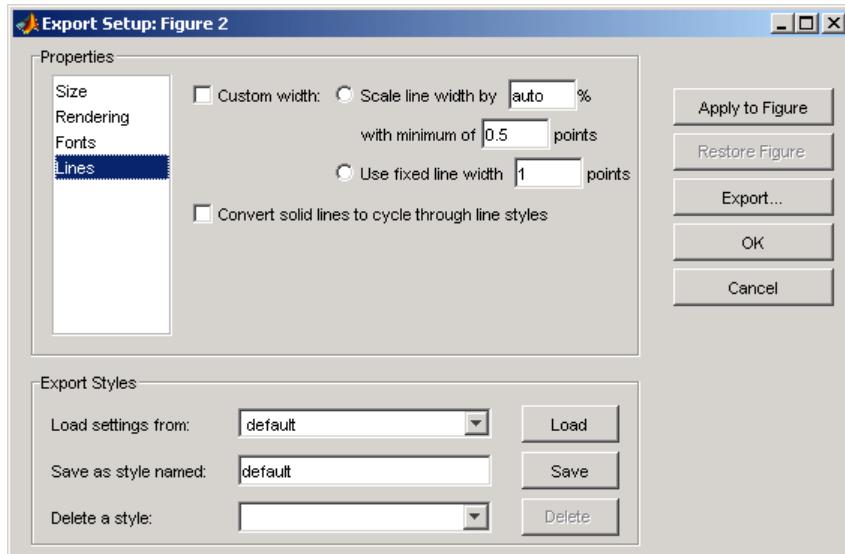
Click the check box and use the drop-down list to select the weight or thickness to be applied to text in the figure. Choose from **normal**, **light**, **demi**, or **bold**.

Custom Angle

Click the check box and use the drop-down list to select the angle to be applied to text in the figure. Choose from **normal**, **italic**, or **oblique**.

Changing Line Characteristics

Click **Lines** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom width

Click the check box and use the radio buttons to select a relative or absolute line size for the figure.

- **Scale line width by N %** — Increases or decreases the width of all lines by a relative amount, N percent. Enter the word **auto** to automatically select the appropriate line width.
- **With minimum of N points** — Specify a minimum line width when scaling the font by a percentage.
- **Use fixed line width N points** — Sets the width of all lines to an absolute value, N points.

Convert solid lines to cycle through line styles

When colored graphics are imported into an application that does not support color, lines that could formerly be distinguished by unique color are likely to appear the same. For example, a red line that shows an input level and a blue line showing output both appear as black when imported into an application that does not support colored graphics.

Clicking this check box causes exported lines to have different line styles, such as solid, dotted, or dashed lines rather than differentiating between lines based on color.

Saving and Loading Settings

If you think you might use these export settings at another time, you can save them now and reload them later. At the bottom of each Export Setup dialog box, there is a panel labeled **Export Styles**. To save your current export styles, type a name into the **Save as style named** text box, and then click **Save**.

If you then click the **Load settings from** drop-down list, the name of the style you just saved appears among the choices of export styles you can load. To load a style, select one of the choices from this list and then click **Load**.

To delete any style you no longer have use for, select that style name from the **Delete a style** drop-down list and click **Delete**.

Exporting the Figure

When you finish setting the export style for your figure, you can export the figure to a file by clicking the **Export** button on the right side of any of the four Export Setup dialog boxes. A new window labeled **Save As** opens.

Select a folder to save the file in from the **Save in** list at the top. Select a file type for your file from the **Save as type** drop-down list at the bottom, and then enter a file name in the **File name** text box. Click the **Save** button to export the file.

Exporting Using MATLAB Commands

Use the **print** function to print from the MATLAB command line or from a program. See “Printing and Exporting with **print**” on page 7-3 for basic information on printing from the command line.

To export the current or most recently active figure, type

```
print -dfileformat filename
```

where `fileformat` is a supported graphics format and `filename` is the name you want to give to the export file. MATLAB selects the filename extension, if you don't specify it.

You can also specify a number of options with the `print` function. These are shown in the Printing Options table on the `print` reference page.

For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution and using the EPS color graphics format, type

```
print -f2 -r600 -depsc spline2d
```

Importing MATLAB Graphics into Other Applications

You can include MATLAB graphics in a wide variety of applications for word processing, slide preparation, modification by a graphics program, presentation on the Internet, and so on. In general, the process is the same for all applications:

- 1** Use MATLAB graphics to create the figure you want to import into another application.
- 2** Export the MATLAB figure to one of the supported graphics file formats, selecting a format that is both appropriate for the type of figure and supported by the target application.
- 3** Use the import features of the target application to import the graphics file.

Edit Before You Export

Vector graphics may be fully editable in a few high-end applications, but most applications do not support editing beyond simple resizing. Bitmaps cannot be edited with quality results unless you use a software package devoted to image processing. In general, you should try to make all the necessary settings while your figure is still in MATLAB.

Importing into Microsoft Applications

To import your exported figure into a Microsoft application, select **Picture** from the **Insert** menu. Then select **From File** and navigate to your exported file. If you use the clipboard to perform your export operations, you can take advantage of the recommended MATLAB settings for Microsoft Word and PowerPoint®.

Example — Importing an EPS Graphic into LaTeX

This example shows how to import an EPS file named `peaks.eps` into LaTeX.

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{peaks.eps}}
\caption{Surface Plot of Peaks}
\end{figure}

\end{document}
```

EPS graphics can be edited after being imported to LaTeX. For example, you can specify the height in any LaTeX-compatible dimension. To set the height to 3.5 inches, use the command

```
height=3.5in
```

You can use the `angle` function to rotate the graph. For example, to rotate the graph 90 degrees, add

```
angle=90
```

to the same line of code that sets the height, i.e., `[height=10cm,angle=90]`.

Exporting to the Windows or Macintosh Clipboard

You can export a figure to the Windows or Macintosh clipboard. The formats used are discussed below.

- “Windows Clipboard Format” on page 7-25
- “Macintosh Clipboard Format” on page 7-26
- “Exporting to the Clipboard Using GUIs” on page 7-26
- “Exporting to the Clipboard Using MATLAB Commands” on page 7-28

Windows Clipboard Format

You can copy graphic data to the system clipboard data on Windows in either of two graphics formats: EMF color vector or a bitmap image.

By default, the graphics format is automatically selected, based on the rendering method used to display the figure. For figures rendered with OpenGL, MATLAB uses the bitmap format. For figures rendered with Painter's, the EMF format is used.

To override the automatic selection, specify the format of your choice using either the Windows Copy Options Preferences dialog, or the `-d` switch in the print command.

Macintosh Clipboard Format

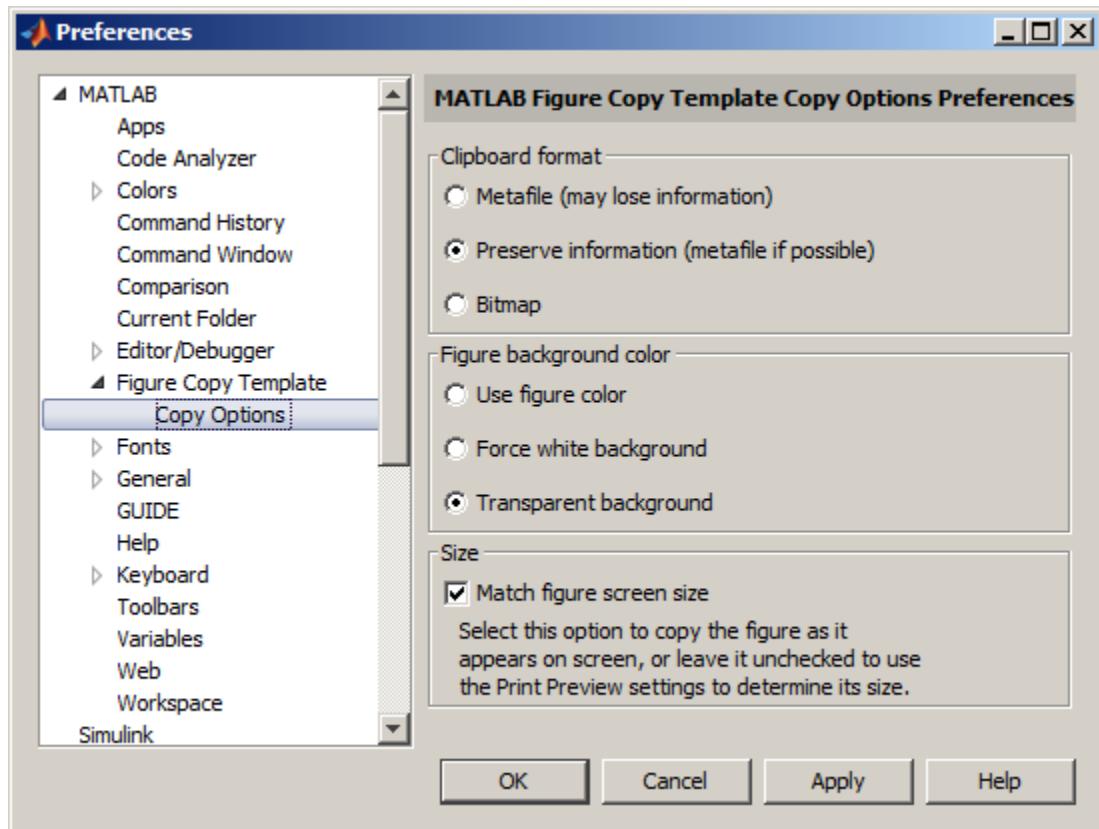
On Macintosh platforms (using Java® figures, the default), MATLAB copies figures to other applications in a high-resolution PDF format. If the figure contains uicontrols, then MATLAB uses a TIFF format instead. Use **Edit > Copy Figure** to copy to the clipboard, not the **print** command. The entire figure window is captured.

Exporting to the Clipboard Using GUIs

Before you export the figure to the clipboard, you can use the Copy Options Preferences dialog box to select a nondefault graphics format, or to adjust certain figure settings. These settings become the new defaults for all figures exported to the clipboard.

Note When exporting to the clipboard in Windows metafile format (**print -dmeta**), the settings from the figure Copy Options Preferences template are ignored.

To open the Windows only Copy Options Preferences dialog box, select **Copy Options** from the figure window's **Edit** menu. Any changes you make with this dialog box affect only the clipboard copy of the figure; they do not affect the way the figure looks on the screen.



Settings you can change in the Copy Options Preferences dialog box are as follows:

Clipboard format

- To copy the figure in EMF color vector format, select **Metafile**. This option places a metafile on the system clipboard.
- The **Preserve information** option selects the format based on the current figure renderer. If the figure renderer is
 - **painters** — clipboard format is metafile
 - **opengl** — clipboard format is a bitmap image
- To use a bitmap format, select **Bitmap**.

Note: On Macintosh platforms, the **Copy Options** dialog box does not have the **Clipboard format** options.

Figure background color

To keep the background color the same as it appears on the screen, select **Use figure color**. To make the background white, select **Force white background**. For a background that is transparent, for example, a slide background to frame the axes part of a figure, select **Transparent background**.

Size

Select **Match figure screen size** to copy the figure as it appears on the screen, or leave it unselected to use the **Width** and **height** options in the Export Setup dialog to determine its size.

- 1 Open the Copy Options Preferences dialog box if you need to make any changes to those preferences used in copying to the clipboard.
- 2 Click **OK** to see the new preferences. These will be used for all future figures exported to the clipboard.
- 3 Select **Copy Figure** from the figure window's **Edit** menu to copy the figure to the clipboard.

Exporting to the Clipboard Using MATLAB Commands

Export to the clipboard on Windows using the **print** function with a graphics format, but no filename. Use one of the following clipboard formats: **-dbitmap**, **-dmeta**, or **-dpdf**. These switches create a bitmap image, an enhanced metafile (EMF), or PDF output.

For example, on Windows:

```
print -dbitmap -clipboard  
print -dmeta -clipboard  
print -dpdf -clipboard
```

On the Macintosh:

```
print -dbitmap -clipboard  
print -dpdf -clipboard
```

On Unix:

```
print -dbitmap -clipboard  
print -dpdf -clipboard
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Printing and Exporting Use Cases

In this section...

- “Printing a Figure at Screen Size” on page 7-30
- “Printing with a Specific Paper Size” on page 7-31
- “Printing a Centered Figure” on page 7-31
- “Exporting in a Specific Graphics Format” on page 7-32
- “PostScript and PDF Font Translations” on page 7-33
- “Exporting in EPS Format with a TIFF Preview” on page 7-34
- “Exporting a Figure to the Clipboard” on page 7-34

Printing a Figure at Screen Size

By default, your figure prints at 8-by-6 inches. This size includes the area delimited by the background. This example shows how to print or export your figure the same size it is displayed on your screen.

Using the Graphical User Interface

- 1 Resize your figure window to the size you want it to be when printed.
- 2 Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
- 3 In the **Placement** panel, select **Auto (Actual Size, Centered)**.
- 4 Click **Print** in the upper right corner to print the figure.
- 5 The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the `PaperPositionMode` property to `auto` before printing the figure.

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

If later you want to print the figure at its original size, set `PaperPositionMode` back to `'manual'`.

Printing with a Specific Paper Size

The MATLAB default paper size is 8.5-by-11 inches. This example shows how to change the paper size to 8.5-by-14 inches by selecting a paper type (Legal).

Using the Graphical User Interface

- 1 Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
- 2 Select the **Legal** paper type from the list in the **Paper** panel. The **Width** and **Height** fields update to **8.5** and **14**, respectively.
- 3 Make sure that **Units** is set to **inches**.
- 4 Click **Print** in the upper right corner to print the figure.
- 5 The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the **PaperUnits** property to **inches** and the **PaperType** property to **Legal**.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperType', 'Legal');
```

Alternatively, you can set the **PaperSize** property to the size of the paper, in the specified units.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [8.5 14]);
```

Printing a Centered Figure

This example sets the size of a figure to 5.5-by-3 inches and centers it on the paper.

Using the Graphical User Interface

- 1 Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
- 2 Make sure **Use manual size and position** is selected.
- 3 Enter **5.5** in the **Width** field and **3** in the **Height** field.
- 4 Make sure that **Units** field is set to **inches**.
- 5 Click **Center**.
- 6 Click **OK**.
- 7 Click **Print** to open the Print dialog box and print the figure.

Using MATLAB Commands

- 1 Start by setting `PaperUnits` to `inches`.

```
set(gcf, 'PaperUnits', 'inches')
```

- 2 Use `PaperSize` to return the size of the current paper.

```
papersize = get(gcf, 'PaperSize')
```

```
papersize =
    8.5000    11.0000
```

- 3 Initialize variables to the desired width and height of the figure.

```
width = 5.5;           % Initialize a variable for width.
height = 3;            % Initialize a variable for height.
```

- 4 Calculate a left margin that centers the figure horizontally on the paper. Use the first element of `papersize` (width of paper) for the calculation.

```
left = (pagesize(1)-width)/2
```

```
left =
    1.5000
```

- 5 Calculate a bottom margin that centers the figure vertically on the paper. Use the second element of `pagesize` (height of paper) for the calculation.

```
bottom = (pagesize(2)-height)/2
```

```
bottom =
    4
```

- 6 Set the figure size and print.

```
myfiguresize = [left, bottom, width, height];
set(gcf, 'PaperPosition', myfiguresize);
print
```

Exporting in a Specific Graphics Format

Export a figure to a graphics-format file when you want to import it at a later time into another application such as a word processor.

Using the Graphical User Interface

- 1 Select **Save As** from the figure window's **File** menu.

- 2 Use the **Save in** field to navigate to the folder in which you want to save your file.

- 3** Select a graphics format from the **Save as type** list.
- 4** Enter a filename in the **File name** field. An appropriate file extension, based on the format you chose, is displayed.
- 5** Click **Save** to export the figure.

Using MATLAB Commands

From the command line, you must specify the graphics format as an option. See the `print` reference page for a complete list of graphics formats and their corresponding option strings.

This example exports a figure to an EPS color file, `myfigure.eps`, in your current folder.

```
print -depsc myfigure
```

This example exports Figure No. 2 at a resolution of 300 dpi to a 24-bit JPEG file, `myfigure.jpg`.

```
print -djpeg -f2 -r300 myfigure
```

This example exports a figure at screen size to a 24-bit TIFF file, `myfigure.tif`.

```
set(gcf, 'PaperPositionMode', 'auto')    % Use screen size
print -dtiff myfigure
```

PostScript and PDF Font Translations

MATLAB translates certain fonts before printing or exporting to PostScript or PDF formats. This table describes the translation:

This Font	Replaced With
Arial	Helvetica
Times New Roman	Times
NewCenturySchoolbook	Sans-Serif
Tahoma	Helvetica
sans serif	Sans-Serif
Any other font	Courier

Exporting in EPS Format with a TIFF Preview

Use the `print` function to export a figure in EPS format with a TIFF preview. When you import the figure, the application can display the TIFF preview in the source document. The preview is color if the exported figure is color, and black and white if the exported figure is black and white.

This example exports a figure to an EPS color format file, `myfigure.eps`, and includes a color TIFF preview.

```
print -depsc -tiff myfigure
```

This example exports a figure to an EPS black-and-white format file, `myfigure.eps`, and includes a black-and-white TIFF preview.

```
print -deps -tiff myfigure
```

Exporting a Figure to the Clipboard

Export a figure to the clipboard in graphics format when you want to paste it into another Windows or Macintosh application such as a word processor.

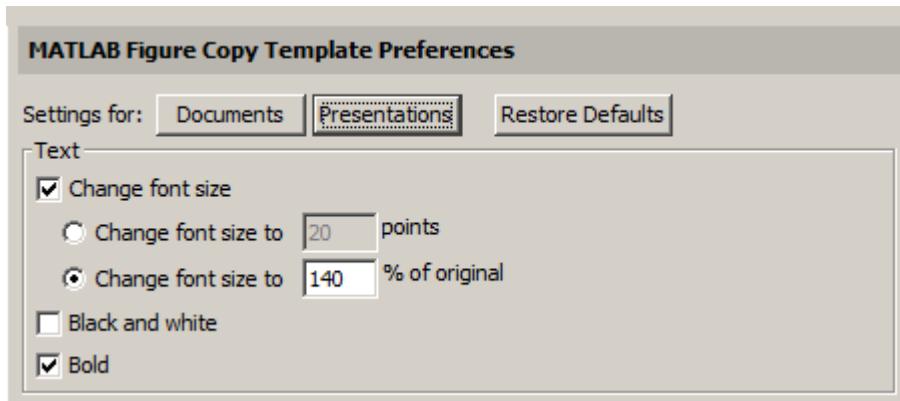
Using the Graphical User Interface

This example exports a figure to the clipboard in enhanced metafile (EMF) format. Figure settings are chosen that would make the exported figure suitable for use in a Microsoft Word or PowerPoint slide. Changing the settings modifies the figure displayed on the screen.

- 1 Create a figure and add a title:

```
x = -pi:0.01:pi;
h = plot(x,sin(x));
title('Sine Plot');
```

- 2 Select **Preferences** from the figure **File** menu. Then select **Figure Copy Template** from the Preferences dialog box.
- 3 In the **Figure Copy Template Preferences** panel, click the **Presentations** button. The suggested settings for presentations include:
 - Increased font size
 - Bold font



Note: In Macintosh®, the **Figure Copy Template Preferences** panel is not displayed. For more information on how to export figures in Macintosh, see “Exporting to the Windows or Macintosh Clipboard” on page 7-25.

- 4 In the **Lines** panel, change the **Custom width** to 4 points.
- 5 In the **Uicontrols and axes** panel, select **Keep axes limits and tick spacing** to prevent tick marks and limits from possibly being rescaled when you export.
- 6 Click **Apply to Figure**. The changes appear in the figure window.

If you don't like the way your figure looks with the new settings, restore it to its original settings by clicking the **Restore Figure** button.

- 7 In the left pane of the Preferences dialog box, expand the **Figure Copy Template** topic. Select **Copy Options**.
- 8 In the **Copy Options** panel, select **Metafile** to export the figure in EMF format.
- 9 Check that **Transparent background** is selected. This choice makes the figure background transparent and allows the slide background to frame the axes part of the figure.
- 10 Clear the **Match figure screen size** check box so that you can use your own figure size settings.
- 11 Click **OK**.
- 12 Select **Export Setup** from the figure window's **File** menu.
- 13 Select the **Size** properties, and set **Width** to 6 and **Height** to 4.5. Make sure that **Units** are set to **inches**.
- 14 Click **Close**.

- 15** Select **Copy Figure** from the **Edit** menu. Your figure is now exported to the clipboard and can be pasted into a Windows application. On Macintosh computers, MATLAB exports the figure in the best format (bit-mapped or vector) based on the figure content.

Using MATLAB Commands

Use the **print** function and one of two clipboard formats (**-dmeta**, **-dbitmap**) to export a figure to the clipboard. Do *not* specify a filename.

This example exports a figure to the clipboard in enhanced metafile (EMF) format.

```
print -dmeta
```

This example exports a figure to the clipboard in bitmap (BMP) 8-bit color format.

```
print -dbitmap
```

Change Figure Settings

In this section...

- “Parameters that Affect Printing” on page 7-37
- “Selecting the Figure” on page 7-39
- “Selecting the Printer” on page 7-39
- “Setting the Figure Size and Position” on page 7-41
- “Setting the Paper Size or Type” on page 7-44
- “Setting the Paper Orientation” on page 7-46
- “Selecting a Renderer” on page 7-48
- “Setting the Resolution” on page 7-50
- “Setting the Axes Ticks and Limits” on page 7-52
- “Setting the Background Color” on page 7-54
- “Setting Line and Text Characteristics” on page 7-55
- “Setting the Line and Text Color” on page 7-58
- “Specifying a Colorspace for Printing and Exporting” on page 7-61
- “Excluding User Interface Controls from Printed Output” on page 7-63
- “Producing Uncropped Figures” on page 7-64

Parameters that Affect Printing

The table below shows parameters that you can set before submitting your figure to the printer.

The Parameter column lists all parameters that you can change.

The Default column shows the MATLAB default setting.

The Dialog Box column shows which dialog box to use to set that parameter. If you can make this setting on only one platform, this is noted in parentheses: (W) for Windows, and (U) for UNIX.

Some dialog boxes have tabs at the top to enable you to select a certain category. These categories are denoted in the table below using the format <dialogbox>/<tabname>.

For example, **Print Preview/Layout...** in this column means to use the Print Preview dialog box, selecting the **Layout** tab.

The **print** Command or **set** Property column shows how to set the parameter using the MATLAB **print** or **set** function. When using **print**, the table shows the appropriate command option (for example, **print -loose**). When using **set**, it shows the property name to set along with the type of object (for example, **(Line)** for line objects).

Parameter	Default	Dialog Box	print Command or set Property
Select figure	Last active window	None	print -fhandle
Select printer	System default	Print	print -pprinter
Figure size	8-by-6 inches	Print Preview/ Layout	PaperSize (Figure) PaperUnits (Figure)
Position on page	0.25 in. from left, 2.5 in. from bottom	Print Preview/ Layout	PaperPosition (Figure) PaperUnits (Figure)
Position mode	Manual	Print Preview/ Layout	PaperPositionMode (Figure)
Paper type	Letter	Print Preview/ Layout	PaperType (Figure)
Paper orientation	Portrait	Print Preview/ Layout	PaperOrientation (Figure)
Renderer	Selected automatically	Print Preview/ Advanced	print -painters -opengl
Renderer mode	Auto	Print Preview/ Advanced	RendererMode (Figure)
Resolution	Depends on driver or graphics format	Print Preview/ Advanced	print -rresolution
Axes tick marks	Recompute	Print Preview/ Advanced	XTickMode , etc. (Axes)
Background color	Force to white	Print Preview/Color	Color (Figure) InvertHardCopy (Figure)

Parameter	Default	Dialog Box	print Command or set Property
Font size	As in the figure	Print Preview/Lines/ Text	FontSize (Text)
Bold font	Regular font	Print Preview/Lines/ Text	FontWeight (Text)
Line width	As in the figure	Print Preview/Lines/ Text	LineWidth (Line)
Line style	Black or white	Figure Copy Template	LineStyle (Line)
Line and text color	Black and white	Print Preview/Lines/ Text	Color (Line, Text)
CMYK color	RGB color	Print Preview/Color (U)	print -cmyk
UI controls	Printed	Print Preview/ Advanced	print -noui
Bounding box	Tight	N/A	print -loose
“Copy background”	Transparent	Copy Options (W)	See “Background color”
“Copy size”	Same as screen size	Copy Options (W)	See “Figure Size”

Selecting the Figure

By default, the current figure prints. If you have more than one figure open, the current figure is the last one that was active. To make a different figure active, click it to bring it to the foreground.

Using MATLAB Commands

Specify a figure using the command

```
print -fhandle
```

This example sends Figure No. 2 to the printer.

```
print -f2
```

Selecting the Printer

You can select the printer you want to use with the Print dialog box or with the **print** function.

Using the Graphical User Interface

- 1 Select **Print** from the figure window's **File** menu.
- 2 Select the printer from the list box near the top of the Print dialog box.
- 3 Click **OK**.

Using MATLAB Commands

You can select the printer using the **-P** switch of the **print** function.

This example prints Figure No. 3 to a printer called **Calliope**.

```
print -f3 -PCalliope
```

If the printer name has spaces in it, put single quotation marks around the **-P** option, as shown here.

```
print '-Pmy local printer'
```

Note: On Macintosh computers, printer names shown in the print dialog might not work when specified at the command line using the **-P** option. Use the following command to get the names of the printers you can use with the **-P** option on your system.

```
[~,printers] = findprinters
```

Using a Network Print Server

On Windows systems, you can print to a network print server using the form shown here for a printer named **trinity** located on a computer named **PRINTERS**.

```
print -P\\PRINTERS\\trinity
```

Note: On Windows platforms, when you use the **-P** option to identify a printer to use, if you specify any driver other than **-dwin** or **-dwinc**, MATLAB output goes to a file with an appropriate extension but does not send it to the printer. You can then copy that file to a printer.

Setting the Figure Size and Position

The default output figure size is 8 inches wide by 6 inches high, which maintains the aspect ratio (width to height) of the MATLAB figure window. The figure's default position is centered both horizontally and vertically when printed to a paper size of 8.5-by-11 inches.

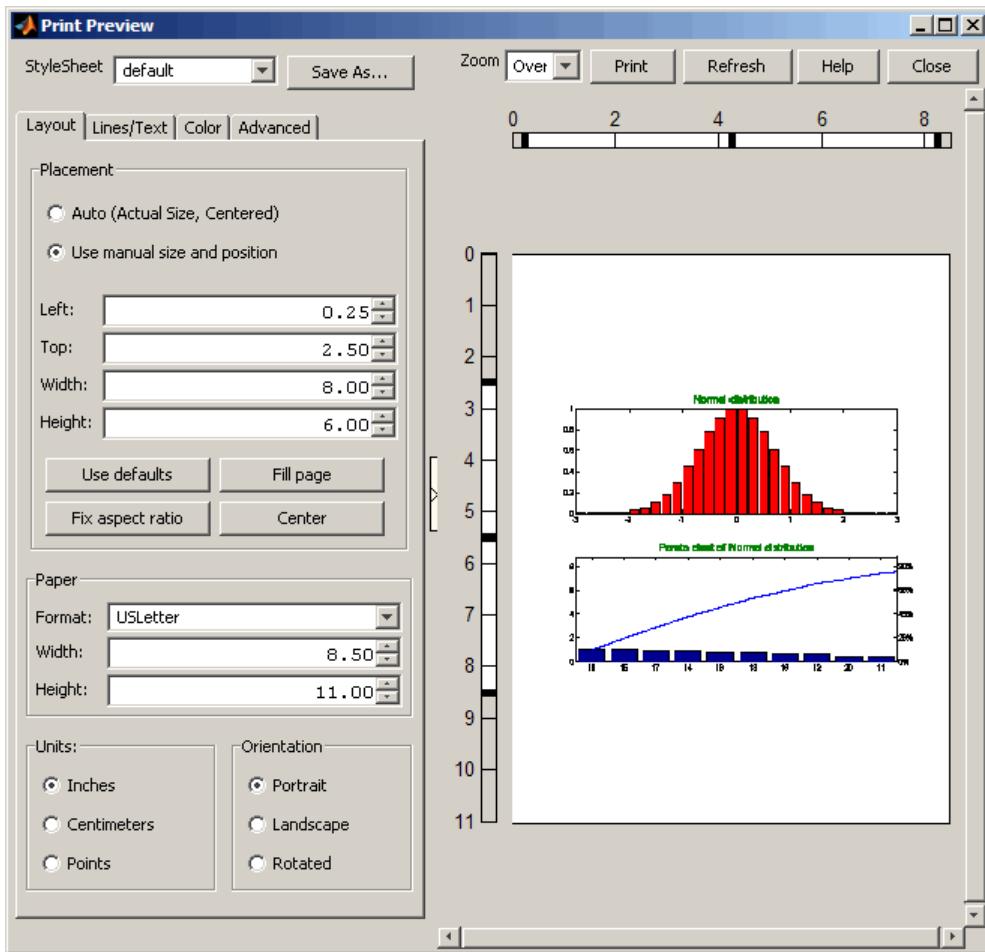
You can change the size and position of the figure:

- “Using the Graphical User Interface” on page 7-41
- “Using MATLAB Commands” on page 7-43

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the size and position of your figure on the printed page.

Use the text edit boxes on the left to enter new dimensions for your figure. Or, use the handlebars on the rulers in the right-hand pane to drag the margins and location of your figure with the mouse. The outer handlebars move the figure toward or away the nearest margin, while the central handlebar repositions the figure on the page without changing its proportions. Guidelines appear while you are using the handlebars.



Settings you can change in the **Layout** tab are as follows:

Placement

Choose whether you want the figure to be the same size as it is displayed on your screen, or you want to manually change its size using the options in the **Layout** pane.

When you select the **Use manual size and position** mode, type the widths of any of the four margins and the preview image responds after each entry you make. Select units

of measure (inches/centimeters/points) with pushbuttons on the **Units** section on the bottom of the pane.

You can use the four buttons at the bottom of the Placement section to expand the figure to fill the page, make its aspect ratio (ratio of y-extent to x-extent) as printed match that of the figure, center the figure on the page, or restore the setup to what it was when you opened the Print Preview dialog. Selecting **Fill page** can alter the aspect ratio of your image. To get the maximum figure size without altering the aspect ratio, select **Fix aspect ratio**.

Auto (actual size, centered)

Select this option to center the figure on the page; it will be the same size as it is in the figure window. The four buttons below the control are dimmed when you select this option.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To print your figure with a specific size or position, make sure the **PaperPositionMode** property is set to **manual** (the default). Then set the **PaperPosition** property to the desired size and position.

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

```
[left bottom width height]
```

where

- **left** specifies the distance from the left edge of the paper to the left edge of the figure.
- **bottom** specifies the distance from the bottom of the paper to the bottom of the figure.
- **width** and **height** specify the figure's width and height.

The default values for **PaperPosition** are

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

This example sets the figure size to a width of 4 inches and height of 2 inches, with the origin of the figure positioned 2 inches from the left edge of the paper and 1 inch from the bottom edge.

```
set(gcf, 'PaperPositionMode', 'manual');  
set(gcf, 'PaperUnits', 'inches');  
set(gcf, 'PaperPosition', [2 1 4 2]);
```

Note `PaperPosition` specifies a bottom margin, rather than a top margin as Print Preview does. When you set the top margin using Print Preview, This setting is used to calculate the bottom margin, and updates the `PaperPosition` property appropriately.

Setting the Paper Size or Type

Set the paper size by specifying the dimensions or by choosing from a list of predefined paper types. If you do not set a paper size or type, the default paper size of 8.5-by-11 inches is used.

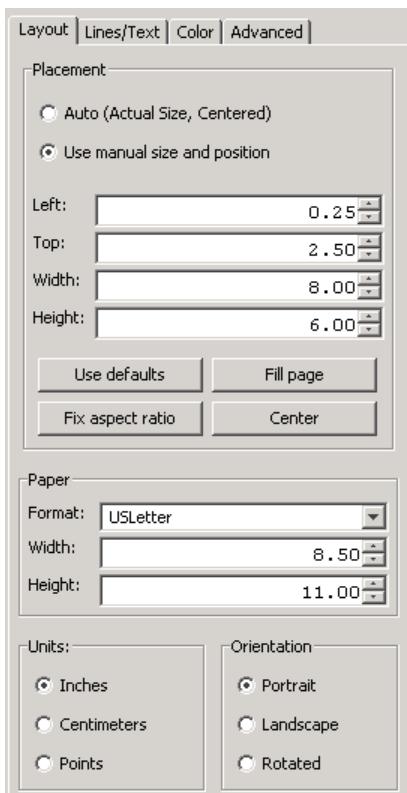
Paper-size and paper-type settings are interrelated—if you set a paper type, the paper size also updates. For example, if you set the paper type to **US Legal**, the width of the paper updates to 8.5 inches and the height to 14 inches.

You can change the paper size and orientation:

- “Using the Graphical User Interface” on page 7-44
- “Using MATLAB Commands” on page 7-46

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the paper type and orientation of the figure on the printed page.



Settings you can change in the **Layout** tab are as follows:

Paper Format, Units and Orientation

Select a paper type from the list under **Format**. If there is no paper type with suitable dimensions, enter your own dimensions in the **Width** and **Height** fields. Make sure **Units** is set appropriately to **inches**, **centimeters**, or **points**. If you change units after setting a paper width and height, the **Width** and **Height** fields update to use the units you just selected. The page region in the preview pane updates to show the new paper format or size when you change them.

Use the **Orientation** buttons to select how you want the figure to be oriented on the printed page. The illustration under “Setting the Paper Orientation” on page 7-46 shows the three types of orientation you can choose from.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

Set the **PaperType** property to one of the built-in MATLAB paper types, or set the **PaperSize** property to the dimensions of the paper.

When you select a paper type, the unit of measure is not automatically updated. We recommend that you set the **PaperUnits** property first.

