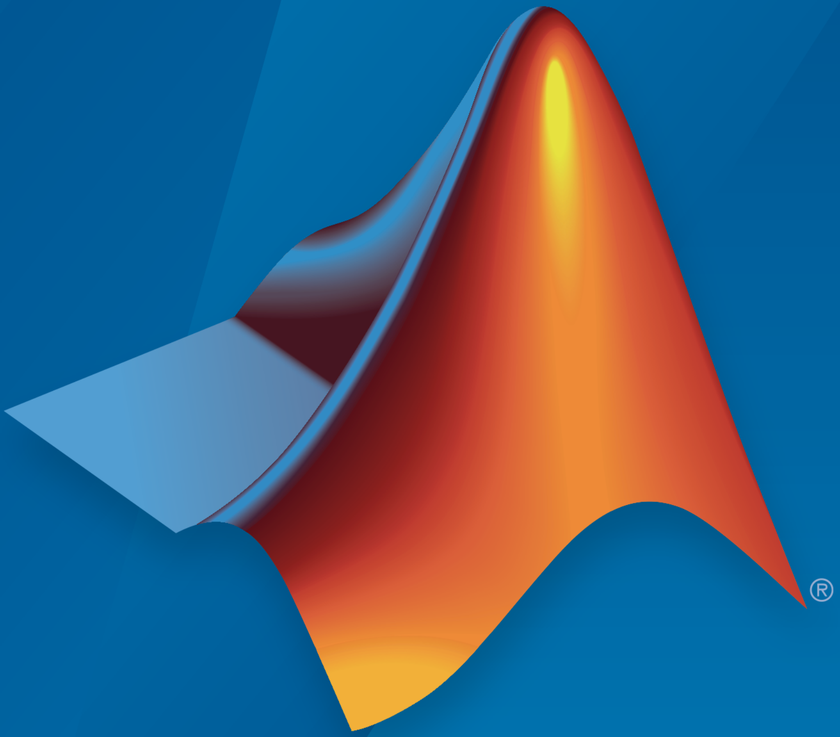
MATLAB®

Graphics



MATLAB®

R2017a

MathWorks®

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

Trademarks

Patents

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)
March 2015	Online only	Revised for Version 8.5 (Release 2015a)
September 2015	Online only	Revised for Version 8.6 (Release 2015b)
March 2016	Online only	Revised for Version 9.0 (Release 2016a)
September 2016	Online only	Revised for Version 9.1 (Release 2016b)
March 2017	Online only	Revised for Version 9.2 (Release 2017a)

Contents

1 Plots and Plotting Tools

Types of MATLAB Plots **1-2**

Create Graph Using Plots Tab **1-4**

Customize Graph Using Plot Tools **1-6**
Open Plot Tools **1-6**
Customize Objects in Graph **1-7**
Control Visibility of Objects in Graph **1-8**
Add Annotations to Graph **1-8**
Close Plot Tools **1-8**

Create Subplots Using Plot Tools **1-9**
Create Simple Line Plot and Open Plot Tools **1-9**
Create Upper and Lower Subplots **1-9**
Add Data to Lower Subplot **1-10**
Add New Plot Without Overwriting Existing Plot **1-11**

2 Basic Plotting Commands

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 **2-7**
Specify Line Style and Color **2-8**
Specify Line Style, Color, and Markers **2-9**

Plot Only Data Points 2-10

Add Title, Axis Labels, and Legend to Graph 2-13

Specify Axis Limits 2-19
Change Axis Limits 2-19
Use Semiautomatic Axis Limits 2-20
Revert Back to Default Axis Limits 2-21
Reverse Axis Direction 2-23
Display Axis Lines through Origin 2-24

Specify Axis Tick Values and Labels 2-28
Change Tick Value Locations and Labels 2-28
Rotate Tick Labels 2-30
Change Tick Label Formatting 2-32
Control Value in Exponent Label 2-33

Add Grid Lines and Edit Placement 2-38
Display Grid Lines 2-38
Display Grid Lines in Specific Direction 2-41
Edit Grid Line Placement 2-43
Modify Visual Appearance of Grid Lines 2-45

Combine Multiple Plots 2-48
Combine Plots in Same Axes 2-48
Create Multiple Axes in Figure Using Subplots 2-49
Add Super Title to Figure with Subplots 2-51

Create Chart with Two *y*-Axes 2-55

Modify Properties of Charts with Two *y*-Axes 2-64
Change Axes Properties 2-64
Change Ruler Properties 2-66
Specify Colors Using Default Color Order 2-68

Plot Imaginary and Complex Data 2-71

Create Heatmap from Tabular Data 2-74

3 Data Exploration Tools

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**
Zooming in 3-D Views **3-16**

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**

4 Annotating Graphs

Exclude Objects from Graph Legend **4-2**

Specify Legend Descriptions During Line Creation **4-5**

Add Text to Specific Points on Graph **4-8**
Add Text to Three Data Points on Graph **4-8**
Determine Minimum and Maximum Points and Add Text . . **4-11**

Include Variable Values in Graph Text **4-15**
Include Variable Value in Axis Label **4-15**
Include Loop Variable Value in Graph Title **4-16**

Text with Mathematical Expression Using LaTeX **4-18**
Add Text with Integral Expression to Graph **4-18**
Add Text with Summation Symbol to Graph **4-20**

Greek Letters and Special Characters in Graph Text **4-23**
Include Greek Letters in Graph Text **4-23**
Include Superscripts and Annotations in Graph Text **4-24**
TeX Markup Options **4-26**

Add Annotations to Graph Interactively **4-30**
Add Annotations **4-30**
Pin Annotations to Points in Graph **4-31**

Add Text to Graph Interactively **4-33**
Add Title and Axis Labels **4-33**
Add Legend **4-35**
Add Annotations to Graph **4-37**

Align Objects in Graph Using Alignment Tools **4-39**

5 Coloring Graphs

Creating Colorbars **5-2**

Change Color Scheme Using a Colormap **5-10**

How Surface Plot Data Relates to a Colormap **5-16**
Relationship Between the Surface and the Colormap **5-16**
Change the Direction or Pattern of Colors **5-17**

How Image Data Relates to a Colormap **5-22**

How Patch Data Relates to a Colormap **5-30**
Relationship of the Colormap to *x*-, *y*-, and *z*-Coordinate Arrays **5-30**

Relationship of the Colormap to Face-Vertex Data 5-34

Control Colormap Limits **5-39**

Differences Between Colormaps and Truecolor **5-45**
Differences in Workflow 5-45
Differences in Visual Presentation 5-47

6 Creating Specialized Plots

Types of Bar Graphs **6-3**

Modify Baseline of Bar Graph **6-10**

Overlay Bar Graphs **6-13**

Combine Line and Bar Charts Using Two y-Axes **6-17**

Color 3-D Bars by Height **6-21**

Compare Data Sets Using Overlayed Area Graphs **6-24**

Offset Pie Slice with Greatest Contribution **6-29**

Add Legend to Pie Chart **6-31**

Label Pie Chart With Text and Percentages **6-34**
Create Pie Chart 6-34
Store Precalculated Percent Values 6-35
Combine Percent Values and Additional Text 6-36
Determine Horizontal Distance to Move Each Label 6-38
Position New Label 6-39

Data Cursors with Histograms **6-42**

Color Analysis with Bivariate Histogram **6-44**

Control Categorical Histogram Display **6-50**

Replace Discouraged Instances of hist and histc **6-58**
Old Histogram Functions (`hist`, `histc`) **6-58**
Recommended Histogram Functions **6-58**
Differences Requiring Code Updates **6-59**

Combine Line and Stem Plots **6-67**

Overlay Stairstep Plot and Line Plot **6-70**

Combine Contour Plot and Quiver Plot **6-73**

Projectile Path Over Time **6-75**

Label Contour Plot Levels **6-77**

Change Fill Colors for Contour Plot **6-79**

Highlight Specific Contour Levels **6-81**

Animation Techniques **6-84**
Updating the Screen **6-84**
Optimizing Performance **6-84**

Trace Marker Along Line **6-86**

Move Group of Objects Along Line **6-89**

Animate Graphics Object **6-93**

Line Animations **6-97**

Record Animation for Playback **6-100**
Record and Play Back Movie **6-100**
Capture Entire Figure for Movie **6-101**

Customize Polar Axes Grid Lines and Appearance **6-104**

Compass Labels on Polar Axes **6-113**

Create Line Plot with Markers **6-115**
Add Markers to Line Plot **6-115**
Specify Marker Size and Color **6-117**
Control Placement of Markers Along Line **6-119**

Display Markers at Maximum and Minimum Data Points . 6-120
Revert to Default Marker Locations 6-121
Supported Marker Symbols 6-123

7 Displaying Bit-Mapped Images

Working with Images in MATLAB Graphics 7-2
What Is Image Data? 7-2
Supported Image Formats 7-3
Functions for Reading, Writing, and Displaying Images 7-4

Image Types 7-5
Indexed Images 7-5
Intensity Images 7-7
RGB (Truecolor) Images 7-8

8-Bit and 16-Bit Images 7-10
Indexed Images 7-10
Intensity Images 7-11
RGB Images 7-11
Mathematical Operations Support for uint8 and uint16 7-12
Other 8-Bit and 16-Bit Array Support 7-12
Converting an 8-Bit RGB Image to Grayscale 7-13
Summary of Image Types and Numeric Classes 7-17

Read, Write, and Query Image Files 7-18
Working with Image Formats 7-18
Reading a Graphics Image 7-19
Writing a Graphics Image 7-19
Subsetting a Graphics Image (Cropping) 7-20
Obtaining Information About Graphics Files 7-21

Displaying Graphics Images 7-22
Image Types and Display Methods 7-22
Controlling Aspect Ratio and Display Size 7-24

The Image Object and Its Properties 7-27
Image CData 7-27
Image CDataMapping 7-27

XData and YData 7-28
Add Text to Image Data 7-30
Additional Techniques for Fast Image Updating 7-32

Printing Images 7-34

Convert Image Graphic or Data Type 7-35

8 Printing and Saving

Print Figure from File Menu 8-2
Simple Printout 8-2
Preserve Background Color and Tick Values 8-2
Figure Size and Placement 8-3
Line Width and Font Size 8-5

Copy Figure to Clipboard from Edit Menu 8-7
Copy Figure to Clipboard 8-7
Specify Format, Background Color, and Size Options 8-9

Customize Figure Interactively Before Saving 8-11
Set Figure Size 8-11
Set Figure Background Color 8-13
Set Figure Font Size and Line Width 8-14
Save Figure to File 8-16
Save Figure Settings for Future Use 8-17
Apply Settings to Another Figure 8-17
Restore Figure to Original Settings 8-18

Save Figure to Open in Another Application 8-19
Choose File Format 8-19
Save Figure for Document or Presentation 8-19
Save Figure for Editing in Another Application 8-20
Customize Figure Before Saving 8-20
Include Figure in Microsoft Application or LaTeX Document 8-21
File Format Options 8-21

Save Figure Preserving Background Color 8-24
Retain Current Background Color 8-24

Change Background Color 8-25

Save Figure at Specific Size and Resolution 8-27
Use Screen Size and Resolution 8-27
Expand Figure to Fill Page 8-28
Use Specific Dimensions 8-29
Preserve Axis Limits and Tick Values 8-31

Save Figure to Reopen in MATLAB Later 8-33
Save Figure to FIG-File 8-33
Generate Code to Recreate Figure 8-35

Save Axes Without Saving UIControls 8-36
Create Figure with UIControls 8-36
Save Axes Without Saving UIControls 8-37
Copy Axes to New Figure and Save 8-38

Save Figure with Minimal White Space 8-40
Create Plot to Save 8-40
Expand Axes to Fill Figure 8-41
Specify Figure Size and Page Size 8-43
Save Figure to File Format 8-43

9 Axes Active Position

Control Axes Layout 9-2
Axes Position-Related Properties 9-2
Position and Margin Boundaries 9-2
Controlling Automatic Resize Behavior 9-4
Stretch-to-Fill Behavior 9-5

10 Controlling Graphics Output

Control Graph Display 10-2
What You Can Control 10-2

Targeting Specific Figures and Axes 10-2

Prepare Figures and Axes for Graphs **10-5**
Behavior of MATLAB Plotting Functions 10-5
How the NextPlot Properties Control Behavior 10-5
Control Behavior of User-Written Plotting Functions 10-7

Use newplot to Control Plotting **10-9**

Responding to Hold State **10-12**

Prevent Access to Figures and Axes **10-14**
Why Prevent Access 10-14
How to Prevent Access 10-14

11 Default Values

Default Property Values **11-2**
Predefined Values for Properties 11-2
Specify Default Values 11-2
Where in Hierarchy to Define Default 11-3
List Default Values 11-3
Set Properties to the Current Default 11-4
Remove Default Values 11-4
Set Properties to Factory-Defined Values 11-4
List Factory-Defined Property Values 11-4
Reserved Words 11-5

Default Values for Automatically Calculated Properties . . **11-6**
What Are Automatically Calculated Properties 11-6
Default Values for Automatically Calculated Properties . . . 11-6

How MATLAB Finds Default Values **11-8**

Factory-Defined Property Values **11-9**

Define Default Line Styles **11-10**

Multilevel Default Values **11-12**

Graphics Object Callbacks

Callbacks — Programmed Response to User Action **12-2**
What Are Callbacks? **12-2**
Window Callbacks **12-2**

Callback Definition **12-4**
Ways to Specify Callbacks **12-4**
Callback Function Syntax **12-4**
Related Information **12-5**
Define a Callback as a Default **12-6**

Button Down Callback Function **12-7**
When to Use a Button Down Callback **12-7**
How to Define a Button Down Callback **12-7**

Define a Context Menu **12-9**
When to Use a Context Menu **12-9**
How to Define a Context Menu **12-9**

Define an Object Creation Callback **12-11**
Related Information **12-12**

Define an Object Deletion Callback **12-13**

Capturing Mouse Clicks **12-14**
Properties That Control Response to Mouse Clicks **12-14**
Combinations of PickablePart/HitTest Values **12-14**
Passing Mouse Click Up the Hierarchy **12-15**

Pass Mouse Click to Group Parent **12-18**
Objective and Design **12-18**
Object Hierarchy and Key Properties **12-18**
MATLAB Code **12-19**

Pass Mouse Click to Obscured Object **12-21**

13 Graphics Objects

Graphics Objects **13-2**
MATLAB Graphics Objects **13-2**
Graphs Are Composed of Specific Objects **13-2**
Organization of Graphics Objects **13-2**

Features Controlled by Graphics Objects **13-7**
Purpose of Graphics Objects **13-7**
Figures **13-7**
Axes **13-8**
Objects That Represent Data **13-9**
Group Objects **13-10**
Annotation Objects **13-11**

14 Group Objects

Object Groups **14-2**

Create Object Groups **14-3**
Parent Specification **14-4**
Visible and Selected Properties of Group Children **14-4**

Transforms Supported by hgtransform **14-5**
Transforming Objects **14-5**
Rotation **14-5**
Translation **14-6**
Scaling **14-6**
The Default Transform **14-7**
Disallowed Transforms: Perspective **14-7**
Disallowed Transforms: Shear **14-7**
Absolute vs. Relative Transforms **14-8**
Combining Transforms into One Matrix **14-8**
Undoing Transform Operations **14-9**

Rotate About an Arbitrary Axis **14-10**
Translate to Origin Before Rotating **14-10**

Rotate Surface 14-10

Nest Transforms for Complex Movements 14-14

15 Control Legend Content

Create Interactive Legends Using Callbacks 15-2
Toggle Chart Visibility 15-2
Toggle Chart Line Width 15-3

16 Working with Graphics Objects

Graphics Object Handles 16-2
What You Can Do with Handles 16-2
What You Cannot Do with Handles 16-3

Preallocate Arrays of Graphics Objects 16-4

Test for Valid Handle 16-5

Handles in Logical Expressions 16-6
If Handle Is Valid 16-6
If Result Is Empty 16-6
If Handles Are Equal 16-7

Graphics Arrays 16-9

17 Object Identification

Special Object Identifiers 17-2
Getting Handles to Special Objects 17-2

The Current Figure, Axes, and Object 17-2
Callback Object and Callback Figure 17-4

Find Objects **17-5**
Find Objects with Specific Property Values 17-5
Find Text by String Property 17-5
Use Regular Expressions with findobj 17-7
Limit Scope of Search 17-9

Copy Objects **17-11**
Copying Objects with copyobj 17-11
Copy Single Object to Multiple Destinations. 17-11
Copying Multiple Objects 17-12

Delete Graphics Objects **17-14**
How to Delete Graphics Objects 17-14
Handles to Deleted Objects 17-15

18 Optimize Performance of Graphics Programs

Finding Code Bottlenecks **18-2**

What Affects Code Execution Speed **18-4**
Potential Bottlenecks 18-4
How to Improve Performance 18-4

Judicious Object Creation **18-6**
Object Overhead 18-6
Do Not Create Unnecessary Objects 18-6
Use NaNs to Simulate Multiple Lines 18-7
Modify Data Instead of Creating New Objects 18-7

Avoid Repeated Searches for Objects **18-8**
Limit Scope of Search 18-8

Screen Updates **18-10**
MATLAB Graphics System 18-10
Managing Updates 18-11

Getting and Setting Properties **18-12**
Automatically Calculated Properties **18-12**
Inefficient Cycles of Sets and Gets **18-13**
Changing Text `Extent` to Rotate Labels **18-14**

Avoid Updating Static Data **18-15**
Segmenting Data to Reduce Update Times **18-15**

Transforming Objects Efficiently **18-17**

Use Low-Level Functions for Speed **18-18**

System Requirements for Graphics **18-19**
Minimum System Requirements **18-19**
Recommended System Requirements **18-19**
Upgrade Your Graphics Drivers **18-19**
Graphics Features with OpenGL Requirements **18-20**