For example, these commands set the units to **centimeters** and the paper type to **A4**.

```
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperType', 'A4');
```

This example sets the units to **inches** and sets the paper size of 5-by-7 inches.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [5 7]);
```

If you set a paper size for which there is no matching paper type, the **PaperType** property is automatically set to '**custom**'.

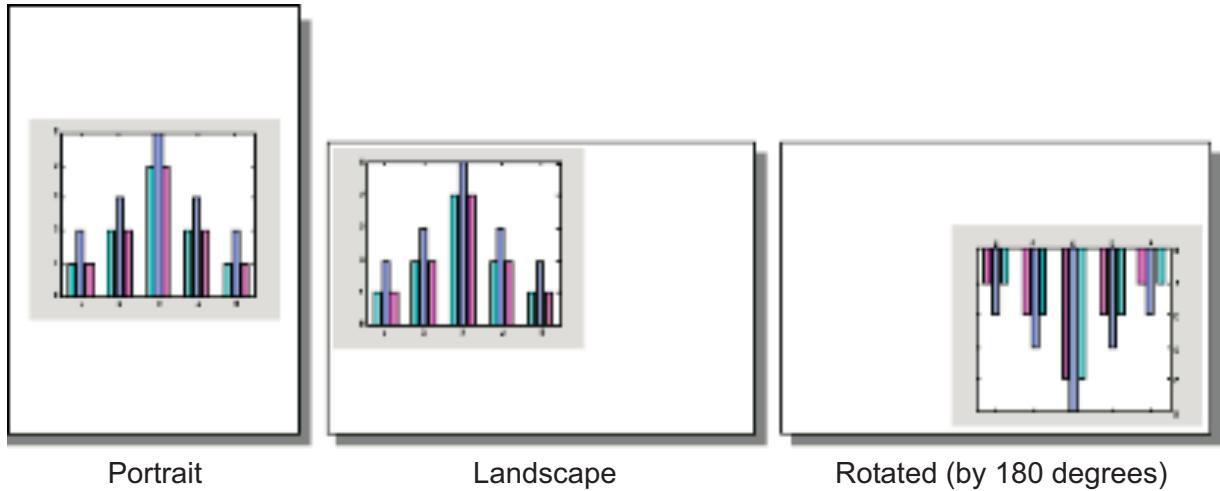
Setting the Paper Orientation

Paper orientation refers to how the paper is oriented with respect to the figure. The choices are **Portrait** (the default), **Landscape**, and **Rotated**.

You can change the orientation of the figure:

- “Using the Graphical User Interface” on page 7-47
- “Using MATLAB Commands” on page 7-47

The figure below shows the same figure printed using the three different orientations.



Note The **Rotated** orientation is not supported by all printers. When the printer does not support it, landscape is used.

Using the Graphical User Interface

- 1 Select **Print Preview** from the figure window's **File** menu and select the **Layout** tab. (See “Using the Graphical User Interface” on page 7-44).
- 2 Select the appropriate option button under **Orientation**.
- 3 Click **Close**.

Using MATLAB Commands

Use the **PaperOrientation** figure property or the **orient** function. Use the **orient** function if you always want your figure centered on the paper.

The following example sets the orientation to **landscape**:

```
set(gcf, 'PaperOrientation', 'landscape');
```

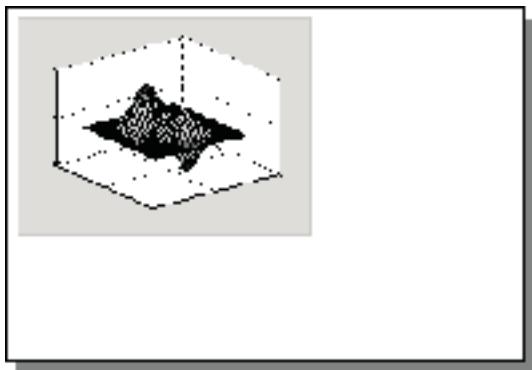
Centering the Figure

If you set the **PaperOrientation** property from **portrait** to either of the other two orientation schemes, you might find that what was previously a centered image

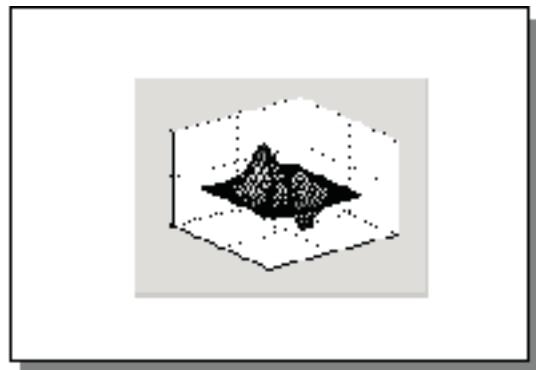
is now positioned near the paper's edge. You can either adjust the position (use the `PaperPosition` property), or you can use the `orient` function, which always centers the figure on the paper.

The `orient` function takes the same argument names as `PaperOrientation`. For example,

```
orient rotated;
```



Orientation set to 'landscape' using
'PaperOrientation' property.



Orientation set to 'landscape' using
`orient` function.

Selecting a Renderer

A renderer is software and/or hardware that enables MATLAB to display, print, or export a figure. With regard to printing, the renderer determines if the output is a vector or a bitmap format.

For information on specifying the renderer, see these sections:

- “Using the Graphical User Interface” on page 7-49
- “Using MATLAB Commands” on page 7-49

Supported Renderers

MATLAB supports two rendering methods with the following characteristics:

Painter's

- Draws figures using vector graphics
- Generally produces results that scale better and can be edited in application programs designed for this purpose
- The renderer for creating PostScript, EPS, PDF, metafile, or SVG files.

OpenGL

- Draws figures using bitmap (raster) graphics
- Generally faster than Painter's
- Can access graphics rendering hardware available on some systems
- Avoids possible limitations of Painter's with regard to sorting graphics objects arranged in 3-D views

For more detailed information about changing rendering methods, see the **Figure Renderer** property.

Manually Set the Renderer

Set the renderer appropriately for the type of output file:

- Use Painter's for vector format
- Use OpenGL for bitmap formats

Using the Graphical User Interface

- 1 Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Advanced** tab.
- 2 In the **Renderer** drop-down menu, select the desired rendering method from the list box.
- 3 Click **Close**.

Using MATLAB Commands

You can use the **Renderer** property or a switch with the **print** function to set the renderer for printing or exporting. These two lines each set the renderer for the current figure to painters.

```
set(gcf, 'Renderer', 'painters');
```

or

```
print -painters
```

The first example saves the new value of `Renderer` with the figure. The second example affects only the current print or export operation.

Renderer Selection

When you export a figure to a graphics file format, MATLAB selects the renderer based on whether the output file type is a vector or bitmap format. However, MATLAB can change the renderer only if the `RendererMode` property is set to `auto`.

If you are exporting to a vector format, MATLAB uses the painters renderer in most cases, which results in true vector graphics that can be edited more easily in other applications.

Setting the Resolution

Resolution refers to how accurately your figure is rendered when printed or exported. Higher resolutions produce higher quality output. The specific definition of resolution depends on whether your figure is output as a bitmap or as a vector graphic.

You can change the resolution used to print a figure:

- “Using the Graphical User Interface” on page 7-52
- “Using the Graphical User Interface on UNIX Platforms” on page 7-63
- “Using MATLAB Commands” on page 7-52

Default Resolution and When You Can Change It

The default resolution depends on the renderer used and the graphics format or printer driver specified. The following two tables summarize the default resolutions and whether you can change them.

Resolutions Used with Graphics Formats

Graphics Format	Default Resolution	Can Be Changed?
Built-in MATLAB export formats, (except for EMF and EPS)	150 dpi (always use OpenGL)	Yes

Graphics Format	Default Resolution	Can Be Changed?
EMF export format (Enhanced Metafile)	150 dpi	Yes
EPS (Encapsulated PostScript)	150 dpi, if OpenGL; 864 dpi if Painter's	Yes

Resolutions Used with Printer Drivers

Printer Driver	Default Resolution	Can Be Changed?
Windows and PostScript drivers	150 dpi, if OpenGL; 864 dpi if Painter's	Yes

Choosing a Setting

You might need to determine your resolution requirements through experimentation, but you can also use the following guidelines.

For Printing

The default resolution of 150 dpi is normally adequate for typical laser-printer output. However, if you are preparing figures for high-quality printing, such as a textbook or color brochures, you might want to use 200 or 300 dpi. The resolution you can use can be limited by the printer's capabilities.

For Exporting

If you are exporting your figure, base your decision on the resolution supported by the final output device. For example, if you will import your figure into a word processing document and print it on a printer that supports a maximum resolution setting of 300 dpi, you could export your figure using 300 dpi to get a precise one-to-one correspondence between pixels in the file and dots on the paper.

Note The only way to set resolution when exporting is with the `print` function.

Impact of Resolution on Size and Memory Needed

Resolution affects file size and memory requirements. For both printing and exporting, the higher the resolution setting, the longer it takes to render your figure.

Using the Graphical User Interface

To set the resolution for built-in MATLAB printer drivers:

- 1 From the Print dialog box, click **Properties**. This opens a new dialog box. (This box can differ from one printer to another.)
- 2 You may be able to set the resolution from this dialog. If not, then click **Advanced** to get to a dialog box that enables you to do this.
- 3 Set the resolution, and then click **OK**. (The resolution setting might be labeled by another name, such as “Print Quality.”)

Using MATLAB Commands

If you use a Windows printer driver, you can only set the resolution using the Windows Document Properties dialog box.

Otherwise, to set the resolution for printing or exporting, the syntax is

```
print -rnumber
```

where **number** is the number of dots per inch. To print or export a figure using screen resolution, set **number** to 0 (zero).

This example prints the current figure with a resolution of 100 dpi:

```
print -r100
```

This example exports the current figure to a TIFF file using screen resolution:

```
print -r0 -dtiff myfile.tif
```

Setting the Axes Ticks and Limits

The default output size, 8-by-6 inches, is normally larger than the screen size. If the size of your printed or exported figure is different from its size on the screen, the number and placement of axes tick marks scale to suit the output size. This section shows you how to lock them so that they are the same as they were when displayed.

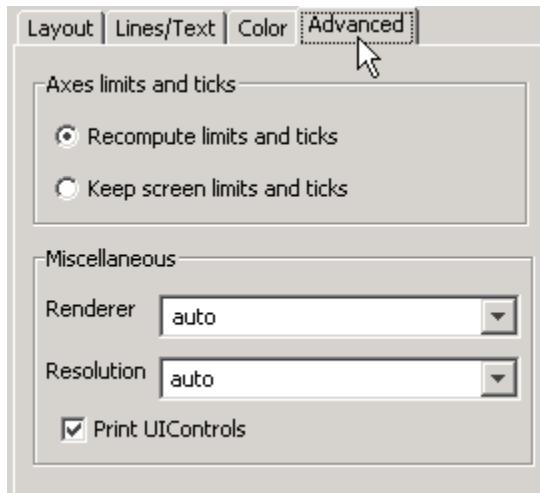
You can change the resolution used when printing a MATLAB figure:

- “Using the Graphical User Interface” on page 7-53

- “Using MATLAB Commands” on page 7-54

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Advanced** tab to make changes to the axes, UI controls, or renderer settings.



Settings you can change in the **Advanced** tab are as follows, by panel:

Axes limits and ticks

If the size of your printed or exported figure is different from its size on the screen, the number and placement of axes tick marks scale to suit the output size. Select **Keep screen limits and ticks** to lock them so that they are the same as they were when displayed. If you want to automatically adjust the ticks and limits when scaling for printing, select **Recompute limits and ticks**.

Miscellaneous

Use the **Renderer** drop-down menu to specify which renderer to use in printing the figure. Set the renderer to **Painters**, or **OpenGL**, or select **auto** to automatically decide which one to use, depending on the characteristics of the figure. (See “Selecting a Renderer” on page 7-48).

Use the **Resolution** drop-down menu to specify the resolution, in dots per inch (DPI), at which to render and print the figure. You can select 150, 300, or 600 DPI, or type in a different value (positive integer).

Figure UI Controls

By default, user interface controls are included in your printed or exported figure. Clear the **Print UIControls** check box to exclude them. (See “Excluding User Interface Controls from Printed Output” on page 7-63).

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To set the **XTickMode**, **YTickMode**, and **ZTickMode** properties to **manual**, type

```
set(gca, 'XTickMode', 'manual')
set(gca, 'YTickMode', 'manual')
set(gca, 'ZTickMode', 'manual')
```

Setting the Background Color

You can keep the background the same as is shown on the screen when printed, or change the background to white. There are two types of background color settings in a figure: the axes background and the figure background. The default displayed color of both backgrounds is gray, but you can set them to any of several colors.

Regardless of the background colors in your displayed figure, by default, they are always changed to white when you print or export. This section shows you how to retain the displayed background colors in your output.

Using the Graphical User Interface

To retain the background color on a per figure basis:

- 1 Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Color** tab.
- 2 Select **Same as figure**.

3 Click Close.

If you are exporting your figure using the clipboard, use the **Copy Options** panel of the Preferences dialog box.

Using MATLAB Commands

To retain your background colors, use

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

The following example sets the figure background color to **blue**, the axes background color to **yellow**, and then sets **InvertHardCopy** to **off** so that these colors appear in your printed or exported figure.

```
set(gcf, 'color', 'blue');
set(gca, 'color', 'yellow');
set(gcf, 'InvertHardCopy', 'off');
```

Setting Line and Text Characteristics

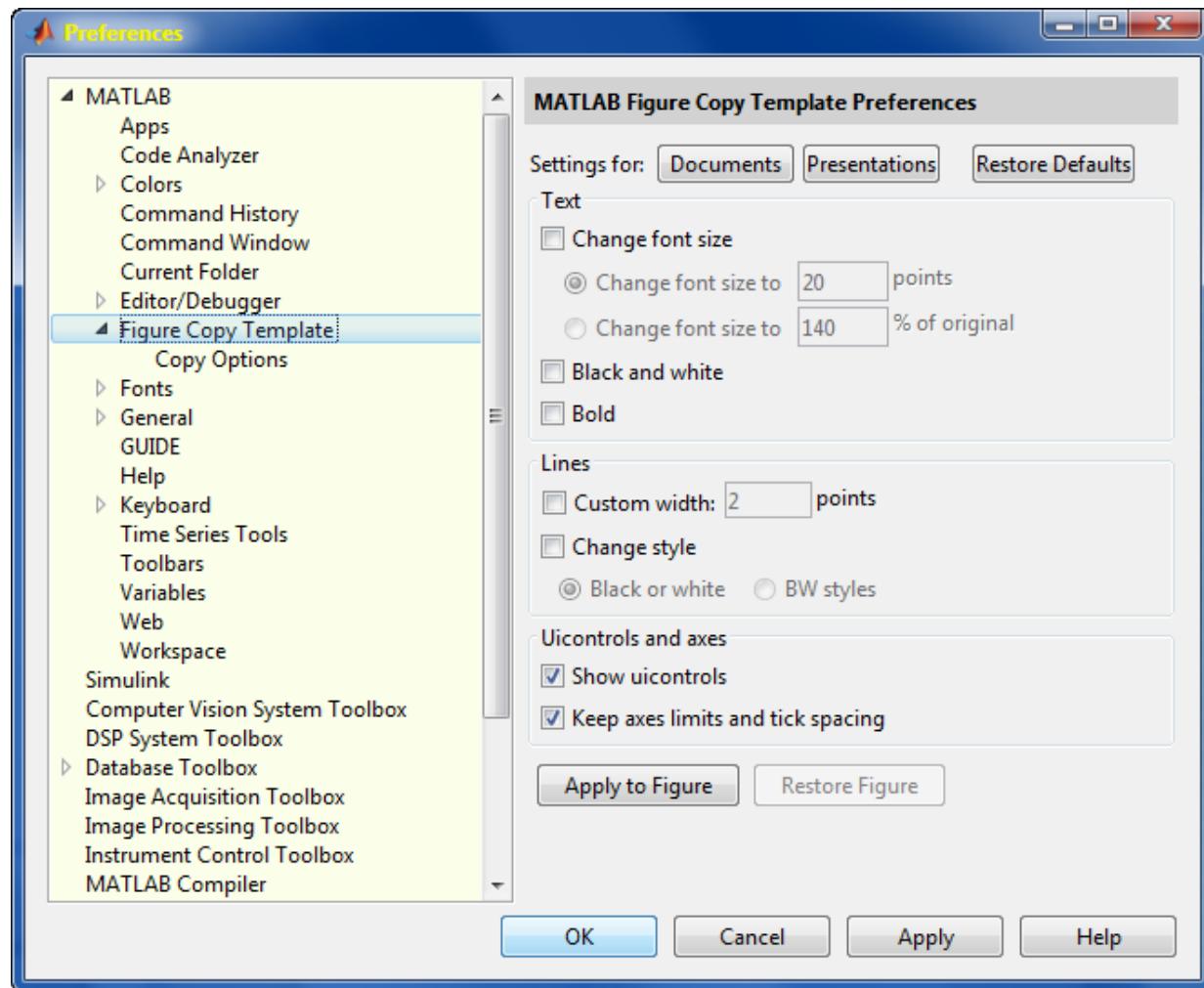
If you transfer your figures to Microsoft Word or PowerPoint applications, you can set line and text characteristics to values recommended for those applications. The Figure Copy Template Preferences dialog box provides Word and PowerPoint options to make these settings, or you can set certain line and text characteristics individually.

You can change line and text characteristics:

- “Using the Graphical User Interface” on page 7-55
- “Using MATLAB Commands” on page 7-57

Using the Graphical User Interface

To open Figure Copy Template Preferences, select **Preferences** from the **File** menu, and then click **Figure Copy Template** in the left pane.



Settings you can change in the Figure Copy Template Preferences dialog box are as follows:

Microsoft Word or PowerPoint

Click **Documents** or **Presentations** to apply recommended MATLAB settings.

Text

Use options under **Text** to modify the appearance of all text in the figure. You can change the font size, change the text color to black and white, and change the font style to bold.

Lines

Use the **Lines** options to modify the appearance of all lines in the figure:

- **Custom width** — Change the line width.
- **Change style (Black or white)** — Change colored lines to black or white.
- **Change style (B&W styles)** — Change solid lines to different line styles (e.g., solid, dashed, etc.), and black or white color.

UIControls and axes

If your figure includes user interface controls, you can choose to show or hide them by clicking **Show uicontrols**. Also, to keep axes limits and tick marks as they appear on the screen, click **Keep axes limits and tick spacing**. To allow automatic scaling of axes limits and tick marks based on the size of the printed figure, clear this box.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

You can use the **set** function on selected graphics objects in your figure to change individual line and text characteristics.

For example, to change line width to 1.8 and line style to a dashed line, use

```
lineobj = findobj('Type','line');
set(lineobj,'LineWidth',1.8);
set(lineobj,'LineStyle','--');
```

To change the font size to 15 points and font weight to bold, use

```
textobj = findobj('Type','text');
set(textobj,'FontUnits','points');
```

```
set(textobj, 'FontSize', 15);  
set(textobj, 'FontWeight', 'bold');
```

Setting the Line and Text Color

When colored lines and text are dithered to gray by a black-and-white printer, it does not produce good results for thin lines and the thin lines that make up text characters. You can, however, force all line and text objects in the figure to print in black and white, thus improving their appearance in the printed copy. When you select this setting, the lines and text are printed all black or all white, depending on the background color.

The default is to leave lines and text in the color that appears on the screen.

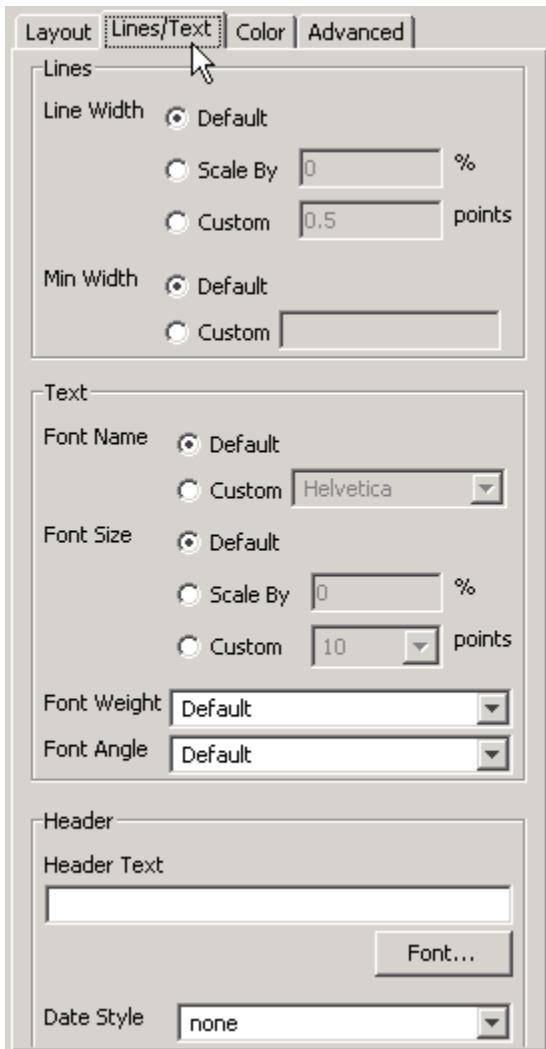
Note Your background color might not be the same as what you see on the screen. See the **Color** tab for an option that preserves the background color when printing.

You can change the resolution used to print a figure:

- “Using the Graphical User Interface” on page 7-58
- “Using MATLAB Commands” on page 7-60

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Lines and Text** tab to make changes to the color of all lines and text on the printed page. The controls for the **Lines and Text** tab are shown below:



Settings you can change in **Lines and Text** are as follows:

Lines

The default option in this panel causes lines to print at the same width they are portrayed in the figure window. You can scale line width from 0 percent upwards

for printing using the **Scale By** field. To print lines at a particular point size, select **Custom**. All lines on the plot will be the same weight when you use the **Custom** option; the **Scale By** option respects relative line weight.

When you scale lines downward, you can prevent them from becoming too faint by setting the **Min Width** option to **Custom** and specifying a minimum line width in points in that field.

Text

The default is to print text in the same font and at the same size as it is in the figure. To change the font (for all text) select **Custom** and choose a new font from the drop-down list that is then enabled. Scale the font size using the **Scale By** option. To print text at a particular point size, select **Custom**. All text on the plot will be printed at the point size you specify when you use the **Custom** option; the **Scale By** option respects relative font size. You can specify the **Font Weight** (normal, light, demi, or bold) and **Font Angle** (normal, italic, or oblique) for all text as well, using the drop-down menus at the bottom of the **Text** panel.

Header

Type any text that you want to appear at the top of the printed figure in the **Header Text** edit field. If you want today's date and/or time appended to the header text, select the appropriate format from the **Date Style** popup menu. To specify and style the header font (which is independent of the font used in the figure), click the **Font** button and choose a font name, size, and style from the **Font** selection window that appears.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

There is no equivalent MATLAB command that sets line and text color depending on background color. Set the color of lines and text using the **set** function on either line or text objects in your figure.

This example sets all lines and text to black:

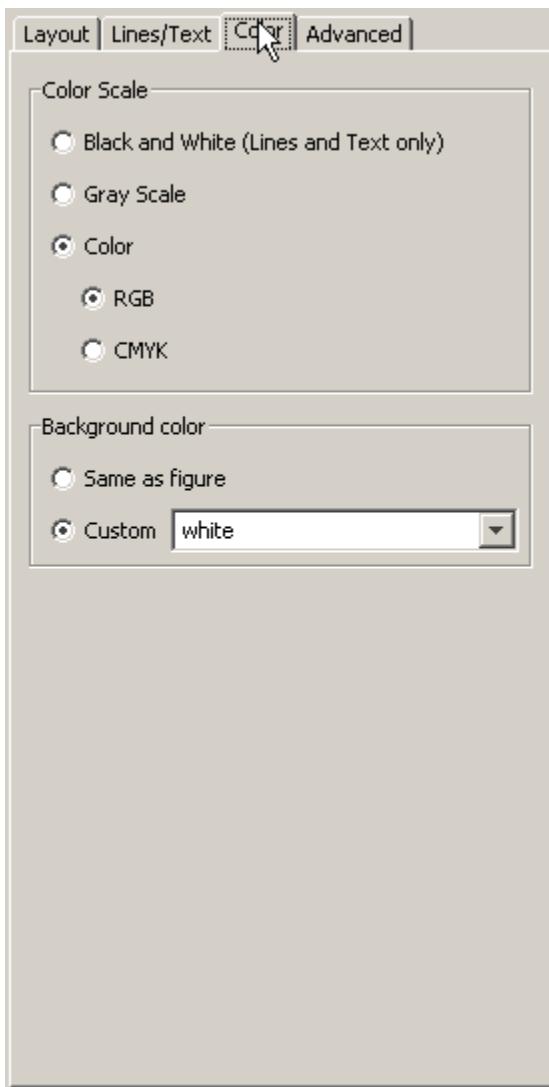
```
set(findobj('Type','line'),'Color','black');
set(findobj('Type','text'),'Color','black');
```

Specifying a Colorspace for Printing and Exporting

By default, color output is in the RGB color space (red, green, blue). If you plan to publish and print MATLAB figures using printing industry standard four-color separation, you might want to use the CMYK color space (cyan, magenta, yellow, black).

Using the Windows Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Color** tab to make changes to the color of all lines and text on the printed page. The controls for the **Color** tab are shown below:



You can print the contents of your figure in color, grayscale, or black-and-white by selecting the appropriate button in the panel. When you select **Color**, you can choose between an **RGB** (red/green/blue) or a **CMYK** (cyan/magenta/yellow/black) color specification, if your printer is capable of it.

Independently of the **Color Scale** controls, you can specify a **Background color** for printing. Select **Same as figure** to use the color used in the figure itself (default is gray), or specify a **Custom** color from the combo box popup menu. The choices are **black**, **white**, and several RGB color triplet values; you type any valid MATLAB colorspec in this field as well, such as **g**, **magenta**, or **.3 .4 .5**.

The background color you specify is respected even if you choose **Black and White** or **Gray Scale** in the **Color Scale** panel.

Using the Graphical User Interface on UNIX Platforms

- 1 Select **Print** from the figure window's **File** menu.
- 2 Click the **Appearance** tab.
- 3 In the **Color Appearance** panel, select **Color**.
- 4 Click **Print**.

On any platform, you can also indicate whether to print in color, grayscale or black-and-white with the Print Preview dialog box.

Using MATLAB Commands

Use the **-cmyk** option with the **print** function. This example prints the current figure in CMYK using a PostScript Level II color printer driver.

```
print -dpsc2 -cmyk
```

Excluding User Interface Controls from Printed Output

User interface controls are objects that you create and add to a figure. For example, you can add a button to a figure that, when clicked, conveniently runs another MATLAB file. By default, user interface controls are included in your printed or exported figure. This section shows how to exclude them.

Using the Graphical User Interface

- 1 Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu, and then select the **Advanced** tab.
- 2 Under **Miscellaneous**, clear the **Print UIControls** check box.
- 3 Click **Close**.

Using MATLAB Commands

Use the `-noui` switch. This example specifies a color PostScript driver and excludes UI controls.

```
print -dpsc -noui
```

This example exports the current figure to a color EPS file and excludes UI controls.

```
print -depsc -noui myfile.eps
```

Producing Uncropped Figures

In most cases, MATLAB crops the background tightly around the objects in the figure. Depending on the printer driver or file format you use, you might be able to produce uncropped output. An uncropped figure has increased background area and is often desirable for figures that contain UI controls.

The setting you make changes the PostScript `BoundingBox` property saved with the figure.

Using MATLAB Commands

Use the `-loose` option with the `print` function. On Windows platforms, the uncropped option is only available if you print to a file.

This example exports the current figure, uncropped, to an EPS file.

```
print -deps -loose myfile.eps
```

Troubleshooting

In this section...

- “Introduction” on page 7-65
- “Common Problems” on page 7-65
- “Printing Problems” on page 7-66
- “Exporting Problems” on page 7-69
- “General Problems” on page 7-72

Introduction

This section describes some common problems you might encounter when printing or exporting your figure. If you don't find your problem listed here, try searching the Knowledge Base maintained by MathWorks® Technical Support Department. Go to <http://www.mathworks.com/support> and enter a topic in the search field.

Common Problems

- **Printing Problems**
 - “Printer Drivers” on page 7-66
 - “Default Settings” on page 7-67
 - “Color vs. Black and White” on page 7-67
 - “Printer Selection” on page 7-68
 - “Rotated Text” on page 7-68
 - “SizeChangedFcn Warning” on page 7-68
- **Exporting Problems**
 - “Background Color” on page 7-69
 - “Default Settings” on page 7-69
 - “Microsoft Word” on page 7-69
 - “File Format” on page 7-70
 - “Size of Exported File” on page 7-71
 - “Making Movies” on page 7-71

- “Extended Operations” on page 7-71
- **General Problems**
 - “Background Color” on page 7-72
 - “Default Settings” on page 7-72
 - “Dimensions of Output” on page 7-72
 - “Axis and Tick Labels” on page 7-73
 - “UI Controls” on page 7-74
 - “Cropping” on page 7-74
 - “Text Object Font” on page 7-74
 - “Printing and Exporting Graphs with Legends” on page 7-74

Printing Problems

Printer Drivers

I am using a Windows printer driver and encountering problems such as segmentation violations, general protection faults, application errors, and unexpected output.

Try one of the following solutions:

- Check the table of `drivers` in the `print` reference page to see if there are other drivers you can try.
- Contact the printer vendor to obtain a different native printer driver. The behavior you are experiencing might occur only with certain versions of the native printer driver. If this doesn't help and you are on a Windows system, try reinstalling the drivers that were shipped with your Windows installation disk.
- Export the figure to a graphics-format file, and then import it into another application before printing it. For information about exporting MATLAB figures, see “Exporting to a File” on page 7-17.

PostScript Output

When I use the `print` function with the `-deps` switch, I receive this error message.

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

As the error message indicates, your figure was saved to a file. EPS is a graphics file format and cannot be sent to a printer using a printer driver. To send your figure directly to a printer, try using one of the PostScript driver switches. See the table of drivers in the print reference page. To print an EPS file, you must first import it into a word processor or other software program.

Default Settings

My printer uses a different default paper type than the MATLAB default type of letter. How can I change the default paper type so that I don't have to set it for each new figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default paper type to A4.

```
set(0, 'DefaultFigurePaperType', 'A4');
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperType`.

I set the paper orientation to landscape, but each time I go to print a new figure, the orientation setting is portrait again. How can I change the default orientation so that I won't have to set it for each new figure?

See the explanation for the previous question. Adding the following line to `startup.m` sets the default paper orientation to landscape.

```
set(0, 'DefaultFigurePaperOrient', 'landscape')
```

Color vs. Black and White

I want the lines in my figure to print in black, but they keep printing in color.

You must be using a color printer driver. You can specify a black-and-white driver using the `print` function or the Print Preview dialog box to force the lines for the current figure to print in black. See “Setting the Line and Text Color” on page 7-58 for instructions.

A white line in my figure keeps coming out black when I print it.

There are two things that can cause this to happen. Most likely, the line is positioned over a dark background. The MATLAB default is to invert your background to white when you print, and changes any white lines over the background to black. To avoid this,

retain your background color when you print. See “Setting the Background Color” on page 7-54.

The other possibility is that you are using a Windows printer driver and the printer is sending inaccurate color information to MATLAB.

I am using a color printer, but my figure keeps printing in black and white.

By default, MATLAB uses a black-and-white printer driver. You need to specify a color printer driver.

Printer Selection

I have more than one printer connected to my system. How do I specify which one to print my figure with?

You can use either the Print dialog box, or the MATLAB `print` function, specifying the printer with the `-P` switch. For instructions using either method, see “Selecting the Printer” on page 7-39.

Rotated Text

I have some rotated text in my figure. It looks fine on the screen, but when I print it, the resolution is poor.

You are probably using bitmapped fonts, which don't rotate well. Try using TrueType fonts instead.

SizeChangedFcn Warning

I get a warning about my `SizeChangedFcn` being used when I print my figure.

By default, MATLAB resizes your figure when converting it to printer coordinates. That means it calls any `SizeChangedFcn` you have created for the figure and issues a warning. You can avoid this warning by setting the figure to print at screen size.

Improper Printer Configuration

I get the following error message on my LINUX/UNIX system ‘Printing failure. There are no properly configured printers on the system.’

This might be a problem with the location of the `lpc` command on your system. If not present, create a symbolic link from `/usr/sbin/lpc` to wherever the `lpc` command resides on your system.

Exporting Problems

Background Color

I generated a figure with a black background and selected “Use figure color” from the **Copy Options** panel of the Preferences dialog box. But when I exported my figure, its background was changed to white.

You must have exported your figure to a file. The settings in **Copy Options** only apply to figures copied to the clipboard.

There are two ways to retain the displayed background color: use the Print Preview dialog box or set the **InvertHardCopy** property to **off**. See “Setting the Background Color” on page 7-54 for instructions on either method.

Default Settings

I want to export all of my figures using the same size. Is there some way to do this so that I don't have to set the size for each individual figure?

You can set the default value for any property by adding a line to **startup.m**. Adding the following line sets the default figure size to 4-by-3 inches.

```
set(0,'DefaultFigurePaperPosition',[0 0 4 3]);
```

In your call to **set**, combine the word **Default** with the name of the object **Figure** and the property name **PaperPosition**.

I use the clipboard to export my figures as metafiles. Is there some way to force all of my copy operations to use the metafile format?

On Windows systems, use the **Copy Options** panel of the Preferences dialog box. Any settings made here, including whether your figure is copied as a metafile or bitmap, apply to all copy operations. See “Exporting to the Windows or Macintosh Clipboard” on page 7-25 for instructions.

Microsoft Word

I exported my figure to an EPS file, and then tried to import it into my Word document. My printout has an empty frame with an error message saying that my EPS picture was not saved with a preview and will only print to a PostScript printer. How do I include a TIFF preview?

Use the `print` command with the `-tiff` switch. For example,

```
print -deps -tiff filename
```

If you print to a non-PostScript printer with Word, the preview image is used for printing. This is a low-resolution image that lacks the quality of an EPS graphic.

When I try to resize my figure in Word, its quality suffers.

You must have used a bitmap format. Bitmap files generally do not resize well. If you are going to export using a bitmap format, try to set the figure's size while it's still in MATLAB. See "Setting the Figure Size and Position" on page 7-41 for instructions.

As an alternative, you can use one of the vector formats, EMF or EPS. Figures exported in these formats can be resized in Word without affecting quality.

I exported my figure as an EMF to the clipboard. When I paste it into Word, some of the labels are printed incorrectly.

This problem occurs with some Microsoft Word and Windows versions. Try editing the labels in Word.

File Format

I tried to import my exported figure into a word processing document, but I got an error saying the file format is unrecognized.

There are two likely causes: you used the `print` function and forgot to specify the export format, or your word processing program does not support the export format. Include a format switch when you use the `print` function; simply including the file extension is not sufficient. For instructions, see "Exporting to a File" on page 7-17.

If this does not solve your problem, check what formats the word processor supports.

I tried to append a figure to an EPS file, and received an error message

You cannot append figures to an EPS file. The `-append` option is only valid for PostScript files, which should not be confused with EPS files. PostScript is a printer driver; EPS is a graphics file format.

Of the supported export formats, only HDF supports storing multiple figures, but you must use the `imwrite` function to append them. For an example, see the reference page for `imwrite`.

Size of Exported File

I've always used the EPS format to export my figures, but recently it started to generate huge files. Some of my files are now several megabytes!

Your graphics have probably become complicated enough that MATLAB is using the OpenGL renderer instead of the Painter's renderer. It does this to improve display time or to handle attributes that Painter's cannot, such as lighting. However, using OpenGL causes a bitmap to be stored in your EPS file, which with large figures leads to a large file.

There are two ways to fix the problem. You can specify the Painter's renderer when you export to EPS, or you can use a bitmap format, such as TIFF. The best renderer and type of format to use depend upon the figure. For information about the rendering methods and how to set them, see “Selecting a Renderer” on page 7-48.

Making Movies

I am using MATLAB functions to process a large number of frames. I would like these frames to be saved as individual files for later conversion into a movie. How can I do this?

Use `getframe` to capture the frames, `imwrite` to write them to a file, and `movie` to create a movie from the files. For more information about creating a movie from the captured frames, see the reference page for `movie`.

You can also save multiple figures to an AVI file. AVI files can be used for animated sequences that do not need MATLAB to run. However, they do require an AVI viewer. For more information, see “Export to Audio and Video” in the MATLAB Programming Fundamentals documentation.

Extended Operations

There are some export operations that cannot be performed using the Export dialog box.

You need to use the `print` function to do any of the following operations:

- Export to a supported file format not listed in the Export dialog box. The formats not available from the Export dialog box include HDF, some variations of BMP and PCX, and the raw data versions of PBM, PGM, and PPM.
- Specify a resolution.

- Specify one of the following options:
 - TIFF preview
 - Loose bounding box for EPS files
 - Compression quality for JPEG files
 - CMYK output on Windows platforms
- Perform batch exporting.

General Problems

Background Color

When I output my figure, its background is changed to white. How can I get it to have the displayed background color?

By default, when you print or export a figure, the background color inverts to white. There are two ways to retain the displayed background color: use the Print Preview dialog box or set the **InvertHardCopy** property to **off**. See “Setting the Background Color” on page 7-54 for instructions on either method.

If you are exporting your figure to the clipboard, you can also use the **Copy Options** panel of the Preferences dialog box. Setting the background here sets it for all figures copied to the clipboard.

Default Settings

I need to produce diagrams for publications. There is a list of requirements that I must meet for size of the figure, fonts types, etc. How can I do this easily and consistently?

You can set the default value for any property by adding a line to **startup.m**. As an example, the following line sets the default axes label font size to 12.

```
set(0,'DefaultAxesFontSize',12);
```

In your call to **set**, combine the word **Default** with the name of the object **Axes** and the property name **FontSize**.

Dimensions of Output

The dimensions of my output are huge. How can I make it smaller?

Check your settings for figure size and resolution, both of which affect the output dimensions of your figure.

The default figure size is 8-by-6 inches. You can use the Print Preview dialog box or the **PaperPosition** property to set the figure size. See “Setting the Figure Size and Position” on page 7-41.

The default resolution depends on the export format or printer driver used. For example, built-in MATLAB bitmap formats, like TIFF, have a default resolution of 150 dpi. You can change the resolution by using the **print** function and the **-r** switch. For default resolution values and instructions on how to change them, see “Setting the Resolution” on page 7-50.

I selected “Auto (actual size, centered)” from the Print Preview menu, but my output looks a little bigger, and my font looks different.

You probably output your figure using a higher resolution than your screen uses. Set your resolution to be the same as the screen's.

As an alternative, if you are exporting your figure, see if your application enables you to select a resolution. If so, import the figure at the same resolution it was exported with. For more information about resolution and how to set it when exporting, see “Setting the Resolution” on page 7-50.

Axis and Tick Labels

When I resize my figure below a certain size, my x-axis label and the bottom half of the x-axis tick labels are missing from the output.

Your figure size might be too small to accommodate the labels. Labels are positioned a fixed distance from the *x*-axis. Since the *x*-axis itself is positioned a relative distance away from the window's edge, the label text might not fit. Try using a larger figure size or smaller fonts. For instructions on setting the size of your figure, see “Setting the Figure Size and Position” on page 7-41. For information about setting font size, see the Text Properties properties page.

In my output, the x-axis has fewer ticks than it did on the screen.

MATLAB has rescaled your ticks because the size of your output figure is different from its displayed size. There are two ways to prevent this: select **Keep screen limits and ticks** from the **Advanced** tab of the Print Preview dialog box, or set the **XTickMode**,

`YTickMode`, and `ZTickMode` properties to `manual`. See “Setting the Axes Ticks and Limits” on page 7-52 for details.

UI Controls

My figure contains UI controls. How do I prevent them from appearing in my output?

Use the `print` function with the `-noui` switch. For details, see “Excluding User Interface Controls from Printed Output” on page 7-63.

Cropping

I can't output my figure using the uncropped setting (i.e., a loose BoundingBox).

Only PostScript printer drivers and the EPS export format support uncropped output. There is a workaround for Windows printer drivers, however. Using the `print` function, save your figure to a file that can be printed later. For an example see “Producing Uncropped Figures” on page 7-64.

Text Object Font

I have a problem with text objects when printing with a PostScript printer driver or exporting to EPS. The fonts are correct on the screen, but are changed in the output.

You have probably used a font that is not supported by EPS and PostScript. All unsupported fonts are converted to Courier.

Printing and Exporting Graphs with Legends

Why does the legend appear in a different location when I set legend Location to Best or BestOutside?

When you print or export a graph containing a legend that you placed in the figure using the `Location` option set to `Best` or `BestOutside`, MATLAB can reposition the legend in the output. This repositioning happens because the process of printing or exporting a figure can result in changes to certain properties that affect the choice of legend location.

To avoid potential repositioning of the legend in printed or exported output, set the legend position explicitly with a four-element vector. For more information on legend position, see the `legend` function.

Saving Figures

In this section...

- “Saving and Loading Graphs” on page 7-75
- “FIG-File Format” on page 7-76
- “Saving Figures From the Menu” on page 7-76
- “Saving to a Different Format — Exporting Figures” on page 7-77
- “Printing Figures” on page 7-78
- “Generating a MATLAB File to Recreate a Graph” on page 7-79

Saving and Loading Graphs

You can save and reload graphs using the `savefig` and `openfig` functions. For example, create a bar graph and save it to a file called `barGraph.fig` (the `.fig` extension is added automatically):

```
figure  
bar(randn(1,5), 'BarWidth', 0.5);  
savefig barGraph
```

You can reload the graph using `openfig`:

```
fig = openfig('barGraph');
```

MATLAB creates a new figure, a new axes, and a new bar object using the same data as the original objects. Most of the property values of the new objects are the same as the original objects.

However, the `Parent` and `Children` properties now contain the new object handles. Also, MATLAB applies any default or system values that are different from those in the environment in which you saved the figure.

For example, suppose you set a default figure color after saving the `barGraph.fig` file. MATLAB uses the default when creating the new figure.

Accessing the New Object

To get the handle to the new bar object, use `findobj` and the figure handle returned by `openfig`:

```
h = findobj(fig, 'Type', 'bar');
```

You can use `h2` to set and get properties on the new bar object:

```
h.BarWidth
```

```
ans =
```

```
0.5000
```

For more information on finding objects, see “Find Objects” on page 16-5

FIG-File Format

The MATLAB FIG-file is a binary format to which you can save figures so that they can be opened in subsequent MATLAB sessions. The whole figure, including graphs, graph data, annotations, data tips, menus and other uicontrols, is saved. (The only exception is highlighting created by data brushing.) These files have a `.fig` filename extension.

If you want to save the figure in a format that can be used by another application, see “Saving to a Different Format — Exporting Figures” on page 7-77.

Saving Figures From the Menu

To save a graph in a figure file,

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

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

You can also use the `saveas` command.

Use the `savefig` command to create backward compatible FIG-files.

Opening a Figure File

To open a figure file, perform these steps:

- 1 Select **Open** from the **File** menu or click the **Open** button on the toolbar.
- 2 Select the figure file you want to open and click **OK**.

The figure file appears in a new figure window.

You can also use the **open** command.

Saving to a Different Format — Exporting Figures

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

- 1 Select **Export Setup** from the **File** menu. This dialog provides options you can specify for the output file, such as the figure size, fonts, line size and style, and output format.
- 2 Select **Export** from the Export Setup dialog. A standard Save As dialog appears.
- 3 Select the graphic format from the list of formats in the **Save as type** drop-down menu. This selects the format of the exported file and adds the standard filename extension given to files of that type.
- 4 Enter the name you want to give the file, less the extension.
- 5 Click **Save**.

Export from Save As Dialog

You can use the Save As dialog to export a figure to a standard file format:

- 1 Select **Save As** from the **File** menu.
- 2 Enter a file name in the **File name** text field
- 3 Select a file type from the **Save as type** drop-down menu.
- 4 Click **Save**.

You can also use the **saveas** function to export figure to specific file formats. However, the **saveas** function and the Save As dialog do not produce identical results:

- The **Save As** dialog produces images at screen resolution and at screen size.
- The `saveas` function uses the default resolution of 150 DPI and honors the figure `PaperPosition` and `PaperSizeMode` properties to determine the size of the image.

Copying a Figure to the Clipboard

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

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

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

Printing Figures

Before printing a figure,

- 1 Select **Print Preview** from the **File** menu to set printing options, including plot size and position, and paper size and orientation.

The **Print Preview** dialog box opens.
- 2 Make changes in the dialog box. Changes you can make are arranged by tabs on the left-hand pane. If you want the printed output to match the annotated plot you see on the screen exactly,
 - a On the **Layout** tab, click **Auto (Actual Size, Centered)**.
 - b On the **Advanced** tab, click **Keep screen limits and ticks**.

For information about other options for print preview, click the **Help** button in the dialog box.

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

You can also use the `print` command.

Generating a MATLAB File to Recreate a Graph

You can generate a MATLAB file from a graph, which you can then use to reproduce the graph. This feature is particularly useful for capturing modifications you make using the plot tools.

- 1 Select **Generate Code** from the **File** menu.

The generated code displays in the MATLAB Editor.

- 2 Save the file using **Save As** from the Editor **File** menu.

Running the Saved File

Generated files do not store the data necessary to recreate the graph, so you must supply the data arguments. The data arguments do not need to be identical to the original data. Comments at the beginning of the file state the type of data expected.

For example, the following statements illustrate a case where three input vectors are required.

```
function createfigure(yvector1)
    %CREATEFIGURE(YVECTOR1)
    % YVECTOR1: bar yvector

    % Auto-generated by MATLAB on 08-Jul-2014 17:19:46

    % Create figure
    figure1 = figure;

    % Create axes
    axes1 = axes('Parent',figure1,'XTick',[1 2 3 4 5]);
    box(axes1,'on');
    hold(axes1,'on');

    % Create bar
    bar(yvector1,'BarWidth',0.5);
```


Axes Active Position

Axes Resize to Accommodate Titles and Labels

In this section...

[“Axes Layout” on page 8-2](#)

[“Properties Controlling Axes Size” on page 8-2](#)

[“Using OuterPosition as the ActivePositionProperty” on page 8-5](#)

[“ActivePositionProperty = OuterPosition” on page 8-5](#)

[“ActivePositionProperty = Position” on page 8-6](#)

[“Axes Resizing in Subplots” on page 8-7](#)

Axes Layout

Axes properties control the layout of titles and axis labels within the figure. You can control which dimensions axes can change to accommodate the titles and labels by setting the appropriate properties.

Properties Controlling Axes Size

When you create a graph, MATLAB creates an axes to display the graph. The axes is sized to fit in the figure and automatically resizes as you resize the figure. MATLAB applies the automatic resize behavior only when the axes `Units` property is set to `normalized` (the default).

You can control the resize behavior of the axes using the following axes properties:

- `OuterPosition` — Defines the boundary of the axes including the axis labels, title, and a margin. For figures with only one axes, `OuterPosition` encompasses the interior of the figure.
- `Position` — The boundary of the axes, excluding the tick marks and labels, title, and axis labels.
- `ActivePositionProperty` — Specifies whether to use the `OuterPosition` or the `Position` property as the size to preserve when resizing the figure containing the axes.
- `TightInset` — The margins MATLAB adds to the width and height of the `Position` property to include text labels, title, and axis labels. This property is read only.

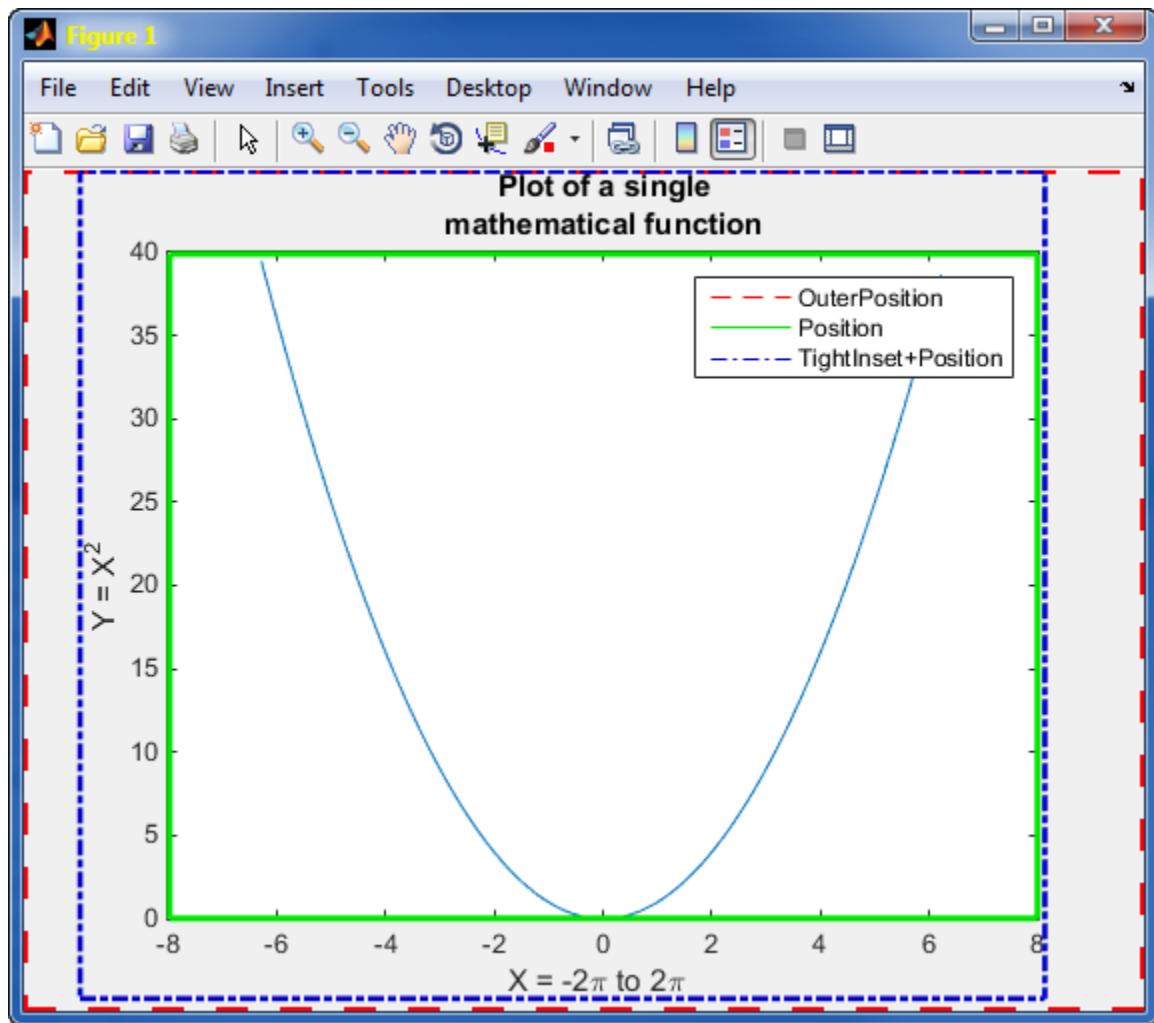
- **Units** — Keep this property set to 'normalized' to enable automatic axes resizing.

Note: MATLAB changes only the current axes' properties by default. If your plot has multiple axes, MATLAB does not automatically resize any secondary axes.

The following graph shows the areas defined by the `OuterPosition`, the `Position` expanded by `TightInset`, and the `Position` properties.

When you add axis labels and a title, the `TightInset` changes to accommodate the additional text.

8 Axes Active Position

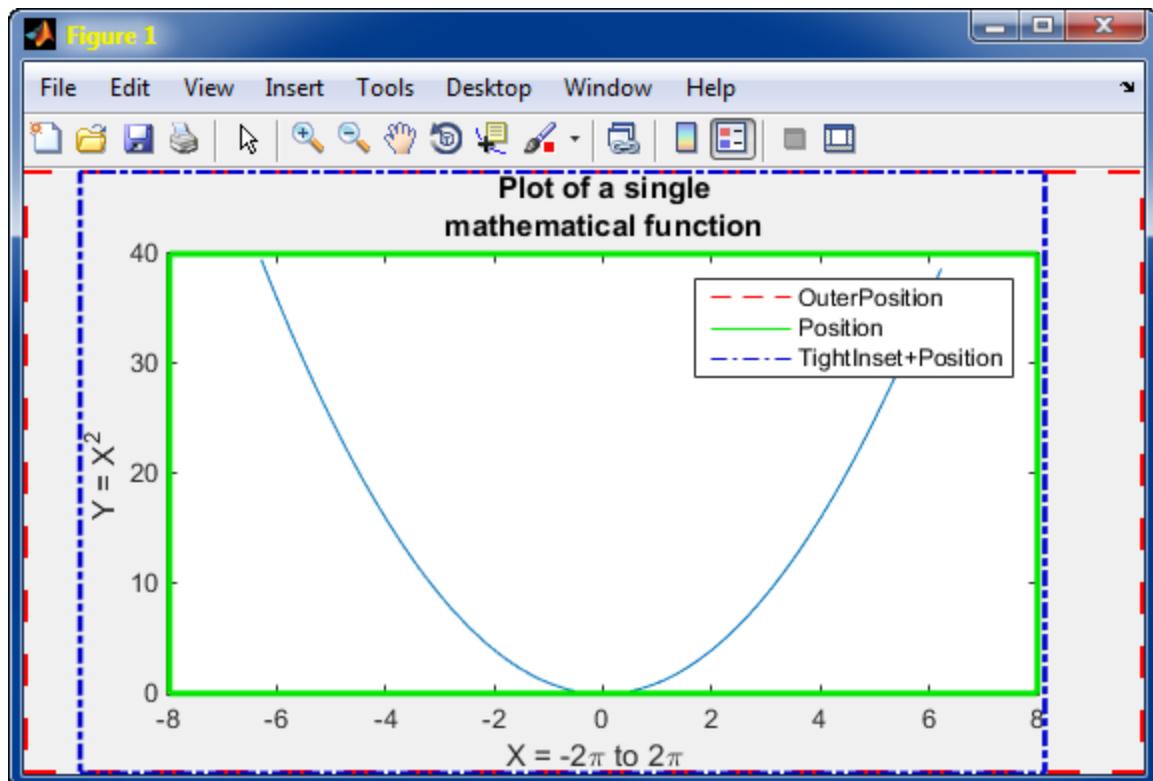


The size of the rectangle defined by the `TightInset` and `Position` properties includes all graph text. The `Position` and `OuterPosition` properties remain unchanged.

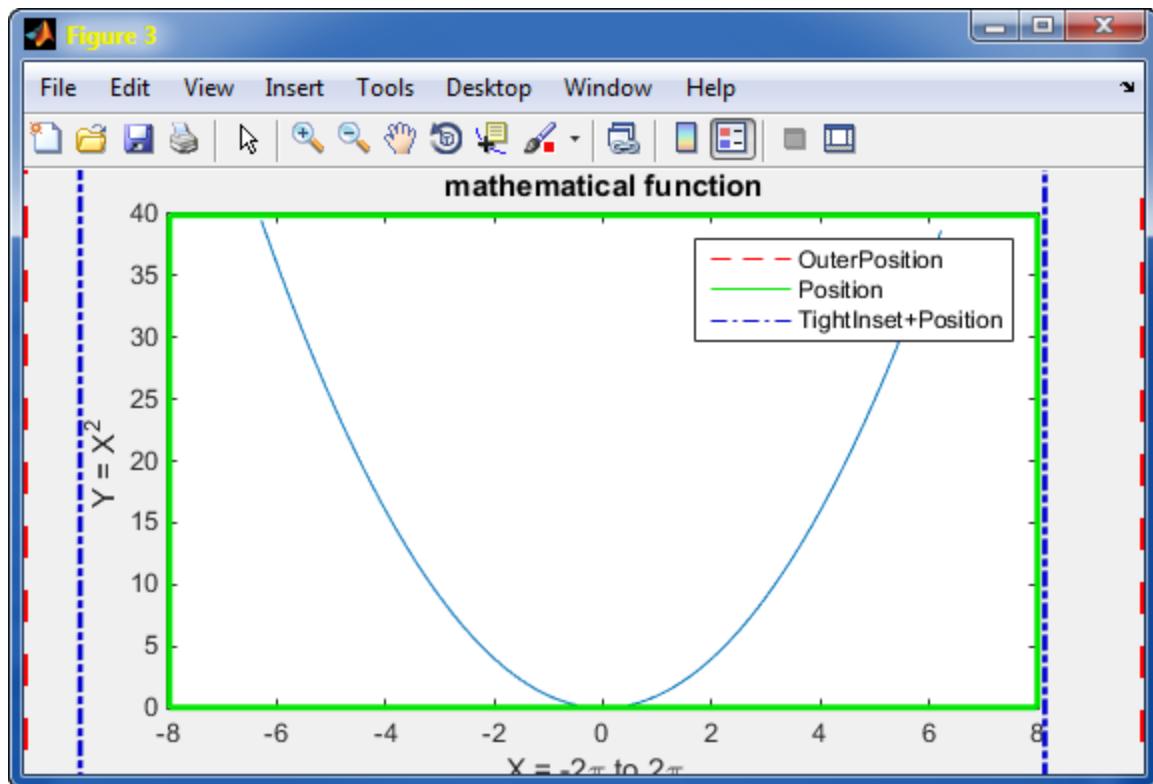
Using OuterPosition as the ActivePositionProperty

As you resize the figure, MATLAB maintains the area defined by the `TightInset` and `Position` so that the text is not cut off. Compare the next two graphs, which have both been resized to the same figure size.

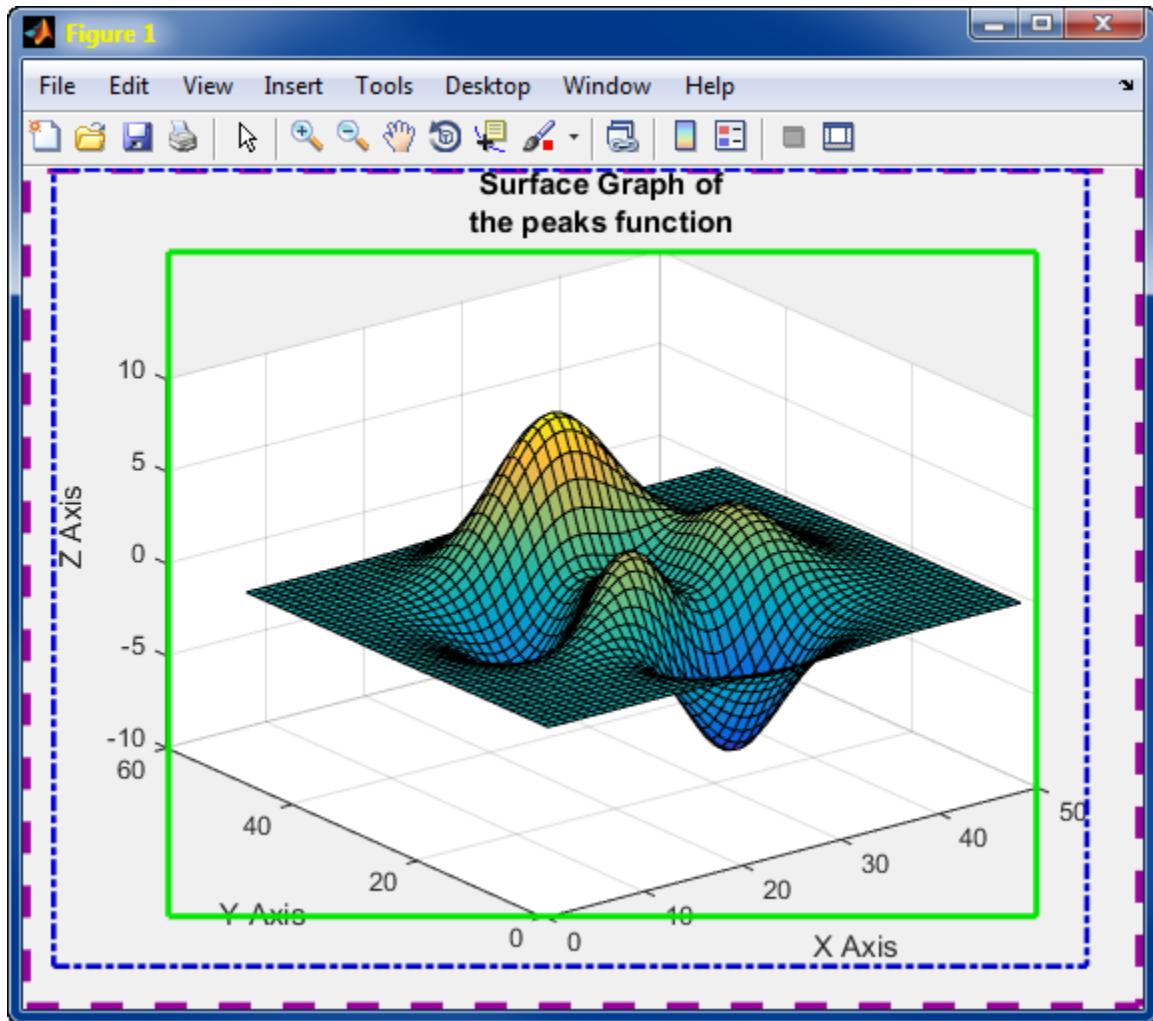
ActivePositionProperty = OuterPosition



ActivePositionProperty = Position



The following figure shows how the default property values apply to 3-D graphs.

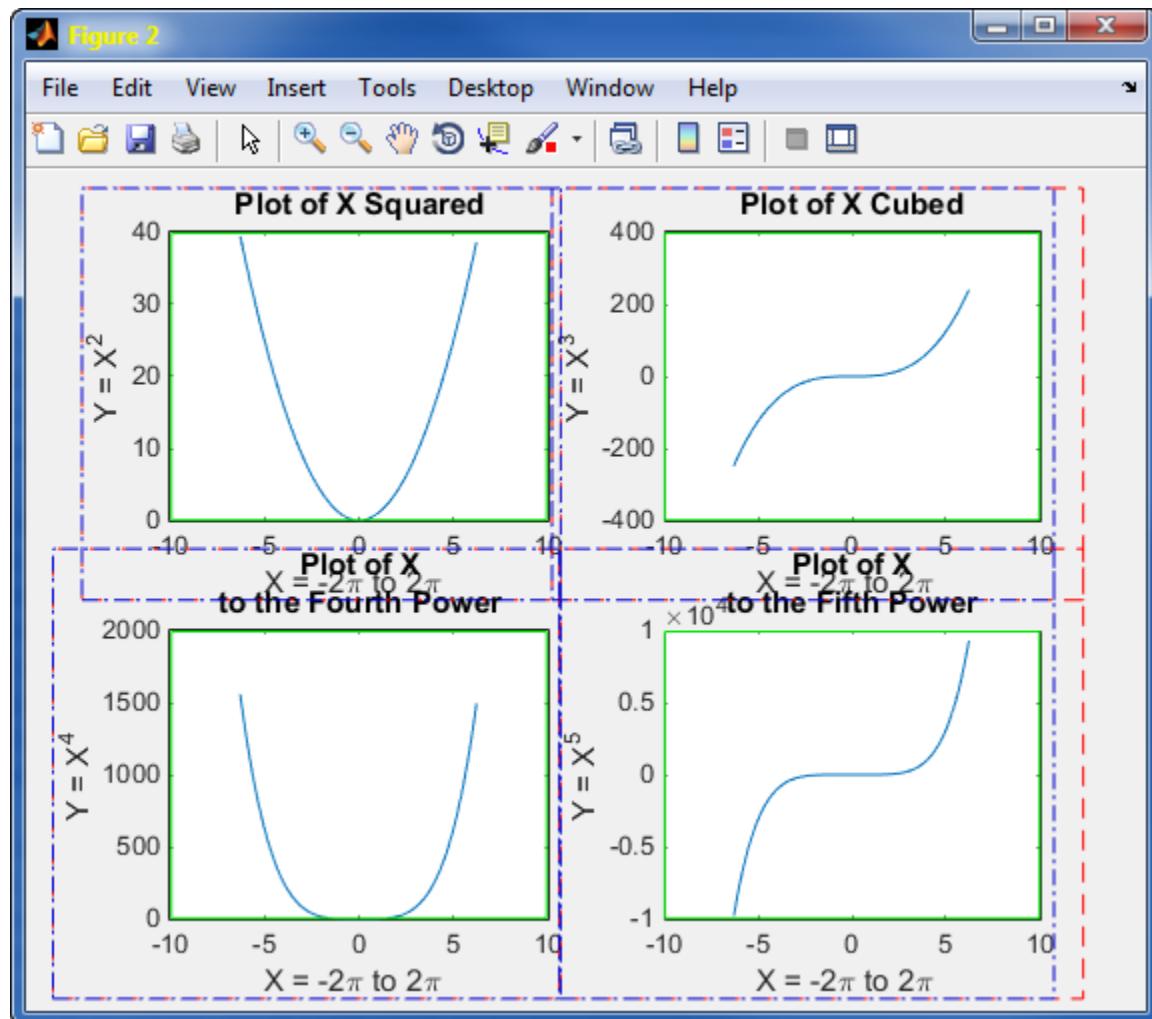


Axes Resizing in Subplots

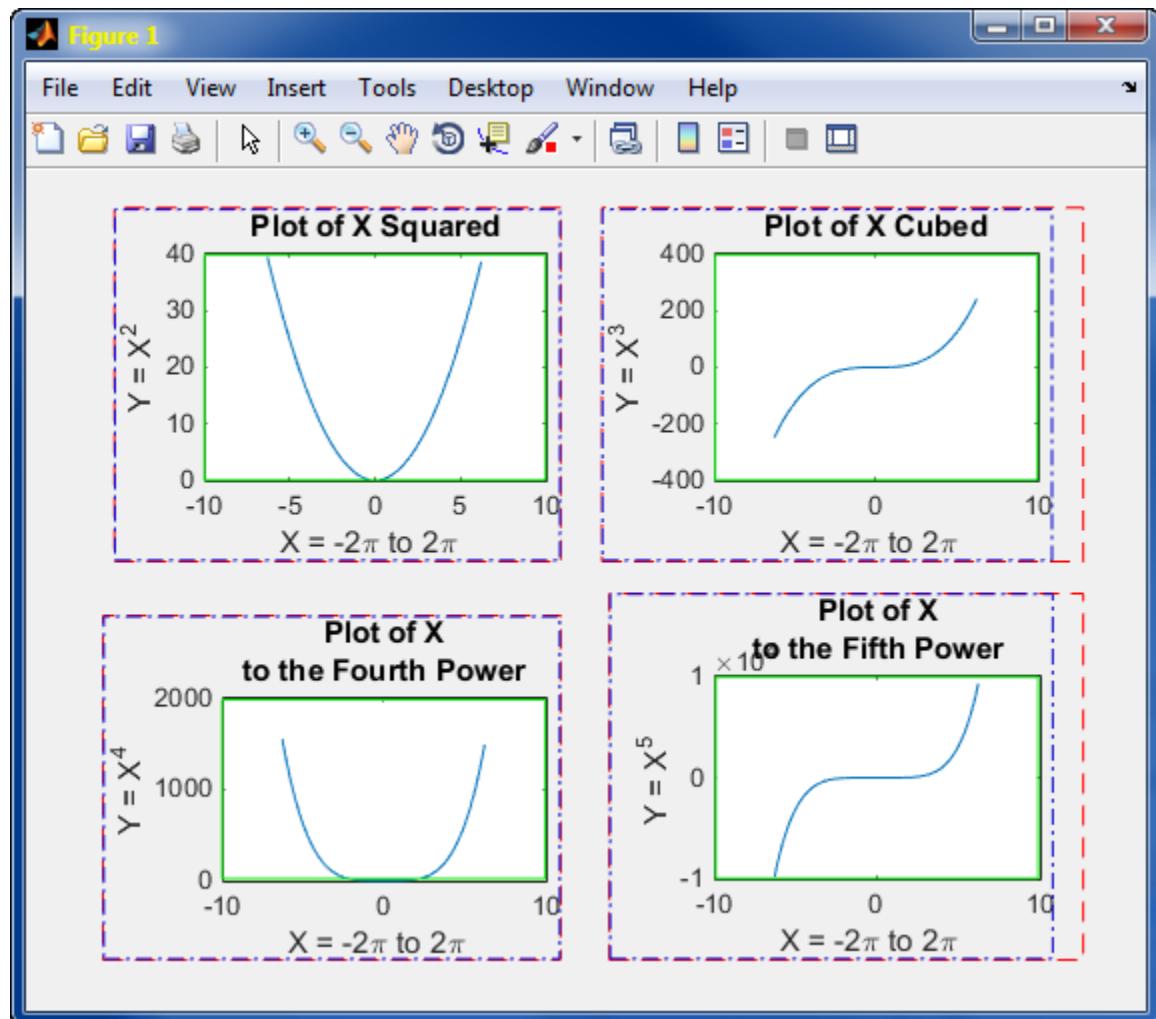
When there are multiple axes in a figure use the `ActivePositionProperty` to prevent titles and labels from being overwritten .

The following figure illustrates how MATLAB resizes the axes to accommodate the multiline titles on the lower two axes when the `ActivePositionProperty` is '`position`'.

8 Axes Active Position



Setting the `ActivePositionProperty` property to '`outerposition`' reduces the height of the two upper axes to provide better spacing of the subplots.



Controlling Graphics Output

- “Control Graph Display” on page 9-2
- “Prepare Figures and Axes for Graphs” on page 9-5
- “Use newplot to Control Plotting” on page 9-9
- “Responding to Hold State” on page 9-12
- “Prevent Access to Figures and Axes” on page 9-14

Control Graph Display

In this section...

[“What You Can Control” on page 9-2](#)

[“Targeting Specific Figures and Axes” on page 9-2](#)

What You Can Control

MATLAB allows many figure windows to be open simultaneously during a session. You can control which figures and which axes MATLAB uses to display the result of plotting functions. You can also control to what extent MATLAB clears and resets the properties of the targeted figures and axes.

You can modify the way MATLAB plotting functions behave and you can implement specific behaviors in plotting functions that you write.

Consider these aspects:

- Can you prevent a specific figure or axes from becoming the target for displaying graphs?
- What happens to an existing graph when you plot more data to that graph? Is the existing graph replaced or are new graphics objects added to the existing graph?

Targeting Specific Figures and Axes

By default, MATLAB plotting functions display graphs in the current figure and current axes (the objects returned by `gcf` and `gca` respectively). You can direct the output to another figure and axes by:

- Explicitly specifying the target axes with the plotting function.
- Making the target axes the current axes.

Specify the Target Axes

Suppose you create a figure with four axes and save the handles in the array `ax`:

```
for k = 1:4
    ax(k) = subplot(2,2,k);
end
```

Call `plot` with the axes handle as the first argument:

```
plot(ax(1),1:10)
```

For plotting functions that do not support the axes first argument, set the `Parent` property:

```
t = 0:pi/5:2*pi;
patch(sin(t),cos(t),'y','Parent',ax(2))
```

Make the Target Current

To specify a target, you can make a figure the current figure and an axes in that figure the current axes. Plotting functions use the current figure and its current axes by default. If the current figure has no current axes, MATLAB creates one.

If `fig` is the handle to a figure, then the statement

```
figure(fig)
```

- Makes `fig` the current figure.
- Restacks `fig` to be the frontmost figure displayed.
- Makes `fig` visible if it was not (sets the `Visible` property to `on`).
- Updates the figure display and processes any pending callbacks.

The same behavior applies to axes. If `ax` is the handle to an axes, then the statement

```
axes(ax)
```

- Makes `ax` the current axes.
- Restacks `ax` to be the frontmost axes displayed.
- Makes `ax` visible if it was not.
- Updates the figure containing the axes and process any pending callbacks.

Make Figure or Axes Current Without Changing Other State

You can make a figure or axes current without causing a change in other aspects of the object state. Set the root `CurrentFigure` property or the figure object's `CurrentAxes` property to the handle of the figure or axes that you want to target.

If `fig` is the handle to an existing figure, the statement

```
r = groot;  
r.CurrentFigure = fig;
```

makes `fig` the current figure. Similarly, if `ax` is the handle of an axes object, the statement

```
fig.CurrentAxes = ax;
```

makes it the current axes, if `fig` is the handle of the axes' parent figure.

Prepare Figures and Axes for Graphs

In this section...

[“Behavior of MATLAB Plotting Functions” on page 9-5](#)

[“How the NextPlot Properties Control Behavior” on page 9-5](#)

[“Control Behavior of User-Written Plotting Functions” on page 9-7](#)

Behavior of MATLAB Plotting Functions

MATLAB plotting functions either create a new figure and axes if none exist, or reuse an existing figure and axes. When reusing existing axes, MATLAB

- Clears the graphics objects from the axes.
- Resets most axes properties to their default values.
- Calculates new axes limits based on the new data.

When a plotting function creates a graph, the function can:

- Create a figure and an axes for the graph and set necessary properties for the particular graph (default behavior if no current figure exists)
- Reuse an existing figure and axes, clearing and resetting axes properties as required (default behavior if a graph exists)
- Add new data objects to an existing graph without resetting properties (if `hold` is on)

The `NextPlot` figure and axes properties control the way that MATLAB plotting functions behave.

How the NextPlot Properties Control Behavior

MATLAB plotting functions rely on the values of the figure and axes `NextPlot` properties to determine whether to add, clear, or clear and reset the figure and axes before drawing the new graph. Low-level object-creation functions do not check the `NextPlot` properties. They simply add the new graphics objects to the current figure and axes.

This table summarizes the possible values for the `NextPlot` properties.

NextPlot	Figure	Axes
new	Creates a new figure and uses it as the current figure.	Not an option for axes.
add	Adds new graphics objects without clearing or resetting the current figure. (Default)	Adds new graphics objects without clearing or resetting the current axes.
replacechildren	Removes all axes objects whose handles are not hidden before adding new objects. Does not reset figure properties. Equivalent to <code>clf</code> .	Removes all axes child objects whose handles are not hidden before adding new graphics objects. Does not reset axes properties. Equivalent to <code>cla</code> .
replace	Removes all axes objects and resets figure properties to their defaults before adding new objects. Equivalent to <code>clf reset</code> .	Removes all child objects and resets axes properties to their defaults before adding new objects. Equivalent to <code>cla reset</code> . (Default)

Plotting functions call the `newplot` function to obtain the handle to the appropriate axes.

The Default Scenario

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 y.
- 2 Determines that MATLAB can draw into the current figure without modifying the figure. If there is no current figure, `newplot` creates one, but does not recheck its `NextPlot` property
- 3 Checks the value of the current axes' `NextPlot` property (which is, `replace`), deletes all graphics objects from the axes, resets all axes properties (except `Position` and `Units`) to their defaults, and returns the handle of the current axes. If there is no current axes, `newplot` creates one, but does not recheck its `NextPlot` property.
- 4 Deletes all graphics objects from the axes, resets all axes properties (except `Position` and `Units`) to their defaults, and returns the handle of the current axes.

If there is no current axes, `newplot` creates one, but does not recheck its `NextPlot` property.

hold Function and NextPlot Properties

The `hold` function provides convenient access to the `NextPlot` properties. When you want add objects to a graph without removing other objects or resetting properties use `hold on`:

- `hold on` — Sets the figure and axes `NextPlot` properties to `add`. Line graphs continue to cycle through the `ColorOrder` and `LineStyleOrder` property values.
- `hold off` — Sets the axes `NextPlot` property to `replace`

Use the `ishold` to determine if `hold` is on or off.

Control Behavior of User-Written Plotting Functions

MATLAB provides the `newplot` function to simplify writing plotting functions 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. Place `newplot` at the beginning of any function that calls object creation functions.

When your function calls `newplot`, `newplot` first queries the figure `NextPlot` property. Based on the property values `newplot` then takes the action described in the following table based on the property value.

Figure NextPlot Property Value	newplot Function
No figures exist	Creates a figure and makes this figure the current figure.
<code>add</code>	Makes the figure the current figure.
<code>new</code>	Creates a new figure and makes it the current figure.
<code>replacechildren</code>	Deletes the figure's children (axes objects and their descendants) and makes this figure the current figure.
<code>replace</code>	Deletes the figure's children, resets the figure's properties to their defaults, and makes this figure the current figure.

Then `newplot` checks the current axes' `NextPlot` property. Based on the property value `newplot` takes the action described in the following table.

Axes <code>NextPlot</code> Property Value	<code>newplot</code> Function
No axes in current figure	Creates an axes and makes it the current axes
<code>add</code>	Makes the axes the current axes and returns its handle.
<code>replacechildren</code>	Deletes the axes' children and makes this axes the current axes.
<code>replace</code>	Deletes the axes' children, reset the axes' properties to their defaults, and makes this axes the current axes.

Use newplot to Control Plotting

This example shows how to prepare figures and axes for user-written plotting functions.