Resolving Low-Level Graphics Issues **18-22**
Upgrade Your Graphics Hardware Drivers **18-22**
Choose OpenGL Implementation for Your System **18-23**
Fix Out-of-Memory Issues **18-24**
Contact Technical Support **18-24**

19 **set and get**

Access Property Values **19-2**
Object Properties and Dot Notation **19-2**
Graphics Object Variables Are Handles **19-4**
Listing Object Properties **19-6**
Modify Properties with set and get **19-6**
Multi Object/Property Operations **19-7**

20 Using Axes Properties

Control Ratio of Axis Lengths and Data Unit Lengths **20-2**
Plot Box Aspect Ratio **20-2**
Data Aspect Ratio **20-5**
Revert Back to Default Ratios **20-8**

Create Chart with Multiple *x*-Axes and *y*-Axes **20-10**

Display Text Outside Axes **20-14**

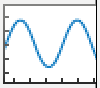
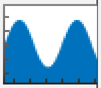
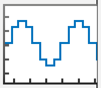
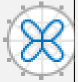
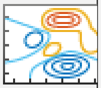
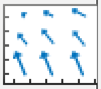
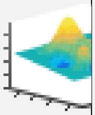
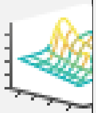
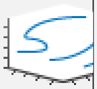
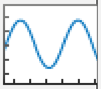
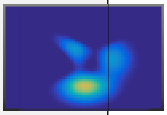
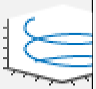
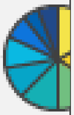
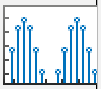
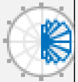
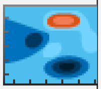
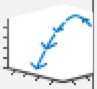
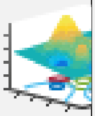
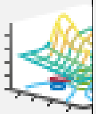
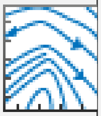
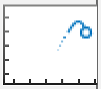
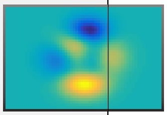
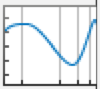
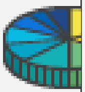
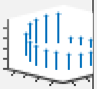
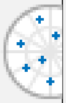
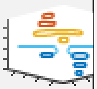
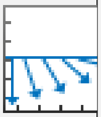
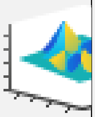
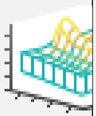
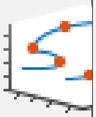
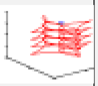
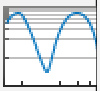
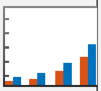
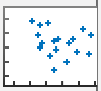
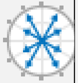
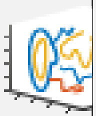
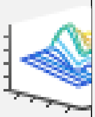
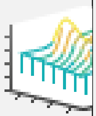
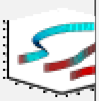
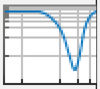
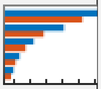
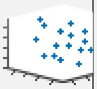
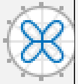
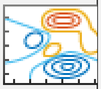
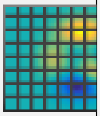
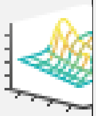
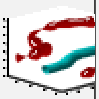
Line Styles Used for Plotting — LineStyleOrder **20-17**

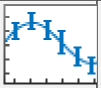
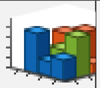
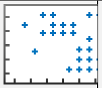
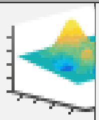
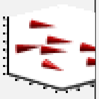
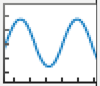
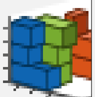
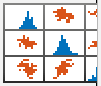
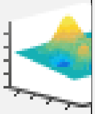
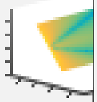
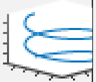
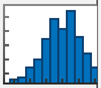
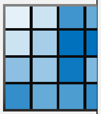
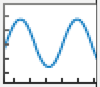
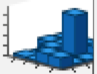
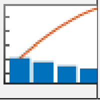
Plots and Plotting Tools

- “Types of MATLAB Plots” on page 1-2
- “Create Graph Using Plots Tab” on page 1-4
- “Customize Graph Using Plot Tools” on page 1-6
- “Create Subplots Using Plot Tools” on page 1-9

Types of MATLAB Plots

There are various functions that you can use to plot data in MATLAB®. This table classifies and illustrates the common graphics functions.

“Line Plots”	“Pie Charts, Bar Plots, and Histogra	“Discret Data Plots”	“Polar Plots”	“Contou Plots”	“Vector Fields”	“Surface and Mesh Plots”		“Volume Visualiz	“Animat	“Images”
plot 	area 	stairs 	polarp 	contou 	quiver 	surf 	mesh 	stream 	animat 	image 
plot3 	pie 	stem 	polarh 	contou 	quiver 	surfc 	meshc 	stream 	comet 	imagesc 
semilo 	pie3 	stem3 	polars 	contou 	feathe 	surfl 	meshz 	stream 	comet3 	
semilo 	bar 	scatte 	compas 	contou 		ribbon 	waterf 	stream 		
loglog 	barh 	scatte 	ezpola 	fconto 		pcolor 	fmesh 	stream 		
errorb	bar3	spy				fsurf		conepl		

"Line Plots"	"Pie Charts, Bar Plots, and Histogra	"Discret Data Plots"	"Polar Plots"	"Contou Plots"	"Vector Fields"	"Surface and Mesh Plots"		"Volume Visualiz	"Animat	"Images"
										
fplot 	bar3h 	plotma 				fimpli 		slice 		
fplot3 	histog 	heatma 								
fimpli 	histog 									
	pareto 									

Related Examples

- "Create 2-D Graph and Customize Lines" on page 2-2
- "Create Graph Using Plots Tab" on page 1-4

External Websites

- MATLAB Plot Gallery

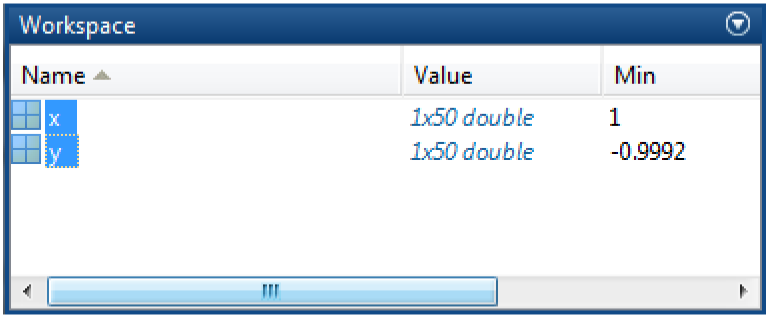
Create Graph Using Plots Tab

This example shows how to create a 2-D line plot interactively using the **Plots** tab in the MATLAB toolstrip. 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);
```

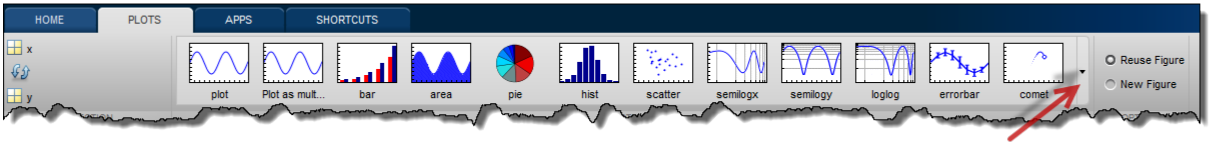
2 In the Workspace panel in the MATLAB desktop, select the variables to plot. Use **Ctrl** + click to select multiple variables.



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

area | bar | histogram | pie | plot | scatter

Related Examples

- “Customize Graph Using Plot Tools” on page 1-6

External Websites

- MATLAB Plot Gallery

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.

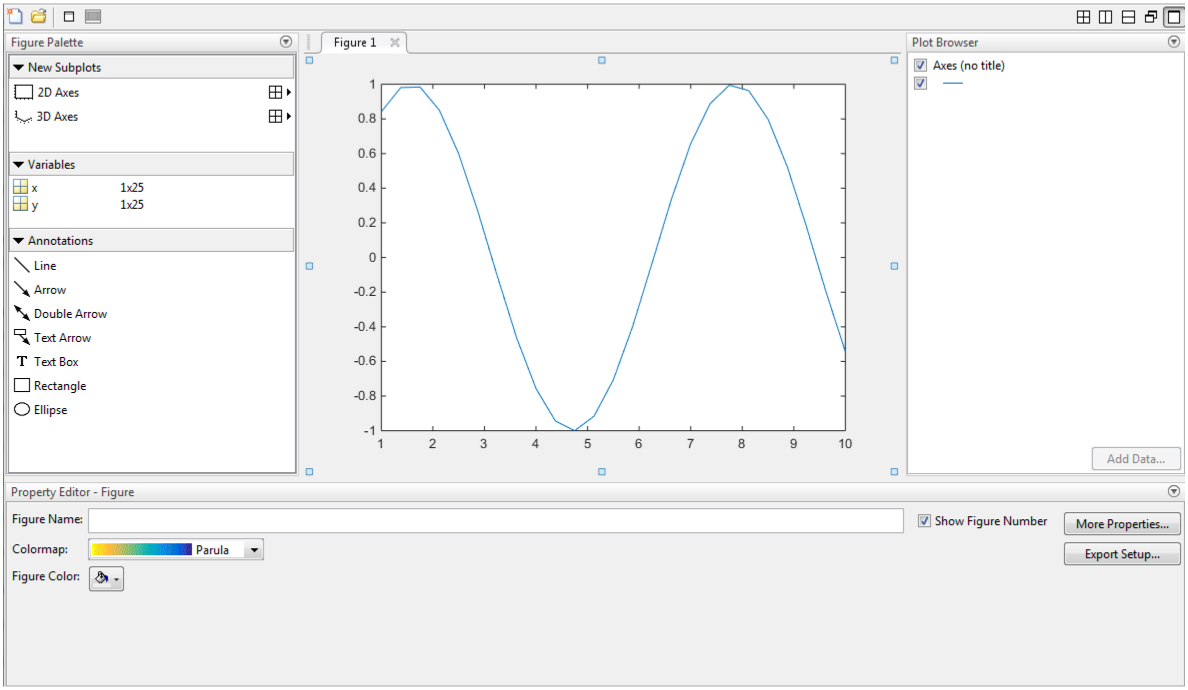
In this section...

"Open Plot Tools" on page 1-6
"Customize Objects in Graph" on page 1-7
"Control Visibility of Objects in Graph" on page 1-8
"Add Annotations to Graph" on page 1-8
"Close Plot Tools" on page 1-8

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
```



Plot Browser: ☑ Axes (no title) ☑

Figure Name:

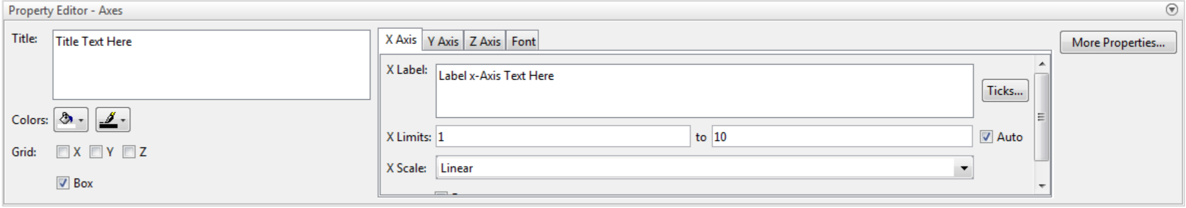
Colormap: Parula

☑ Show Figure Number

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.



Title: Title Text Here

Grid: □ X □ Y □ Z

☑ Box

X Label: Label x-Axis Text Here

X Limits: 1 to 10 ☑ Auto

X Scale: Linear

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 clearing 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 1-9