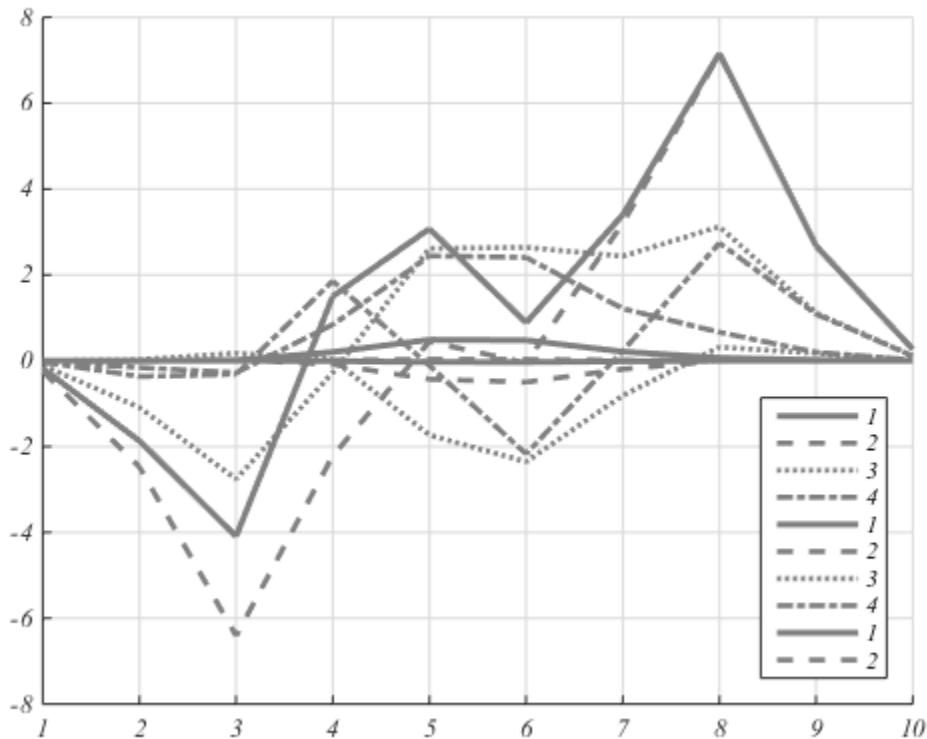
Use `newplot` to manage the output from specialized plotting functions. The `myPlot2D` function:

- Customizes the axes and figure appearance for a particular publication requirement.
- Uses revolving line styles and a single color for multiline graphs.
- Adds a legend with specified display names.

```
function myPlot2D(x,y)
    % Call newplot to get the axes handle
    cax = newplot;
    % Customize axes
    cax.FontName = 'Times';
    cax.FontAngle = 'italic';
    % Customize figure
    fig = cax.Parent;
    fig.MenuBar= 'none';
    % Call plotting commands to
    % produce custom graph
    hLines = line(x,y, ...
        'Color',[.5,.5,.5],...
        'LineWidth',2);
    lso = [ '-' ; '--' ; ':' ; '-.' ];
    setLineStyle(hLines)
    grid on
    legend('show','Location','SouthEast')
    function setLineStyle(hLines)
        style = 1;
        for ii = 1:length(hLines)
            if style > length(lso)
                style = 1;
            end
            hLines(ii).LineStyle = lso(style,:);
            hLines(ii).DisplayName = num2str(style);
            style = style + 1;
        end
    end
end
```

This graph shows typical output for the `myPlot2D` function:

```
x = 1:10;
y = peaks(10);
myPlot2D(x,y)
```



The `myPlot2D` function shows the basic structure of a user-written plotting functions:

- Call `newplot` to get the handle of the target axes and to apply the settings of the `NextPlot` properties of the axes and figure.
- Use the returned axes handle to customize the axes or figure for this specific plotting function.
- Call plotting functions (for example, `line` and `legend`) to implement the specialized graph.

Because `myPlot2D` uses the handle returned by `newplot` to access the target figure and axes, this function:

- Adheres to the behavior of MATLAB plotting functions when clearing the axes with each subsequent call.
- Works correctly when `hold` is set to `on`

The default settings for the `NextPlot` properties ensure that your plotting functions adhere to the standard MATLAB behavior — reuse the figure window, but clear and reset the axes with each new graph.

Responding to Hold State

This example shows how to test for `hold` state and respond appropriately in user-defined plotting functions.

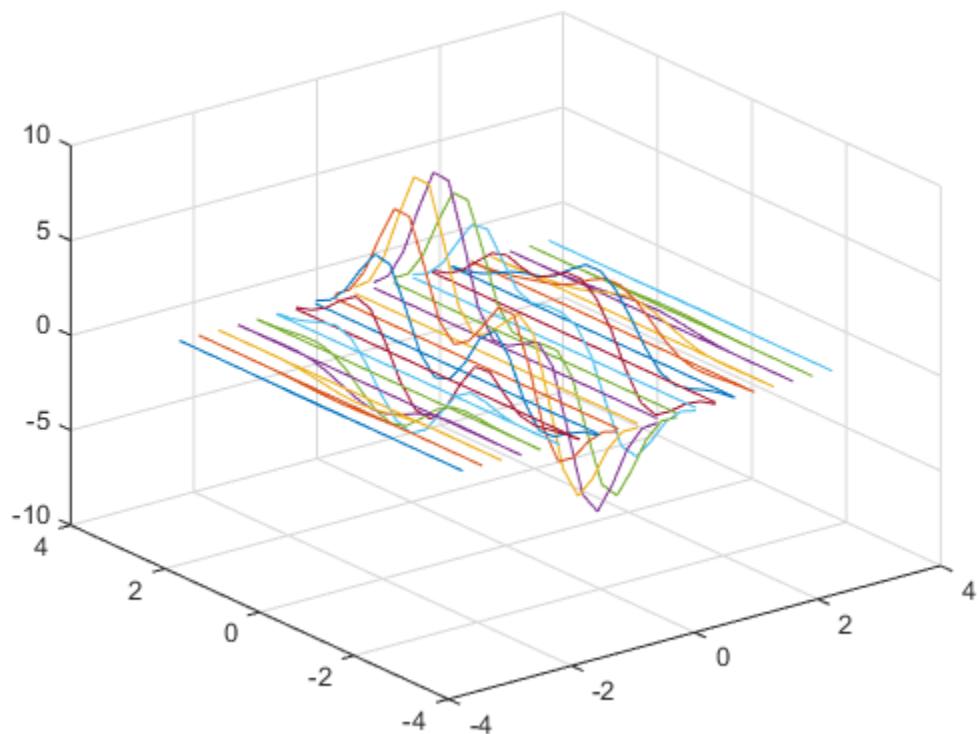
Plotting functions usually change various axes parameters to accommodate different data. The `myPlot3D` function:

- Uses a 2-D or 3-D view depending on the input data.
- Respects the current `hold` state, to be consistent with the behavior of MATLAB plotting functions.

```
function myPlot3D(x,y,z)
    % Call newplot to get the axes handle
    cax = newplot;
    % Save current hold state
    hold_state = ishold;
    % Call plotting commands to
    % produce custom graph
    if nargin == 2
        line(x,y);
        % Change view only if hold is off
        if ~hold_state
            view(cax,2)
        end
    elseif nargin == 3
        line(x,y,z);
        % Change view only if hold is off
        if ~hold_state
            view(cax,3)
        end
    end
    grid on
end
```

For example, the first call to `myPlot3D` creates a 3-D graph. The second call to `myPlot3D` adds the 2-D data to the 3-D view because `hold` is `on`.

```
[x,y,z] = peaks(20);
myPlot3D(x,y,z)
hold on
myPlot3D(x,y)
```



Prevent Access to Figures and Axes

In this section...

[“Why Prevent Access” on page 9-14](#)

[“How to Prevent Access” on page 9-14](#)

Why Prevent Access

In some situations it is important to prevent particular figures or axes from becoming the target for graphics output. That is, prevent them from becoming the current figure, as returned by `gcf`, or the current axes, as returned by `gca`.

You might want to prevent access to a figure containing the controls that implement a user interface. Or, you might want to prevent access to an axes that is part of an application program accessed only by the application.

How to Prevent Access

Prevent MATLAB functions from targeting a particular figure or axes by removing their handles from the list of visible handles.

Two properties control handle visibility: `HandleVisibility` and `ShowHiddenHandles`

`HandleVisibility` is a property of all graphics objects. It controls the visibility of the object's handle to three possible values:

- `on` — You can obtain the object's handle with functions that return handles, such as (`gcf`, `gca`, `gco`, `get`, and `findobj`). This is the default behavior.
- `callback` — The object's handle is visible only within the workspace of a callback function.
- `off` — The handle is hidden from all functions executing in the command window and in callback functions.

Properties Affected by Handle Visibility

When an object's `HandleVisibility` is set to `callback` or `off`:

- The object's handle does not appear in its parent's `Children` property.

- Figures do not appear in the root's `CurrentFigure` property.
- Axes do not appear in the containing figure's `CurrentAxes` property.
- Graphics objects do not appear in the figure's `CurrentObject` property.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, functions that obtain handles by searching the object hierarchy cannot return the handle. These functions include `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Values Returned by `gca` and `gcf`

When a hidden-handle figure is topmost on the screen, but has visible-handle figures stacked behind it, `gcf` returns the topmost visible-handle figure in the stack. The same behavior is true for `gca`. If no visible-handle figures or axes exist, calling `gcf` or `gca` creates one.

Access Hidden-Handle Objects

The root `ShowHiddenHandles` property enables and disables handle visibility control. By default, `ShowHiddenHandles` is `off`, which means MATLAB follows the setting of every object's `HandleVisibility` property.

Setting `ShowHiddenHandles` to `on` is equivalent to setting the `HandleVisibility` property of all objects in the graphics hierarchy to `on`.

Note: Axes title and axis label text objects are not children of the axes. To access the handles of these objects, use the axes `Title`, `XLabel`, `YLabel`, and `ZLabel` properties.

The `close` function also allows access to hidden-handle figures using the `hidden` option. For example:

```
close('hidden')
```

closes the topmost figure on the screen, even if its handle is hidden.

Combining `all` and `hidden` options:

```
close('all','hidden')
```

closes all figures.

Handle Validity Versus Handle Visibility

All handles remain valid regardless of the state of their HandleVisibility property. If you have assigned an object handle to a variable, you can always set and get its properties using that handle variable.

Default Values

- “Default Property Values” on page 10-2
- “Default Values for Automatically Calculated Properties” on page 10-6
- “How MATLAB Finds Default Values” on page 10-8
- “Factory-Defined Property Values” on page 10-9
- “Define Default Line Styles” on page 10-10
- “Multilevel Default Values” on page 10-12

Default Property Values

In this section...

- “Predefined Values for Properties” on page 10-2
- “Specify Default Values” on page 10-2
- “Where in Hierarchy to Define Default” on page 10-3
- “List Default Values” on page 10-3
- “Set Properties to the Current Default” on page 10-4
- “Remove Default Values” on page 10-4
- “Set Properties to Factory-Defined Values” on page 10-4
- “List Factory-Defined Property Values” on page 10-4
- “Reserved Words” on page 10-5

Predefined Values for Properties

Nearly all graphics object properties have predefined values. Predefined values originate from two possible sources:

- Default values defined on an ancestor of the object
- Factory values defined on the root of the graphics object hierarchy

Users can create default values for an object property, which take precedence over the factory-defined values. Objects use default values when:

- Created in a hierarchy where an ancestor defines a default value
- Parented into a hierarchy where an ancestor defines a default value

Specify Default Values

Define a default property value using a string with these three parts:

`'default' ObjectType propertyName`

- The word **default**
- The object type (for example, **Line**)

- The property name (for example, `LineWidth`)

A string that specified the default line `LineWidth` would be:

```
'defaultLineLineWidth'
```

Use this string to specify the default value. For example, to specify a default value of 2 points for the line `LineWidth` property, use the statement:

```
set(groot, 'defaultLineLineWidth', 2)
```

The string `defaultLineLineWidth` identifies the property as a line property. To specify the figure color, use `defaultFigureColor`.

```
set(groot, 'defaultFigureColor', 'b')
```

Where in Hierarchy to Define Default

In general, you should define a default value on the root level so that all subsequent plotting function use those defaults. Specify the root in `set` and `get` statements using the `groot` function, which returns the handle to the root.

You can define default property values on three levels:

- Root — values apply to objects created in during MATLAB session
- Figure — use for default values applied to children of the figure defining the defaults.
- Axes — use for default values applied only to children of the axes defining the defaults and only when using low-level functions (`light`, `line`, `.patch`, `rectangle`, `surface`, `text`, and the low-level form of `image`).

For example, specify a default figure color only on the root level.

```
set(groot, 'defaultFigureColor', 'b')
```

List Default Values

Use `get` to determine what default values are currently set on any given object level:

```
get(groot, 'default')
```

returns all default values set in your current MATLAB session.

Set Properties to the Current 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(groot, 'defaultSurfaceEdgeColor', 'k')
h = surface(peaks);
set(gcf, 'defaultSurfaceEdgeColor', 'g')
set(h, 'EdgeColor', 'default')
```

Because a default value for surface EdgeColor exists on the figure level, MATLAB encounters this value first and uses it instead of the default EdgeColor defined on the root.

Remove Default Values

Specifying a property value of 'remove' gets rid of user-defined default values. The statement

```
set(groot, 'defaultSurfaceEdgeColor', 'remove')
```

removes the definition of the default surface EdgeColor from the root.

Set Properties to Factory-Defined Values

Specifying a property value of 'factory' sets the property to its factory-defined value. 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')
```

List Factory-Defined Property Values

You can list factory values:

- `get(groot, 'factory')` — List all factory-defined property values for all graphics objects

- `get(groot, 'factoryObjectType')` — List all factory-defined property values for a specific object
- `get(groot, 'factoryObjectTypePropertyName')` — List factory-defined value for the specified property.

Reserved Words

Setting a property value to `default`, `remove`, or `factory` produces the effects described in the previous sections. To set a property to one of these words (for example, a text `String` property set to the word `default`), precede the word with the backslash character:

```
h = text('String', '\default');
```

Default Values for Automatically Calculated Properties

In this section...

[“What Are Automatically Calculated Properties” on page 10-6](#)

[“Default Values for Automatically Calculated Properties” on page 10-6](#)

What Are Automatically Calculated Properties

When you create a graph, MATLAB sets certain property values appropriately for the particular graph. These properties, such as those controlling axis limits and the figure renderer, have an associated mode property.

The mode property determines if MATLAB calculates a value for the property (mode is `auto`) or if the property uses a specified value (mode is `manual`). For more information on automatically calculated properties, see “Automatically Calculated Properties” on page 19-16.

Default Values for Automatically Calculated Properties

Defining a default value for an automatically calculated property requires two steps:

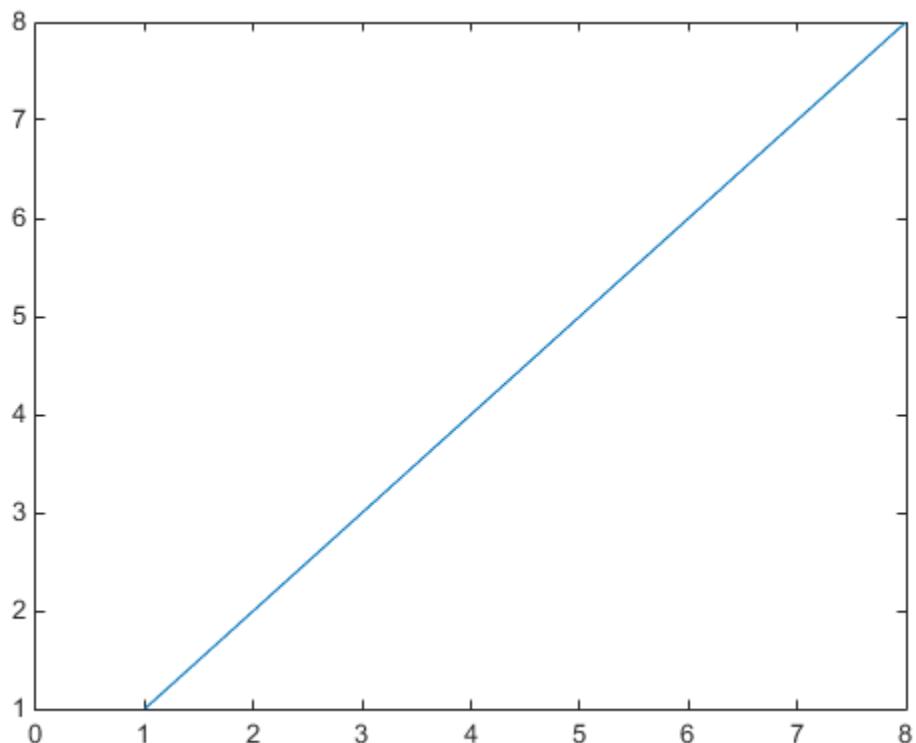
- Define the property default value
- Define the default value of the mode property as `manual`

Setting X-Axis Limits

Suppose you want to define default values for the x-axis limits. Because the axes `XLim` property is usually automatically calculated, you must set the associated mode property (`XLimMode`) to `manual`.

```
set(groot, 'defaultAxesXLim',[0 8])
set(groot, 'defaultAxesXLimMode','manual')
plot(1:20)
```

The axes uses the default x-axis limits of [0 8]:



How MATLAB Finds Default Values

All graphics object properties have values built into MATLAB. These values are called factory-defined values. Any property for which you do not specify a value uses the predefined value.

You can also define your own default values. MATLAB uses your default value unless you specify a value for the property when you create the object.

MATLAB searches for a default value beginning with the current object and continuing through the object's ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

MATLAB determines the value to use for a given property according to this sequence of steps:

- 1 Property default value specified as argument to the plotting function
- 2 If object is a line created by a high-level plotting function like `plot`, the axes `ColorOrder` and `LineStyleOrder` definitions override default values defined for the `Color` or `LineStyle` properties.
- 3 Property default value defined by axes (defaults can be cleared by plotting functions)
- 4 Property default value defined by figure
- 5 Property default value defined by root
- 6 If not default is defined, use factory default value

Setting default values affects only those objects created after you set the default. Existing graphics objects are not affected.

Factory-Defined Property Values

MATLAB defines values for all graphics object properties. Plotting functions use these values if you do not specify values as arguments or as defaults. Generate a list of all factory-defined values with the statement

```
a = get(groot, 'Factory');
```

`get` returns a structure array whose field names are the object type and property name concatenated, and field values are the factory value for the indicated object and property. For example, this field,

```
factoryAxesVisible: 'on'
```

indicates that the factory value for the `Visible` property of axes objects is `on`.

You can get the factory value of an individual property with

```
get(groot, 'factoryObjectTypePropertyName')
```

For example:

```
get(groot, 'factoryTextFontName')
```

Define Default Line Styles

This example shows how to set default line styles.

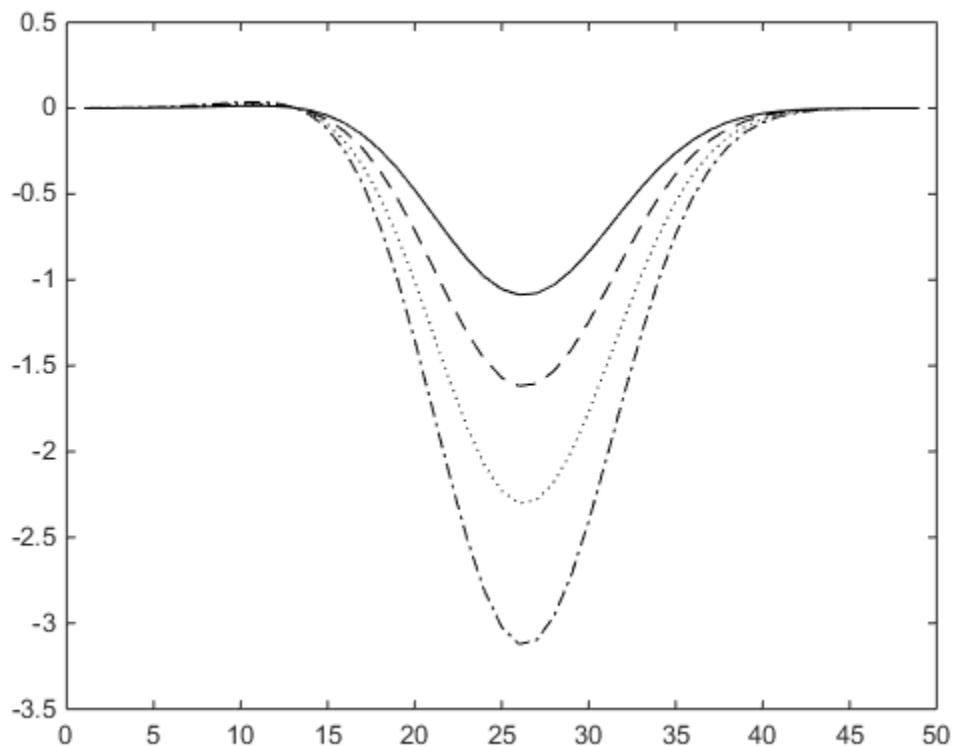
The `plot` function cycles through the colors defined by the axes `ColorOrder` property when displaying multiline plots. If you define more than one value for the axes `LineStyleOrder` property, `plot` increments the line style after each cycle through the colors.

This example sets default values for axes objects on the root level:

```
set(groot, 'DefaultAxesColorOrder',[0 0 0],...
    'DefaultAxesLineStyleOrder',' -|-.|:-|-.|-.')
```

Now, whenever you call `plot`, it uses black for all data plotted because the axes `ColorOrder` contains only one color, but it cycles through the line styles defined for `LineStyleOrder`.

```
Z = peaks;
x = 1:length(Z);
y = Z(4:7,:);
plot(x,y)
```

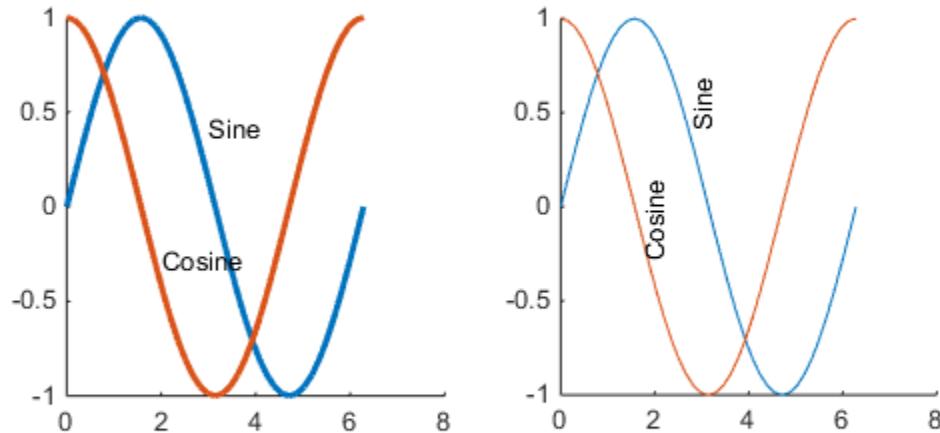


Multilevel Default Values

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);
figure('defaultAxesPlotBoxAspectRatio',[1 1 1],...
    'defaultAxesPlotBoxAspectRatioMode','manual');
subplot(1,2,1,'defaultLineLineWidth',2);
hold on
plot(t,s,t,c)
text('Position',[3 0.4], 'String', 'Sine')
text('Position',[2 -0.3], 'String', 'Cosine')

subplot(1,2,2,'defaultTextRotation',90);
hold on
plot(t,s,t,c)
text('Position',[3 0.4], 'String', 'Sine')
text('Position',[2 -0.3], 'String', 'Cosine')
```



Issuing the same `plot` and `text` statements to each subplot region results in a different display, reflecting different default values defined for the axes. The default defined on the figure applies to both axes.

It is necessary to call `hold on` to prevent the `plot` function from resetting axes properties.

Note: If a property has an associated mode property (for example, `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode`), you must define a default value of `manual` for the mode property when you define a default value for the associated property.

Graphics Object Callbacks

- “Callbacks — Programmed Response to User Action” on page 11-2
- “Callback Definition” on page 11-4
- “Button Down Callback Function” on page 11-7
- “Define a Context Menu” on page 11-9
- “Define an Object Creation Callback” on page 11-11
- “Define an Object Deletion Callback” on page 11-13
- “Capturing Mouse Clicks” on page 11-14
- “Pass Mouse Click to Group Parent” on page 11-18
- “Pass Mouse Click to Obscured Object” on page 11-21

Callbacks – Programmed Response to User Action

In this section...

[“What Are Callbacks?” on page 11-2](#)

[“Window Callbacks” on page 11-2](#)

What Are Callbacks?

A *callback* is a function that executes in response to some predefined user action, such as clicking on a graphics object or closing a figure window. Associate a callback with a specific user action by assigning a function to the callback property for that user action.

All graphics objects have the following properties for which you can define callback functions:

- `ButtonDownFcn` — Executes when you press the left mouse button while the cursor is over the object or is within a few pixels of the object.
- `CreateFcn` — Executes during object creation after MATLAB set all properties
- `DeleteFcn` — Executes just before MATLAB deletes the object

Note: When you call a plotting function, such as `plot` or `bar`, MATLAB creates new graphics objects and resets most figure and axes properties. Therefore, callback functions that you have defined for graphics objects can be removed by MATLAB. To avoid this problem, see “[Define a Callback as a Default](#)” on page 11-6.

Window Callbacks

Figures have additional properties that execute callbacks with specific user actions:

- `CloseRequestFcn` — Executes when a request is made to close the figure (by a `close` command, by the window manager menu, or by quitting MATLAB).
- `KeyPressFcn` — Executes when you press a key while the cursor is in the figure window.
- `ResizeFcn` — Executes when you resize the figure window.

- **WindowButtonDownFcn** — Executes when you press a mouse button while the cursor is over the figure background, a disabled user-interface control, or the axes background.
- **WindowButtonMotionFcn** — Executes when you move the cursor in the figure window (but not over menus or title bar).
- **WindowButtonUpFcn** — Executes when you release the mouse button, after having pressed the mouse button in the figure.

Callback Definition

In this section...

- “Ways to Specify Callbacks” on page 11-4
- “Callback Function Syntax” on page 11-4
- “Related Information” on page 11-5
- “Define a Callback as a Default” on page 11-6

Ways to Specify Callbacks

To use callback properties, assign the callback code to the property. Use one of the following techniques:

- A function handle that references the function to execute.
- A cell array containing a function handle and additional arguments
- A string that evaluates to a valid MATLAB expression. MATLAB evaluates the string in the base workspace.

Defining a callback as a string is not recommended. The use of a function specified as function handle enables MATLAB to provide important information to your callback function.

For more information, see “Callback Function Syntax” on page 11-4.

Callback Function Syntax

Graphics callback functions must accept at least two input arguments:

- The handle of the object whose callback is executing. Use this handle within your callback function to refer to the callback object.
- The event data structure, which can be empty for some callbacks or contain specific information that is described in the property description for that object.

Whenever the callback executes as a result of the specific triggering action, MATLAB calls the callback function and passes these two arguments to the function .

For example, define a callback function called `lineCallback` for the lines created by the `plot` function. With the `lineCallback` function on the MATLAB path, use the

@ operator to assign the function handle to the `ButtonDownFcn` property of each line created by `plot`.

```
plot(x,y,'ButtonFcn',@lineCallback)
```

Define the callback to accept two input arguments. Use the first argument to refer to the specific line whose callback is executing. Use this argument to set the line `Color` property:

```
function lineCallback(src,evt)
    src.Color = 'red';
end
```

The second argument is empty for the `ButtonDownFcn` callback. The ~ character indicates that this argument is not used.

Passing Additional Input Arguments

To define additional input arguments for the callback function, add the arguments to the function definition, maintaining the correct order of the default arguments and the additional arguments:

```
function lineCallback(src,evt,arg1,arg2)
    src.Color = 'red';
    src.LineStyle = arg1;
    src.Marker = arg2;
end
```

Assign a cell array containing the function handle and the additional arguments to the property:

```
plot(x,y,'ButtonFcn',{@lineCallback,'--','*'})
```

You can use an anonymous function to pass additional arguments. For example:

```
plot(x,y,'ButtonFcn',@(src eventdata)lineCallback(src,eventdata,'--','*'))
```

Related Information

For more information about function handles, see [function handle](#).

For information on using anonymous functions, see [“Anonymous Functions”](#).

For information about using class methods as callbacks, see “Class Methods for Graphics Callbacks”.

For information on how MATLAB resolves multiple callback execution, see the **BusyAction** and **Interruptible** properties of the objects defining callbacks.

Define a Callback as a Default

You can assign a callback to the property of a specific object or you can define a default callback for all objects of that type.

To define a **ButtonDownFcn** for all line objects, set a default value on the root level.

- Use the **groot** function to specify the root level of the object hierarchy.
- Define a callback function that is on the MATLAB path
- Assign a function handle referencing this function to the **defaultLineButtonDownFcn**.

```
set(groot, 'defaultLineButtonDownFcn', @lineCallback)
```

The default value remains assigned for the MATLAB session. You can make the default value assignment in your **startup.m** file

For more information, see “Default Property Values”

Button Down Callback Function

In this section...

[“When to Use a Button Down Callback” on page 11-7](#)

[“How to Define a Button Down Callback” on page 11-7](#)

When to Use a Button Down Callback

Button down callbacks execute when users left-click on the graphics object for which the callback is assigned. Button down callbacks provide a simple way for users to interact with an object without requiring you to program additional user-interface objects, like push buttons or popup menu.

Program a button down callback when you want users to be able to:

- Perform a single operation on a graphics object by left-clicking
- Select among different operations performed on a graphics object using modifier keys in conjunction with a left-click

How to Define a Button Down Callback

- Create the callback function that MATLAB executes when users left-click on the graphics object.
- Assign a function handle that references the callback function to the `ButtonDownFcn` property of the object.

```
... 'ButtonDownFcn' ,@callbackFcn
```

Define the Callback Function

In this example, the callback function is called `lineCallback`. When you assign the function handle to the `ButtonDownFcn` property, this function must be on the MATLAB path.

Values used in the callback function include:

- `src` — The handle to the line object that the user clicks. MATLAB passes this handle .

- `src.Color` — The line object `Color` property.

```
function lineCallback(src,~)
    src.Color = rand(1,3);
end
```

Using the Callback

Here is a call to the `plot` function that creates line graphs and defines a button down callback for each line created.

```
plot(rand(1,5), 'ButtonDownFcn', @lineCallback)
```

To use the callback, create the plot and left-click on a line.

Define a Context Menu

This example shows how to define a context menu.

In this section...

[“When to Use a Context Menu” on page 11-9](#)

[“How to Define a Context Menu” on page 11-9](#)

When to Use a Context Menu

Context menus are displayed when users right-click the graphics object for which you assign the context menu. Context menus enable you to provide choices to users for interaction with graphics objects.

Program a context menu when you want user to be able to:

- Choose among specific options by right-clicking a graphics object.
- Provide an indication of what each option is via the menu label.
- Produce a specific result without knowing key combinations.

How to Define a Context Menu

- Create a `uicontextmenu` object and save its handle.
- Create each menu item using `uimenu`.
- Define callbacks for each menu item in the context menu.
- Parent the individual menu items to the context menu and assign the respective callback.
- Assign the context menu handle to the `UIContextMenu` property of the object for which you are defining the context menu.

```
function cmHandle = defineCM
    cmHandle = uicontextmenu;
    uimenu(cmHandle,'Label','Wider','Callback',@increaseLW);
    uimenu(cmHandle,'Label','Inspect','Callback',@inspectLine);
end
function increaseLW(~,~)
```

```
% Increase line width
h = gco;
orgLW = h.LineWidth;
h.LineWidth = orgLW+1;
end
function inspectLine(~,~)
% Open the property inspector
h = gco;
inspect(h)
end
```

The `defineCM` function returns the handle to the context menu that it creates. Assign this handle to the `UIContextMenu` property of the line objects as they are created by the `plot` function:

```
plot(rand(1,5), 'UIContextMenu', defineCM)
```

Use a similar programming pattern for your specific requirements.

Define an Object Creation Callback

This example shows how to define an object creation callback.

Define an object creation callback that specifies values for the `LineWidth` and `Marker` properties of line objects.

```
function lineCreate(src,~)
    src.LineWidth = 2;
    src.Marker = 'o';
end
```

Assign this function as the default line creation callback using the `line CreateFcn` property:

```
set(groot,'defaultLineCreateFcn',@lineCreate)
```

The `groot` function specifies the root of the graphics object hierarchy. Therefore, all lines created in any given MATLAB session acquire this callback. All plotting functions that create lines use these defaults.

An object's creation callback executes directly after MATLAB creates the object and sets all its property values. Therefore, the creation callback can override property name/value pairs specified in a plotting function. For example:

```
set(groot,'defaultLineCreateFcn',@lineCreate)
h = plot(1:10,'LineWidth',.5,'Marker','none')
```

The creation callback executes after the `plot` function execution is complete. The `LineWidth` and `Marker` property values of the resulting line are those values specified in the creation callback:

```
h =
Line with properties:
    Color: [0 0 1]
    LineStyle: '--'
    LineWidth: 2
    Marker: 'o'
    MarkerSize: 6
    MarkerFaceColor: 'none'
    XData: [1 2 3 4 5 6 7 8 9 10]
```

```
YData: [1 2 3 4 5 6 7 8 9 10]
ZData: []
```

Related Information

For information about default values, see “Default Property Values” on page 10-2

For information about defining callback functions, see “Callback Definition” on page 11-4

Define an Object Deletion Callback

This example shows how to define an object deletion callback.

The **figDelete** function is an example of a object deletion callback. Whenever the figure is deleted, it performs the following steps :

- Displays a **questdlg** asking if you want to save the figure
- If yes, displays a **uiputfile** dialog to get the location and name of the saved figure file
- Use **savefig** to save the figure to the file returned by **uiputfile**.