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 1-9

"Create Upper and Lower Subplots" on page 1-9

"Add Data to Lower Subplot" on page 1-10

"Add New Plot Without Overwriting Existing Plot" on page 1-11

Create Simple Line Plot and Open Plot Tools

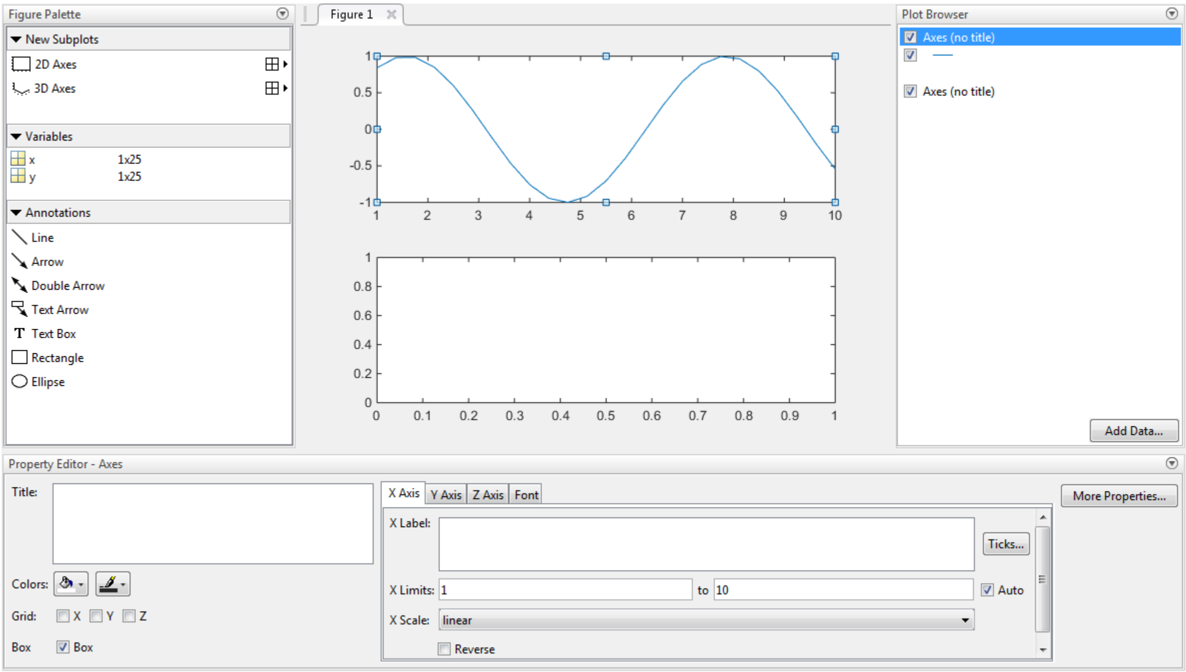
Define variables `x` and `y` in the Command Window and create a line plot using the `plot` function. Open the plot tools using the `plottools` command or by clicking the Show Plot Tools icon ▣ in the figure window.

```
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 ⊞▸.



x 1x25
y 1x25

Grid: □ X □ Y □ Z

Box ☑ Box

X Limits: 1 to 10 ☑ Auto

X Scale: linear

□ Reverse

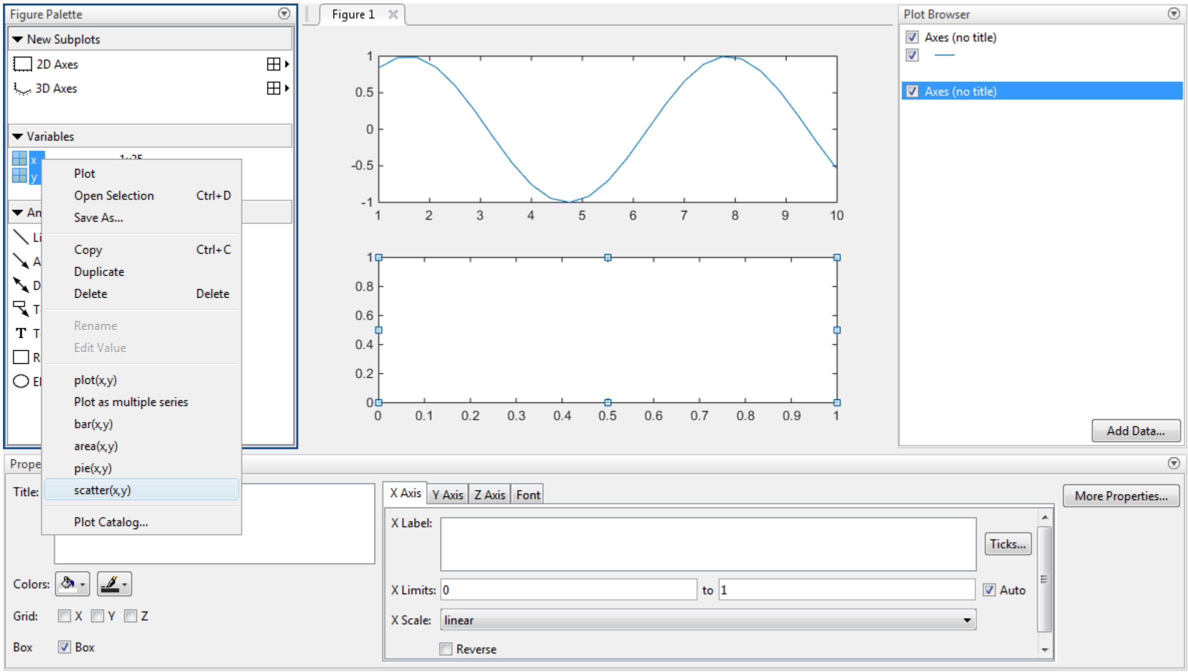
Add Data to Lower Subplot

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.



X Limits: 0 to 1 ☑ Auto

X Scale: linear

Grid: □ X □ Y □ Z

Box ☑ Box

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.

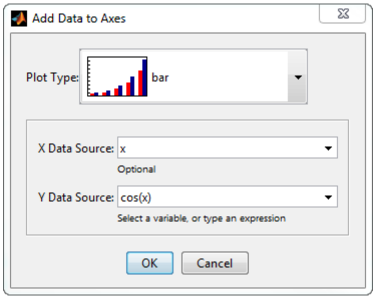
Add New Plot Without Overwriting Existing Plot

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**.

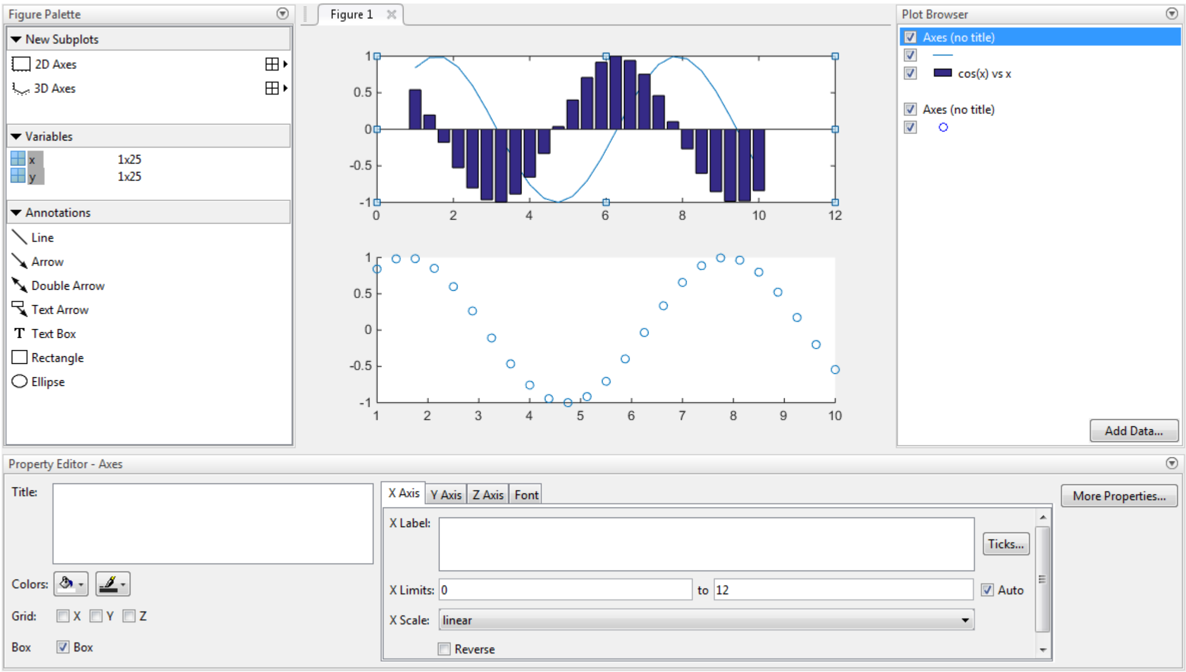


Plot Type: bar

X Data Source: x

Y Data Source: cos(x)

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



x 1x25
y 1x25

☑ Axes (no title)
☑ —
☑ cos(x) vs x
☑ Axes (no title)
☑ ○

Title:

Colors:

Grid: □ X □ Y □ Z

Box ☑ Box

X Label:

X Limits: 0 to 12 ☑ Auto

X Scale: linear

□ Reverse

See Also

`bar` | `figurepalette` | `plot` | `plottools` | `propertyeditor` | `scatter` | `subplot`

Related Examples

- “Customize Graph Using Plot Tools” on page 1-6

Basic Plotting Commands

- “Create 2-D Graph and Customize Lines” on page 2-2
- “Add Title, Axis Labels, and Legend to Graph” on page 2-13
- “Specify Axis Limits” on page 2-19
- “Specify Axis Tick Values and Labels” on page 2-28
- “Add Grid Lines and Edit Placement” on page 2-38
- “Combine Multiple Plots” on page 2-48
- “Create Chart with Two *y*-Axes” on page 2-55
- “Modify Properties of Charts with Two *y*-Axes” on page 2-64
- “Plot Imaginary and Complex Data” on page 2-71
- “Create Heatmap from Tabular Data” on page 2-74

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 2-7

“Specify Line Style and Color” on page 2-8

“Specify Line Style, Color, and Markers” on page 2-9

“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.

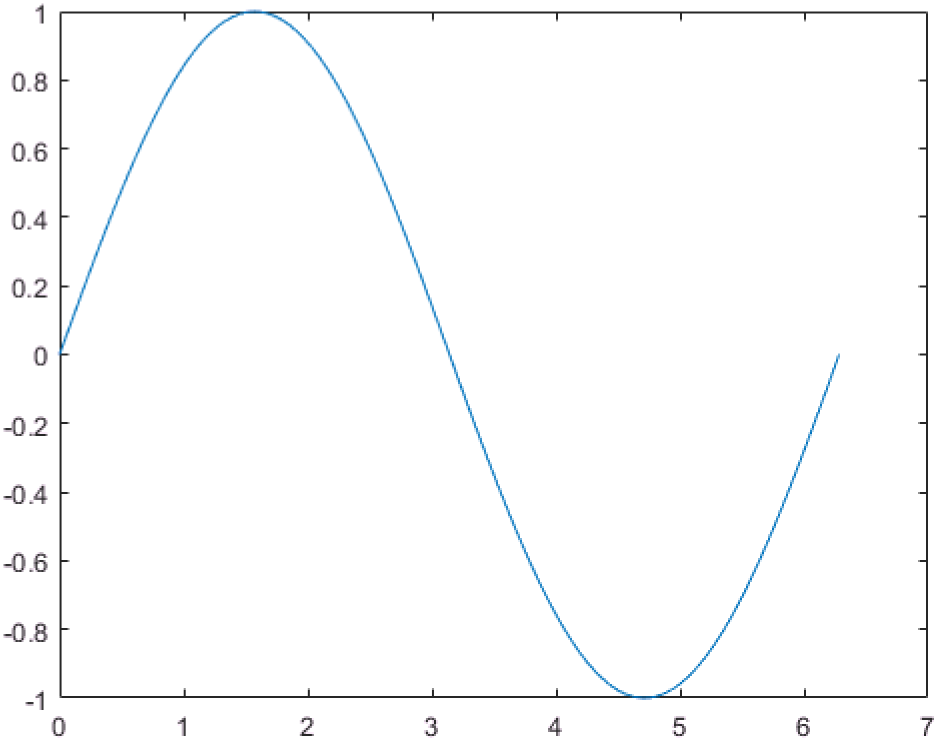
```
x = linspace(0,2*pi,100);
```

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

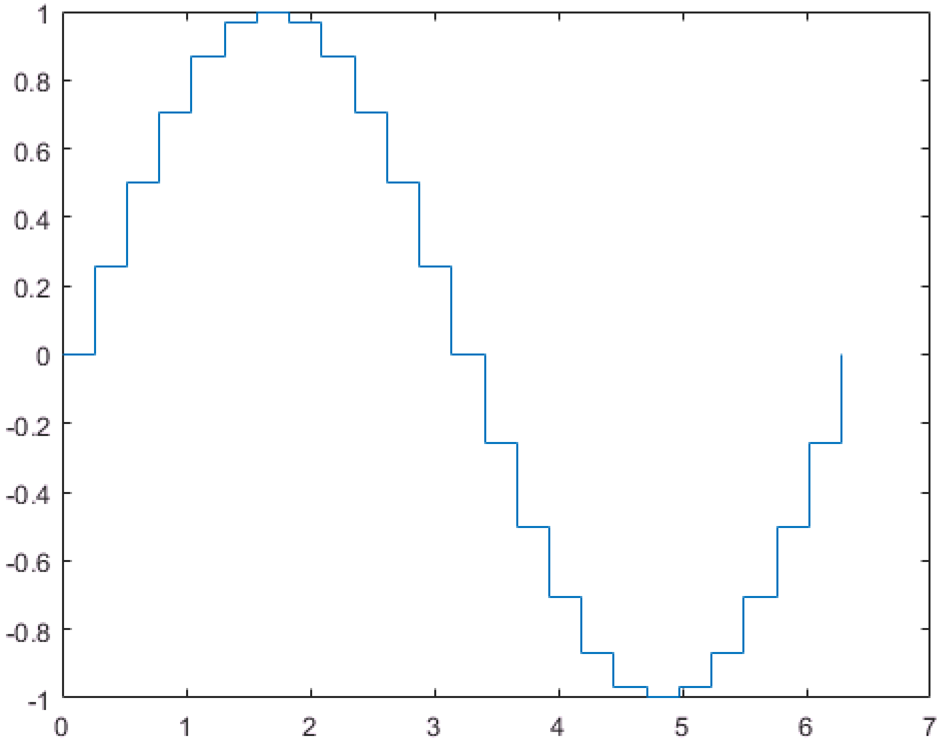
This example shows how to create a graph in a new figure window, instead of plotting into the current figure.

Define `x` and `y`.

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

Create a stairstep 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

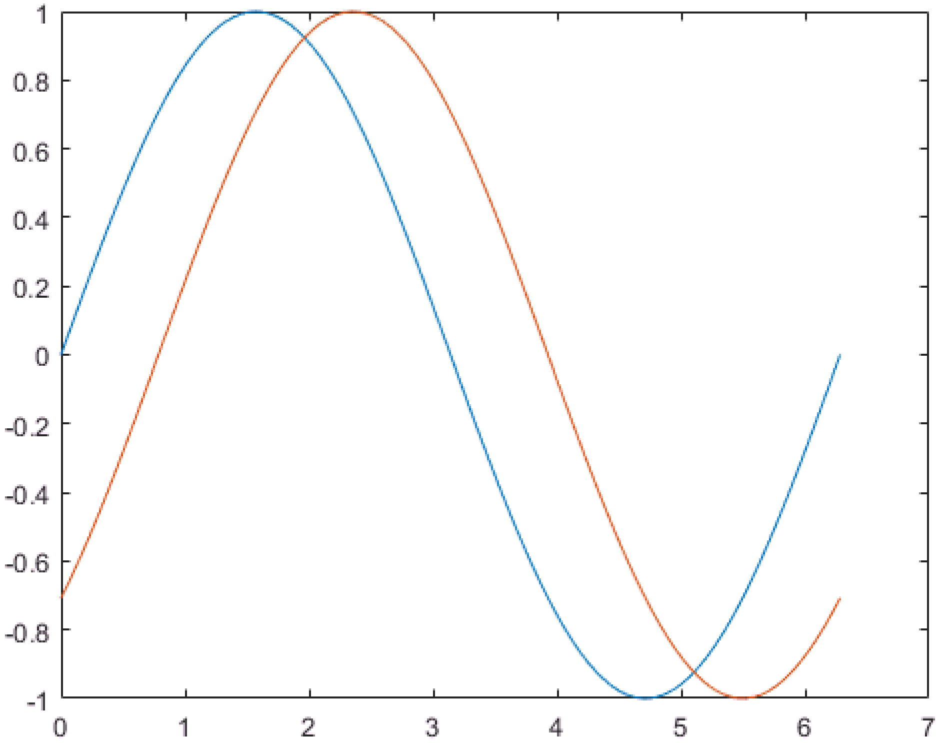
This example shows how to plot more than one line by passing multiple `x,y` pairs to the `plot` function.

Define `y1` and `y2` 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

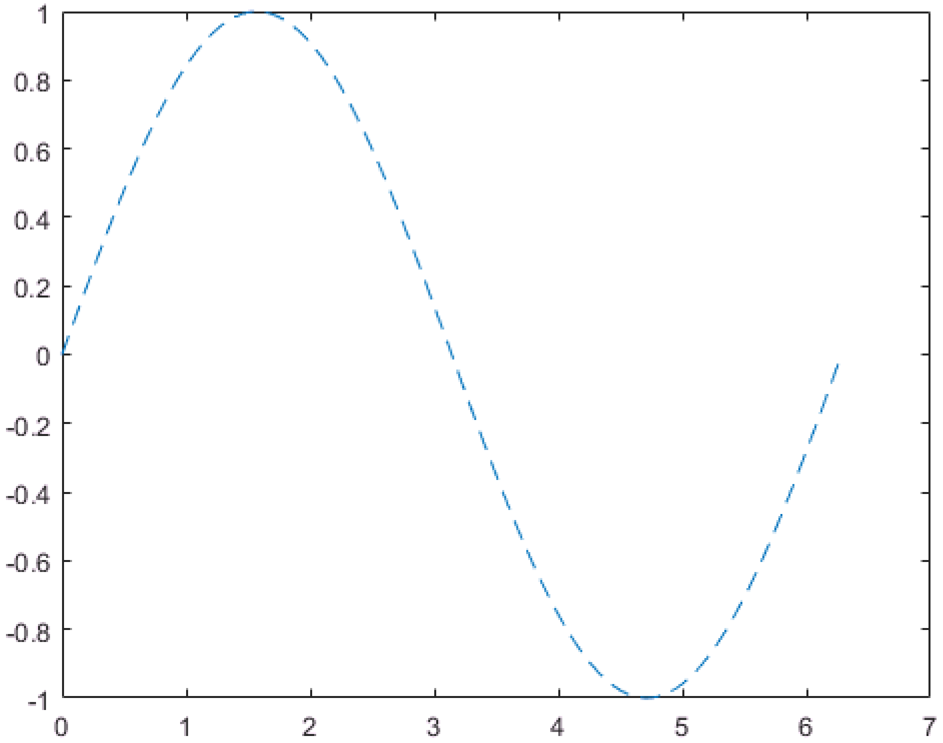
To change the line color, line style, and marker type, add a line specification input argument to the `x,y` pair. For example, `'g:*'` plots a green dotted line with star markers. You can omit one or more options from the line specification, such as `'g:'` for a green dotted line with no markers. To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

For more information, see the LineSpec input argument for plot.

Specify Line Style

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

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

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



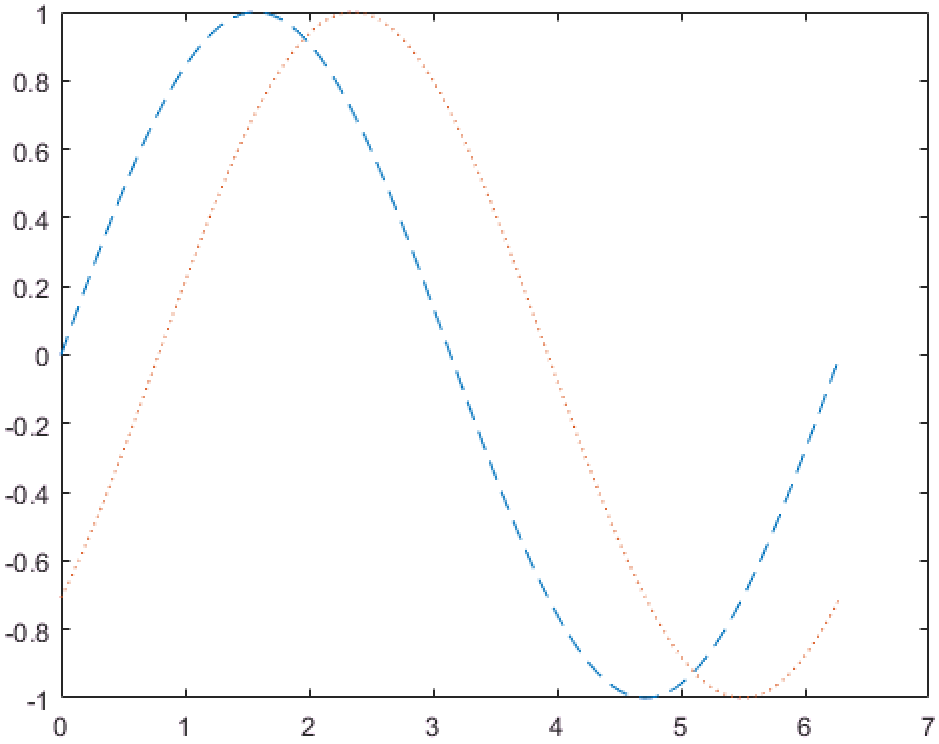
Specify Different Line Styles for Multiple Lines

This example shows how to plot two sine waves with different line styles by adding a line specification 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,':')
```