```
function figDelete(src,~)
    yn = questdlg('Save this graph?',...
        'Graph Being Destroyed',...
        'Yes','No','Yes');
    switch yn
        case 'Yes'
            fileName = uiputfile('.fig','Save graph as');
            savefig(src,fileName)
        case 'No'
            return
    end
end
```

Assign the **figDelete** function to the figure **DeleteFcn** in the **figure** function:

```
figure('DeleteFcn',@figDelete)
```

Note: Do not define the **figDelete** function as a default value for the figure **DeleteFcn** property because dialogs like **questdlg** are also figures. When MATLAB deletes the **questdlg**, the **figDelete** function would execute recursively .

Capturing Mouse Clicks

In this section...

[“Properties That Control Response to Mouse Clicks” on page 11-14](#)

[“Combinations of PickablePart/HitTest Values” on page 11-14](#)

[“Passing Mouse Click Up the Hierarchy” on page 11-15](#)

Properties That Control Response to Mouse Clicks

There are two properties that determine if and how objects respond to mouse clicks:

- `PickableParts` — Determines if an object captures mouse clicks
- `HitTest` — Determines if the object can respond to the mouse click it captures or passes the click to its closest ancestor.

Objects pass the click through the object hierarchy until reaching an object that can respond.

Programming a Response to a Mouse Click

When an object captures and responds to a mouse click, the object:

- Executes its button down function in response to a mouse left-click — If the object defines a callback for the `ButtonDownFcn` property, MATLAB executes this callback.
- Displays context menu in response to a mouse right-click — If the object defined a context menu using the `UIContextMenu` property, MATLAB invokes this context menu.

Note: Figures do not have a `PickableParts` property. Figures execute button callback functions regardless of the setting of their `HitTest` property.

Note: If the axes `PickableParts` property is set to '`none`', the axes children cannot capture mouse clicks. In this case, all mouse clicks are captured by the figure.

Combinations of PickablePart/HitTest Values

Use the `PickableParts` and `HitTest` properties to implement the following behaviors:

- Clicked object captures mouse click and responds with button down callback or context menu.
- Clicked object captures mouse click and passes the mouse click to one of its ancestors, which can respond with button down callback or context menu.
- Clicked object does not capture mouse click. Mouse click can be captured by objects behind the clicked object.

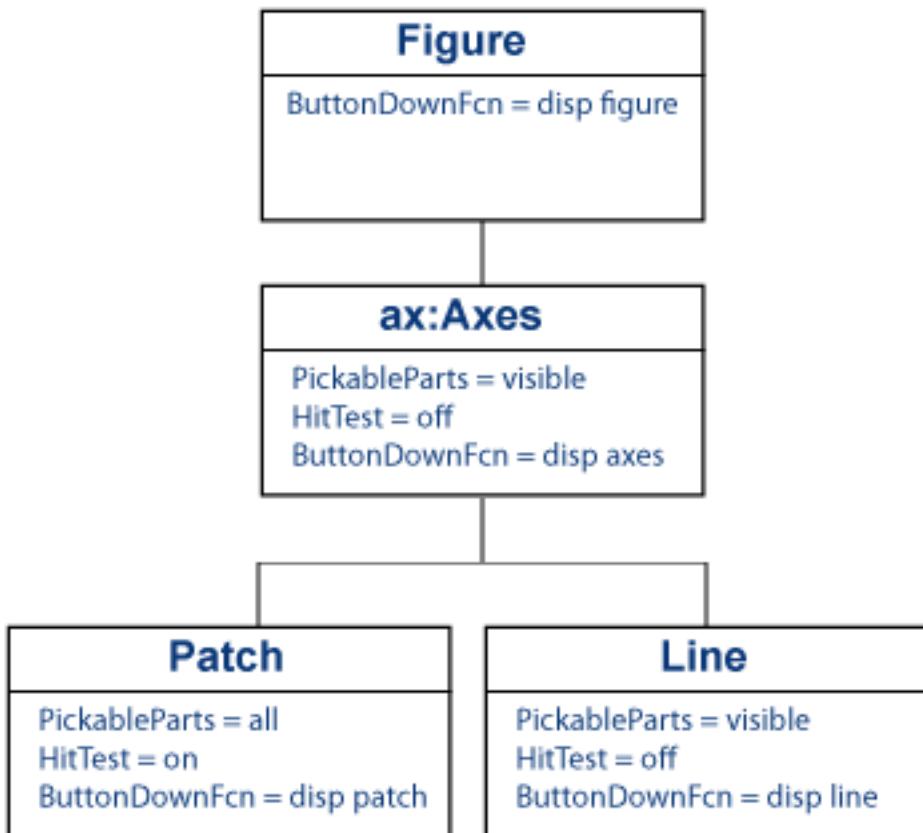
This table summarizes the response to a mouse click based on property values.

Axes PickableParts	PickableParts	HitTest	Result of Mouse Click
visible/all	visible (default)	on (default)	Clicking visible parts of object executes button down callback or invokes context menu
visible/all	all	on	Clicking any part of the object, even if not visible, makes object current and executes button down callback or invokes context menu
visible/all/none	none	on/off	Clicking the object never makes it the current object and can never execute button down callback or invoke context menu
none	visible/all/none	on/off	Clicking any axes child objects never executes button down callback or invokes context menu

MATLAB searches ancestors using the **Parent** property of each object until finding a suitable ancestor or reaching the figure.

Passing Mouse Click Up the Hierarchy

Consider the following hierarchy of objects and their **PickableParts** and **HitTest** property settings.



This code creates the hierarchy:

```
function pickHit
f = figure;
ax = axes;
p = patch(rand(1,3),rand(1,3),'g');
l = line([1 0],[0 1]);
set(f,'ButtonDownFcn',@(~,~)disp('figure'),...
    'HitTest','off')
set(ax,'ButtonDownFcn',@(~,~)disp('axes'),...
    'HitTest','off')
set(p,'ButtonDownFcn',@(~,~)disp('patch'),...
    'PickableParts','all','FaceColor','none')
```

```
set(l, 'ButtonDownFcn', @(~,~)disp('line')), ...
    'HitTest', 'off')
end
```

Click the Line

Left-click the line:

- The line becomes the current object, but cannot execute its `ButtonDownFcn` callback because its `HitTest` property is `off`.
- The line passes the hit to the closest ancestor (the parent axes), but the axes cannot execute its `ButtonDownFcn` callback, so the axes passes the hit to the figure.
- The figure can execute its callback, so MATLAB displays `figure` in the Command Window.

Click the Patch

The patch `FaceColor` is `none`. However, the patch `PickableParts` is `all`, so you can pick the patch by clicking the empty face and the edge.

The patch `HitTest` property is `on` so the patch can become the current object. When the patch becomes the current object, it executes its button down callback.

Pass Mouse Click to Group Parent

This example shows how a group of objects can pass a mouse click to a parent, which operates on all objects in the group.

In this section...

[“Objective and Design” on page 11-18](#)

[“Object Hierarchy and Key Properties” on page 11-18](#)

[“MATLAB Code” on page 11-19](#)

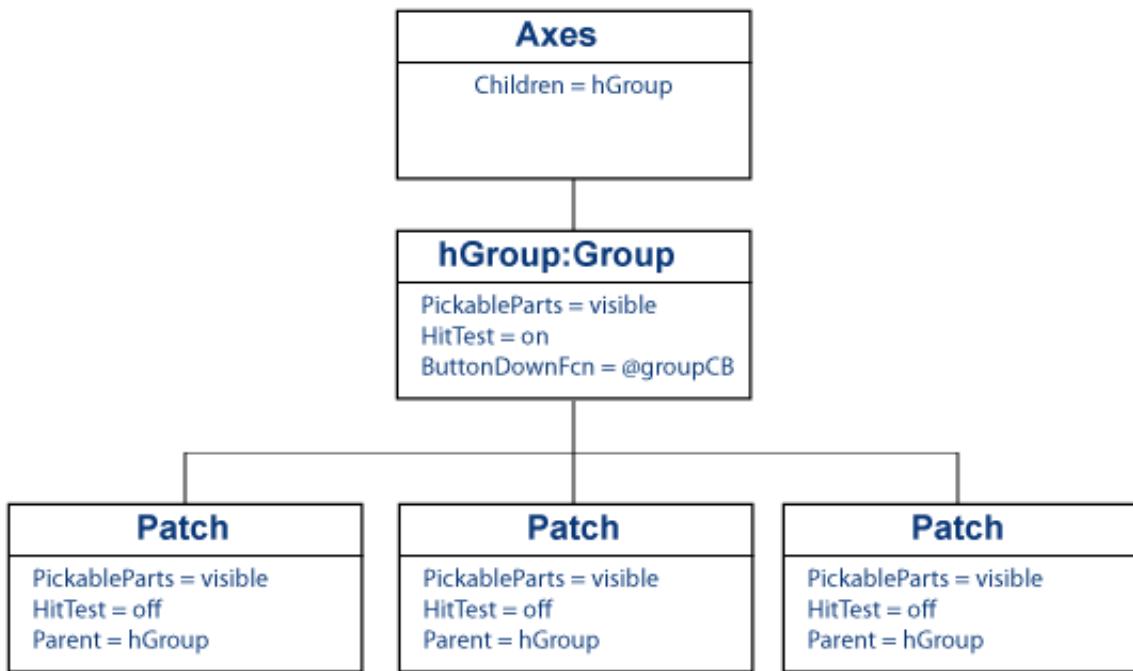
Objective and Design

Suppose you want a single mouse click on any member of a group of objects to execute a single button down callback affecting all objects in the group.

- Define the graphics objects to be added to the group.
- Assign an `hggroup` object as the parent of the graphics objects.
- Define a function to execute when any of the objects are clicked. Assign its function handle to the `hggroup` object's `ButtonDownFcn` property.
- Set the `HitTest` property of every object in the group to `off` so that the mouse click is passed to the object's parent.

Object Hierarchy and Key Properties

This example uses the following object hierarchy.



MATLAB Code

Create a file with two functions:

- `pickPatch` — The main function that creates the graphics objects.
- `groupCB` — The local function for the `hggroup` callback.

The `pickPatch` function creates three patch objects and parents them to an `hggroup` object. Setting the `HitTest` property of each patch to off directs mouse clicks to the parent.

```

function pickPatch
    figure
    x = [0 1 2];
    y = [0 1 0];
    hGroup = hggroup('ButtonDownFcn', @groupCB);
    patch(x,y, 'b', ...
        'Parent', hGroup, ...
  
```

```
'HitTest','off')
patch(x+2,y,'b',...
    'Parent',hGroup,....
    'HitTest','off')
patch(x+3,y,'b',...
    'Parent',hGroup,....
    'HitTest','off')
end
```

The `groupCB` callback operates on all objects contained in the `hggroup`. The `groupCB` function uses the callback source argument passed to the callback (`src`) to obtain the handles of the patch objects.

Using the callback source argument (which is the handle to `hggroup` object) eliminates the need to create global data or pass additional arguments to the callback.

A left-click on any patch changes the face color of all three patches to a random RGB color value.

```
function groupCB(src,~)
    s = src.Children;
    set(s,'FaceColor',rand(1,3))
end
end
```

For more information on callback functions, see “Callback Definition” on page 11-4

Pass Mouse Click to Obscured Object

This example shows how to pass mouse clicks to an obscured object.

Set the **PickableParts** property of a graphics object to **none** to prevent the object from capturing a mouse click. This example:

- Defines a context menu for the axes that calls **hold** with values **on** or **off**
- Creates graphs in which none of the data objects can capture mouse clicks, enabling all right-clicks to pass to the axes and invoke the context menu.

The **axesHoldCM** function defines a context menu and returns its handle.

```
function cmHandle = axesHoldCM
    cmHandle = uicontextmenu;
    uimenu(cmHandle,'Label','hold on','Callback',@holdOn);
    uimenu(cmHandle,'Label','hold off','Callback',@holdOff);
end
function holdOn(~,~)
    fig = gcbf;
    ax = fig.CurrentAxes;
    hold(ax,'on')
end
function holdOff(~,~)
    fig = gcbf;
    ax = fig.CurrentAxes;
    hold(ax,'off')
end
```

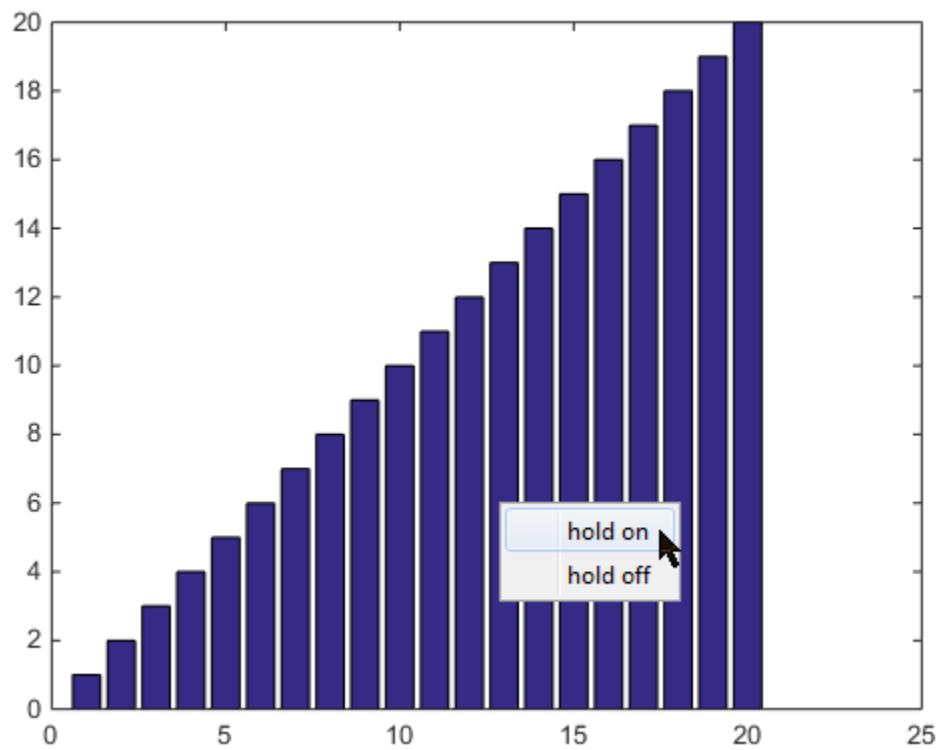
Create a bar graph and set the **PickableParts** property of the Bar objects:

```
bar(1:20,'PickableParts','none')
```

Create the context menu for the current axes:

```
ax = gca;
ax.UIContextMenu = axesHoldCM
```

Right-click over the bars in the graph and display the axes context menu:



Graphics Objects

- “Graphics Objects” on page 12-2
- “Features Controlled by Graphics Objects” on page 12-7

Graphics Objects

In this section...

- “MATLAB Graphics Objects” on page 12-2
- “Graphs Are Composed of Specific Objects” on page 12-2
- “Organization of Graphics Objects” on page 12-2

MATLAB Graphics Objects

Graphics objects are the visual components used by MATLAB to display data graphically. For example, a graph can contain lines, text, and axes, all displayed in a figure window.

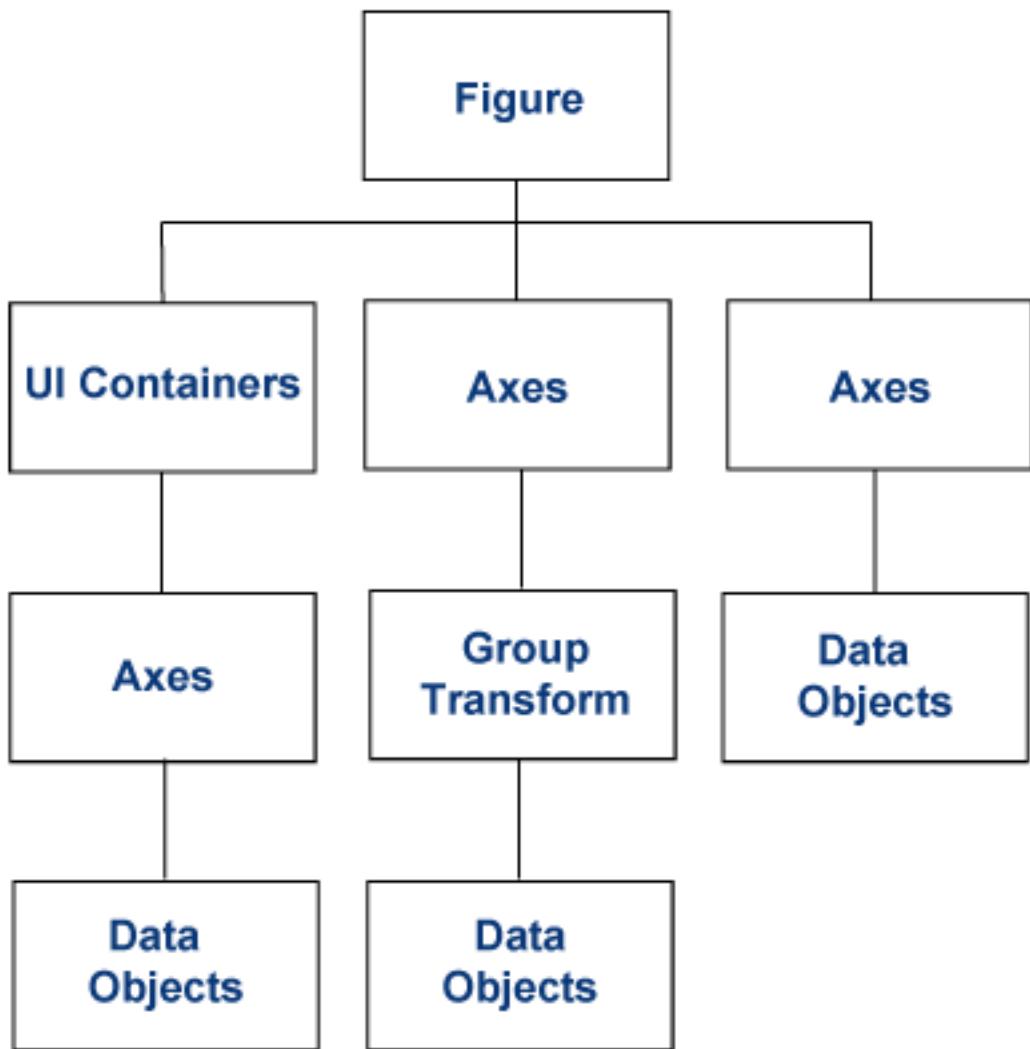
Each object has a unique identifier called a *handle*. Using this handle, you can manipulate the characteristics of an existing graphics object by setting object *properties*. You can also specify values for properties when you create a graphics object. Typically, you create graphics objects using plotting functions like `plot`, `bar`, `scatter`, and so on.

Graphs Are Composed of Specific Objects

When you create a graph, for example by calling the `plot` function, MATLAB automatically performs a number of steps to produce the graph. These steps involve creating objects and setting the properties of these objects to appropriate values for your specific graph.

Organization of Graphics Objects

Graphics objects are organized into a hierarchy, as shown by the following diagram.



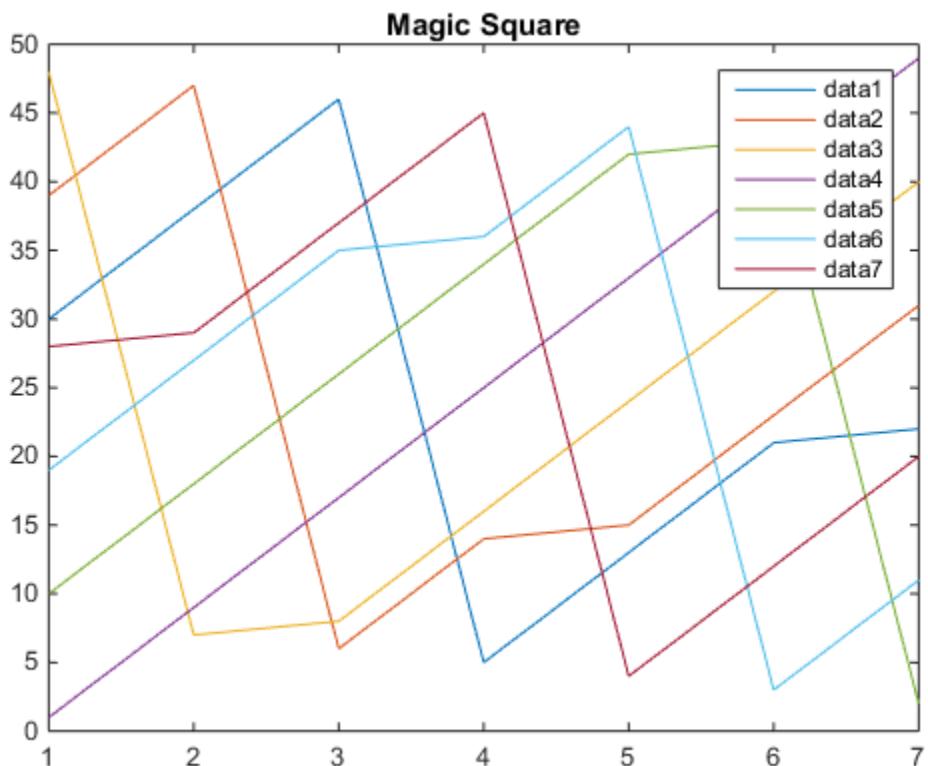
The hierarchical nature of graphics objects reflects the containment of objects by other objects. Each object plays a specific role in the graphics display.

For example, suppose you create a line graph with the `plot` function. An axes object defines a frame of reference for the lines that represent data. A figure is the window to

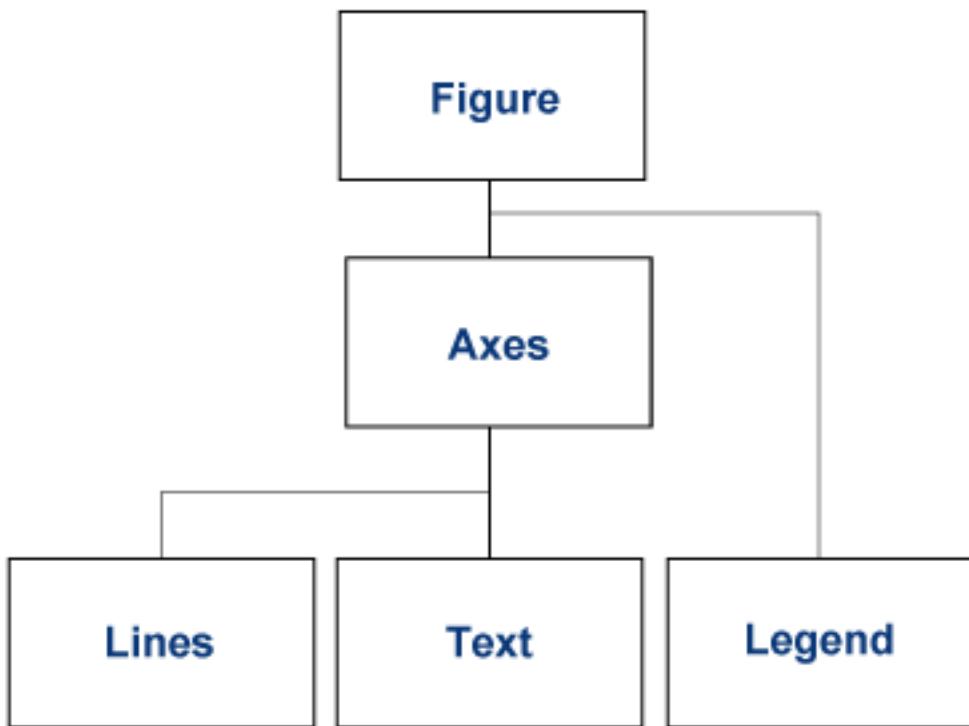
display the graph. The figure contains the axes and the axes contains the lines, text, legends, and other objects used to represent the graph.

Note: An *axes* is a single object that represents x-, y-, and z-axis scales, tick marks, tick labels, axis labels, and so on.

Here is a simple graph.



This graph forms a hierarchy of objects.



Parent-Child Relationship

The relationship among objects is held in the **Parent** and **Children** properties. For example, the parent of an axes is a figure. The **Parent** property of an axes contains the handle to the figure in which it is contained.

Similarly, the **Children** property of a figure contains any axes that the figure contains. The figure **Children** property also contains the handles of any other objects it contains, such as legends and user-interface objects.

You can use the parent-child relationship to find object handles. For example, if you create a plot, the current axes **Children** property contains the handles to all the lines:

```
plot(rand(5))
ax = gca;
ax.Children
```

```
ans =  
5x1 Line array:  
Line  
Line  
Line  
Line  
Line
```

You can also specify the parent of objects. For example, create a group object and parent the lines from the axes to the group:

```
hg = hggroup;  
plot(rand(5), 'Parent', hg)
```

Features Controlled by Graphics Objects

In this section...

- “Purpose of Graphics Objects” on page 12-7
- “Figures” on page 12-7
- “Axes” on page 12-8
- “Objects That Represent Data” on page 12-9
- “Group Objects” on page 12-10
- “Annotation Objects” on page 12-11

Purpose of Graphics Objects

Graphics objects represent data in intuitive and meaningful ways, such as line graphs, images, text, and combinations of these objects. Graphics objects act as containers for other objects or as representations of data.

- Containers — Figures display all graphics objects. Panels and groups enable collections of objects to be treated as one entity for some operations.
- Axes are containers that define a coordinate system for the objects that represent the actual data in graphs.
- Data visualization objects — Lines, text, images, surfaces, and patches that implement various types of graphs.

Figures

Figures are the windows in which MATLAB displays graphics. Figures contain menus, toolbars, user-interface objects, context menus, and axes.

Figures play two distinct roles in MATLAB:

- Containing graphs of data
- Containing user interfaces (which can include graphs in the interface)

Graphics Features Controlled by Figures

Figure properties control certain characteristics that affect graphs:

- Color and transparency of surfaces and patches — **Alphamap** and **Colormap**
- Appearance of plotted lines and axes grid lines — **GraphicsSmoothing**
- Printing and exporting graphs — figure printing properties
- Drawing speed and rendering features — **Renderer**

Figures use different drawing methods called renderers. There are two renderers:

- OpenGL — The default renderer used by MATLAB for most applications. For more information, see [opengl](#).
- Painters — Use when OpenGL has problems on a computer with particular graphics hardware that has software defects or outdated software drivers. Also used for exporting graphics for certain formats, such as PDF.

Note: For best results, ensure that your computer has the latest graphics hardware drivers supplied by the hardware vendor.

For a list of all figure properties, see [Figure Properties](#)

Axes

MATLAB creates an axes to define the coordinate system of each graph. Axes are always contained by a figure object. Axes themselves contain the graphics objects that represent data.

Axes control many aspects of how MATLAB displays graphical information.

Graphics Features Controlled by Axes

Much of what you can customize in a graph is controlled by axes properties.

- Axis limits, orientation, and tick placement
- Axis scales (linear or logarithmic)
- Grid control
- Font characteristics for the title and axis labels.
- Default line colors and line styles for multiline graphs
- Axis line and grid control

- Color scaling of objects based on colormap
- View and aspect ratio
- Clipping graphs to axis limits
- Controlling axes resize behavior
- Lighting and transparency control

For a list of all axes properties, see Axes Properties

Objects That Represent Data

Data objects are the lines, images, text, and polygons that graphs use to represent data. For example:

- Lines connect data points using specified x- and y-coordinates.
- Markers locate scattered data in some range of values.
- Rectangular bars indicate distribution of values in a histogram.

Because there are many kinds of graphs, there are many types of data objects. Some are general purpose, such as lines and rectangles and some are highly specialized, such as errorbars, colorbars, and legends.

Graphics Features Controlled by Data Objects

Data object properties control the appearance of the object and also contain the data that defines the object. Data object properties can also control certain behaviors.

- Data — Change the data to update the graph. Many data objects can link their data properties to workspace variables that contain the data.
- Color Data — Objects can control how data maps to colors by specifying color data.
- Appearance — Specify colors of line, markers, polygon faces as well as line styles, marker types.
- Specific behaviors — Properties can control how the object interprets or displays its data. For example, Bar objects have a property called `BarLayout` that determines if the bars are grouped or stacked. Contour objects have a `LevelList` property that specifies the contour intervals at which to draw contour lines.

High-Level vs. Low-Level Functions

Plotting functions create data objects in one of two ways:

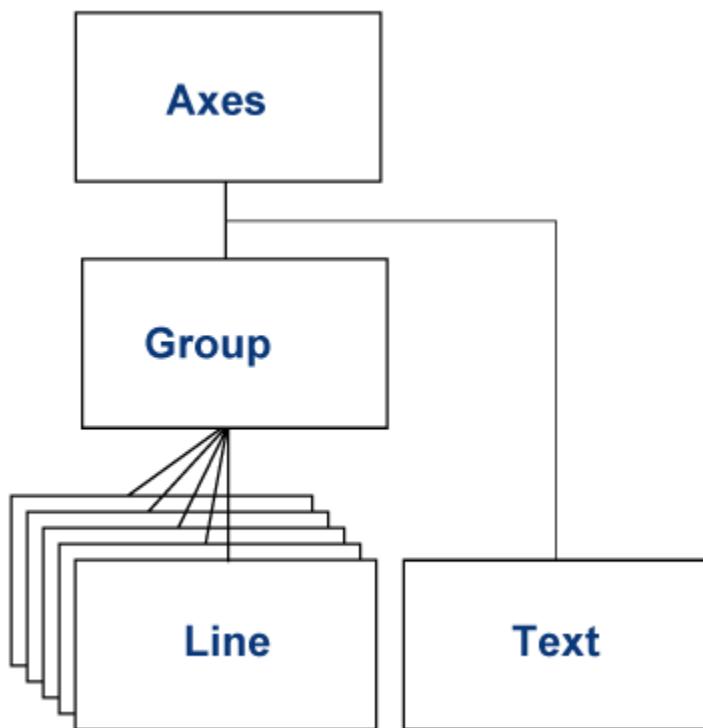
- High-level functions — Create complete graphs that replace existing graphs with new ones. High-level functions include `plot`, `bar`, `scatter`, and so on. For a summary of high-level functions, see “Types of MATLAB Plots” on page 1-2.
- Low-level functions — Add graphics objects with minimal changes to the existing graph. Low-level functions include `line`, `patch`, `rectangle`, `surface`, `text`, `image`, and `light`.

Group Objects

Group objects enable you to treat a number of data objects as one entity. For example, you can make the entire group visible or invisible, select all objects when only one is clicked, or apply a transform matrix to rotate, translate, or scale all the objects in the group.

This code parents the plotted lines to the group object returned by the `hggroup` function. The text object is not part of the group.

```
y = magic(5);
hg = hggroup;
plot(y, 'Parent', hg)
text(2.5, 10, 'Plot of 5x5 magic square')
```



Annotation Objects

Annotation objects comprise arrows, text boxes, and combinations of both. Annotation objects have special features that overcome the limitations of data objects used to annotate graphs:

- Annotation objects are children of the figure.
- You can easily locate annotations anywhere in the figure.
- Define the location of annotation objects in normalized figure coordinates: lower left = (0,0), upper right = (1,1), making their placement independent of range of data represented by the axes.

Note: MATLAB parents annotation objects to a special layer. Do not attempt to parent objects to this layer. MATLAB automatically assigns annotation objects to the appropriate parent.

Group Objects

- “Object Groups” on page 13-2
- “Create Object Groups” on page 13-3
- “Transforms Supported by hgtransform” on page 13-5
- “Rotate About an Arbitrary Axis” on page 13-10
- “Nest Transforms for Complex Movements” on page 13-14

Object Groups

Group objects are invisible containers for graphics objects. Use group objects to form a collection of objects that can behave as one object in certain respects. When you set properties of the group object, the result applies to the objects contained in the group.

For example, you can make the entire group visible or invisible, select all objects when only one is clicked, or apply a transform matrix to reposition the objects.

Group objects can contain any of the objects that axes can contain, such as lines, surfaces, text, and so on. Group objects can also contain other group objects. Group objects are always parented to an axes object or another group object.

There are two kinds of group objects:

- Group — Use when you want to create a group of objects and control the visibility or selectability of the group based on what happens to any individual object in the group. Create group objects with the `hggroup` function.
- Transform — Use when you want to transform a group of objects. Transforms include rotation, translation, and scaling. For an example, see “Nest Transforms for Complex Movements” on page 13-14. Create transform objects with the `hgtransform` function.

The difference between the group and transform objects is that the transform object can apply a transform matrix (via its `Matrix` property) to all objects for which it is the parent.

Create Object Groups

In this section...

[“Parent Specification” on page 13-4](#)

[“Visible and Selected Properties of Group Children” on page 13-4](#)

Create an object group by parenting objects to a group or transform object. For example, call `hggroup` to create a group object and save its handle. Assign this group object as the parent of subsequently created objects:

```
hg = hggroup;
plot(rand(5), 'Parent', hg)
text(3,0.5, 'Random lines', 'Parent', hg)
```

Setting the visibility of the group to off makes the line and text objects it contains invisible.

```
hg.Visible = 'off';
```

You can add objects to a group selectively. For example, the following call to the `bar` function returns the handles to five separate bar objects:

```
hb = bar(randn(5))

hb =
1x5 Bar array:

Bar      Bar      Bar      Bar      Bar
```

Parent the third, fourth, and fifth bar object to the group:

```
hg = hggroup;
set(hb(3:5), 'Parent', hg)
```

Group objects can be the parent of any number of axes children, including other group objects. For examples, see “Rotate About an Arbitrary Axis” on page 13-10 and “Nest Transforms for Complex Movements” on page 13-14.

Parent Specification

Plotting functions clear the axes before generating their graph. However, if you assign a group or transform as the **Parent** in the plotting function, the group or transform object is not cleared.

For example:

```
hg = hggroup;
hb = bar(randn(5));
set(hb, 'Parent', hg)
```

```
Error using matlab.graphics.chart.primitive.Bar/set
Cannot set property to a deleted object
```

The **bar** function cleared the axes. However, if you set the **Parent** property as a name/value pair in the **bar** function arguments, the **bar** function does not delete the group:

```
hg = hggroup;
hb = bar(randn(5), 'Parent', hg);
```

Visible and Selected Properties of Group Children

Setting the **Visible** property of a group or transform object controls whether all the objects in the group are visible or not visible. However, changing the state of the **Visible** property for the group object does not change the state of this property for the individual objects. The values of the **Visible** property for the individual objects are preserved.

For example, if the **Visible** property of the group is set to off and subsequently set to on, only the objects that were originally visible are displayed.

The same behavior applies to the **Selected** and **SelectionHighlight** properties. The children of the group or transform object show the state of the containing object properties without actually changing their own property values.

Transforms Supported by hgtransform

In this section...

- “Transforming Objects” on page 13-5
- “Rotation” on page 13-5
- “Translation” on page 13-6
- “Scaling” on page 13-6
- “The Default Transform” on page 13-7
- “Disallowed Transforms: Perspective” on page 13-7
- “Disallowed Transforms: Shear” on page 13-7
- “Absolute vs. Relative Transforms” on page 13-8
- “Combining Transforms into One Matrix” on page 13-8
- “Undoing Transform Operations” on page 13-9

Transforming Objects

The transform object's `Matrix` property applies a transform to all the object's children in unison. Transforms include rotation, translation, and scaling. Define a transform with a four-by-four transformation matrix.

Creating a Transform Matrix

The `makehgform` function simplifies the construction of matrices to perform rotation, translation, and scaling. For information on creating transform matrices using `makehgform`, see “Nest Transforms for Complex Movements” on page 13-14.

Rotation

Rotation transforms follow the right-hand rule — rotate objects about the x -, y -, or z -axis, with positive angles rotating counterclockwise, while sighting along the respective axis toward the origin. If the angle of rotation is theta, the following matrix defines a rotation of theta about the x -axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x & 0 \\ 0 & \sin \theta_x & \cos \theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To create a transform matrix for rotation about an arbitrary axis, use the `makehgtform` function.

Translation

Translation transforms move objects with respect to their current locations. Specify the translation as distances t_x , t_y , and t_z in data space units. The following matrix shows the location of these elements in the transform matrix.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling transforms change the sizes of objects. Specify scale factors s_x , s_y , and s_z and construct the following matrix.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You cannot use scale factors less than or equal to zero.

The Default Transform

The default transform is the identity matrix, which you can create with the `eye` function. Here is the identity matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See “Undoing Transform Operations” on page 13-9.

Disallowed Transforms: Perspective

Perspective transforms change the distance at which you view an object. The following matrix is an example of a perspective transform matrix, which MATLAB graphics does not allow.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & p_x & 0 \end{bmatrix}$$

In this case, p_y is the perspective factor.

Disallowed Transforms: Shear

Shear transforms keep all points along a given line (or plane, in 3-D coordinates) fixed while shifting all other points parallel to the line (plane) proportional to their perpendicular distance from the fixed line (plane). The following matrix is an example of a shear transform matrix, which `hgtransform` does not allow.

$$\begin{bmatrix} 1 & s_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case, s_x is the shear factor and can replace any zero element in an identity matrix.

Absolute vs. Relative Transforms

Transforms are specified in absolute terms, not relative to the current transform. For example, if you apply a transform that translates the transform object 5 units in the x direction, and then you apply another transform that translates it 4 units in the y direction, the resulting position of the object is 4 units in the y direction from its original position.

If you want transforms to accumulate, you must concatenate the individual transforms into a single matrix. See “Combining Transforms into One Matrix” on page 13-8.

Combining Transforms into One Matrix

It is usually more efficient to combine various transform operations into one matrix by concatenating (multiplying) the individual matrices and setting the **Matrix** property to the result. Matrix multiplication is not commutative, so the order in which you multiply the matrices affects the result.

For example, suppose you want to perform an operation that scales, translates, and then rotates. Assuming **R**, **T** and **S** are your individual transform matrices, multiply the matrices as follows:

```
C = R*T*S % operations are performed from right to left
```

S is the scaling matrix, **T** is the translation matrix, **R** is the rotation matrix, and **C** is the composite of the three operations. Then set the transform object's **Matrix** property to **C**:

```
hg = hgtransform('Matrix',C);
```

Multiplying the Transform by the Identity Matrix

The following sets of statements are not equivalent. The first set:

```
hg.Matrix = C;  
hg.Matrix = eye(4);
```

results in the removal of the transform C. The second set:

```
I = eye(4);  
C = I*R*T*S;  
hg.Matrix = C;
```

applies the transform C. Concatenating the identity matrix to other matrices has no effect on the composite matrix.

Undoing Transform Operations

Because transform operations are specified in absolute terms (not relative to the current transform), you can undo a series of transforms by setting the current transform to the identity matrix. For example:

```
hg = hgtransform('Matrix',C);  
...  
hg.Matrix = eye(4);
```

returns the objects contained by the transform object, hg, to their orientation before applying the transform C.

Rotate About an Arbitrary Axis

This example shows how to rotate an object about an arbitrary axis.

In this section...

[“Translate to Origin Before Rotating” on page 13-10](#)

[“Rotate Surface” on page 13-10](#)

Translate to Origin Before Rotating

Rotations are performed about the origin. Therefore, you need to perform a translation so that the intended axis of rotation is temporarily at the origin. After applying the rotation transform matrix, you then translate the object back to its original position.

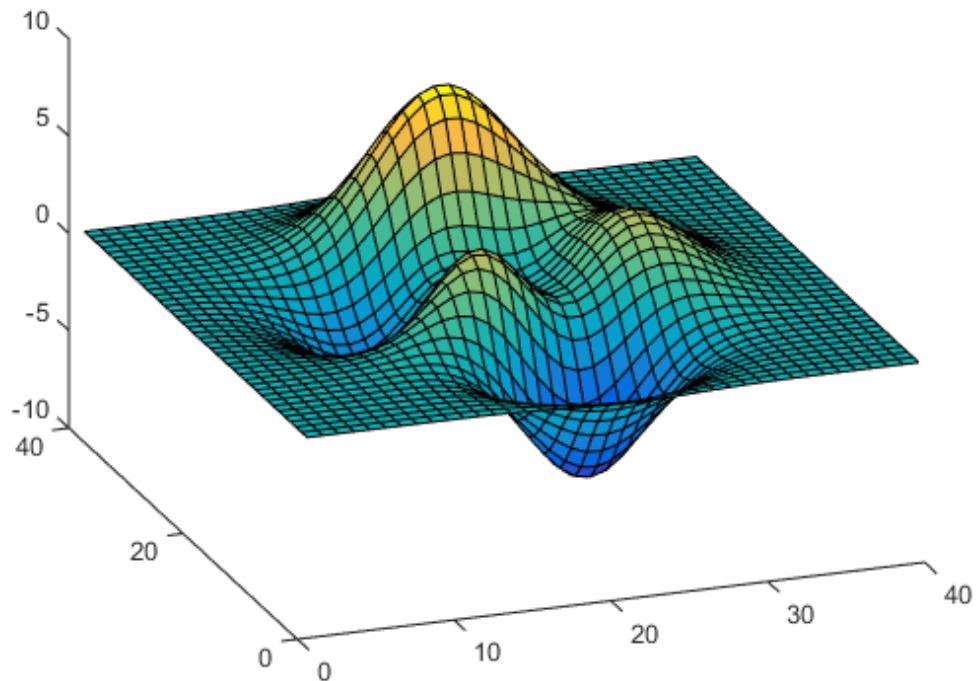
Rotate Surface

This example shows how to rotate a surface about the *y*-axis.

Create Surface and Hgtransform

Parent the surface to the hgtransform.

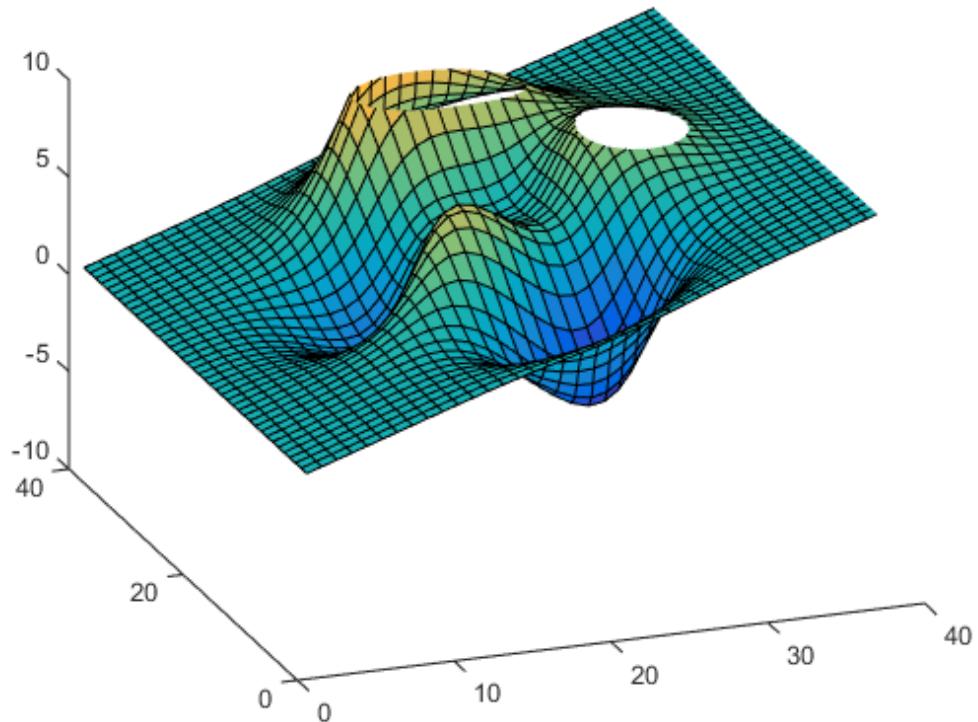
```
t = hgtransform;
surf(peaks(40), 'Parent', t)
view(-20,30)
axis manual
```



Create Transform

Set a y-axis rotation matrix to rotate the surface by -15 degrees.