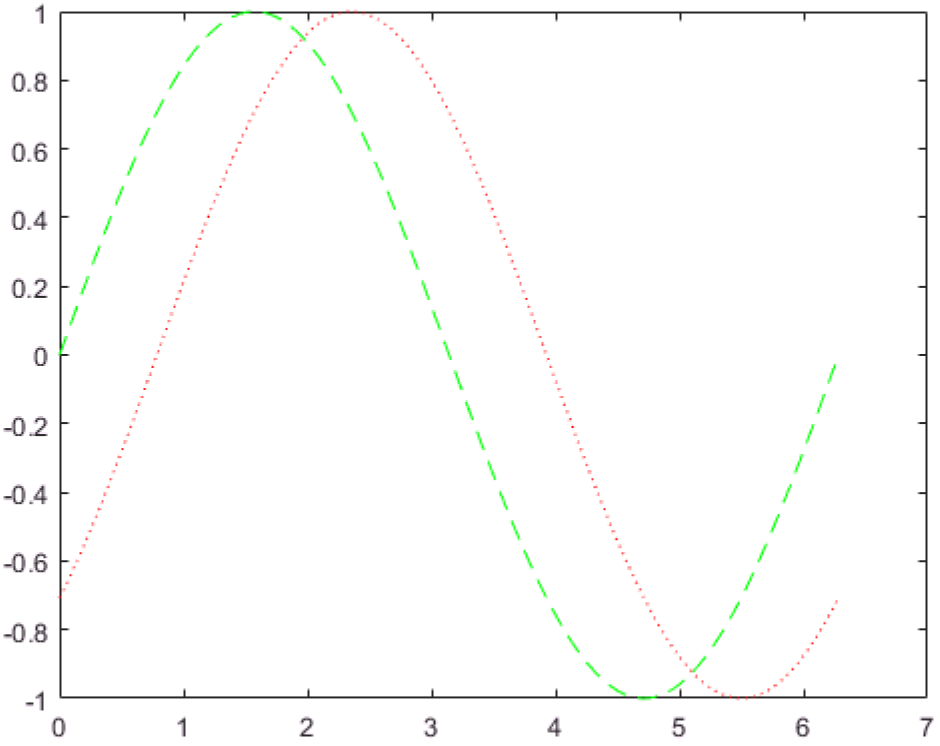
Specify Line Style and Color

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 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')
```

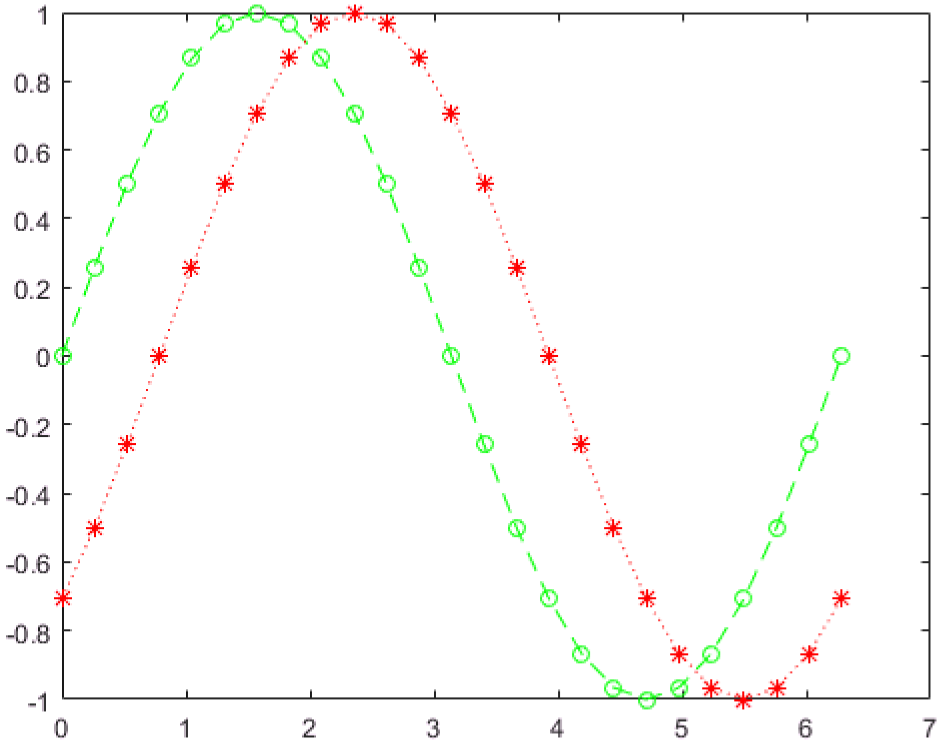


Specify Line Style, Color, and Markers

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*')
```

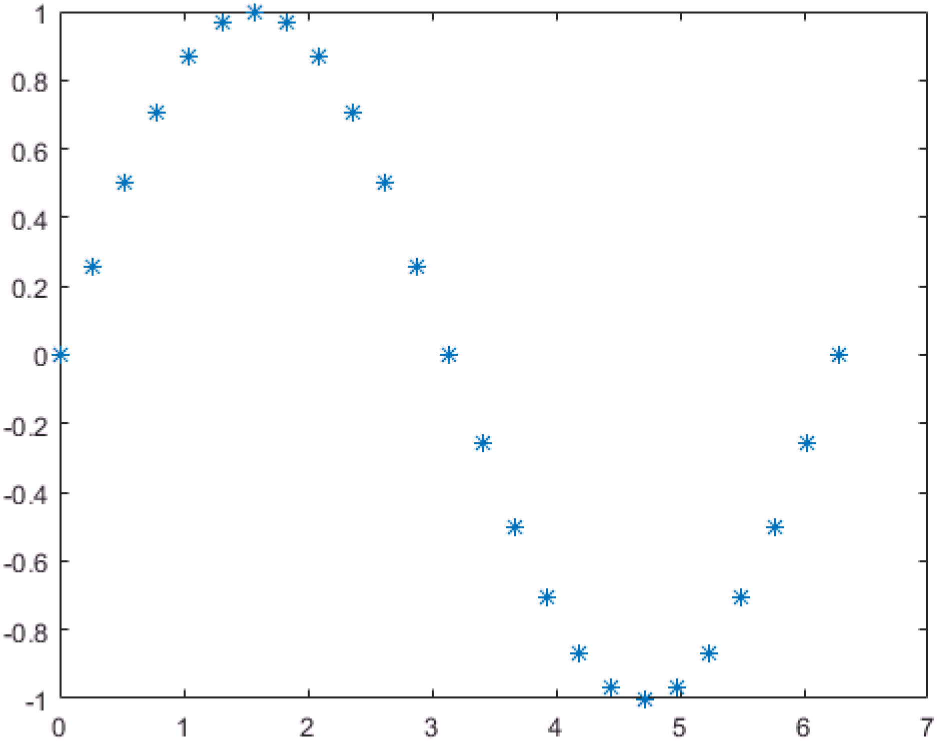


Plot Only Data Points

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

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 | linspace | loglog | plot | plotyy | scatter | semilogx | semilogy | stairs | stem

Related Examples

- "Add Title, Axis Labels, and Legend to Graph" on page 2-13
- "Specify Axis Limits" on page 2-19
- "Specify Axis Tick Values and Labels" on page 2-28

External Websites

- MATLAB Plot Gallery

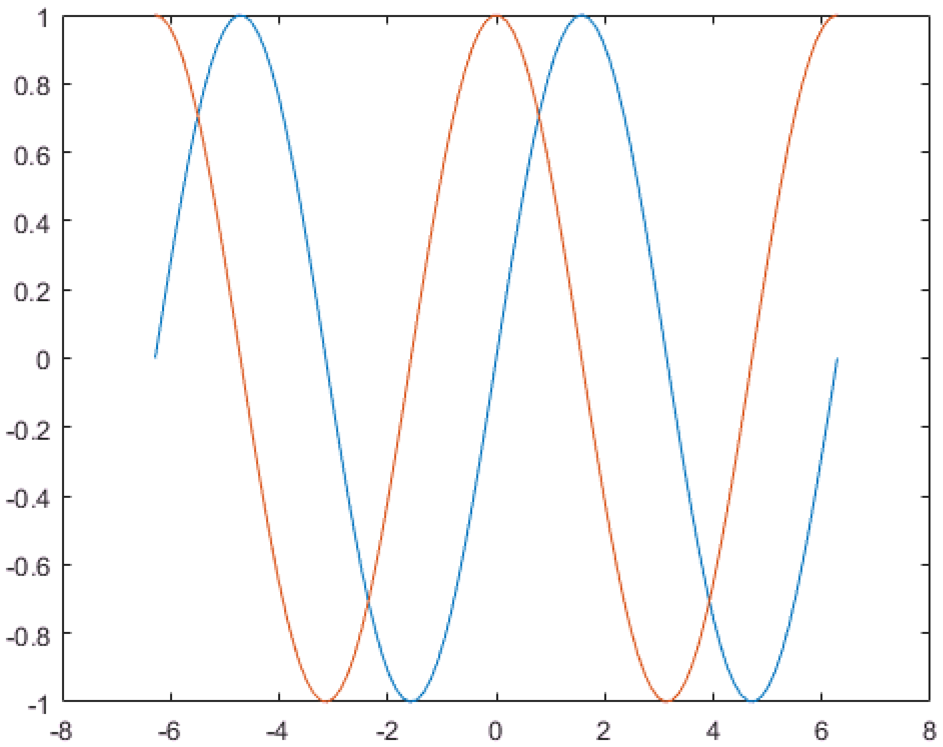
Add Title, Axis Labels, and Legend to Graph

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.

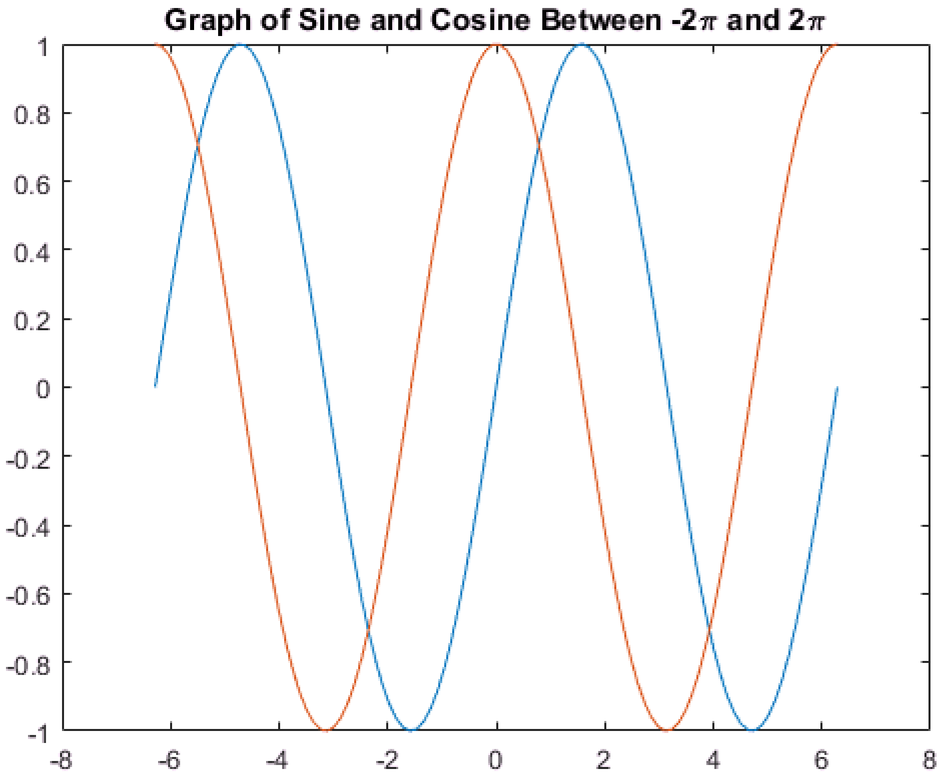
```
x = linspace(-2*pi,2*pi,100);
y1 = sin(x);
y2 = cos(x);

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



Add Title

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

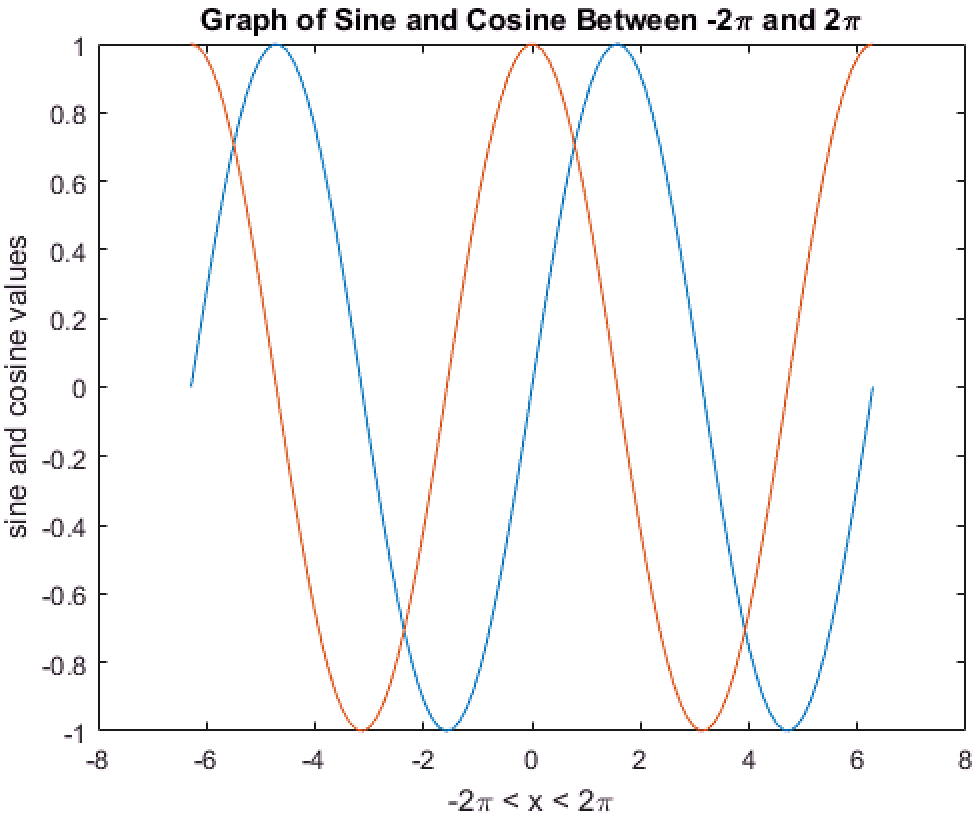
```
title('Graph of Sine and Cosine Between -2\pi and 2\pi')
```



Add Axis Labels

Add axis labels to the graph using the `xlabel` and `ylabel` functions.

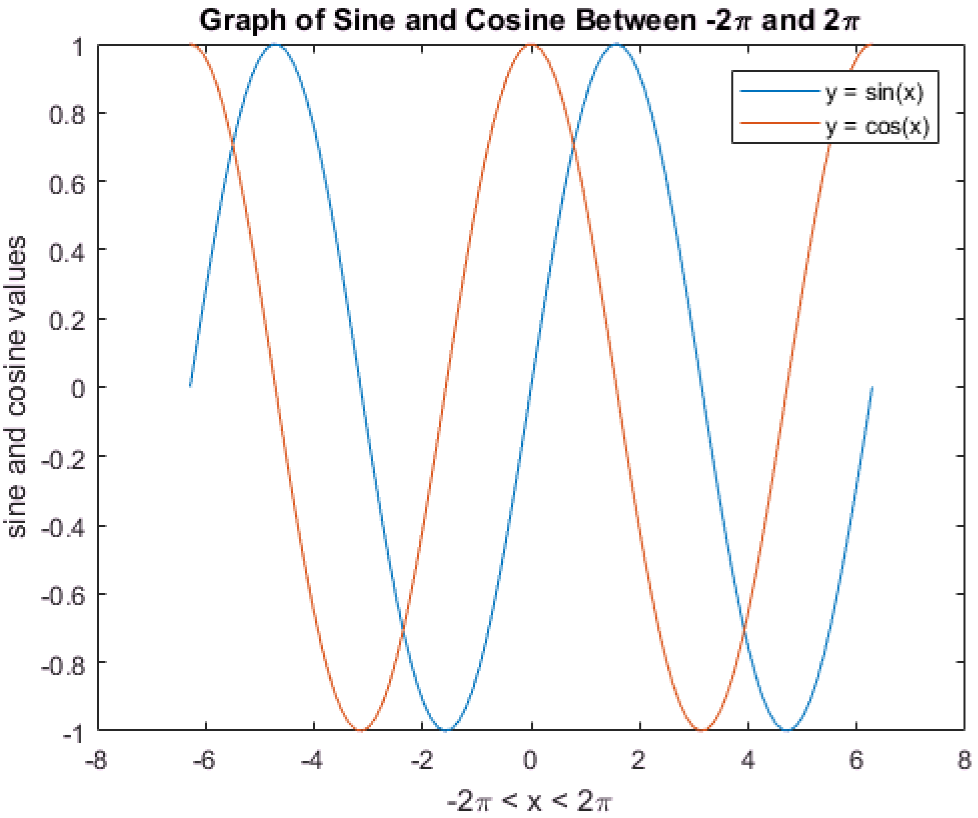
```
xlabel('-2\pi < x < 2\pi') % x-axis label
ylabel('sine and cosine values') % y-axis label
```



Add Legend

Add a legend to the graph identifying each data set using the `legend` function. Specify legend descriptions in the order that you plot the lines.

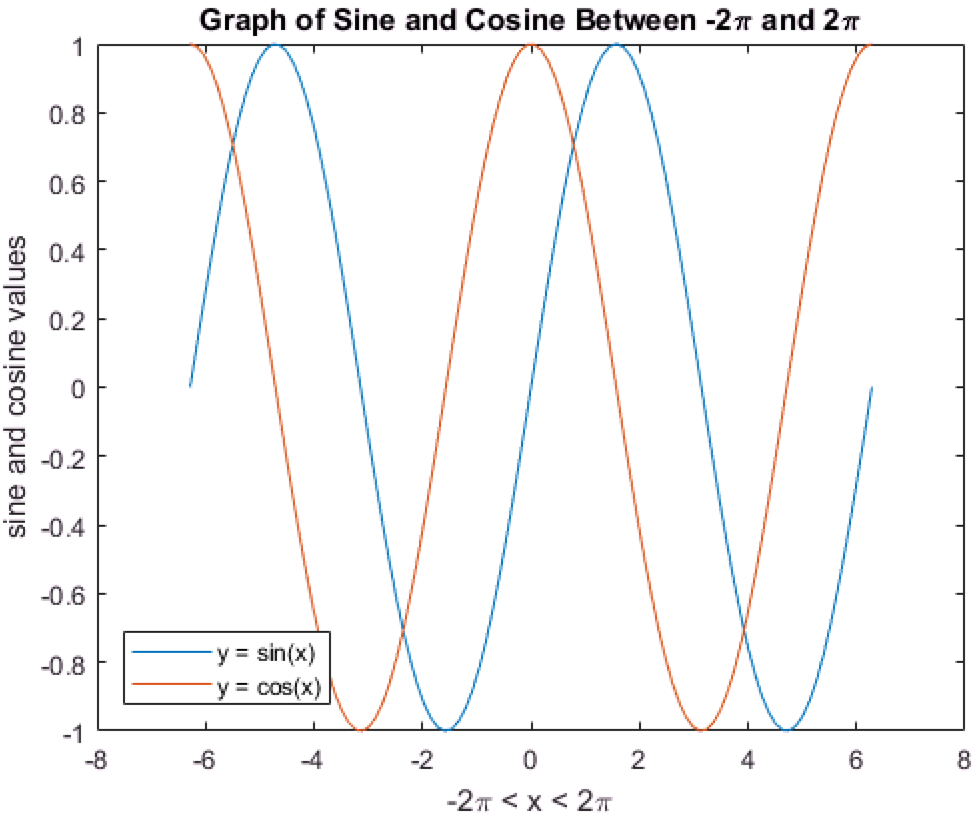
```
legend('y = sin(x)','y = cos(x)')
```



Specify Legend Location

Specify 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 the location as `'southwest'`.

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



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

See Also

`legend` | `linspace` | `title` | `xlabel` | `ylabel`

Related Examples

- "Specify Axis Limits" on page 2-19
- "Specify Axis Tick Values and Labels" on page 2-28

Specify Axis Limits

In this section...

“Change Axis Limits” on page 2-19

“Use Semiautomatic Axis Limits” on page 2-20

“Revert Back to Default Axis Limits” on page 2-21

“Reverse Axis Direction” on page 2-23

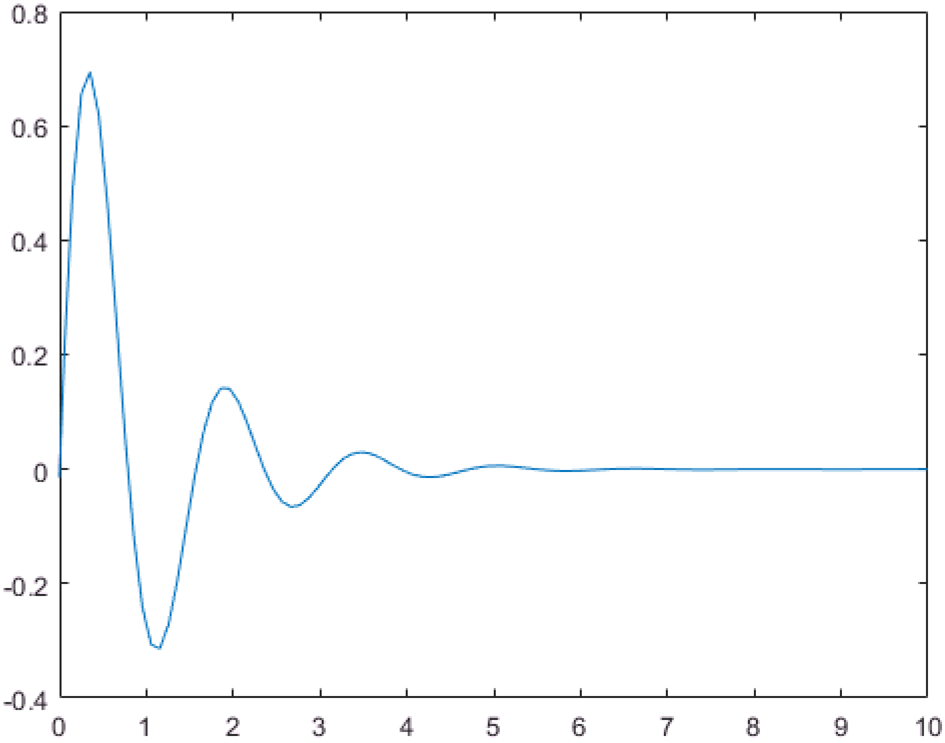
“Display Axis Lines through Origin” on page 2-24

You can control where data appears in the axes by setting the *x*-axis, *y*-axis, and *z*-axis limits. You also can change where the *x*-axis and *y*-axis lines appear (2-D plots only) or reverse the direction of increasing values along each axis.

Change Axis Limits

Create a line plot. Specify the axis limits using the `xlim` and `ylim` functions. For 3-D plots, use the `zlim` function. Pass the functions a two-element vector of the form `[min max]`.