```
ry_angle = -15*pi/180;  
Ry = makehgtform('yrotate',ry_angle);  
t.Matrix = Ry;
```



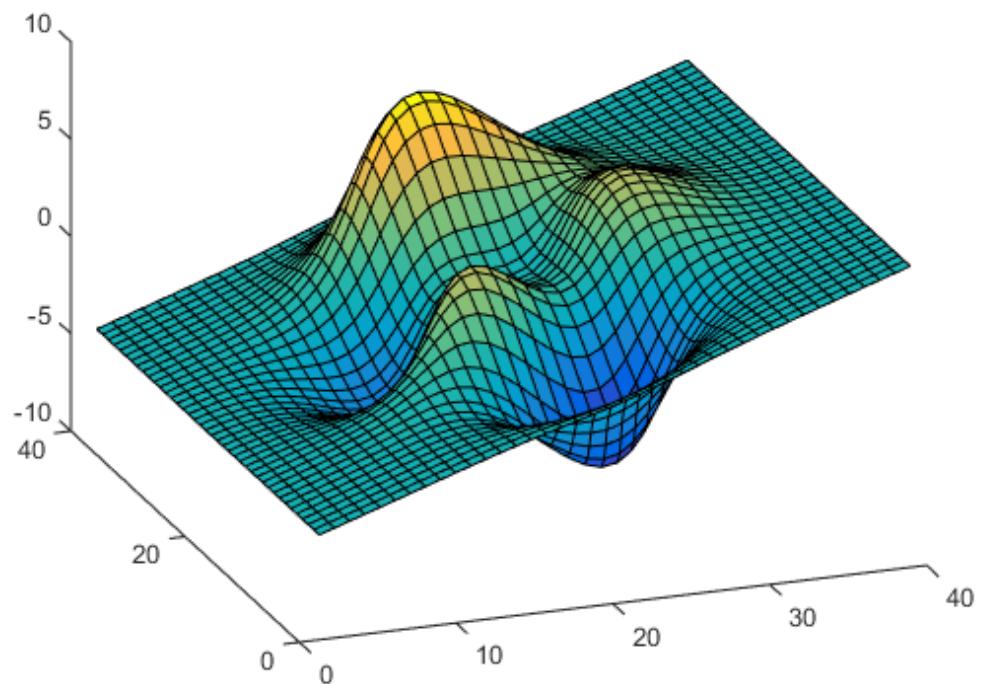
The surface rotated -15 degrees about the y -axis that passes through the origin.

Translate the Surface and Rotate

Now rotate the surface about the y -axis that passes through the point $x = 20$.

Create two translation matrices, one to translate the surface -20 units in x and another to translate 20 units back. Concatenate the two translation matrices with the rotation matrix in the correct order and set the transform.

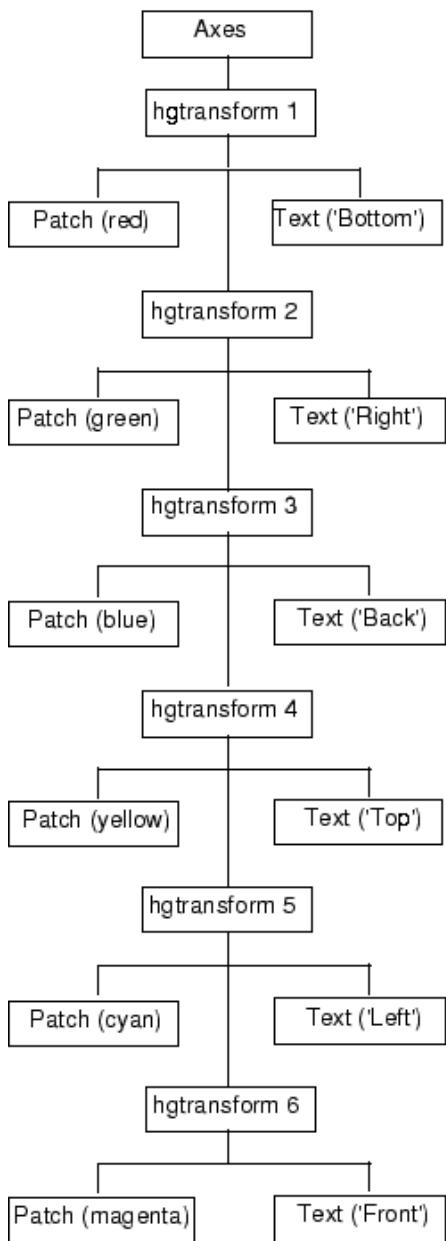
```
Tx1 = makehgtform('translate', [-20 0 0]);
Tx2 = makehgtform('translate', [20 0 0]);
t.Matrix = Tx2*Ry*Tx1;
```



Nest Transforms for Complex Movements

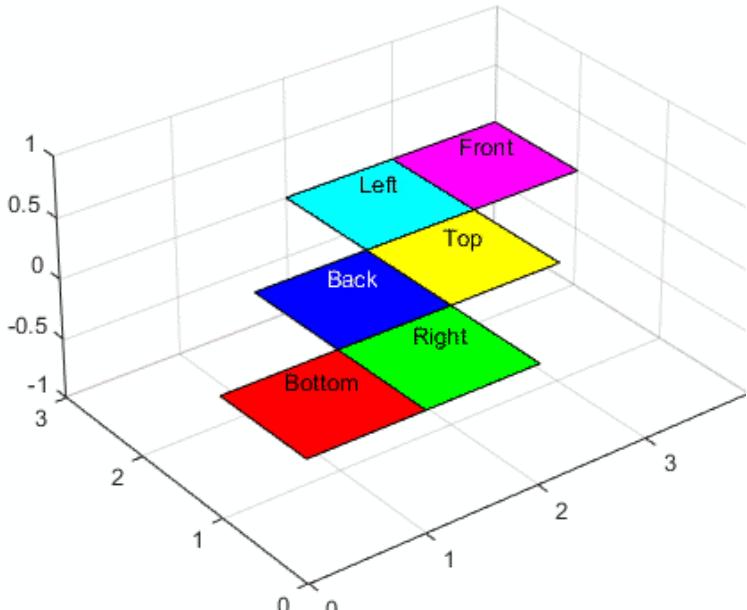
This example creates a nested hierarchy of hgtransform objects, which are then transformed in sequence to create a cube from six squares. The example illustrates how you can parent hgtransform objects to other hgtransform objects to create a hierarchy, and how transforming members of a hierarchy affects subordinate members.

Here is an illustration of the hierarchy.



The diagram on the left represents the object hierarchy in the picture below.

Through a series of simple rotations and translations, the six squares are folded into a cube.



The `transform_foldbox` function implements the transform hierarchy. The `doUpdate` function renders each step. Place both functions in a `.m` file and execute `transform_foldbox`.

```
function transform_foldbox
    % Create six square and fold
    % them into a cube

    figure

    % Set axis limits and view
    axes('Projection','perspective',...
        'XLim',[0 4],...
        'YLim',[0 4],...
        'ZLim',[0 3])
    view(3); axis equal; grid on

    % Create a hierarchy of hgtransforms
    t(1) = hgtransform;
    t(2) = hgtransform('parent',t(1));
    t(3) = hgtransform('parent',t(2));
    t(4) = hgtransform('parent',t(3));
    t(5) = hgtransform('parent',t(4));
    t(6) = hgtransform('parent',t(5));

    % Patch data
    X = [0 0 1 1];
    Y = [0 1 1 0];
    Z = [0 0 0 0];

    % Text data
    Xtext = .5;
    Ytext = .5;
    Ztext = .15;

    % Corresponding pairs of objects (patch and text)
    % are parented into the object hierarchy
    p(1) = patch('FaceColor','red','Parent',t(1));
    txt(1) = text('String','Bottom','Parent',t(1));
    p(2) = patch('FaceColor','green','Parent',t(2));
    txt(2) = text('String','Right','Parent',t(2));
    p(3) = patch('FaceColor','blue','Parent',t(3));
    txt(3) = text('String','Back','Color','white','Parent',t(3));
    p(4) = patch('FaceColor','yellow','Parent',t(4));
```

```
txt(4) = text('String','Top','Parent',t(4));
p(5) = patch('FaceColor','cyan','Parent',t(5));
txt(5) = text('String','Left','Parent',t(5));
p(6) = patch('FaceColor','magenta','Parent',t(6));
txt(6) = text('String','Front','Parent',t(6));

% All the patch objects use the same x, y, and z data
set(p,'XData',X,'YData',Y,'ZData',Z)

% Set the position and alignment of the text objects
set(txt,'Position',[Xtext Ytext Ztext],...
    'HorizontalAlignment','center',...
    'VerticalAlignment','middle')

% Display the objects in their current location
doUpdate(1)

% Set up initial translation transforms
% Translate 1 unit in x
Tx = makehgtform('translate',[1 0 0]);
% Translate 1 unit in y
Ty = makehgtform('translate',[0 1 0]);

% Translate the unit squares to the desired locations
% The drawnow and pause commands display
% the objects after each translation
set(t(2),'Matrix',Tx);
doUpdate(1)
set(t(3),'Matrix',Ty);
doUpdate(1)
set(t(4),'Matrix',Tx);
doUpdate(1)
set(t(5),'Matrix',Ty);
doUpdate(1)
set(t(6),'Matrix',Tx);
doUpdate(1)

% Specify rotation angle (pi/2 radians = 90 degrees)
fold = pi/2;

% Rotate -y, translate x
Ry = makehgtform('yrotate',-fold);
RyTx = Tx*Ry;
```

```
% Rotate x, translate y
Rx = makehgtform('xrotate',fold);
RxTy = Ty*Rx;

% Set the transforms
% Draw after each group transform and pause
set(t(6), 'Matrix', RyTx);
doUpdate(1)
set(t(5), 'Matrix', RxTy);
doUpdate(1)
set(t(4), 'Matrix', RyTx);
doUpdate(1)
set(t(3), 'Matrix', RxTy);
doUpdate(1)
set(t(2), 'Matrix', RyTx);
doUpdate(1)
end

function doUpdate(delay)
drawnow
pause(delay)
end
```

Control Legend Content

Control Legend Content

In this section...

[“Properties for Controlling Legend Content” on page 14-2](#)

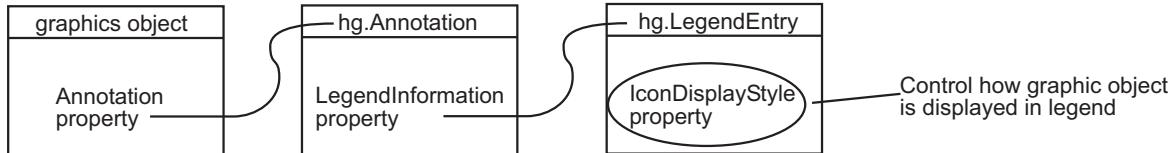
[“Updating a Legend” on page 14-3](#)

Properties for Controlling Legend Content

Graphics objects have two properties that control these options:

- **Annotation** — Controls whether the graphics object appears in the legend and determines if the object or its children appear in the legend.
- **DisplayName** — Specifies the text label used by the legend for the object. However, specifying a string with the legend commands resets the value of **DisplayName** property.

Accessing the Annotation Control Objects



Querying the **Annotation** property returns the handle of an **hg.Annotation** object. The **hg.Annotation** object has a property called **LegendInformation**, which contains an **hg.LegendEntry** object. The **hg.LegendEntry** object has a property called **IconDisplayStyle** that you can set to one of three values.

IconDisplayStyle Value	Behavior
on	Represent this object in a figure legend.
off	Do not include this object in a figure legend .
children	Display legend entries for this object's children and not the object itself (applies only to objects that have children, otherwise, the same as on).

For example, if `object_handle` is the handle of a graphics object, use the following statements to set the object's **IconDisplayStyle**. In this case, the graphics object,

`object_handle`, is not included in the legend because its `IconDisplayStyle` property is `off`.

```
hAnnotation = get(object_handle,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Updating a Legend

If a legend exists and you change its `IconDisplayStyle` setting, you must call `legend` to update the display. See the `legend` command for the options available.

Working with Graphics Objects

- “Graphics Object Handles” on page 15-2
- “Preallocate Arrays” on page 15-4
- “Test for Valid Handle” on page 15-5
- “Handles in Logical Expressions” on page 15-6
- “Graphics Arrays” on page 15-9

Graphics Object Handles

In this section...

[“What You Can Do with Handles” on page 15-2](#)

[“What You Cannot Do with Handles” on page 15-3](#)

What You Can Do with Handles

A handle refers to a specific instance of a graphics object. Use the object handle to set and query the values of the object properties.

When you create graphics objects, you can save the handle to the object in a variable. For example:

```
x = 1:10;
y = x.^2;
plot(x,y);
h = text(5,25, '*'(5,25));
```

The variable `h` refers to this particular text object `'*(5,25)'`, which is located at the point `5, 25`. Use the handle `h` to query and set the properties of this text object.

Set font size

```
h.FontSize = 12;
```

Get font size

```
h.FontSize
```

```
ans =
```

```
12
```

Make a copy of the variable `h`. The copy refers to the same object. For example, the following statements create a copy of the handle, but not the object:

```
hNew = h;
hNew.FontAngle = 'italic';
h.FontAngle
```

```
ans =
```

italic

What You Cannot Do with Handles

Handles variables are objects. Do not attempt to perform operations involving handles that convert the handles to a numeric, character, or any other type. For example, you cannot:

- Perform arithmetic operations on handles.
- Use handles directly in logical statements without converting to a logical value.
- Rely on the numeric values of figure handles (integers) in logical statements.
- Combine handles with data in numeric arrays.
- Convert handles to strings or use handles in string operations.

More About

- “Graphics Arrays” on page 15-9
- “Dominant Argument in Overloaded Plotting Functions”

Preallocate Arrays

Use the `gobjects` function to preallocate arrays for graphics objects. You can fill in each element in the array with a graphics object handle.

Preallocate a 4-by-1 array:

```
h = gobjects(4,1);
```

Assign axes handles to the array elements:

```
for k=1:4
    h(k) = subplot(2,2,k);
end
```

`gobjects` returns a `GraphicsPlaceholder` array. You can replace these placeholders elements with any type of graphics object. You must use `gobjects` to preallocate graphics object arrays to ensure compatibility among all graphics objects that are assigned to the array.

Test for Valid Handle

Use `isgraphics` to determine if a variable is a valid graphics object handle. A handle variable (`h` in this case) can still exist, but not be a valid handle if the object to which it refers has been deleted.

```
h = plot(1:10);
...
close % Close the figure containing the plot
whos
```

Name	Size	Bytes	Class	Attributes
h	1x1	104	matlab.graphics.chart.primitive.Line	

Test the validity of `h`:

```
isgraphics(h)
```

```
ans =
```

```
0
```

For more information on deleted handles, see “Deleted Handle Objects”.

Handles in Logical Expressions

In this section...

[“If Handle Is Valid” on page 15-6](#)

[“If Result Is Empty” on page 15-6](#)

[“If Handles Are Equal” on page 15-7](#)

Handle objects do not evaluate to logical `true` or `false`. You must use the function that tests for the state of interest and returns a logical value.

If Handle Is Valid

Use `isgraphics` to determine if a variable contains a valid graphics object handle. For example, suppose `hobj` is a variable in the workspace. Before operating on this variable, test its validity:

```
if isgraphics(hobj)
    ...
end
```

You can also determine the type of object:

```
if isgraphics(hobj, 'figure')
    ...% hobj is a figure handle
end
```

If Result Is Empty

You cannot use empty objects directly in logical statements. Use `isempty` to return a logical value that you can use in logical statements.

Some properties contain the handle to other objects. In cases where the other object does not exist, the property contains an empty object:

```
close all
hRoot = groot;
hRoot.CurrentFigure

ans =
```

0x0 empty GraphicsPlaceholder array.

For example, to determine if there is a current figure by querying the root CurrentFigure property, use the isempty function:

```
hRoot = groot;
if isempty(hRoot.CurrentFigure)
    ... % There is a current figure
end
```

Another case where code can encounter an empty object is when searching for handles. For example, suppose you set a figure's Tag property with the string myFigure and you use findobj to get the handle of this figure:

```
if isempty(findobj('Tag','myFigure'))
    ... % That figure was NOT found
end
```

findobj returns an empty object if there is no match.

If Handles Are Equal

There are two states of being equal for handles:

- Any two handles refer to the same object (test with ==).
- The objects referred to by any two handles are the same class, and all properties have the same values (test with isequal).

Suppose you want to determine if h is a handle to a particular figure that has a value of myFigure for its Tag property:

```
if h == findobj('Tag','myFigure')
    ...% h is correct figure
end
```

If you want to determine if different objects are in the same state, use isequal:

```
hLine1 = line;
hLine2 = line;
isequal(hLine1,hLine2)

ans =
```


Graphics Arrays

Graphics arrays can contain the handles of any graphics objects. For example, this call to the `plot` function returns an array containing five line object handles:

```
y = rand(20,5);  
h = plot(y)  
  
h =
```

5x1 Line array:

```
Line  
Line  
Line  
Line  
Line  
Line
```

This array contains only handles to line objects. However, graphics arrays can contain more than one type of graphics object. That is, graphics arrays can be heterogeneous.

For example, you can concatenate the handles of the figure, axes, and line objects into one array, `harray`:

```
hf = figure;  
ha = axes;  
h1 = plot(1:10);  
harray = [hf,ha,h1]  
  
harray =  
  
1x3 graphics array:  
  
Figure      Axes      Line
```

Graphics arrays follow the same rules as any MATLAB array. For example, array element dimensions must agree. In this code, `plot` returns a 5-by-1 Line array:

```
hf = figure;  
ha = axes;  
h1 = plot(rand(5));  
harray = [hf,ha,h1];  
Error using horzcat  
Dimensions of matrices being concatenated are not consistent.
```

To form an array, you must transpose `h1`:

```
harray = [hf,ha,h1']  
harray =  
  
1x7 graphics array:  
  
Figure Axes Line Line Line Line Line
```

You cannot concatenate numeric data with object handles, with the exception of the unique identifier specified by the figure `Number` property. For example, if there is an existing figure with its `Number` property set to 1, you can refer to that figure by this number:

```
figure(1)  
aHandle = axes;  
[aHandle,1]  
  
ans =  
  
1x2 graphics array:  
  
Axes Figure
```

The same rules for array formation apply to indexed assignment. For example, you can build a handle array with a `for` loop:

```
harray = gobjects(1,7);  
hf = figure;  
ha = axes;  
h1 = plot(rand(5));  
harray(1) = hf;  
harray(2) = ha;  
for k = 1:length(h1)  
    harray(k+2) = h1(k);  
end
```

Object Identification

- “Special Object Identifiers” on page 16-2
- “Find Objects” on page 16-5
- “Copy Objects” on page 16-11
- “Delete Graphics Objects” on page 16-14

Special Object Identifiers

In this section...

[“Getting Handles to Special Objects” on page 16-2](#)

[“The Current Figure, Axes, and Object” on page 16-2](#)

[“Callback Object and Callback Figure” on page 16-4](#)

Getting Handles to Special Objects

MATLAB provides functions that return important object handles so that you can obtain these handles whenever you require them.

These object include:

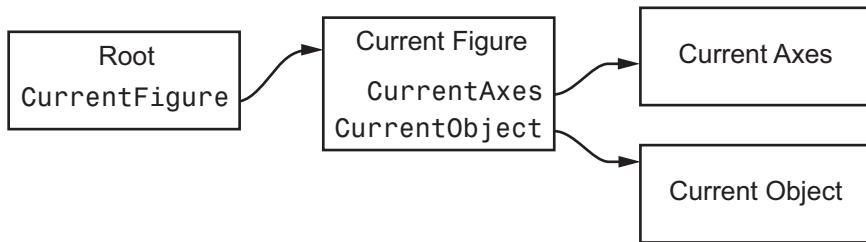
- Current figure — Handle of the figure that is the current target for graphics commands.
- Current axes— Handle of the axes in the current figure that is the target for graphics commands.
- Current object — Handle of the object that is selected
- Callback object — Handle of the object whose callback is executing.
- Callback figure — Handle of figure that is the parent of the callback object.

The Current Figure, Axes, and Object

An important concept in MATLAB graphics is that of being the current object. Being current means that object is the target for any action that affects objects of that type. There are three objects designated as current at any point in time:

- The *current figure* is the window designated to receive graphics output.
- The *current axes* is the axes in which plotting functions display graphs.
- The *current object* is the most recent object created or selected.

MATLAB stores the three handles corresponding to these objects in the ancestor's corresponding property.



These properties enable you to obtain the handles of these key objects:

```

hRoot = groot;
hFigure = hRoot.CurrentFigure;
hAxes = hFigure.CurrentAxes;
hobj = hFigure.CurrentObject;

```

Convenience Functions

The following commands are shorthand notation for the property queries.

- **gcf** — Returns the value of the root **CurrentFigure** property or creates a figure if there is no current figure. A figure with its **HandleVisibility** property set to **off** cannot become the current figure.
- **gca** — Returns the value of the current figure's **CurrentAxes** property or creates an axes if there is no current axes. An axes with its **HandleVisibility** property set to **off** cannot become the current axes.
- **gco** — Returns the value of the current figure's **CurrentObject** property.

Use these commands as input arguments to functions that require object handles. For example, you can click a line object and then use **gco** to specify the handle to the **set** command,

```
set(gco, 'Marker', 'square')
```

or click in an axes to set an axes property:

```
set(gca, 'Color', 'black')
```

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 code files. Especially true if your code is part of an application layered on MATLAB where you do not have knowledge of user actions that can change these values.

For information on how to prevent users from accessing the handles of graphics objects that you want to protect, see “Prevent Access to Figures and Axes” on page 9-14.

Callback Object and Callback Figure

Callback functions often require information about the object that defines the callback or the figure that contains the objects whose callback is executing. To obtain handles, these objects, use these convenience functions:

- `gcbo` — Returns the value of the `CallbackObject` property. This property contains the handle of the object whose callback is executing. `gcbo` optionally returns the handle of the figure containing the callback object.
- `gcbf` — Returns the handle of the figure containing the callback object.

MATLAB keeps the value of the `CallbackObject` property in sync with the currently executing callback. If one callback interrupts an executing callback, MATLAB updates the value of `CallbackObject` property.

When writing callback functions for the `CreateFcn` and `DeleteFcn`, always use `gcbo` to reference the callback object

For more information on writing callback functions, see “Callback Definition” on page 11-4

Find Objects

In this section...

- “Find Objects with Specific Property Values” on page 16-5
- “Find Text by String Property” on page 16-5
- “Use Regular Expressions with `findobj`” on page 16-7
- “Limit Scope of Search” on page 16-9

Find Objects with Specific Property Values

The `findobj` function can scan the object hierarchy to obtain the handles of objects that have specific property values.

For identification, all graphics objects have a `Tag` property that you can set to any string. You can then search for the specific property/value pair. For example, suppose you create a check box that is sometimes inactivated in the UI. By assigning a unique value for the `Tag` property, you can find that particular object:

```
uicontrol('Style','checkbox','Tag','save option')
```

Use `findobj` to locate the object whose `Tag` property is set to '`save option`' and disable it:

```
hCheckbox = findobj('Tag','save option');  
hCheckbox.Enable = 'off'
```

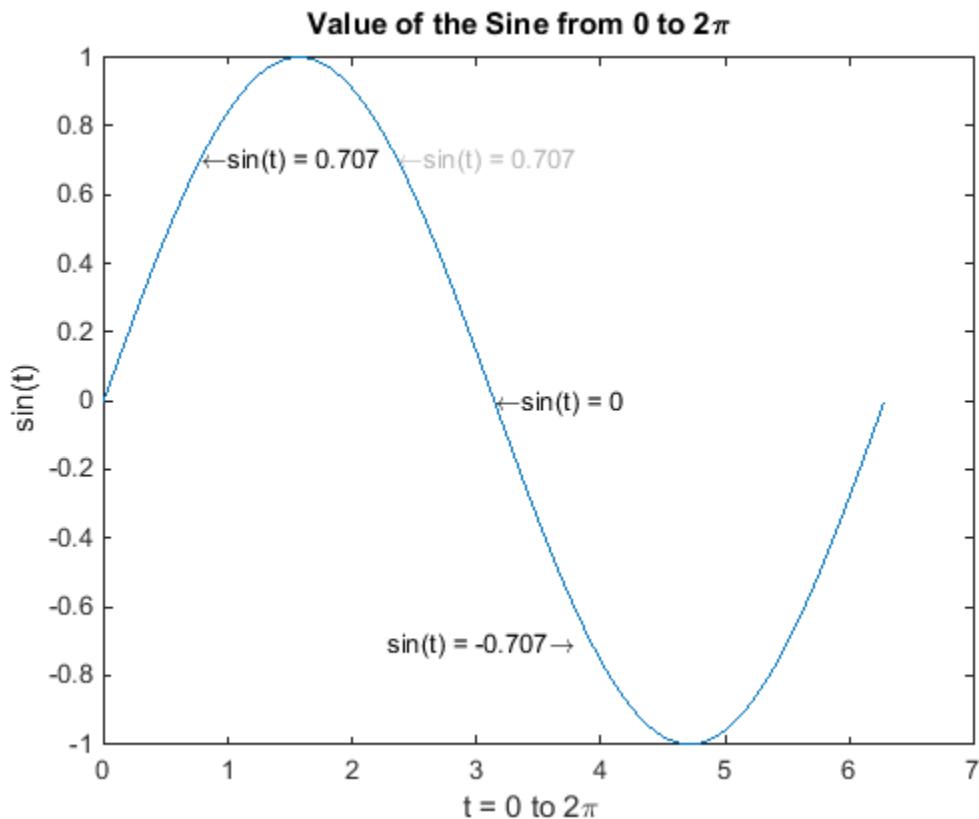
If you do not specify a starting object, `findobj` searches from the root object, finding all occurrences of the property name/property value combination that you specify.

To find objects with hidden handles, use `findall`.

Find Text by String Property

This example shows how to find the handles of text objects using the text string.

The following graph contains text objects labeling particular values of the function.



Suppose that 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 in light gray out in the graph).

Determine the handle of the text object labeling the point $[\pi/4, \sin(\pi/4)]$ and change its **Position** property.

To use **findobj**, pick a property value that uniquely identifies the object. This example uses the text **String** property:

```
hText = findobj('String','\leftarrow sin(t) = .707');
```

Move the object to the new position, defining the text **Position** in axes units.

```
hText.Position = [3*pi/4,sin(3*pi/4),0];
```

`findobj` lets you restrict the search by specifying a starting point in the hierarchy, instead of beginning with the root object. If there are many objects in the object tree, this capability results in faster searches. In the previous example, you know the text object of interest is in the current axes, so you can type:

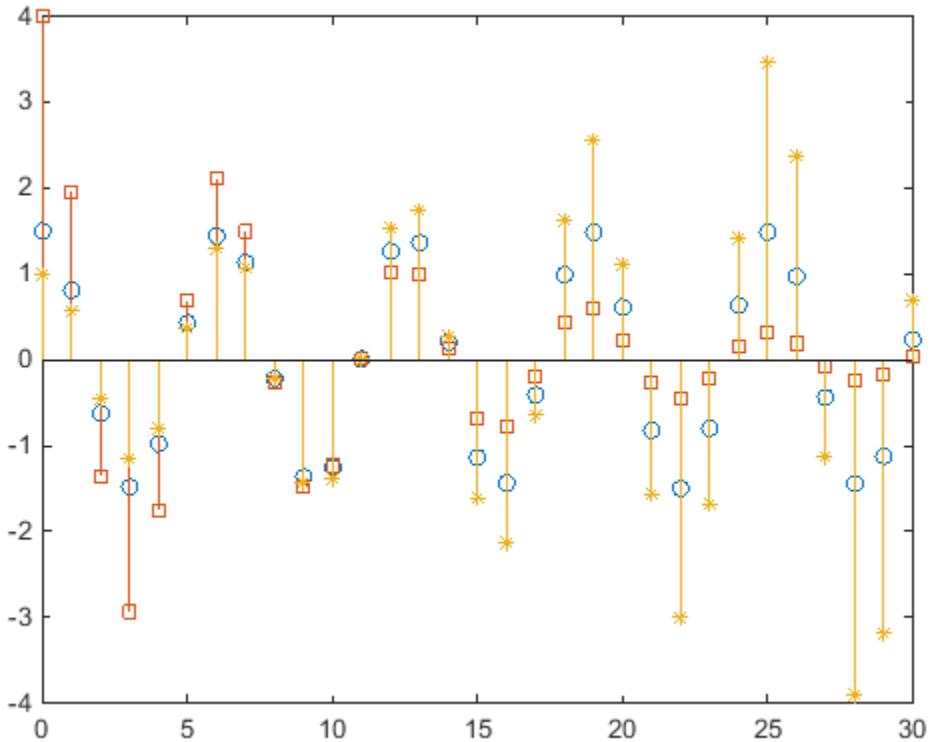
```
hText = findobj(gca,'String','\leftarrowsin(t) = .707');
```

Use Regular Expressions with `findobj`

This example shows how to find object handles using regular expressions to identify specific property values. For more information about regular expressions, see `regexp`.

Suppose you create the following graph and want to modify certain properties of the objects created.

```
x = 0:30;
y = [1.5*cos(x);4*exp(-.1*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
h(1).Marker = 'o';
h(1).Tag = 'Decaying Exponential';
h(2).Marker = 'square';
h(2).Tag = 'Growing Exponential';
h(3).Marker = '*';
h(3).Tag = 'Steady State';
```



Passing a regular expression to `findobj` enables you to match very specific patterns. For example, suppose you want to set the value of the `MarkerFaceColor` property to green on all stem objects that do *not* have their `Tag` property set to 'Steady State' (that is, stems that represent decaying and growing exponentials).

```
hStems = findobj( '-regexp' , 'Tag' , '^(!Steady State$) . ' );
for k = 1:length(hStems)
    hStems(k).MarkerFaceColor = 'green'
end
```

Limit Scope of Search

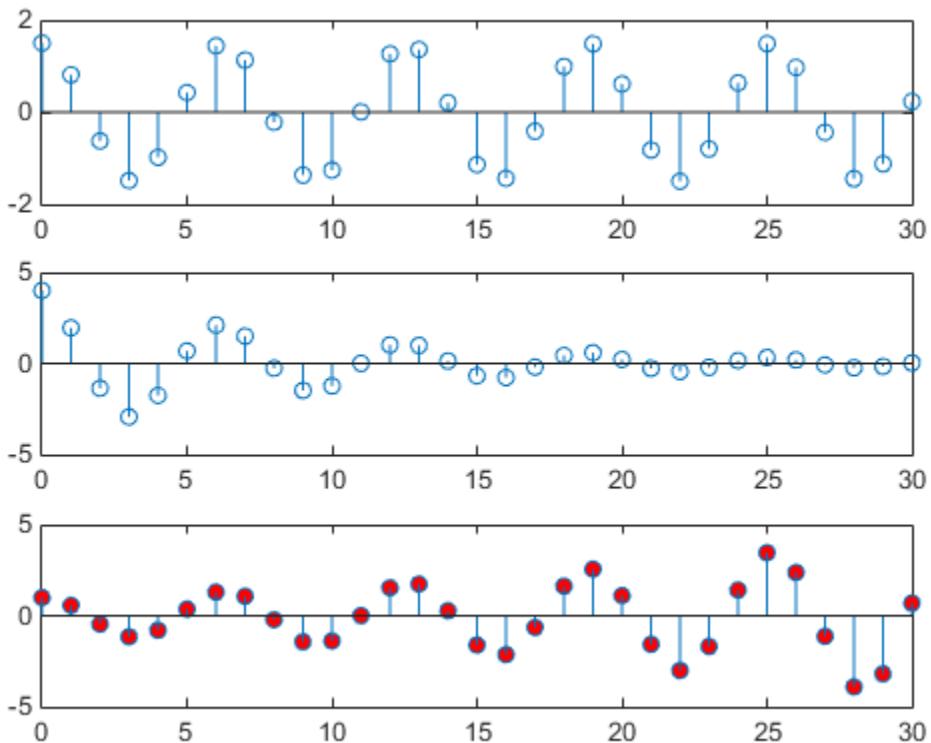
Specify the starting point in the object tree to limit the scope of the search. The starting point can be the handle of a figure, axes, or a group of object handles.

For example, suppose you want to change the marker face color of the stems in a specific axes:

```
x = 0:30;
y = [1.5*cos(x);4*exp(-.1*x).*cos(x);exp(.05*x).*cos(x)]';
ax(1) = subplot(3,1,1);
stem(x,y(:,1))
ax(2) = subplot(3,1,2);
stem(x,y(:,2))
ax(3) = subplot(3,1,3);
stem(x,y(:,3))
```

Set the marker face color of the stems in the third axes only.

```
h = findobj(ax(3), 'Type', 'stem');
h.MarkerFaceColor = 'red';
```



For more information on limiting the scope and depth of an object search, see **findobj** and **findall**.

Copy Objects

In this section...

[“Copying Objects with copyobj” on page 16-11](#)

[“Copy Single Object to Multiple Destinations.” on page 16-11](#)

[“Copying Multiple Objects” on page 16-12](#)

Copying Objects with `copyobj`

Copy objects from one parent to another using the `copyobj` function. The copy differs from the original:

- The `Parent` property is now the new parent.
- The copied object's handle is different from the original.
- `copyobj` does not copy the original object's callback properties
- `copyobj` does not copy any application data associated with the original object.

Therefore, `==` and `isequal` return false when comparing original and new handles.