```
x = linspace(-10,10,200);
y = sin(4*x)./exp(x);
plot(x,y)
xlim([0 10])
ylim([-0.4 0.8])
```

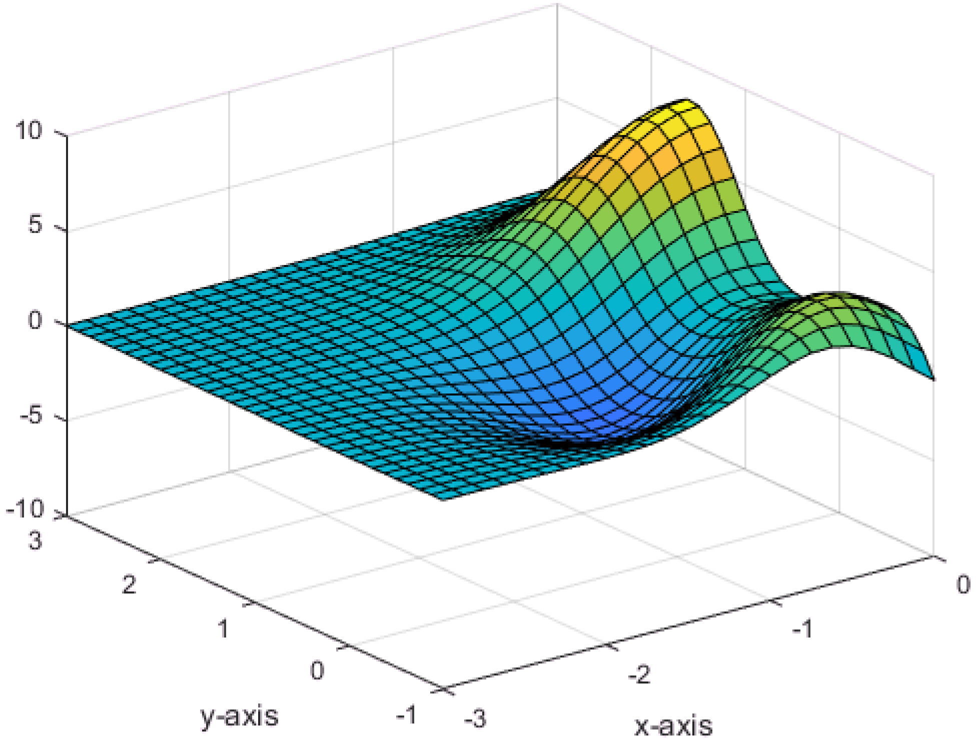


Use Semiautomatic Axis Limits

Set the maximum *x*-axis limit to 0 and the minimum *y*-axis limit to -1. Let MATLAB choose the other limits. For an automatically calculated minimum or maximuim limit, use `-inf` or `inf`, respectively.

```
[X,Y,Z] = peaks;
surf(X,Y,Z)
xlabel('x-axis')
ylabel('y-axis')
xlim([-inf 0])
```

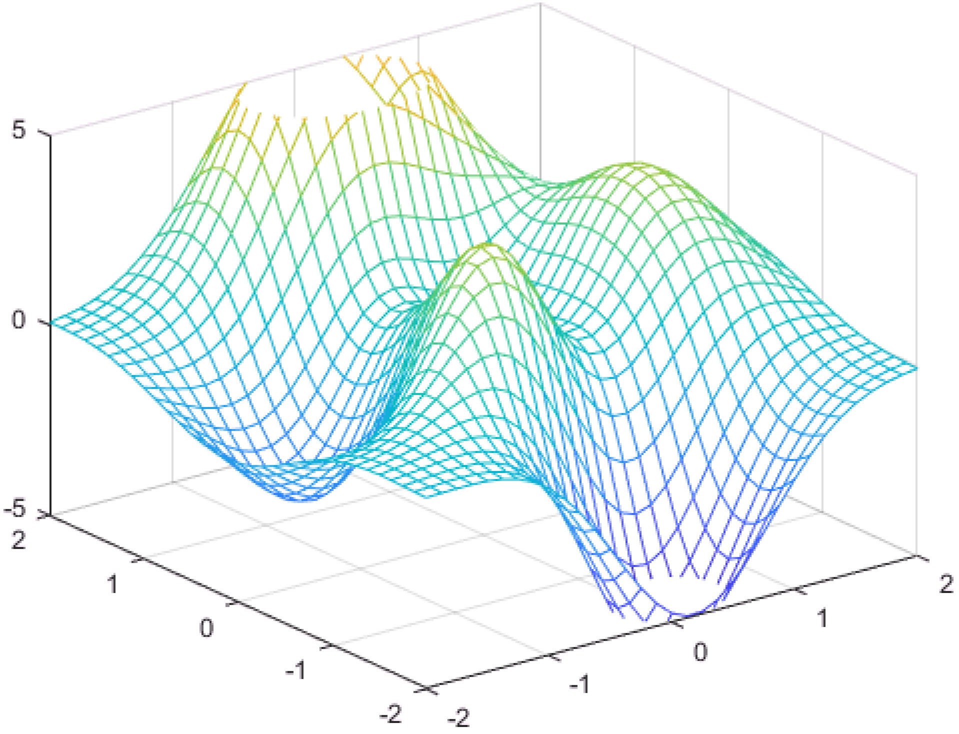
```
ylim([-1 inf])
```



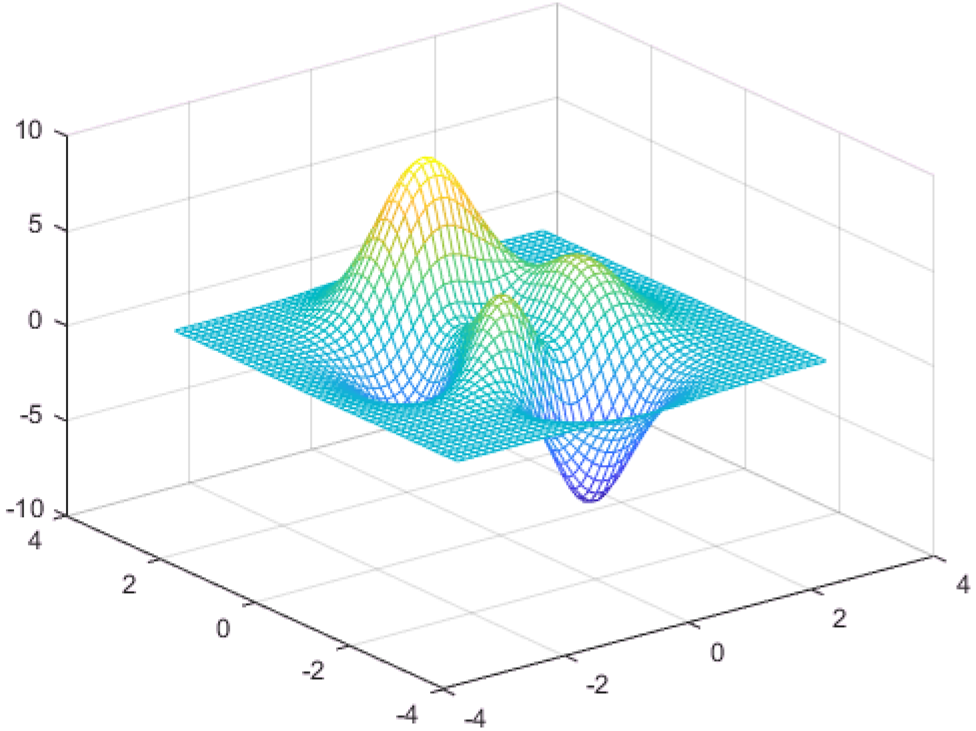
Revert Back to Default Axis Limits

Create a mesh plot and change the axis limits. Then revert back to the default limits.

```
[X,Y,Z] = peaks;
mesh(X,Y,Z)
xlim([-2 2])
ylim([-2 2])
zlim([-5 5])
```



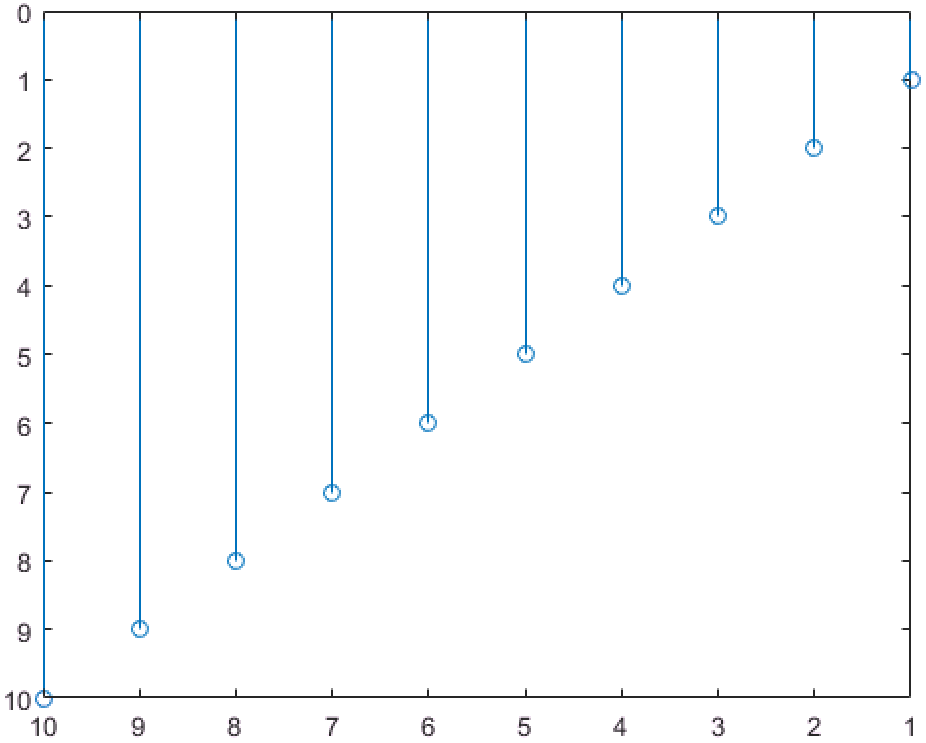
```
xlim auto
ylim auto
zlim auto
```



Reverse Axis Direction

Control the direction of increasing values along the *x*-axis and *y*-axis by setting the `XDir` and `YDir` properties of the `Axes` object. Set these properties to either `'reverse'` or `'normal'` (the default). Use the `gca` command to access the `Axes` object.

```
stem(1:10)
ax = gca;
ax.XDir = 'reverse';
ax.YDir = 'reverse';
```

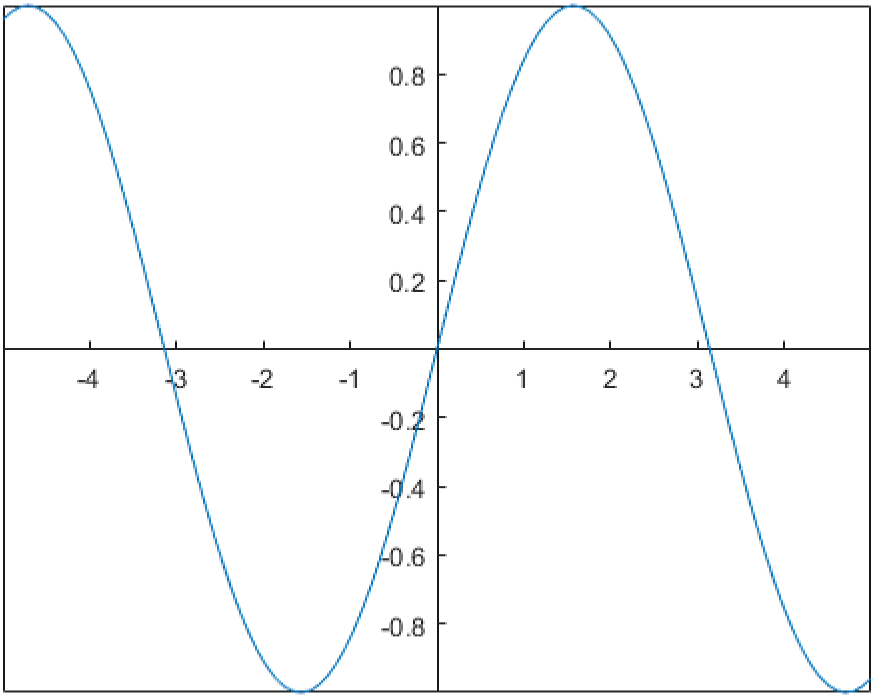


Display Axis Lines through Origin

By default, the *x*-axis and *y*-axis appear along the outer bounds of the axes. Change the location of the axis lines so that they cross at the origin point `(0,0)` by setting the `XAxisLocation` and `YAxisLocation` properties of the `Axes` object. Set `XAxisLocation` to either `'top'`, `'bottom'`, or `'origin'`. Set `YAxisLocation` to either `'left'`, `'right'`, or `'origin'`. These properties only apply to axes in a 2-D view.

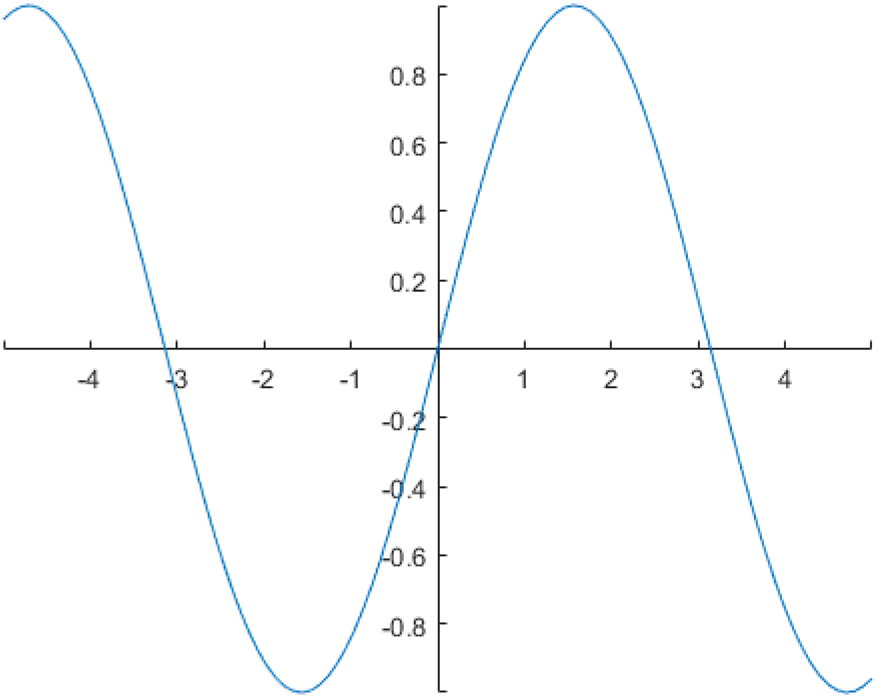
```
x = linspace(-5,5);
y = sin(x);
```

```
plot(x,y)

ax = gca;
ax.XAxisLocation = 'origin';
ax.YAxisLocation = 'origin';
```



Remove the axes box outline.

```
box off
```



See Also

Functions

`axis` | `grid` | `xlim` | `xticks` | `ylim` | `yticks` | `zlim` | `zticks`

Properties

Axes Properties

Related Examples

- “Specify Axis Tick Values and Labels” on page 2-28

- “Add Grid Lines and Edit Placement” on page 2-38
- “Add Title, Axis Labels, and Legend to Graph” on page 2-13

Specify Axis Tick Values and Labels

In this section...

“Change Tick Value Locations and Labels” on page 2-28

“Rotate Tick Labels” on page 2-30

“Change Tick Label Formatting” on page 2-32

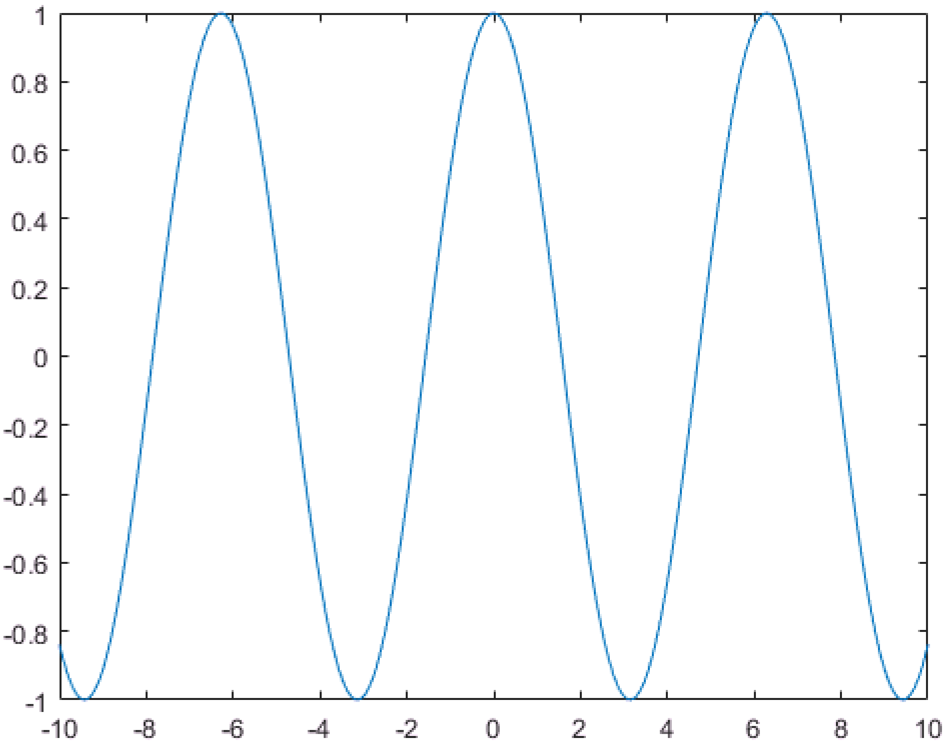
“Control Value in Exponent Label” on page 2-33

Customizing the tick values and labels along an axis can help highlight particular aspects of your data. These examples show some common customizations, such as modifying the tick value placement, changing the tick label text and formatting, and rotating the tick labels.

Change Tick Value Locations and Labels

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

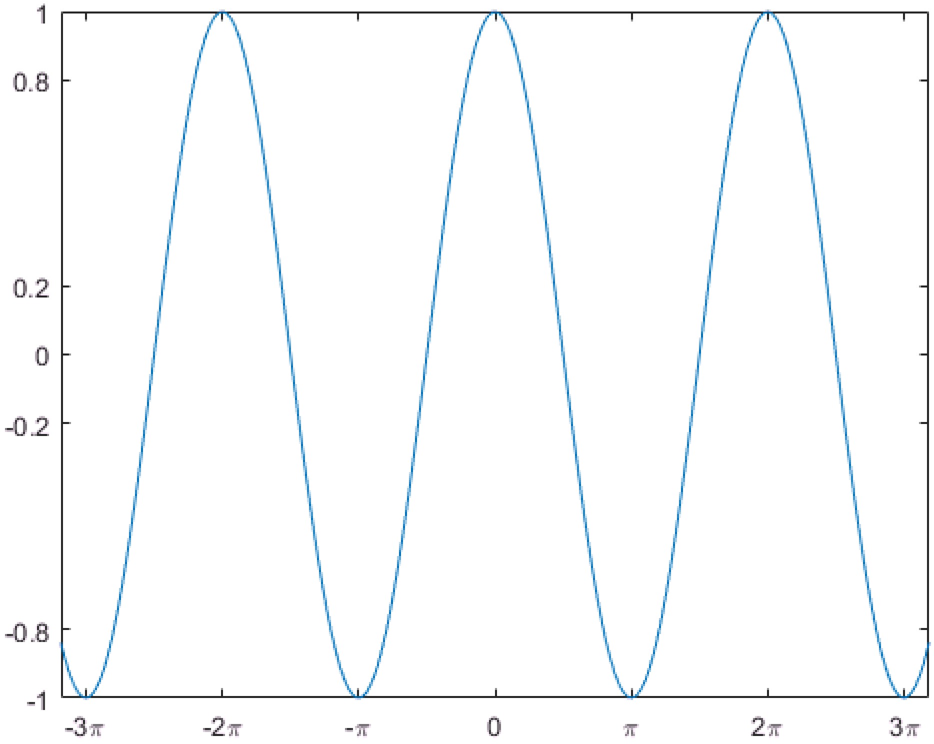
```
x = linspace(-10,10,200);
y = cos(x);
plot(x,y)
```



Change the tick value locations along the *x*-axis and *y*-axis. Specify the locations as a vector of increasing values. The values do not need to be evenly spaced.

Also, change the labels associated with each tick value along the *x*-axis. Specify the labels using a cell array of character vectors. To include special characters or Greek letters in the labels, use TeX markup, such as `\pi` for the π symbol.

```
xticks([-3*pi -2*pi -pi 0 pi 2*pi 3*pi])
xticklabels({'-3\pi','-2\pi','-\pi','0','\pi','2\pi','3\pi'})
yticks([-1 -0.8 -0.2 0 0.2 0.8 1])
```

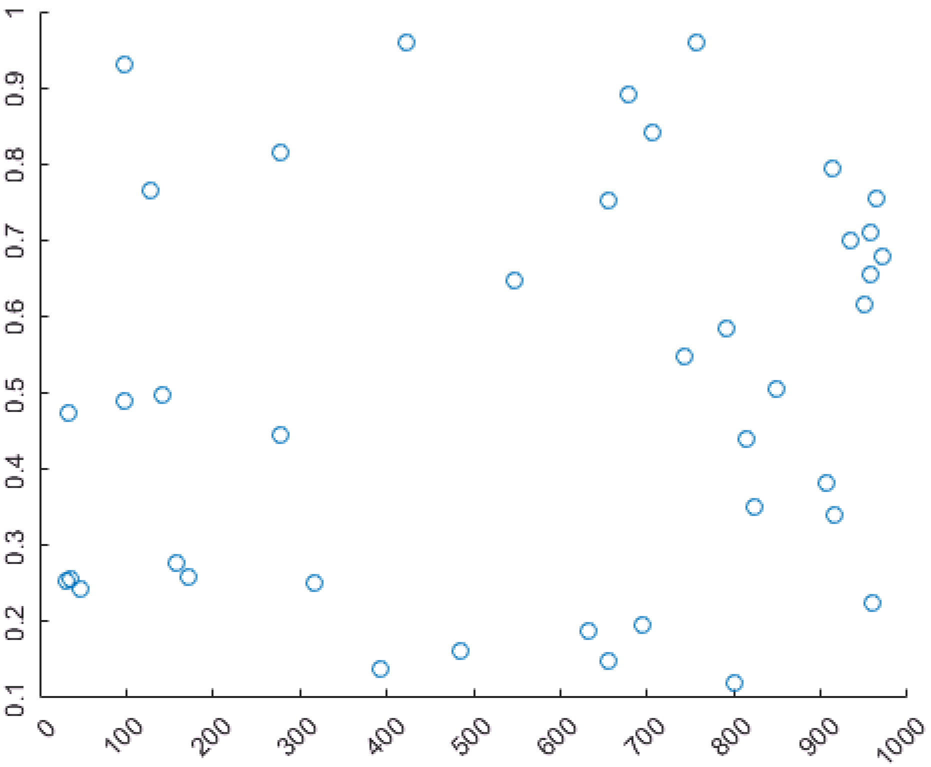


For releases prior to R2016b, instead set the tick values and labels using the `XTick`, `XTickLabel`, `YTick`, and `YTickLabel` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTick` property using dot notation, such as `ax.XTick = [-3*pi -2*pi -pi 0 pi 2*pi 3*pi]`. For releases prior to R2014b, use the `set` function to set the property instead.

Rotate Tick Labels

Create a scatter plot and rotate the tick labels along each axis. Specify the rotation as a scalar value. Positive values indicate counterclockwise rotation. Negative values indicate clockwise rotation.

```
x = 1000*rand(40,1);
y = rand(40,1);
scatter(x,y)
xtickangle(45)
ytickangle(90)
```

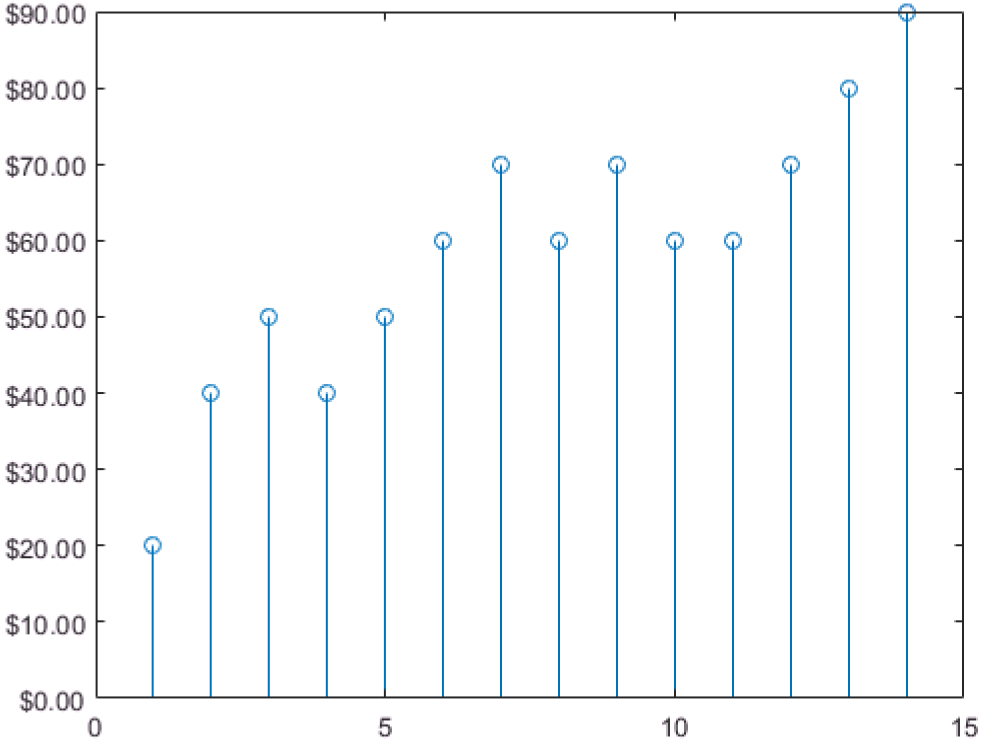


For releases prior to R2016b, specify the rotation using the `XTickLabelRotation` and `YTickLabelRotation` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTickLabelRotation` property using dot notation, such as `ax.XTickLabelRotation = 45`.

Change Tick Label Formatting

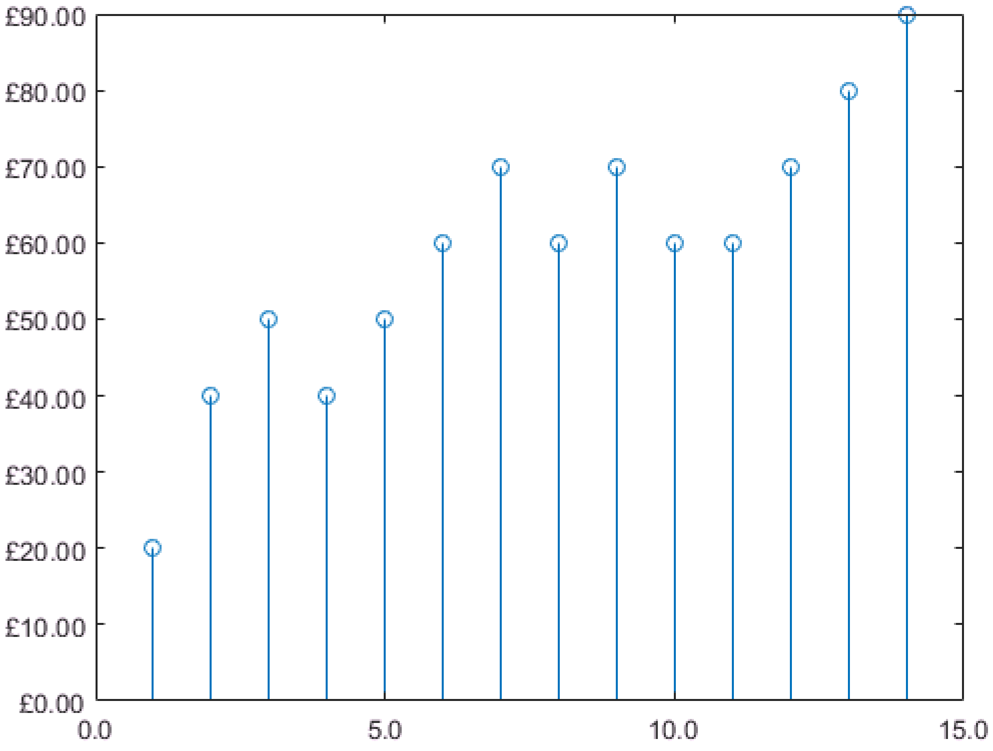
Create a stem chart and display the tick label values along the *y*-axis as US dollar values.

```
profit = [20 40 50 40 50 60 70 60 70 60 60 70 80 90];
stem(profit)
xlim([0 15])
ytickformat('usd')
```



For more control over the formatting, specify a custom format. For example, show one decimal value in the *x*-axis tick labels using `'%.1f'`. Display the *y*-axis tick labels as British Pounds using `'\xA3%.2f'`. The option `\xA3` indicates the Unicode character for the Pound symbol. For more information on specifying a custom format, see the xtickformat function.

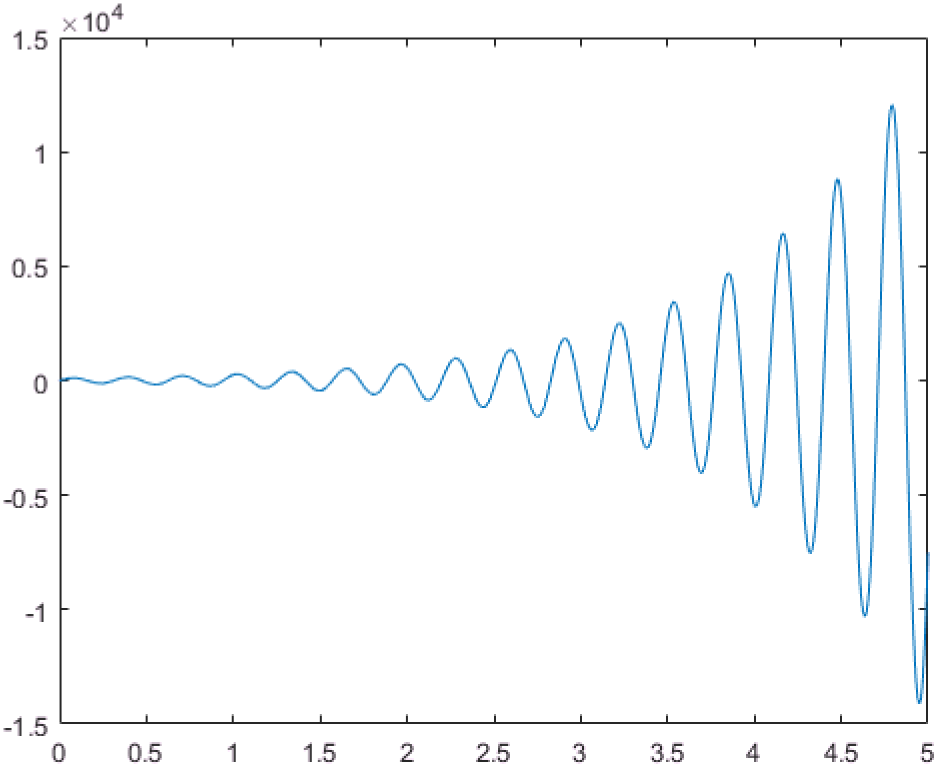
```
xtickformat('%.1f')
ytickformat('\xA3%.2f')
```



Control Value in Exponent Label

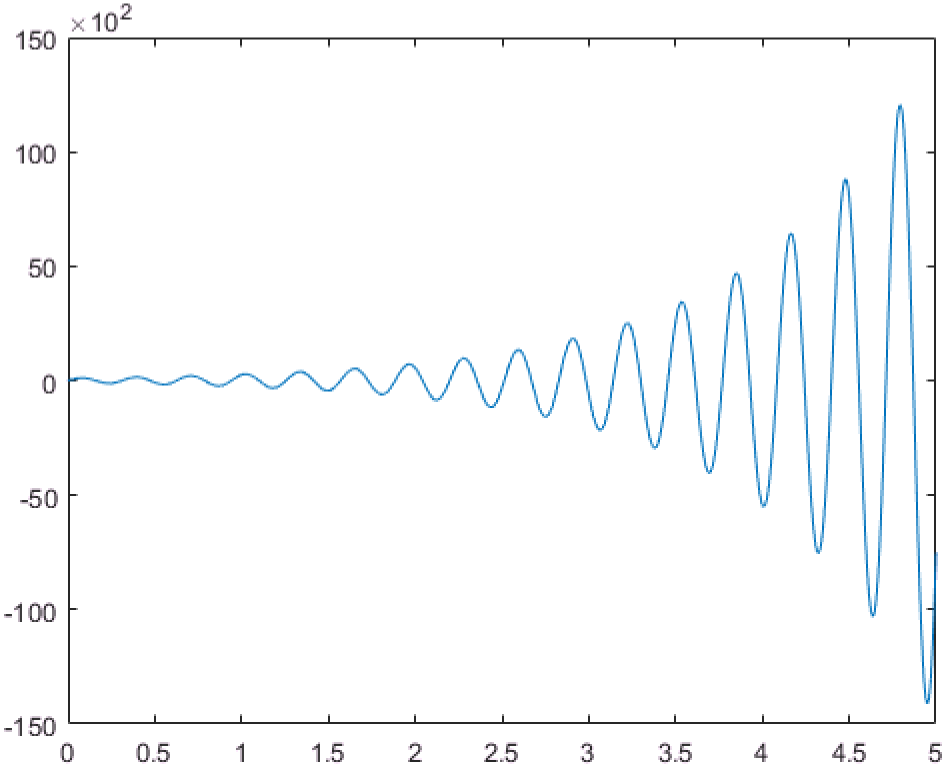
Plot data with *y* values that range between -15,000 and 15,000. By default, the *y*-axis tick labels use exponential notation with an exponent value of 4 and a base of 10.