You can copy a number of objects to a new parent, or one object to a number of new parents, as long as the result maintains the correct parent/child relationship. When you copy an object having child objects, MATLAB copies all children too.

Note: You cannot copy the same object more than once to the same parent in a single call to `copyobj`.

Copy Single Object to Multiple Destinations.

When copying a single object to multiple destinations, the new handles returned by `copyobj` are in the same order as the parent handles.

```
h = copyobj(cobj,[newParent1,newParent2,newParent3])
```

The returned array `h` contains the new object handles in the order shown:

```
h(1) -> newParent1  
h(2) -> newParent2  
h(3) -> newParent3
```

Copying Multiple Objects

This example shows how to copy multiple objects to a single parent.

Suppose you create a set of similar graphs and want to label the same data point on each graph. You can copy the text and marker objects used to label the point in the first graph to each subsequent graph.

Create and label the first graph:

```
x = 0:.1:2*pi;
plot(x,sin(x))
hText = text('String','\{5\pi\div4, sin(5\pi\div4)\}\rightarrow',...
    'Position',[5*pi/4,sin(5*pi/4),0],...
    'HorizontalAlignment','right');
hMarker = line(5*pi/4,sin(5*pi/4),0,'Marker','*');
```

Create two more graphs without labels:

```
figure
x = pi/4:.1:9*pi/4;
plot(x,sin(x))
hAxes1 = gca;

figure
x = pi/2:.1:5*pi/2;
plot(x,sin(x))
hAxes2 = gca;
```

Copy the text and marker (`hText` and `hMarker`) to each graph by parenting them to the respective axes. Return the new handles for the text and marker copies:

```
newHandles1 = copyobj([hText,hMarker],hAxes1);
newHandles2 = copyobj([hText,hMarker],hAxes2);
```

Because the objective is to copy both objects to each axes, you must call `copyobj` twice, each time with a single destination axes.

Copy Multiple Objects to Multiple Destinations

When you call `copyobj` with multiple objects to copy and multiple parent destinations, `copyobj` copies respective objects to respective parents. That is, if `h` and `p` are handle arrays of length `n`, then this call to `copyobj`:

`copyobj(h,p)`

results in an element-by-element copy:

```
h(1) -> p(1);  
h(2) -> p(2);  
...  
h(n) -> p(n);
```

Delete Graphics Objects

In this section...

[“How to Delete Objects” on page 16-14](#)

[“Handles to Deleted Objects” on page 16-15](#)

How to Delete Objects

You can remove a graphics object with the **delete** function, using the object's handle as an argument. For example, delete the current axes, and all the objects contained in the axes, with the statement:

```
delete(gca)
```

If you want to delete multiple objects, pass an array of handles to **delete** as a single argument. For example, if **h1**, **h2**, and **h3** are handles to objects that you want to delete, for a single array:

```
hToDelete = [h1,h2,h3];
delete(hToDelete)
```

Closing a figure deletes all the objects contained in the figure. For example, create a bar graph:

```
f = figure;
y = rand(1,5);
bar(y)
```

The figure now contains axes and bar objects:

```
ax = f.Children;
b = ax.Children;
whos
```

Name	Size	Bytes	Class	Attributes
ax	1x1	112	matlab.graphics.axis.Axes	
b	1x1	112	matlab.graphics.chart.primitive.Bar	
f	1x1	112	matlab.ui.Figure	
y	1x5	40	double	

Close the figure:

```
close(f)
```

MATLAB deletes each object, but the handle variable still exists:

```
>>f  
f =  
    handle to deleted Figure  
>> ax  
ax =  
    handle to deleted Axes  
b  
b =  
    handle to deleted Bar
```

Handles to Deleted Objects

When you delete a graphics object, MATLAB does not delete the variable or variables that contained the object handle. However, these variables become invalid handles because the object they referred to no longer exists.

The handle to a deleted object no longer refers to a valid object. Use `isgraphics` to determine the validity of a handle to a graphics object:

```
y = rand(1,5);  
h = bar(y);  
delete(h)  
isgraphics(h)  
  
ans =  
  
0
```

You can also use the `isValid` method to determine if any handle variable is a valid handle.

You cannot access properties with the invalid handle variable:

```
h.FaceColor
```

Invalid or deleted object.

To remove the variable, use `clear`:

```
clear h
```

For information on getting the handle of a valid object for which you have not saved a handle variable, see “Find Objects” on page 16-5.

Optimize Performance of Graphics Programs

- “Finding Code Bottlenecks” on page 17-2
- “What Affects Code Execution Speed” on page 17-4
- “Judicious Object Creation” on page 17-6
- “Avoid Repeated Searches for Objects” on page 17-8
- “Screen Updates” on page 17-10
- “Getting and Setting Properties” on page 17-12
- “Avoid Updating Static Data” on page 17-15
- “Animating Line Graphs” on page 17-17
- “Transforming Objects Efficiently” on page 17-18
- “Use Low-Level Functions for Speed” on page 17-19
- “Using `drawnow` Efficiently” on page 17-20
- “System Requirements for Graphics” on page 17-23
- “Workarounds for Older Graphics Hardware” on page 17-25

Finding Code Bottlenecks

Use the code profiler to determine which functions contribute the most time to execution time. You can make performance improvements by reducing the execution times of your algorithms and calculations wherever possible.

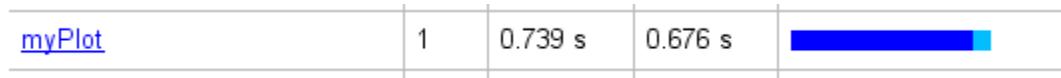
Once you have optimized your code, use the following techniques to reduce the overhead of object creation and updating the display.

For example, suppose you are plotting 10-by-1000 element arrays using the `myPlot` function:

```
function myPlot
    x = rand(10,1000);
    y = rand(10,1000);
    plot(x,y,'LineStyle','none','Marker','o','Color','b');
end

profile on
myPlot
profile viewer
```

When you profile this code, you see that most time is spent in the `myPlot` function:

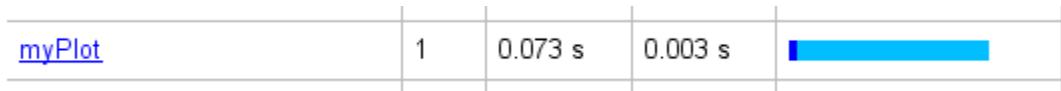


Because the `x` and `y` arrays contain 1000 columns of data, the `plot` function creates 1000 line objects. In this case, you can achieve the same results by creating one line with 10000 data points:

```
function myPlot
    x = rand(10,1000);
    y = rand(10,1000);
    % Pass x and y as 1-by-1000 vectors
    plot(x(:,1),y(:,1),'LineStyle','none','Marker','o','Color','b');
end

profile on
myPlot
profile viewer
```

Object creation time is a major factor in this case:



You can often achieve improvements in execution speed by understanding how to avoid or minimize inherently slow operations. For information on how to improve performance using this tool, see the documentation for the `profile` function.

What Affects Code Execution Speed

In this section...

[“Potential Bottlenecks” on page 17-4](#)

[“How to Improve Performance” on page 17-4](#)

Potential Bottlenecks

Performance becomes an issue when working with large amounts of data and large numbers of objects. In such cases, you can improve the execution speed of graphics code by minimizing the effect of two factors that contribute to total execution time:

- Object creation — Adding new graphics objects to a scene.
- Screen updates — Updating the graphics model and sending changes to be rendered.

It is often possible to prevent these activities from dominating the total execution time of a particular programming pattern. Think of execution time as being the sum of a number of terms:

$$T_{\text{execution time}} = T_{\text{creating objects}} + T_{\text{updating}} + (T_{\text{calculations, etc}})$$

The examples that follow show ways to minimize the time spent in object creation and updating the screen. In the preceding expression, the execution time does not include time spent in the actual rendering of the screen.

How to Improve Performance

Profile your code and optimize algorithms, calculation, and other bottlenecks that are specific to your application. Then determine if the code is taking more time in object creation functions or `drawnow` (updating). You can begin to optimize both operations, beginning with the larger term in the total time equation.

Is your code:

- Creating new objects instead of updating existing objects? See “[Judicious Object Creation](#)” on page 17-6.
- Updating an object that has some percentage of static data? See “[Avoid Updating Static Data](#)” on page 17-15.

- Searching for object handles. See “Avoid Repeated Searches for Objects” on page 17-8.
- Rotating, translating, or scaling objects? See “Transforming Objects Efficiently” on page 17-18.
- Querying and setting properties in the same loop? See “Getting and Setting Properties” on page 17-12.

Judicious Object Creation

In this section...

- “Object Overhead” on page 17-6
- “Do Not Create Unnecessary Objects” on page 17-6
- “Use NaNs to Simulate Multiple Lines” on page 17-7
- “Modify Data Instead of Creating New Objects” on page 17-7

Object Overhead

Graphics objects are complex structures that store information (data and object characteristics), listen for certain events to occur (callback properties), and can cause changes to other objects to accommodate their existence (update to axes limits, and so on). Therefore, creating an object consumes resources.

When performance becomes an important consideration, try to realize your objectives in a way that consumes a minimum amount of resources.

You can often improve performance by following these guidelines:

- Do not create unnecessary objects
- Avoid searching the object hierarchy

Do Not Create Unnecessary Objects

Look for cases where you can create fewer objects and achieve the same results. For example, suppose you want to plot a 10-by-1000 array of points showing only markers.

This code creates 1000 line objects:

```
x = rand(10,1000);
y = rand(10,1000);
plot(x,y,'LineStyle','none','Marker','.', 'Color','b');
```

Convert the data from 10-by-1000 to 1000-by-1. This code creates a graph that looks the same, but creates only one object:

```
plot(x(:),y(:),'LineStyle','none','Marker','.', 'Color','b')
```

Use NaNs to Simulate Multiple Lines

If coordinate data contains NaNs, MATLAB does not render those points. You can add NaNs to vertex data to create line segments that look like separate lines. Place the NaNs at the same element locations in each vector of data. For example, this code appears to create three separate lines:

```
x = [0:10,NaN,20:30,NaN,40:50];  
y = [0:10,NaN,0:10,NaN,0:10];  
line(x,y)
```

Modify Data Instead of Creating New Objects

To view different data on what is basically the same graph, it is more efficient to update the data of the existing objects (lines, text, etc.) rather than recreating the entire graph.

For example, suppose you want to visualize the effect on your data of varying certain parameters.

- 1 Set the limits of any axis that can be determined in advance, or set the axis limits modes to `manual`.
- 2 Recalculate the data using the new parameters.
- 3 Use the new data to update the data properties of the lines, text, etc. objects used in the graph.
- 4 Call `drawnow` to update the figure (and all child objects in the figure).

For example, suppose you want to update a graph as data changes:

```
figure  
z = peaks;  
h = surf(z);  
drawnow  
zlim([min(z(:)), max(z(:))]);  
for k = 1:50  
    h.ZData = (0.01+sin(2*pi*k/20)*z);  
    drawnow  
end
```

Avoid Repeated Searches for Objects

When you search for handles, MATLAB must search the object hierarchy to find matching handles, which is time-consuming. Saving handles that you need to access later is a faster approach. Array indexing is generally faster than using `findobj` or `findall`.

This code creates 500 line objects and then calls `findobj` in a loop.

```
figure
ax = axes;
for ix=1:500
    line(rand(1,5),rand(1,5), 'Tag', num2str(ix), 'Parent', ax);
end
drawnow;
for ix=1:500
    h = findobj(ax, 'Tag', num2str(ix));
    set(h, 'Color', rand(1,3));
end
drawnow;
```

A better approach is to save the handles in an array and index into the array in the second `for` loop.

```
figure
ax = axes;
h = gobjects(1,500);
for ix = 1:500
    h(ix) = line(rand(1,5),rand(1,5), 'Tag', num2str(ix), 'Parent', ax);
end
drawnow;
% Index into handle array
for ix=1:500
    set(h(ix), 'Color', rand(1,3));
end
drawnow
```

Limit Scope of Search

If searching for handles is necessary, limit the number of objects to be searched by specifying a starting point in the object tree. For example, specify the starting point as the figure or axes containing the objects for which you are searching.

Another way to limit the time expended searching for objects is to restrict the depth of the search. For example, a '`flat`' search restricts the search to the objects in a specific handle array.

Use the `findobj` and `findall` functions to search for handles.

For more information, see “Find Objects”

Screen Updates

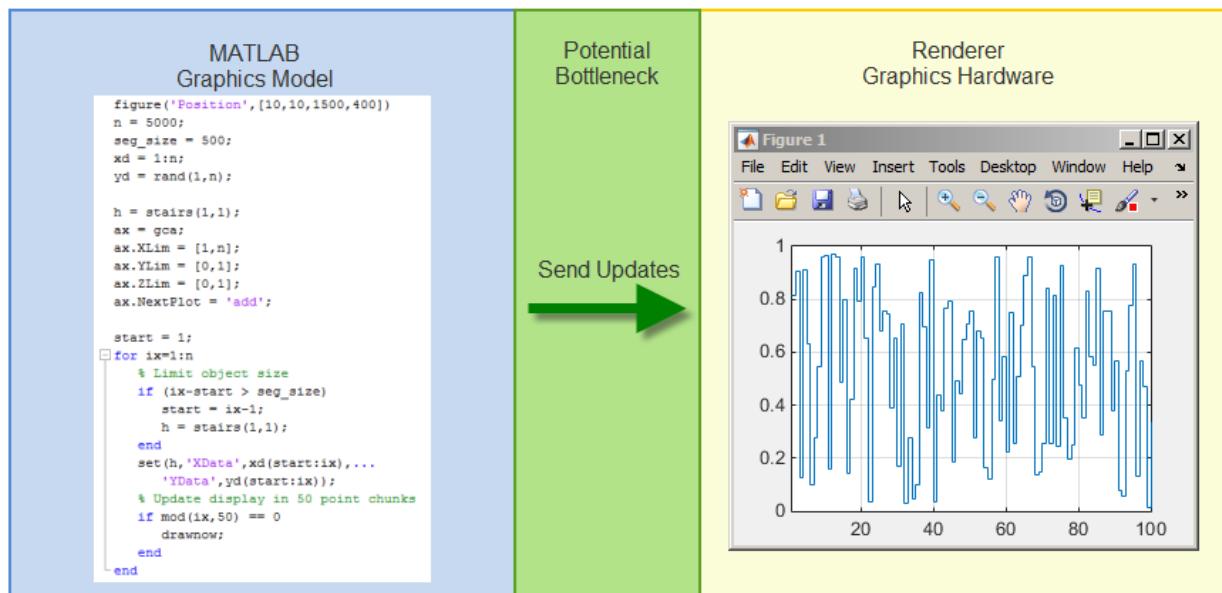
In this section...

[“MATLAB Graphics System” on page 17-10](#)

[“Managing Updates” on page 17-11](#)

MATLAB Graphics System

MATLAB graphics is implemented using multiple threads of execution. The following diagram illustrates how the main and renderer threads interact during the update process. The MATLAB side contains the graphics model, which describes the geometry rendered by the graphics hardware. The renderer side has a copy of the geometry in its own memory system. The graphics hardware can render the screen without blocking MATLAB execution.



When the graphics model changes, these updates must be passed to the graphics hardware. Sending updates can be a bottleneck because the graphics hardware does

not support all MATLAB data types. The update process must convert the data into the correct form.

When geometry is in the graphics hardware memory, you can realize performance advantages by using this data and minimizing the data sent in an update.

Managing Updates

Updates involve these steps:

- Collecting changes that require an update to the screen, such as property changes and objects added.
- Updating dependencies within the graphics model.
- Sending these updates to the renderer.
- Waiting for the renderer to accept these updates before returning execution to MATLAB.

You initiate an update by calling the `drawnow` function. `drawnow` completes execution when the renderer accepts the updates, which can happen before the renderer completes updating the screen.

Explicit Updates

During function execution, adding graphics objects to a figure or changing properties of existing objects does not necessarily cause an immediate update of the screen. The update process occurs when there are changes to graphics that need to be updated, and the code:

- Calls `drawnow`, `pause`, `figure`, or other functions that effectively cause an update (see `drawnow`).
- Queries a property whose value depends on other properties (see “Automatically Calculated Properties” on page 17-12).
- Completes execution and returns control to the MATLAB prompt or debugger.

Getting and Setting Properties

In this section...

- “Automatically Calculated Properties” on page 17-12
- “Inefficient Cycles of Sets and Gets” on page 17-13
- “Changing Text Extent to Rotate Labels” on page 17-14

Automatically Calculated Properties

Certain properties have dependencies on the value of other properties. MATLAB automatically calculates the values of these properties and updates their values based on the current graphics model. For example, axis limits affect the values used for axis ticks, which, in turn, affect the axis tick labels.

When you query a calculated property, MATLAB performs an implicit `drawnow` to ensure all property values are up to date before returning the property value. The query causes a full update of all dependent properties and an update of the screen.

MATLAB calculates the values of certain properties based on other values on which that property depends. For example, plotting functions automatically create an axes with axis limits, tick labels, and a size appropriate for the plotted data and the figure size.

MATLAB graphics performs a full update, if necessary, before returning a value from a calculated property to ensure the returned value is up to date.

Object	Automatically Calculated Properties
Axes	<code>CameraPosition</code> , <code>CameraTarget</code> , <code>CameraUpVector</code> , <code>CameraViewAngle</code>
	<code>Position</code> , <code>OuterPosition</code> , <code>TightInset</code>
	<code>XLim</code> , <code>YLim</code> , <code>ZLim</code>
	<code>XTick</code> , <code>YTick</code> , <code>ZTick</code> , <code>XMinorTick</code> , <code>YMinorTick</code> , <code>ZMinorTick</code>
	<code>XTickLabel</code> , <code>YTickLabel</code> , <code>ZTickLabel</code> , <code>TickDir</code>
	<code>SortMethod</code>
Text	<code>Extent</code>

Inefficient Cycles of Sets and Gets

When you set property values, you change the state of the graphics model and mark it as needing to be updated. When you query an autocalculated property, MATLAB needs to perform an update if the graphics model and graphics hardware are not in sync.

When you get and set properties in the same loop, you can create a situation where updates are performed with every pass through the loop.

- The `get` causes an update.
- The `set` marks the graphics model as needing an update.

The cycle is repeated with each pass through the loop. It is better to execute all property queries in one loop, then execute all property sets in another loop, as shown in the following example.

This example gets and sets the text `Extent` property.

Code with Poor Performance	Code with Better Performance
<pre> h = gobjects(1,500); p = zeros(500,3); for ix = 1:500 h(ix) = text(ix/500,ix/500,num2str(ix)); end drawnow % Gets and sets in the same loop, % prompting a full update at each pass for ix = 1:500 pos = get(h(ix), 'Position'); ext = get(h(ix), 'Extent'); p(ix,:) = [pos(1)+(ext(3)+ext(1)), ... pos(2)+ext(2)+ext(4),0]; set(h(ix), 'Position',p(ix,:)) end drawnow </pre>	<pre> h = gobjects(1,500); p = zeros(500,3); for ix = 1:500 h(ix) = text(ix/500,ix/500,num2str(ix)); end drawnow % Get and save property values for ix=1:500 pos = get(h(ix), 'Position'); ext = get(h(ix), 'Extent'); p(ix,:) = [pos(1)+(ext(3)+ext(1)), ... pos(2)+ext(2)+ext(4),0]; end % Set the property values and % call a drawnow after the loop for ix=1:500 set(h(ix), 'Position',p(ix,:)); end drawnow </pre>
<p>This code performs poorly because:</p> <ul style="list-style-type: none"> • The <code>Extent</code> property depends on other values, such as screen resolution, figure size, and axis limits, so querying this property can cause a full update. 	<p>The performance is better because this code:</p> <ul style="list-style-type: none"> • Queries all property values in one loop and stores these values in an array. • Sets all property values in a separate loop. • Calls <code>drawnow</code> after the second loop finishes.

Code with Poor Performance	Code with Better Performance
<ul style="list-style-type: none">• Each set of the <code>Position</code> property makes a full update necessary when the next get of the <code>Extent</code> property occurs.	

Changing Text Extent to Rotate Labels

In cases where you change the text `Extent` property to rotate axes labels, it is more efficient to use the axes properties `XTickLabelRotation`, `YTickLabelRotation`, and `ZTickLabelRotation`.

Avoid Updating Static Data

If only a small portion of the data defining a graphics scene changes with each update of the screen, you can improve performance by updating only the data that changes. The following example illustrates this technique.

Code with Poor Performance	Code with Better Performance
<p>In this example, a marker moves along the surface by creating both objects with each pass through the loop.</p> <pre>[sx,sy,sz] = peaks(500); nframes = 490; for t = 1:nframes surf(sx,sy,sz,'EdgeColor','none') hold on plot3(sx(t+10,t),sy(t,t+10),... sz(t+10,t+10)+0.5,'o',... 'MarkerFaceColor','red',... 'MarkerSize',14) hold off drawnow end</pre>	<p>Create the surface, then update the XData, YData, and ZData of the marker in the loop. Only the marker data changes in each iteration.</p> <pre>[sx,sy,sz] = peaks(500); nframes = 490; surf(sx,sy,sz,'EdgeColor','none') hold on h = plot3(sx(1,1),sy(1,1),sz(1,1),'o',... 'MarkerFaceColor','red',... 'MarkerSize',14); hold off for t = 1:nframes set(h,'XData',sx(t+10,t),... 'YData',sy(t,t+10),... 'ZData',sz(t+10,t+10)+0.5) drawnow end</pre>

Segmenting Data to Reduce Update Times

Consider the case where an object's data grows very large while code executes in a loop, such as a line tracing a signal over time.

With each call to **drawnow**, the updates are passed to the renderer. The performance decreases as the data arrays grow in size. If you are using this pattern, adopt the segmentation approach described in the example on the right.

Code with Poor Performance	Code with Better Performance
<pre>% Grow data figure('Position',[10,10,1500,400]) n = 5000; h = stairs(1,1); ax = gca; ax.XLim = [1,n]; ax.YLim = [0,1]; ax.ZLim = [0,1];</pre>	<pre>% Segment data figure('Position',[10,10,1500,400]) n = 5000; seg_size = 500; xd = 1:n; yd = rand(1,n); h = stairs(1,1); ax = gca; ax.XLim = [1,n];</pre>

Code with Poor Performance	Code with Better Performance
<pre> ax.NextPlot = 'add'; xd = 1:n; yd = rand(1,n); tic for ix = 1:n set(h, 'XData',xd(1:ix), 'YData',yd(1:ix)); drawnow; end toc </pre>	<pre> ax.YLim = [0,1]; ax.ZLim = [0,1]; ax.NextPlot = 'add'; tic start = 1; for ix=1:n % Limit object size if (ix-start > seg_size) start = ix-1; h = stairs(1,1); end set(h, 'XData',xd(start:ix),... 'YData',yd(start:ix)); % Update display in 50 point chunks if mod(ix,50) == 0 drawnow; end end toc </pre> <p>The performance of this code is better because the limiting factor is the amount of data sent during updates.</p>

Animating Line Graphs

The `matlab.graphics.animation.AnimatedLine` class defines an object that is designed specifically for animating line graphs. `AnimatedLine` objects provide a method to add points to an existing object. This method provides data segmentation and accepts vectors of points so that you can minimize the effects of the update.

Here is an example that uses an `AnimatedLine` object:

```
%% Segment with animatedline
figure('Position',[10,10,1000,400])
n = 10000;
axes('XLim',[1,n], 'YLim',[0,1], 'ZLim',[0,1]);
xd = 1:n;
yd = rand(1,n);
h = animatedline;
for ix = 1:50:n-50
    addpoints(h,xd(ix:ix+50),yd(ix:ix+50));
    drawnow update;
end
```

Interactive Lines with Markers

If you are creating lines that use many markers and you are clicking objects in the same axes to, for example, execute a button down callback function, you can improve performance by:

- Using the ' .' (the dot) marker.
- Setting the `PickableParts` property to 'none' for all objects in the axes that you do not want to capture mouse clicks.

Transforming Objects Efficiently

Moving objects, for example by rotation, requires transforming the data that defines the objects. You can improve performance by taking advantage of the fact that graphics hardware can apply transforms to the data. You can then avoid sending the transformed data to the renderer. Instead, you send only the four-by-four transform matrix.

To realize the performance benefits of this approach, use the `hgtransform` container to group the objects that you want to move.

The following examples define a sphere and rotate it using two techniques to compare performance:

- The `rotate` function transforms the sphere's data and sends the data to the renderer thread with each call to `drawnow`.
- The `hgtransform` sends the transform matrix for the same rotation to the renderer thread.

Code with Poor Performance	Code with Better Performance
<p>When object data is large, the update bottleneck becomes a limiting factor.</p> <pre>% Using rotate figure [x,y,z] = sphere(270); axis vis3d for ang = 1:360 rotate(s,[1,1,1],1) drawnow end</pre>	<p>Using an <code>hgtransform</code> applies the transform on the renderer side of the bottleneck.</p> <pre>% Using transform figure ax = axes; [x,y,z] = sphere(270); % hgtransform contains the surface grp = hgtransform('Parent',ax); s = surf(ax,x,y,z,'Parent',grp,... ... 'EdgeColor','none'); view(3) grid on axis vis3d % Apply the transform to the hgtransform content tic for ang = linspace(0,2*pi,360) tm = makehgform('axisrotate',[1,1,1],ang); grp.Matrix = tm; drawnow end toc</pre>

Use Low-Level Functions for Speed

The features that make plotting functions easy to use also consume computer resources. If you want to maximize graphing performance, use low-level functions and disable certain automatic features.

Low-level graphics functions (e.g., `line` vs. `plot`, `surface` vs. `surf`) perform fewer operations and therefore are faster when you are creating many graphics objects.

The low-level graphics functions are `line`, `patch`, `rectangle`, `surface`, `text`, `image`, `axes`, and `light`

Using `drawnow` Efficiently

In this section...

[“What Does `drawnow` Do?” on page 17-20](#)

[“How to Use `drawnow`” on page 17-21](#)

[“Achieve a Specific Frame Rate” on page 17-21](#)

What Does `drawnow` Do?

Calling `drawnow` updates everything. It updates the screen and processes all events.

Calling `drawnow` initiates a sequence of steps:

- 1 Perform an update — Update and collect all changes made to the state of displayed objects.
- 2 Send updates — Send data to the screen renderer.
- 3 Execute callbacks — Allow any pending callbacks to run.
- 4 Wait for pending events — Process pending events before returning execution to the calling function.
- 5 `drawnow` returns — MATLAB resumes execution.
- 6 Renderer updates screen — UI and graphics objects are updated.

`drawnow` has options that limit the steps to make updating quicker:

- `drawnow update` — Maximum loop speed when updating graphs. Useful for keeping up with user input or real-time data acquisition, but updates can be lost if renderer is busy.
 - Update graphics model and send updates if renderer is free. Otherwise return and discard the update, which can cause frames to be lost in an animation.
 - Send updates for both UI and graphics objects.
 - Do not process events.
- `drawnow expose` — Synchronized loop and movie playback.
 - Perform update.
 - Send updates for both UI and graphics objects.
 - Do not process events.

MATLAB performs an effective `drawnow` in a number of conditions (see `drawnow` documentation). However, judicious use of explicit calls to `drawnow` can improve performance of code that frequently updates the screen.

How to Use `drawnow`

Calling `drawnow` blocks MATLAB execution until updates are passed to the screen and pending events are processed. Some things to consider when using `drawnow`:

- What is your responsiveness criteria? For example, if you are animating a line graph, can you update the display every 50 points instead of every point? See “Segmenting Data to Reduce Update Times” on page 17-15.
- Are you streaming data and want to:
 - Update a graph as quickly as possible. Some updates might be lost — use `drawnow update`
 - Update a graph as quickly as possible without loosing updates — use `drawnow expose`
 - Maintain a fully interactive UI application, with possible slower performance — use `drawnow`
- Are you animating a sequence of graphs — use the `getframe` function to record a set of frames and the `movie` function to playback the set.

Achieve a Specific Frame Rate

Animations typically do not need to run at frame rates exceeding 30 `fps`. You can achieve a specific frame rate using the `clock`, `etime`, and `drawnow` functions.

This sample code animates a line graph at a specified frame rate (30 `fps`).

```
%> Create animatedline graph
x = linspace(0,100*pi,1e4);
ax = axes;
xlim([0 100*pi]);
ylim([-1 1]);
l = animatedline('Parent',ax);
drawnow;
%> Get start time
t1 = tic;
%> Update display at 30 fps
```

```
for i = 1:length(x)
    addpoints(l,x(i),sin(x(i)));
    t2 = toc(t1);
    if(t2 > 1/30)
        drawnow
        t1 = tic;
    end
end
```

If your objective is to update the display as fast as possible, use `drawnow update` and reduce the for loop to:

```
for ix = 1:50:n-50
    addpoints(l,x(i),sin(x(i)));
    drawnow update;
end
```

For an example, see “Animating Line Graphs” on page 17-17.

System Requirements for Graphics

In this section...

[“Recommended System Requirements” on page 17-23](#)

[“Upgrade Your Graphics Drivers” on page 17-23](#)

[“Features with OpenGL Requirements” on page 17-24](#)

Recommended System Requirements

All systems support most of the common MATLAB graphics features. For the best results with graphics, your system must have:

- At least 1 GB of GPU memory.
- Graphics hardware that supports a hardware-accelerated implementation of OpenGL 2.1 or later. Most graphics hardware released since 2006 is compliant. For advanced graphics, OpenGL 3.0 or later is recommended.
- The latest versions of graphics drivers available from your computer manufacturer or graphics hardware vendor.

Upgrade Your Graphics Drivers

Upgrade your graphics drivers to the latest versions available.

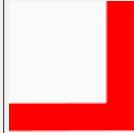
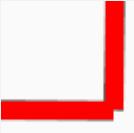
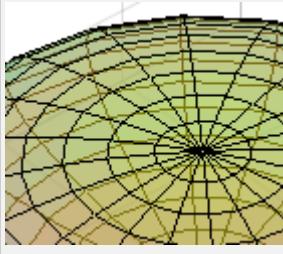
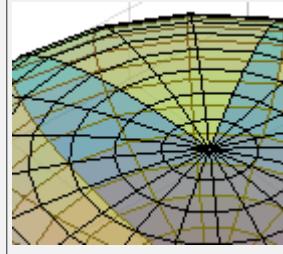
- On Windows systems, check your computer manufacturer website for driver updates, such as Dell® or HP. If no updates are provided, then check your graphics hardware vendor website, such as the AMD® website, NVIDIA® website, or Intel® website.
- On Linux® systems, use proprietary vendor drivers instead of open-source replacements.
- On Mac OS X systems, the graphics drivers are part of the operating system. Use the latest updates provided.

For more information on determining your graphics hardware, see `opengl`.

If MATLAB detects graphics drivers with known issues or graphics virtualization, then it uses software OpenGL instead of your graphics hardware.

Features with OpenGL Requirements

The graphics features in this table require certain versions of hardware-accelerated OpenGL.

Feature	Required OpenGL Version	With Required OpenGL Version	Without Required OpenGL Version
Sharp corners when using wide lines	OpenGL 2.1 or later		
Correct rendering of transparent objects in 3-D views	OpenGL 2.1 or later		
Graphics smoothing to reduce the appearance of jagged lines	OpenGL 3.0 or later		

Alternatively, you can use software OpenGL or Painters to render graphics. However, these rendering methods have some limitations:

- Software OpenGL is slower than hardware-accelerated OpenGL in some cases and does not support some graphics features.
- Painters works well for 2-D graphics. However, it might not correctly draw intersecting polygons.

For more information about renderers, see the `Renderer` property of the figure.

Workarounds for Older Graphics Hardware

Older graphics hardware or hardware with limited graphics memory can cause poor performance. It is possible to improve performance with these changes:

- Use smaller figure windows.
- Set the figure `GraphicsSmoothing` property to 'off'.
- Set the axes `SortMethod` property to 'childorder'.
- Do not use transparency or, if you must use transparency, set the axes `SortMethod` property to 'childorder'.

For the best results with graphics, upgrade your graphics drivers to the latest version available. For more information, see “System Requirements for Graphics” on page 17-23.

set and get

Access Property Values

In this section...

- “Object Properties and Dot Notation” on page 18-2
- “Graphics Object Variables Are Handles” on page 18-4
- “Listing Object Properties” on page 18-6
- “Modify Properties with set and get” on page 18-6
- “Multi Object/Property Operations” on page 18-7

Object Properties and Dot Notation

Graphing functions return the object or objects created by the function. For example:

```
h = plot(1:10);
```

`h` refers to the line drawn in the graph of the values 1 through 10.

Dot notation is a syntax for accessing object properties. This syntax uses the object variable and the case-sensitive property name connected with a dot (.) to form an object dot property name notation:

object.PropertyName

If the object variable is nonscalar, use indexing to refer to a single object:

object(n).PropertyName

Scalar Object Variable

If `h` is the line created by the `plot` function, the expression `h.Color` is the value of this particular line's `Color` property:

```
h.Color
```

```
ans =
```

```
0      0.4470    0.7410
```

If you assign the color value to a variable:

```
c = h.Color;
```

The variable `c` is a double.

```
whos
```

Name	Size	Bytes	Class
c	1x3	24	double
h	1x1	112	matlab.graphics.chart.primitive.Line

You can change the value of this line's `Color` property with an assignment statement:

```
h.Color = [0 0 1];
```

Use dot notation property references in expressions:

```
meanY = mean(h.YData);
```

Or to change the property value:

```
h.LineWidth = h.LineWidth + 0.5;
```

Reference other objects contained in properties with multiple dot references:

```
h.Annotation.LegendInformation.IconDisplayStyle
```

```
ans =
```

```
on
```

Set the properties of objects contained in properties:

```
ax = gca;
ax.Title.FontWeight = 'normal';
```

Nonscalar Object Variable

Graphics functions can return an array of objects. For example:

```
y = rand(5);
h = plot(y);
size(h)
```

```
ans =
```

```
5      1
```

Access the line representing the first column in `y` using the array index:

```
h(1).LineStyle = '---';
```

Use the `set` function to `set` the `LineStyle` of all the lines in the array:

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

Appending Data to Property Values

With dot notation, you can use “end” indexing to append data to properties that contain data arrays, such as line `XData` and `YData`. For example, this code updates the line `XData` and `YData` together to grow the line. You must ensure the size of line’s x- and y-data are the same before rendering with the call to `drawnow` or returning to the MATLAB prompt.

```
h = plot(1:10);
for k = 1:5
    h.XData(end + 1) = h.XData(end) + k;
    h.YData(end + 1) = h.YData(end) + k;
    drawnow
end
```

Graphics Object Variables Are Handles

The object variables returned by graphics functions are *handles*. Handles are references to the actual objects. Object variables that are handles behave in specific ways when copied and when the object is deleted.

Copy Object Variable

For example, create a graph with one line:

```
h = plot(1:10);
```

Now copy the object variable to another variable and set a property value with the new object variable:

```
h2 = h;
h2.Color = [1,0,0]
```

Assigning the object variable `h` to `h2` creates a copy of the handle, but not the object referred to by the variable. The value of the `Color` property accessed from variable `h` is the same as that accessed from variable `h2`.

```
h.Color  
ans =  
1 0 0
```

`h` and `h2` refer to the same object. Copying a handle object variable does not copy the object.

Delete Object Variables

There are now two object variables in the workspace that refer to the same line.

```
whos
```

Name	Size	Bytes	Class
<code>h</code>	1x1	112	<code>matlab.graphics.chart.primitive.Line</code>
<code>h2</code>	1x1	112	<code>matlab.graphics.chart.primitive.Line</code>

Now close the figure containing the line graph:

```
close gcf
```

The line object no longer exists, but the object variables that referred to the line do still exist:

```
whos
```

Name	Size	Bytes	Class
<code>h</code>	1x1	112	<code>matlab.graphics.chart.primitive.Line</code>
<code>h2</code>	1x1	112	<code>matlab.graphics.chart.primitive.Line</code>

However, the object variables are no longer valid:

```
h.Color
```

Invalid or deleted object.

```
h2.Color = 'blue'
```

Invalid or deleted object.

To remove the invalid object variables, use `clear`:

```
clear h h2
```

Listing Object Properties

To see what properties an object contains, use the `get` function:

```
get(h)
```

MATLAB returns a list of the object properties and their current value:

```
AlignVertexCenters: 'off'
    Annotation: [1x1 matlab.graphics.eventdata.Annotation]
BeingDeleted: 'off'
    BusyAction: 'queue'
ButtonDownFcn: ''
    Children: []
    Clipping: 'on'
    Color: [0 0.4470 0.7410]

...
LineStyle: '-'
LineWidth: 0.5000
Marker: 'none'

...
```

You can see the values for properties with an enumerated set of possible values using the `set` function:

```
set(h, 'LineStyle')

'-'
'--'
':'
'-.''
'none'
```

To display all settable properties including possible values for properties with an enumerated set of values, use `set` with the object variable:

```
set(h)
```

Modify Properties with `set` and `get`

You can also access and modify properties using the `set` and `get` functions.