```
x = linspace(0,5,1000);
y = 100*exp(x).*sin(20*x);
plot(x,y)
```



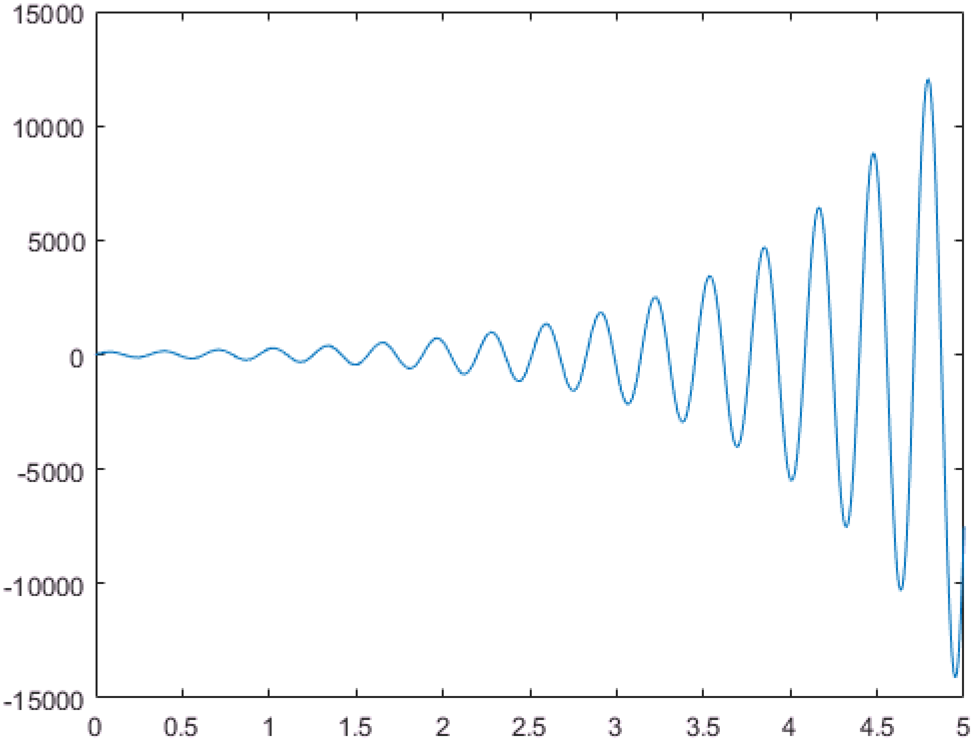
Change the exponent value to 2. Set the `Exponent` property of the ruler object associated with the *y*-axis. Access the ruler object through the `YAxis` property of the `Axes` object. The exponent label and the tick labels change accordingly.

```
ax = gca;
ax.YAxis.Exponent = 2;
```



Change the exponent value to 0 so that the tick labels do not use exponential notation.

```
ax.YAxis.Exponent = 0;
```



See Also

Functions

`xlim` | `xtickangle` | `xtickformat` | `xticks` | `yticks` | `zticks`

Properties

Axes Properties | NumericRuler Properties

Related Examples

- "Add Grid Lines and Edit Placement" on page 2-38

- “Specify Axis Limits” on page 2-19
- “Add Title, Axis Labels, and Legend to Graph” on page 2-13

Add Grid Lines and Edit Placement

In this section...

"Display Grid Lines" on page 2-38

"Display Grid Lines in Specific Direction" on page 2-41

"Edit Grid Line Placement" on page 2-43

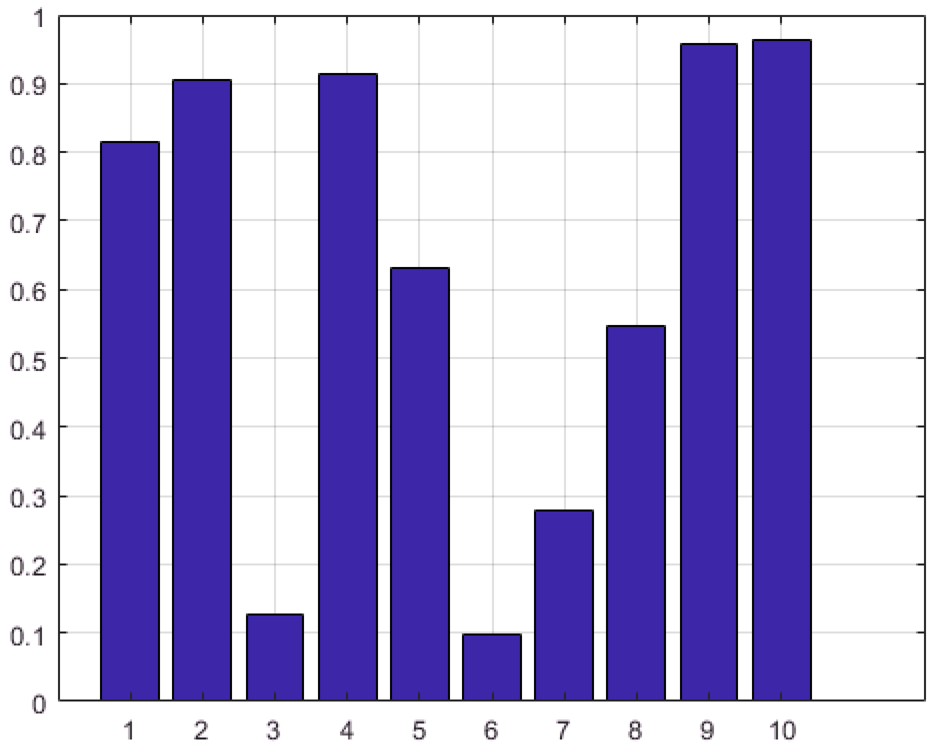
"Modify Visual Appearance of Grid Lines" on page 2-45

This example shows how to add grid lines to a graph. It also describes how to edit the placement of the grid lines and modify their appearance.

Display Grid Lines

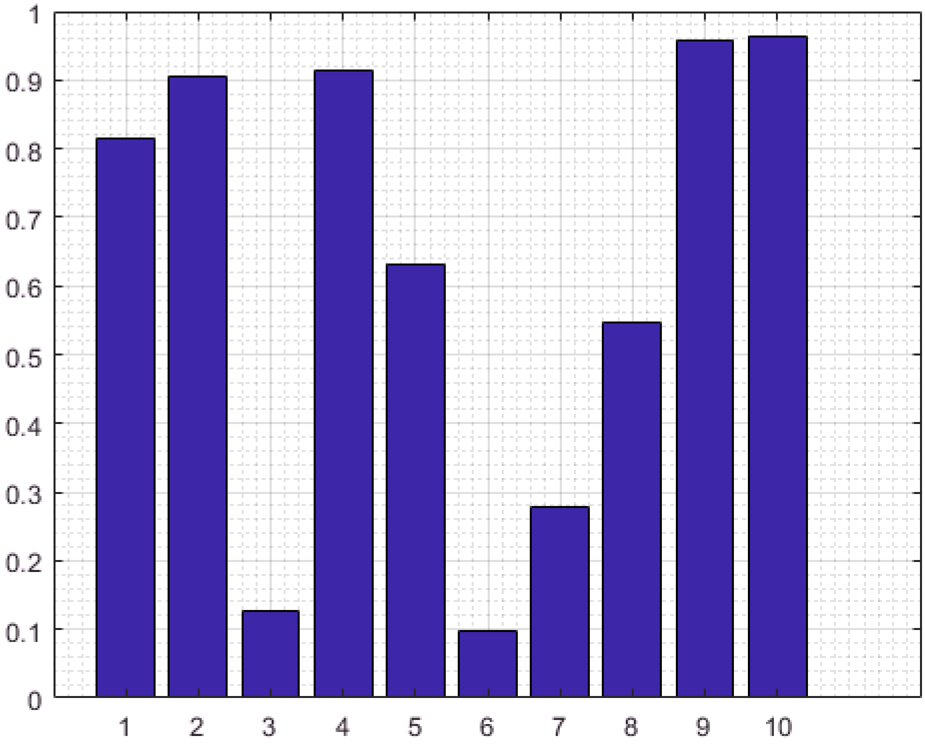
Create a bar chart and display grid lines. The grid lines appear at the tick marks.

```
y = rand(10,1);
bar(y)
grid on
```



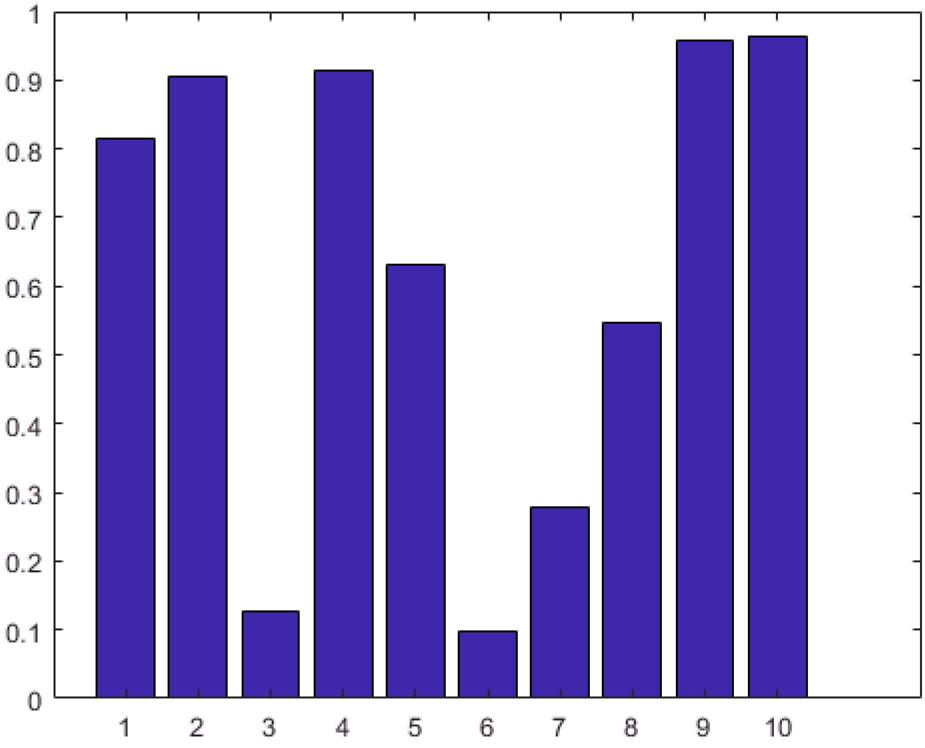
Add minor grid lines between the tick marks.

```
grid minor
```



Turn off all the grid lines.

```
grid off
```

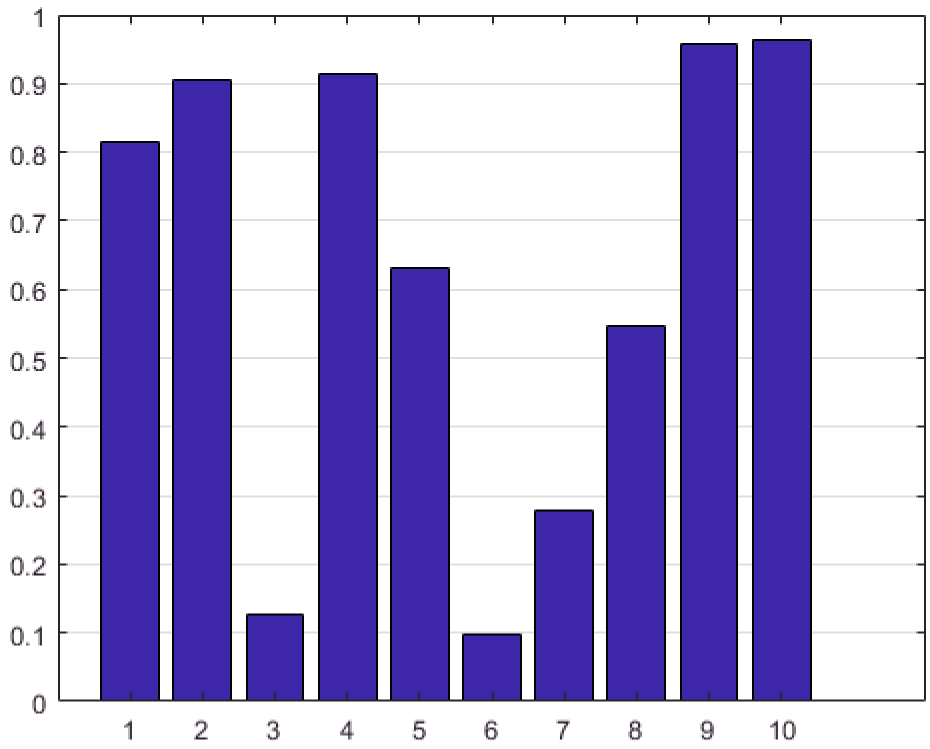


Display Grid Lines in Specific Direction

Display the grid lines in a particular direction by accessing the `Axes` object and setting the `XGrid`, `YGrid`, and `ZGrid` properties. Set these properties to either `'on'` or `'off'`.

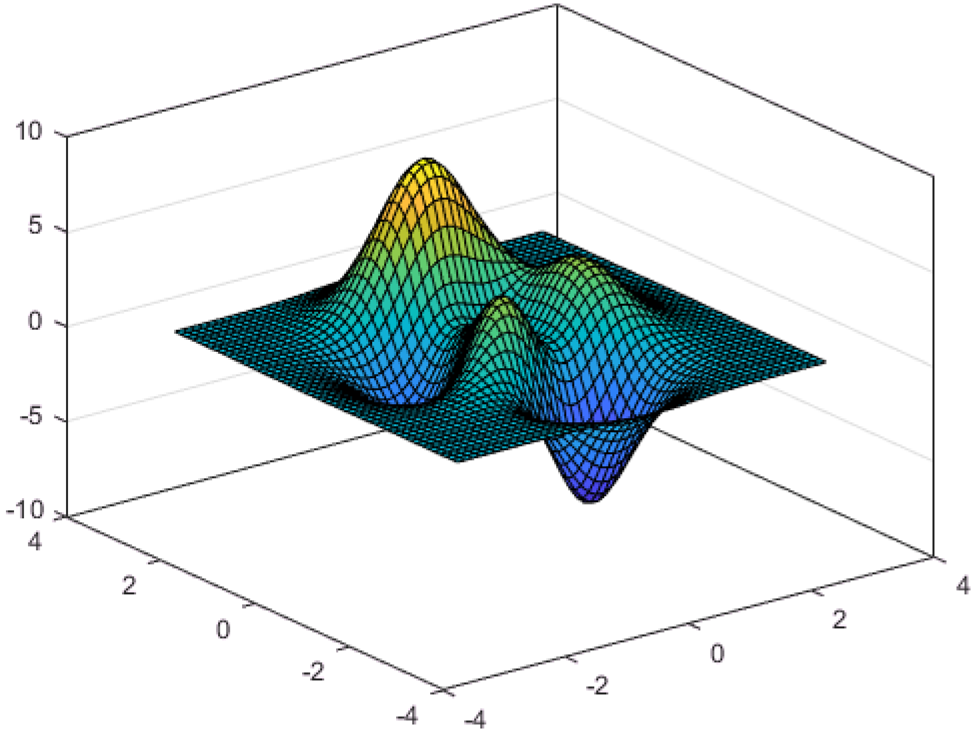
Create a 2-D plot and display the grid lines only in the *y* direction.