The basic syntax for setting the value of a property on an existing object is:

```
set(object,'PropertyName',NewPropertyValue)
```

To query the current value of a specific object property, use a statement of the form:

```
returned_value = get(object,'PropertyName');
```

Property names are always character strings. You can use quoted strings or a variable that is a character string. Property values depend on the particular property.

Multi Object/Property Operations

If the object argument is an array, MATLAB sets the specified value on all identified objects. For example:

```
y = rand(5);  
h = plot(y);
```

Set all the lines to red:

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

To set the same properties on a number of objects, specify property names and property values using a structure or cell array. For example, define a structure to set axes properties appropriately to display a particular graph:

```
view1.CameraViewAngleMode = 'manual';  
view1.DataAspectRatio = [1 1 1];  
view1.Projection = 'Perspective';
```

To set these values on the current axes, type:

```
set(gca,view1)
```

Query Multiple Properties

You can define a cell array of property names and 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:

```
camModes = {'CameraPositionMode','CameraTargetMode',...
'CameraUpVectorMode','CameraViewAngleMode'};
```

Use this cell array as an argument to obtain the current values of these properties:

```
get(gca, camModes)
```

```
ans =  
'auto' 'auto' 'auto' 'auto'
```

Using Axes Properties

- “Axes Aspect Ratio” on page 19-2
- “Display Text Outside Axes” on page 19-6
- “Overlay Axes with Different Sizes” on page 19-9
- “Graph with Multiple x-Axes and y-Axes” on page 19-12
- “Automatically Calculated Properties” on page 19-16
- “Line Styles Used for Plotting — LineStyleOrder” on page 19-20

Axes Aspect Ratio

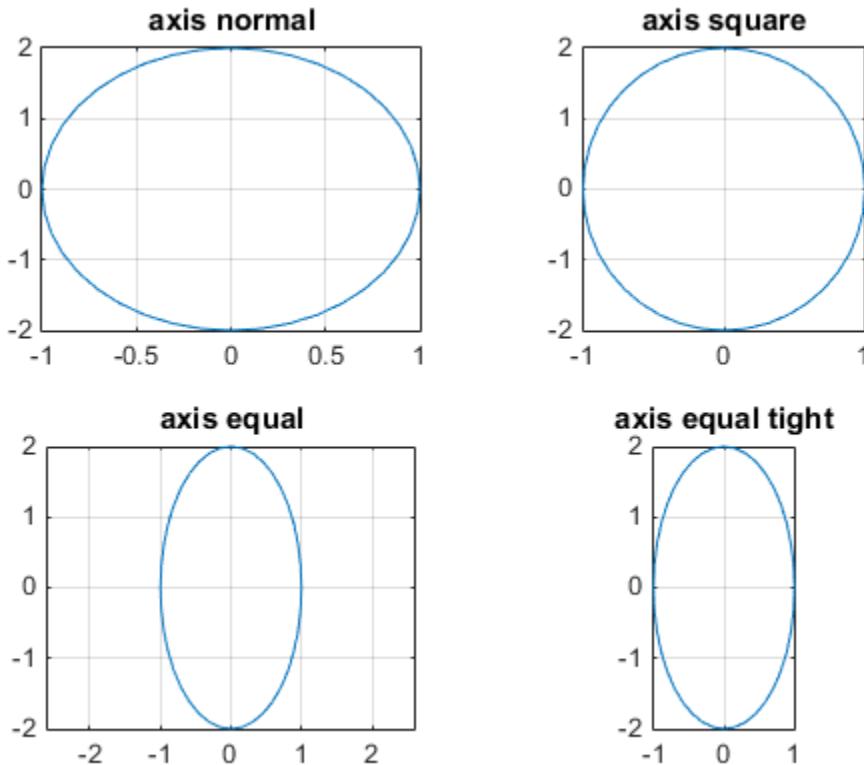
By default, 2-D graphs display in a rectangular axes that has the same aspect ratio as the figure window. This makes optimum use of space available for plotting. Set the aspect ratio with the `axis` function:

- `axis normal` — Sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window. This is the default behavior.
- `axis square` — makes the current axes region square
- `axis equal` — sets the aspect ratio so that the data units are the same in every direction
- `axis equal tight` — sets the aspect ratio so that the data units are the same in every direction and then sets the axis limits to the minimum and maximum values of the data.

For example, these statements create a elongated circle.

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

These graphs show the effects of various axis command options:

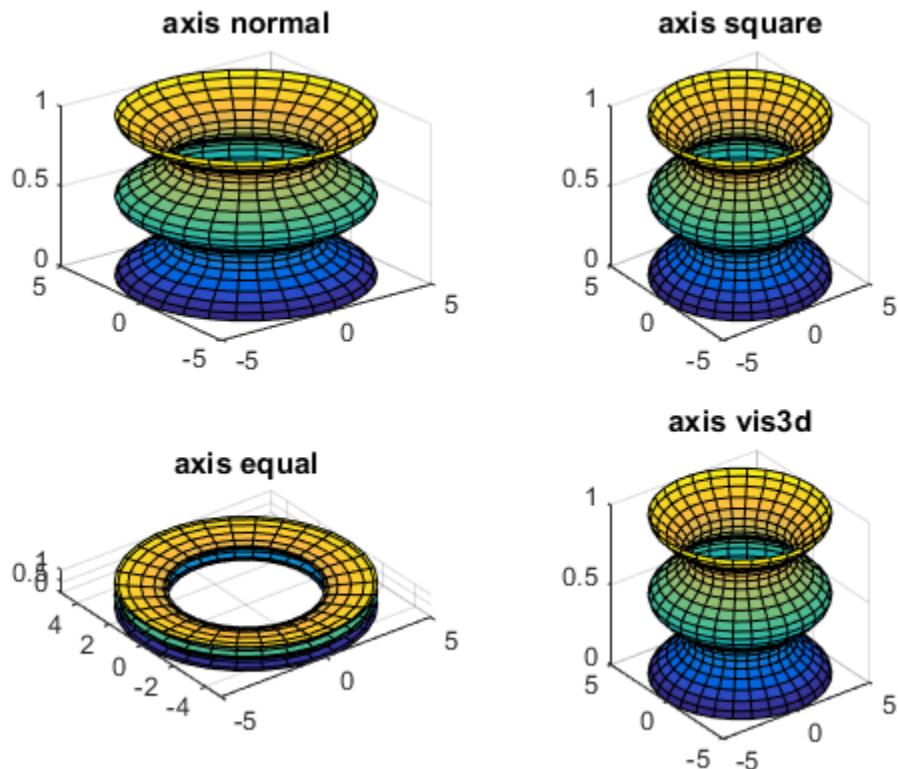


3-D Views

These statements create a cylindrical surface:

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

These graphs show the effects of various axis command options:



- **axis normal** — Sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window. This is the default behavior.
- **axis square** — Creates an axes that is square regardless of the shape of the figure window. The cylindrical surface is no longer distorted because it is not warped to fit the window. However, the size of one data unit is not equal along all axes (the z-axis spans only one unit while the x-axes and y-axes span 10 units each).
- **axis equal** — Makes the length of one data unit equal along each axis while maintaining a nearly square plot box. It also prevents warping of the axis to fill the window's shape.
- **axis vis3d** — Freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretching the axes to fill the figure. Use this option to keep settings from changing while you rotate the scene.

Note: To format aspect ratio using the `axis` function, call `axis` after creating the graph or use the `hold on` command before plotting data.

Additional Commands for Setting Aspect Ratio

You can also control the aspect ratio of your graph more precisely using these functions:

- Specifying the relative scales of the x -, y -, and z -axes (data aspect ratio)
- Specifying the shape of the space defined by the axes (plot box aspect ratio)
- Specifying the axis limits

The following commands enable you to set these values.

Command	Purpose
<code>daspect</code>	Set or query the data aspect ratio
<code>pbaspect</code>	Set or query the plot box aspect ratio
<code>xlim</code>	Set or query x -axis limits
<code>ylim</code>	Set or query y -axis limits
<code>zlim</code>	Set or query z -axis limits

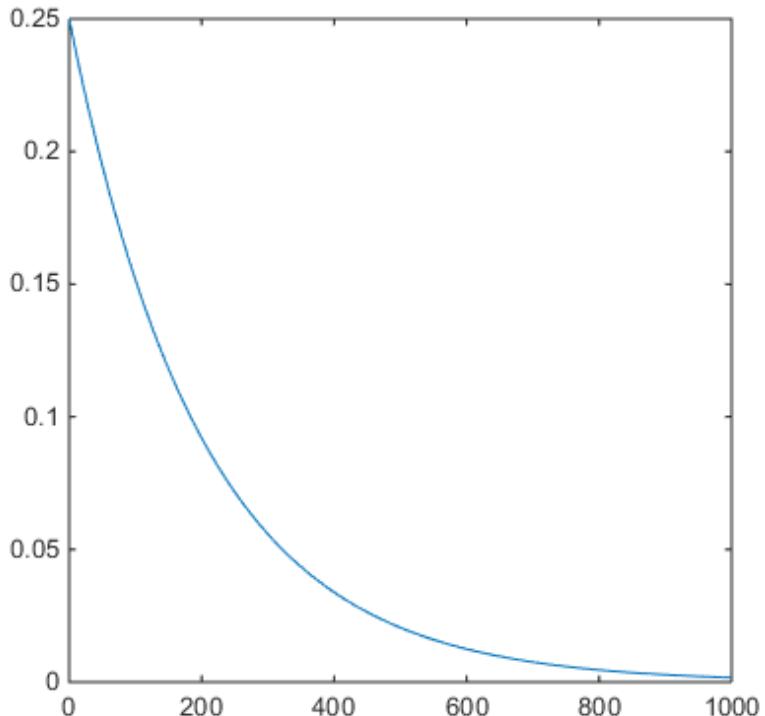
Display Text Outside Axes

This example shows how to display text outside an axes by creating a second axes for the text. MATLAB® always displays text objects within an axes. If you want to place a text description alongside an axes, then you must create another axes to position the text.

Create an invisible axes, `ax1`, that encompasses the entire figure window by specifying its position as `[0,0,1,1]`. Then, create a smaller axes, `ax2`, to contain the actual plot. Create a line plot in the smaller axes by passing its axes handle, `ax2`, to the `plot` function.

```
fig = figure;
ax1 = axes('Position',[0 0 1 1], 'Visible','off');
ax2 = axes('Position',[.3 .1 .6 .8]);

t = 0:1000;
y = 0.25*exp(-0.005*t);
plot(ax2,t,y)
```

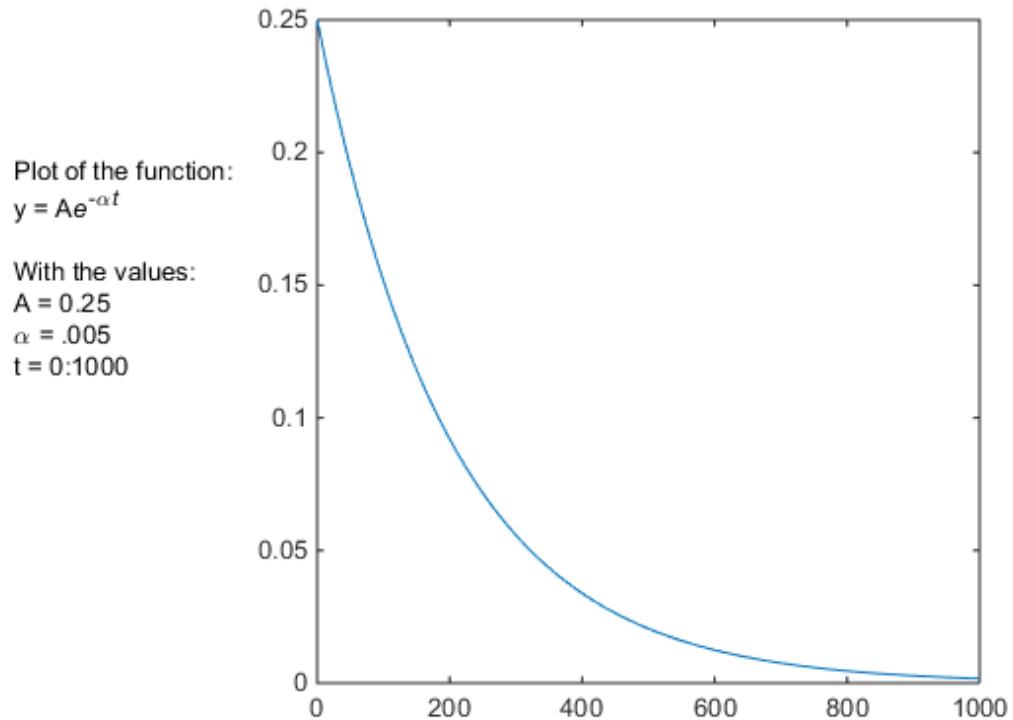


Define the string for the text description. Use a cell array to define multiline text.

```
descr = {'Plot of the function:';  
        'y = A{\ite}^{-\alpha{\itt}}';  
        ' ';  
        'With the values:';  
        'A = 0.25';  
        '\alpha = .005';  
        't = 0:1000'};
```

Set the larger axes to be the current axes since the `text` function places text in the current axes. Then, display the text.

```
axes(ax1) % sets ax1 to current axes  
text(.025,0.6,descr)
```



The figure contains text next to the line plot.

Overlay Axes with Different Sizes

This example shows how to display the same set of data using different size axes.

Create a figure with five axes of different sizes by setting their **Position** properties. In each axes plot the **sphere** function.

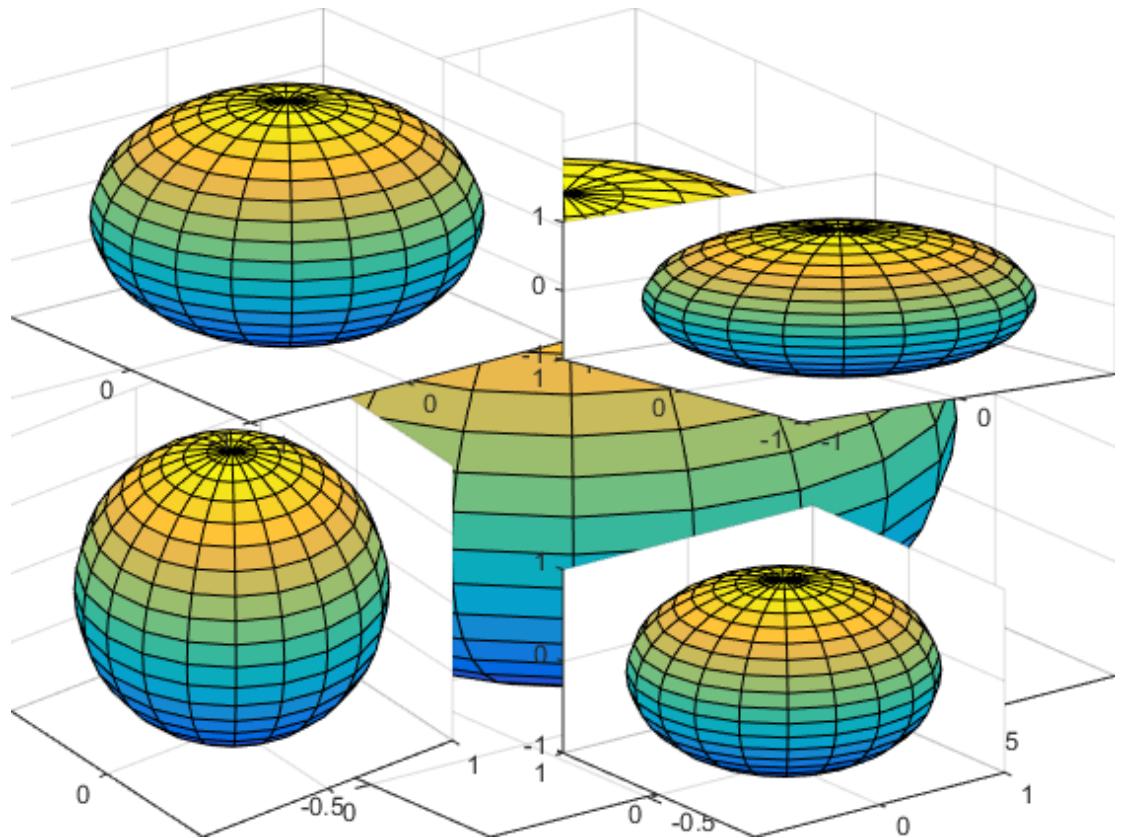
```
figure
ax(1) = axes('Position',[0 0 1 1]);
sphere

ax(2) = axes('Position',[0 0 .4 .6]);
sphere

ax(3) = axes('Position',[0 .5 .5 .5]);
sphere

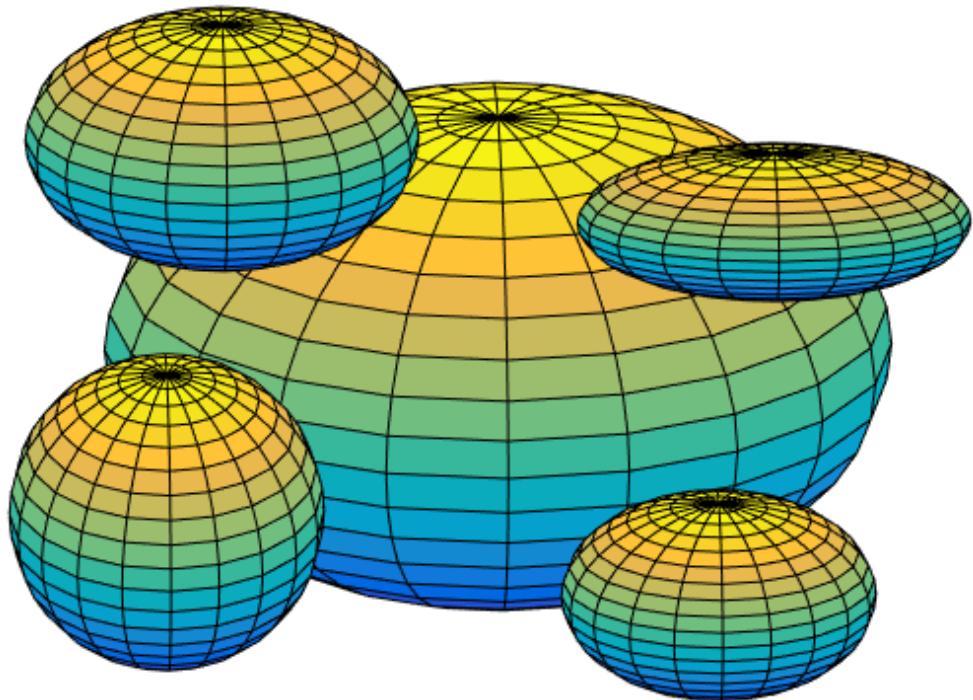
ax(4) = axes('Position',[.5 0 .4 .4]);
sphere

ax(5) = axes('Position',[.5 .5 .5 .3]);
sphere
```



Use the axes handles stored in array `ax` to turn off the display of the axes boxes so that only the spheres are visible.

```
set(ax, 'Visible', 'off')
```



Using five axes of different sizes gives the effect that the spheres appear different shapes and sizes, even though each sphere is defined by the same data.

Graph with Multiple x-Axes and y-Axes

This example shows how to create a graph using the bottom and left sides of the axes for the first plot, and the top and right sides of the axes for the second plot.

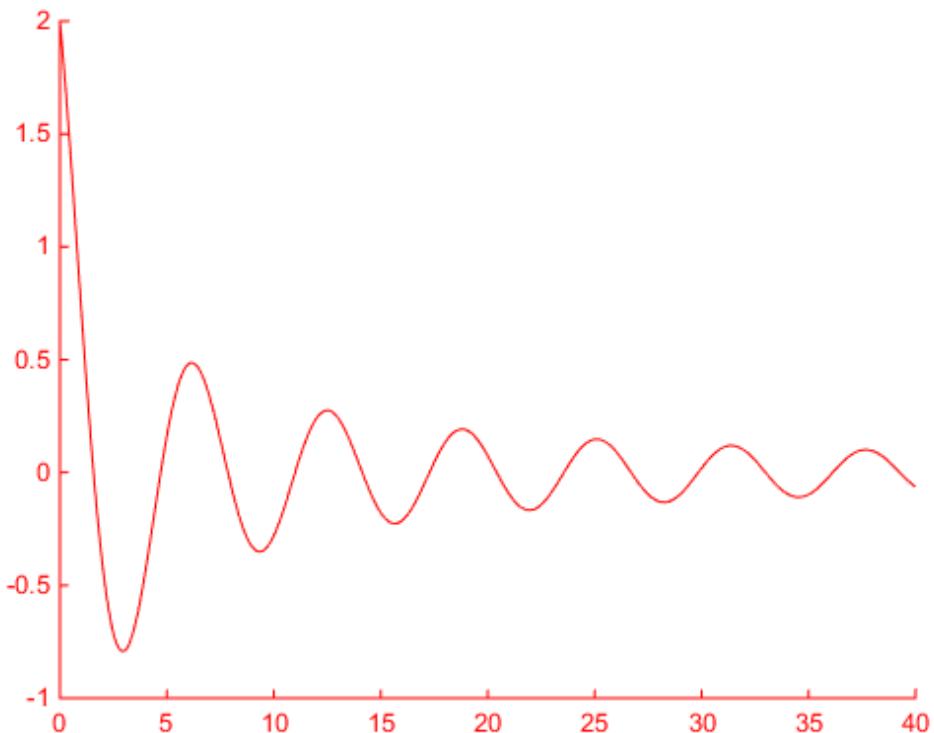
Define the data to plot.

```
x1 = 0:0.1:40;
y1 = 4.*cos(x1)./(x1+2);
x2 = 1:0.2:20;
y2 = x2.^2./x2.^3;
```

Use the `line` function to plot `y1` versus `x1` using a red line. Set the color for the *x*-axis and *y*-axis to red.

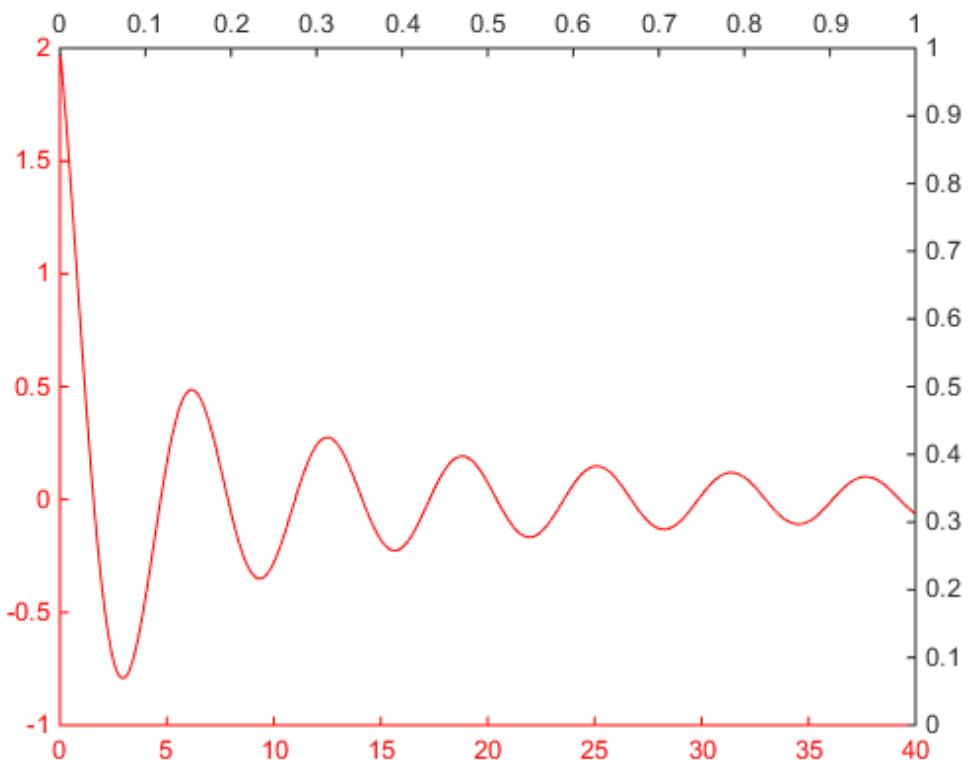
```
figure
line(x1,y1,'Color','r')

ax1 = gca; % current axes
ax1.XColor = 'r';
ax1.YColor = 'r';
```



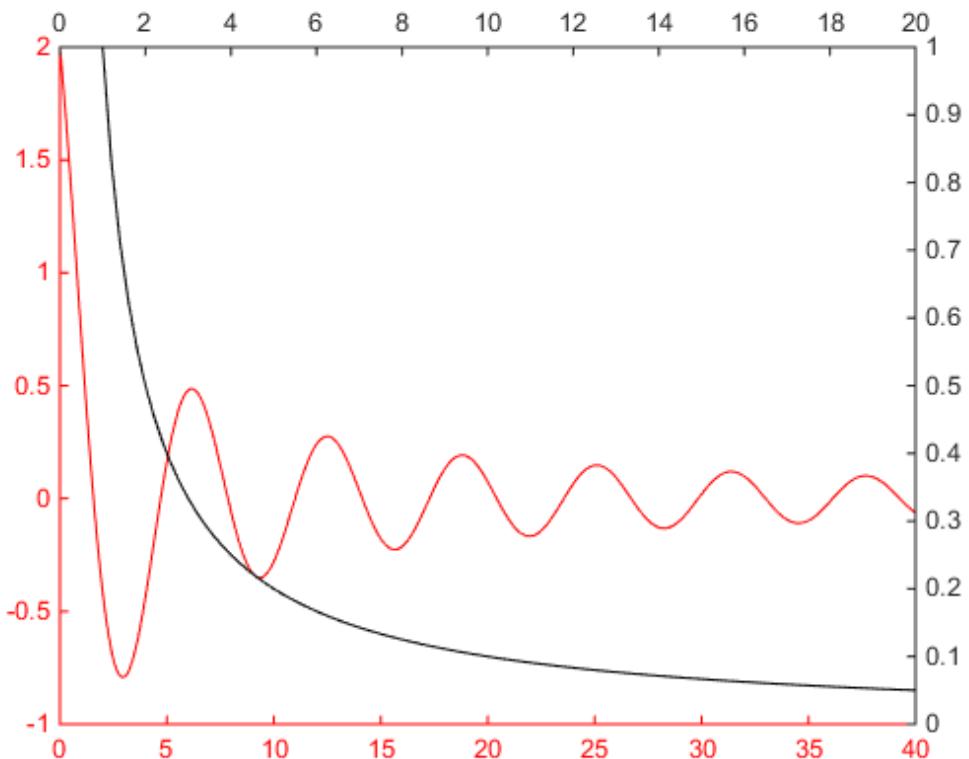
Create a second axes in the same location as the first axes by setting the position of the second axes equal to the position of the first axes. Specify the location of the x -axis as the top of the graph and the y -axis as the right side of the graph. Set the axes `Color` to '`none`' so that the first axes is visible underneath the second axes.

```
ax1_pos = ax1.Position; % position of first axes  
ax2 = axes('Position',ax1_pos,...  
    'XAxisLocation','top',...  
    'YAxisLocation','right',...  
    'Color','none');
```



Use the `line` function to plot y_2 versus x_2 on the second axes. Set the line color to black so that it matches the color of the corresponding x -axis and y -axis.

```
line(x2,y2,'Parent',ax2,'Color','k')
```



The graph contains two lines that correspond to different axes. The red line corresponds to the red axes. The black line corresponds to the black axes.

See Also

[axes](#) | [gca](#) | [line](#)

Related Examples

- “Create Graph with Two y-Axes”

Automatically Calculated Properties

When plotting functions create graphs, many of the axes properties that are under automatic control adjust to best display the graph. These properties adjust automatically when their associated mode property is set to `auto` (which is the default). The following table lists the axes automatic-mode properties.

Note: When setting any mode property to '`manual`' from within a function, you should call `drawnow` first to ensure the corresponding property has been updated to the latest value.

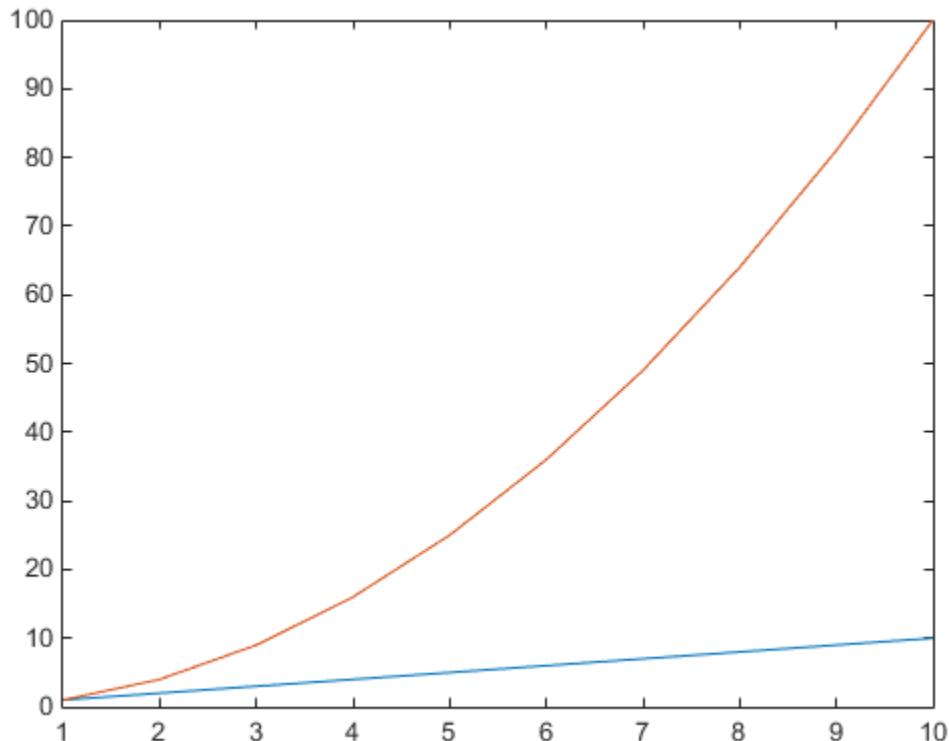
Mode Property	What It Controls
<code>CameraPositionMode</code>	Positioning of the viewpoint
<code>CameraTargetMode</code>	Positioning of the camera target in the axes
<code>CameraUpVectorMode</code>	The direction of “up” in 2-D and 3-D views
<code>CameraViewAngleMode</code>	The size of the projected scene and stretch-to-fit behavior
<code>CLimMode</code>	Mapping of data values to colors
<code>DataAspectRatioMode</code>	Relative scaling of data units along x-, y-, and z-axes and stretch-to-fit behavior
<code>PlotBoxAspectRatioMode</code>	Relative scaling of plot box along x-, y-, and z-axes and stretch-to-fit behavior
<code>TickDirMode</code>	Direction of axis tick marks (in for 2-D, out for 3-D)
<code>XLimMode</code>	Limits of the respective x, y, and z axes
<code>YLimMode</code>	
<code>ZLimMode</code>	
<code>XTickMode</code>	Tick mark spacing along the respective x-, y-, and z-axes
<code>YTickMode</code>	
<code>ZTickMode</code>	
<code>XTickLabelMode</code>	Tick mark labels along the respective x-, y-, and z-axes

Mode Property	What It Controls
ZTickLabelMode	
YTickLabelMode	

For example, these statements graph two lines:

```
x = 1:10;  
y = 1:10;  
plot(x,y)  
hold on  
plot(x,y.^2)
```

The second `plot` statement causes the axes `YLim` property to change from `[0,10]` to `[0,100]`.



This is because `YLimMode` is `auto`, which means the axes recompute the axis limits whenever necessary.

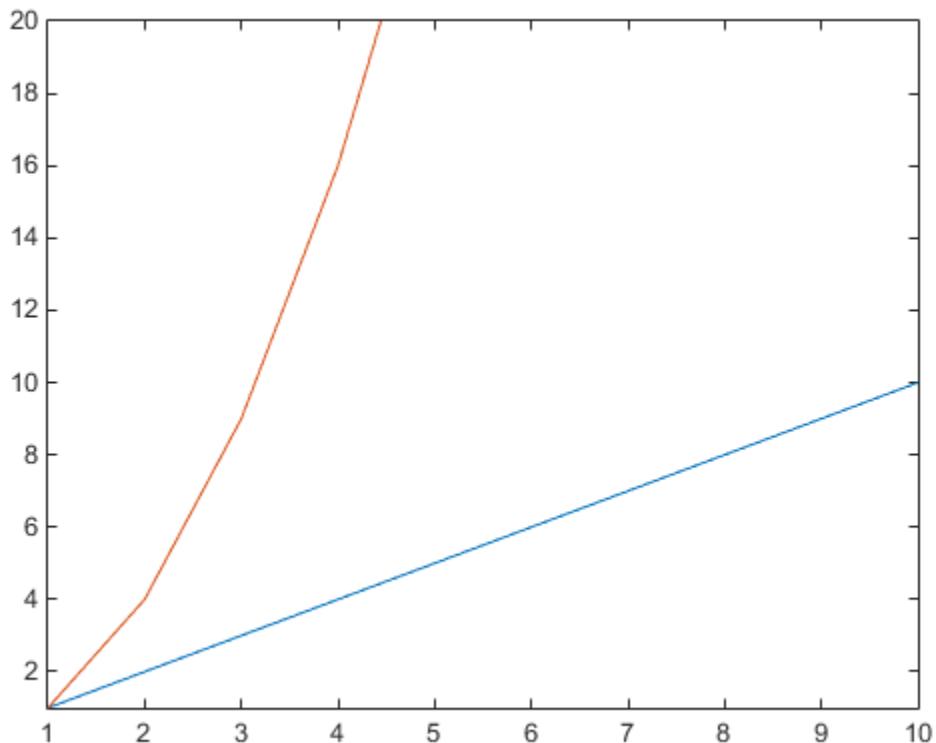
If you assign a value to a property controlled by an automatic-mode property, MATLAB sets the mode property to `manual`. When the mode property is `manual`, the axes does not automatically recompute the property value.

For example,

```
x = 1:10;
y = 1:10;
plot(x,y)
hold on
ax = gca;
```

```
ax.XLim = [1,10];
ax.YLim = [1,20];
plot(x,y.^2)
```

Setting values for the XLim and YLim properties changes the XLimMode and YLimMode properties to `manual`. The second `plot` statement draws a line that is clipped to the axis limits instead of causing the axes to recompute its limits.



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 line-plotting functions.

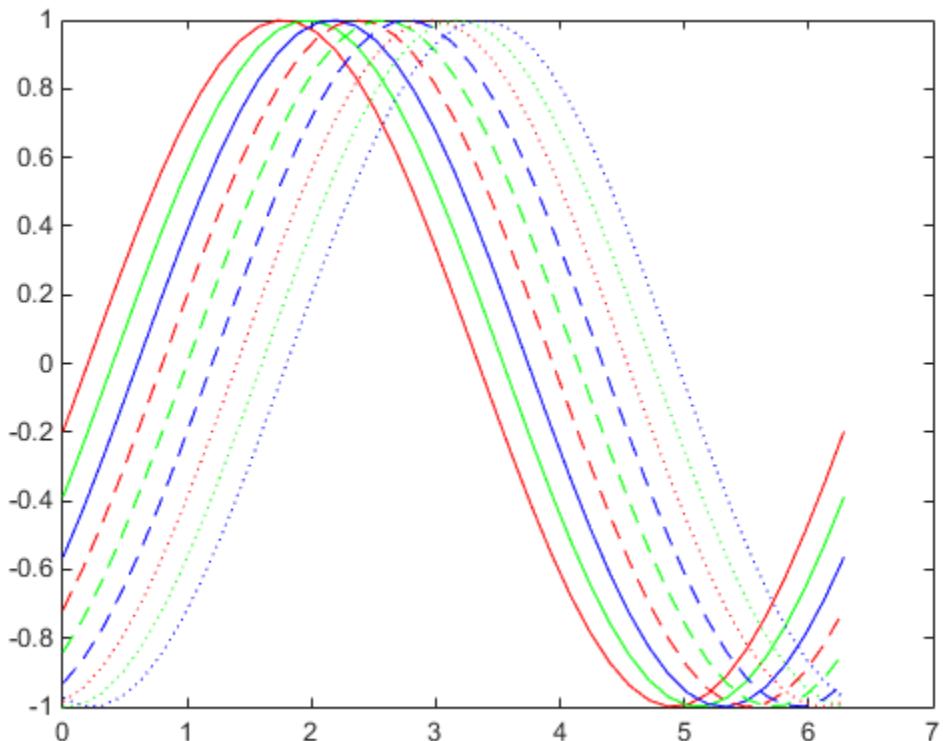
Axes increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

For example, define a default `ColorOrder` of red, green, and blue and a default `LineStyleOrder` of solid, dashed, and dotted lines.

```
set(groot, '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)
```



Plotting functions cycle through all colors for each line style.

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

```
set(groot,'defaultAxesLineStyleOrder','remove')
set(groot,'defaultAxesColorOrder','remove')
```

See “Default Property Values” for more information.