```
y = rand(10,1);
bar(y)
ax = gca;
ax.XGrid = 'off';
ax.YGrid = 'on';
```



Create a 3-D plot and display the grid lines only in the *z* direction. Use the `box on` command to show the box outline around the axes.

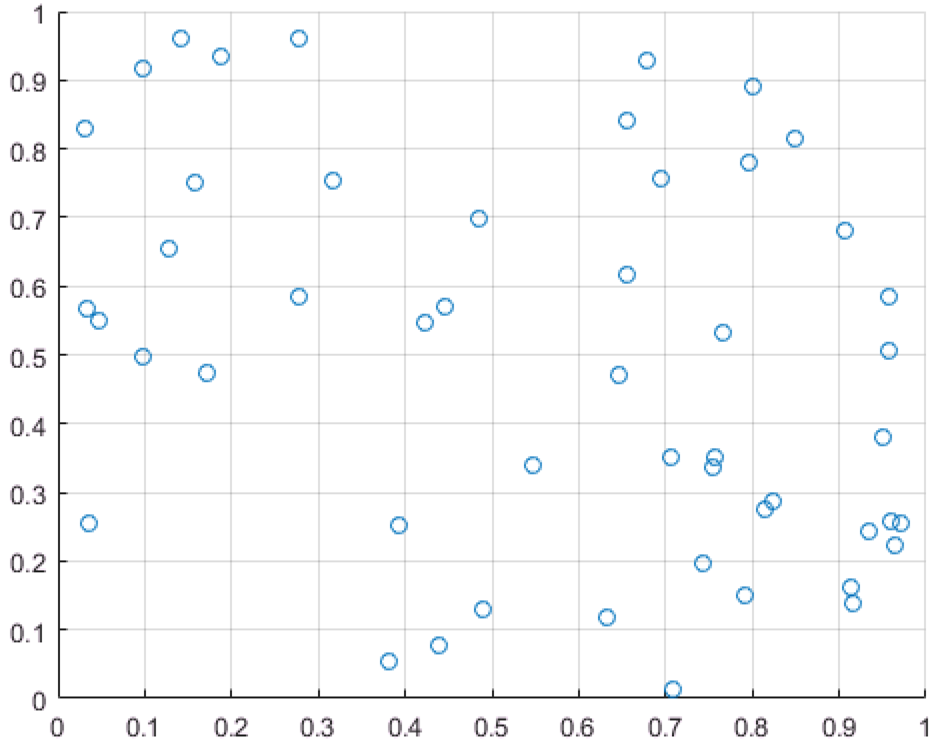
```
[X,Y,Z] = peaks;
surf(X,Y,Z)
box on
ax = gca;
ax.ZGrid = 'on';
ax.XGrid = 'off';
ax.YGrid = 'off';
```



Edit Grid Line Placement

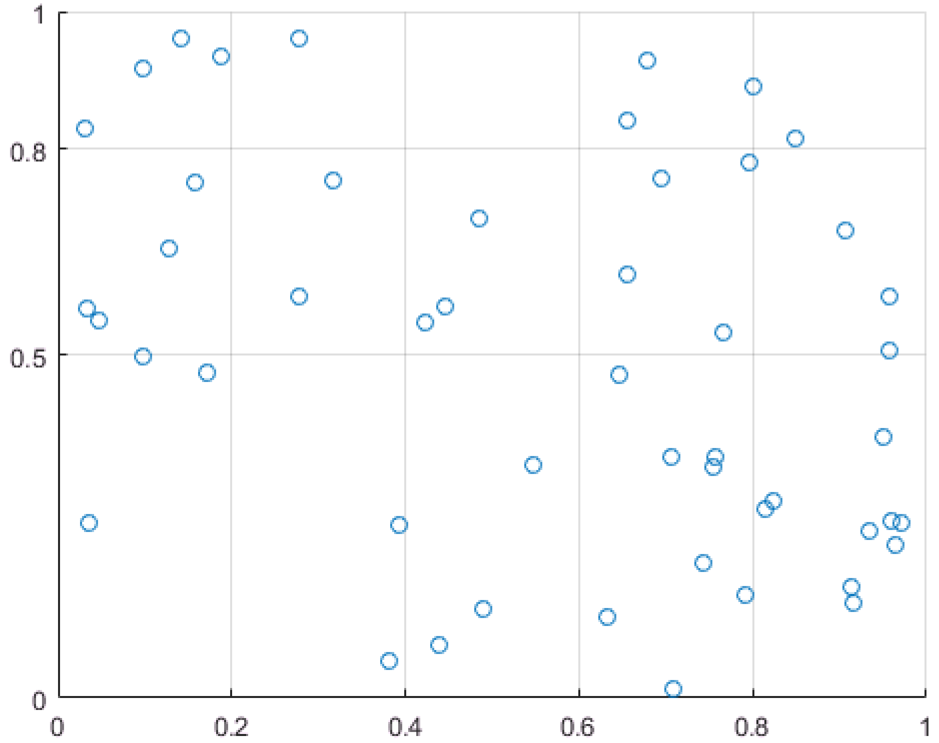
Create a scatter plot of random data and display the grid lines.

```
x = rand(50,1);
y = rand(50,1);
scatter(x,y)
grid on
```



Grid lines appear at the tick mark locations. Edit the placement of the grid lines by changing the tick mark locations.

```
xticks(0:0.2:1)
yticks([0 0.5 0.8 1])
```

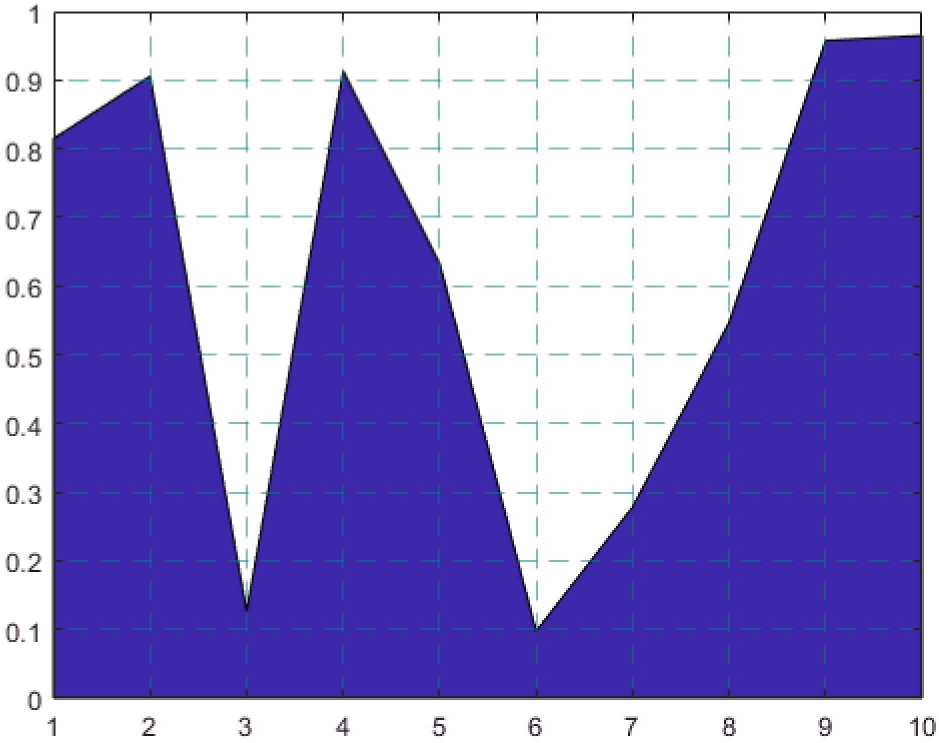


Modify Visual Appearance of Grid Lines

Change the color, line style, and transparency of grid lines for an area plot. Modify the appearance of the grid lines by accessing the `Axes` object. Then set properties related to the grid, such as the `GridColor`, `GridLineStyle`, and `GridAlpha` properties. Display the grid lines on top of the plot by setting the `Layer` property.

```
y = rand(10,1);
area(y)
grid on

ax = gca;
```

```
ax.GridColor = [0 .5 .5];
ax.GridLineStyle = '--';
ax.GridAlpha = 0.5;
ax.Layer = 'top';
```



See Also

Functions

`grid` | `xlim` | `xticks` | `yticks` | `zticks`

Properties

Axes Properties

Related Examples

- “Specify Axis Tick Values and Labels” on page 2-28
- “Add Title, Axis Labels, and Legend to Graph” on page 2-13
- “Specify Axis Limits” on page 2-19

Combine Multiple Plots

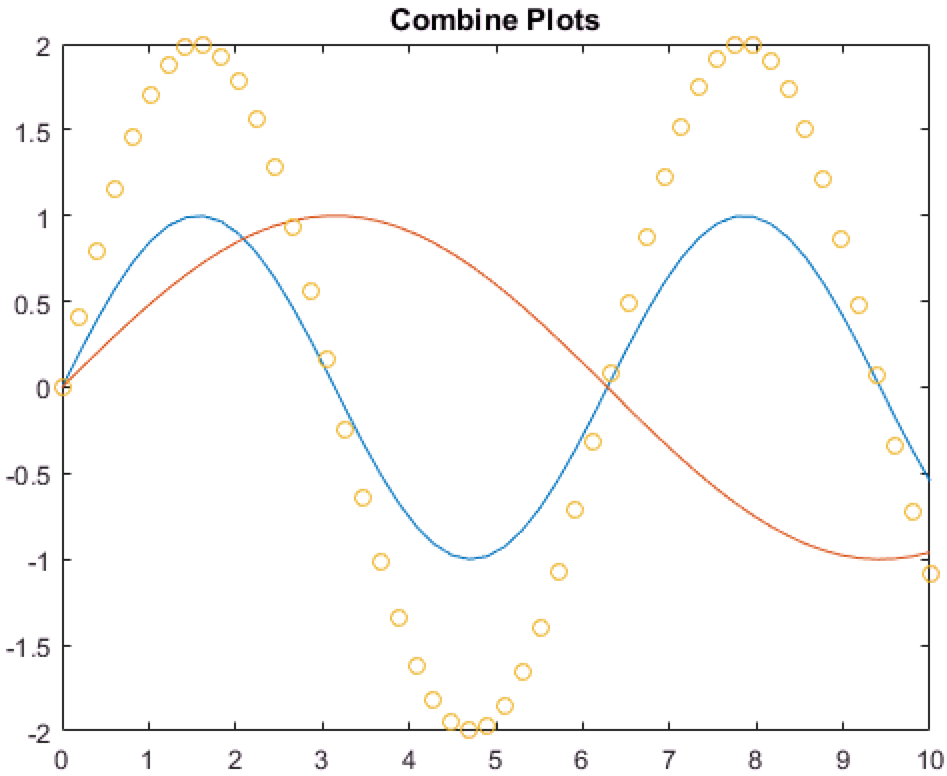
In this section...
"Combine Plots in Same Axes" on page 2-48
"Create Multiple Axes in Figure Using Subplots" on page 2-49
"Add Super Title to Figure with Subplots" on page 2-51

You can combine plots in several ways. Combine plots in the same axes, or create multiple axes in a figure.

Combine Plots in Same Axes

Use the `hold on` command to combine multiple plots in the same axes. For example, plot two lines and a scatter plot. Then reset the hold state to off.

```
x = linspace(0,10,50);
y1 = sin(x);
figure
plot(x,y1)
title('Combine Plots')

hold on
y2 = sin(x/2);
plot(x,y2)

y3 = 2*sin(x);
scatter(x,y3)
hold off
```



When the hold state is on, the plots cycle through colors and lines styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. Also, new plots do not reset axes properties, such as the title or axis labels. However, the axes limits and tick values can adjust to accommodate new data.

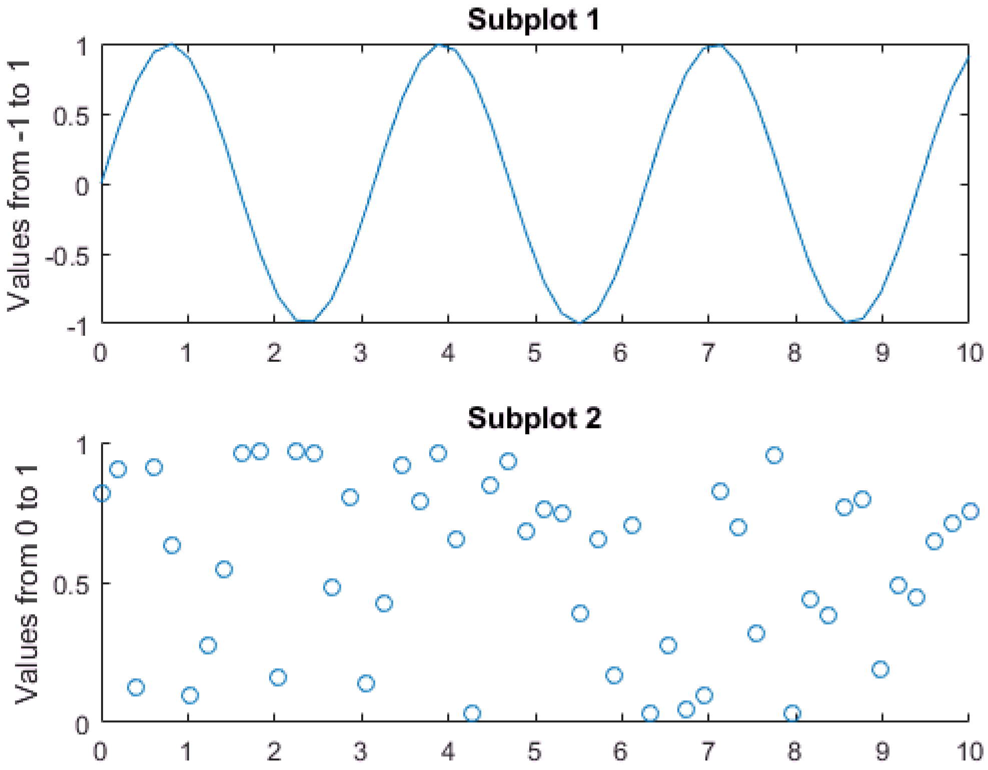
When the hold state is off, the next new plot clears existing plots and resets axes properties, such as the title.

Create Multiple Axes in Figure Using Subplots

Create multiple axes in a single figure using the `subplot` function, which divides the figure into a grid of subplots.

For example, create two stacked subplots and assign the Axes objects to the variables ax1 and ax2. Add a plot, title, and axis labels to each subplot. Specify the Axes objects as the first input arguments to each graphics function to ensure that the function targets the correct axes.

```
figure
ax1 = subplot(2,1,1);
x = linspace(0,10,50);
y1 = sin(2*x);
plot(ax1,x,y1)
title(ax1,'Subplot 1')
ylabel(ax1,'Values from -1 to 1')

ax2 = subplot(2,1,2);
y2 = rand(50,1);
scatter(ax2,x,y2)
title(ax2,'Subplot 2')
ylabel(ax2,'Values from 0 to 1')
```



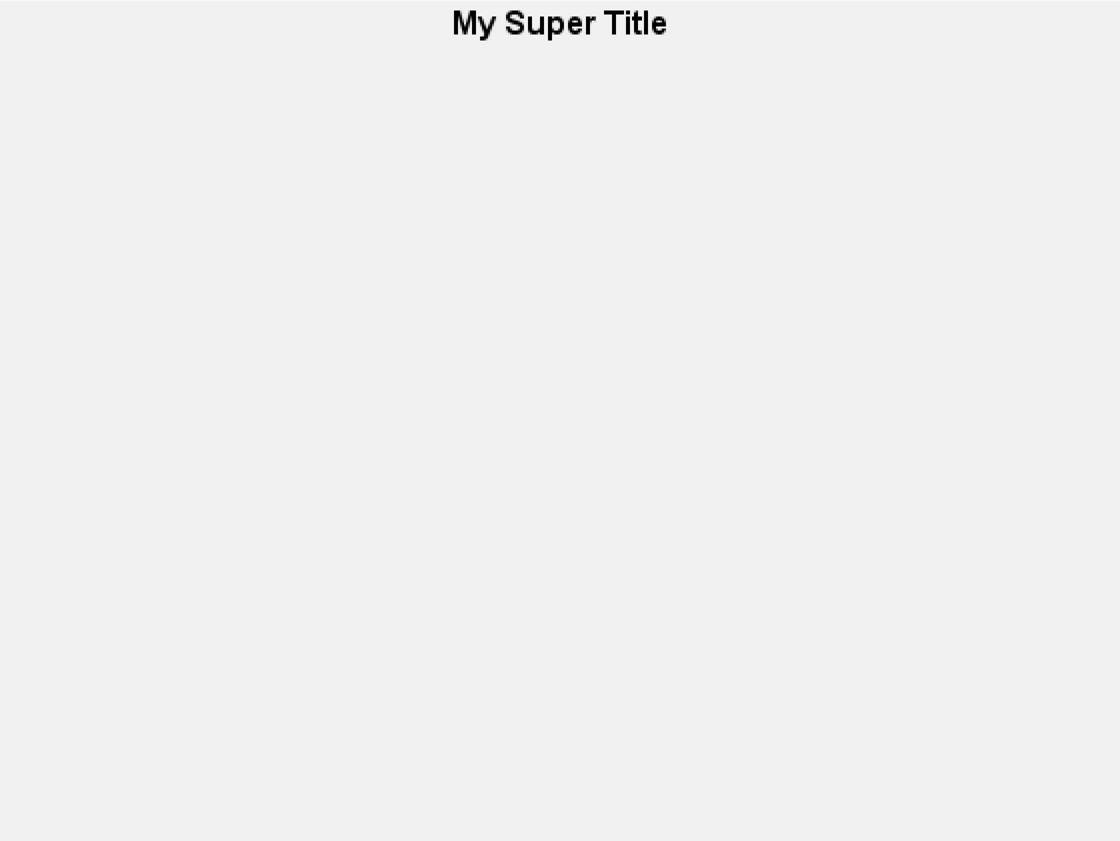
Add Super Title to Figure with Subplots

When you create a figure with subplots, you might want to add a title that applies to all the subplots. You can create the appearance of a super title by creating the subplots in a panel and adding a title to the panel.

Create a panel inside a figure. Specify a title for the panel and adjust some of the font properties.

```
f = figure;
p = uipanel('Parent',f,'BorderType','none');
p.Title = 'My Super Title';
```

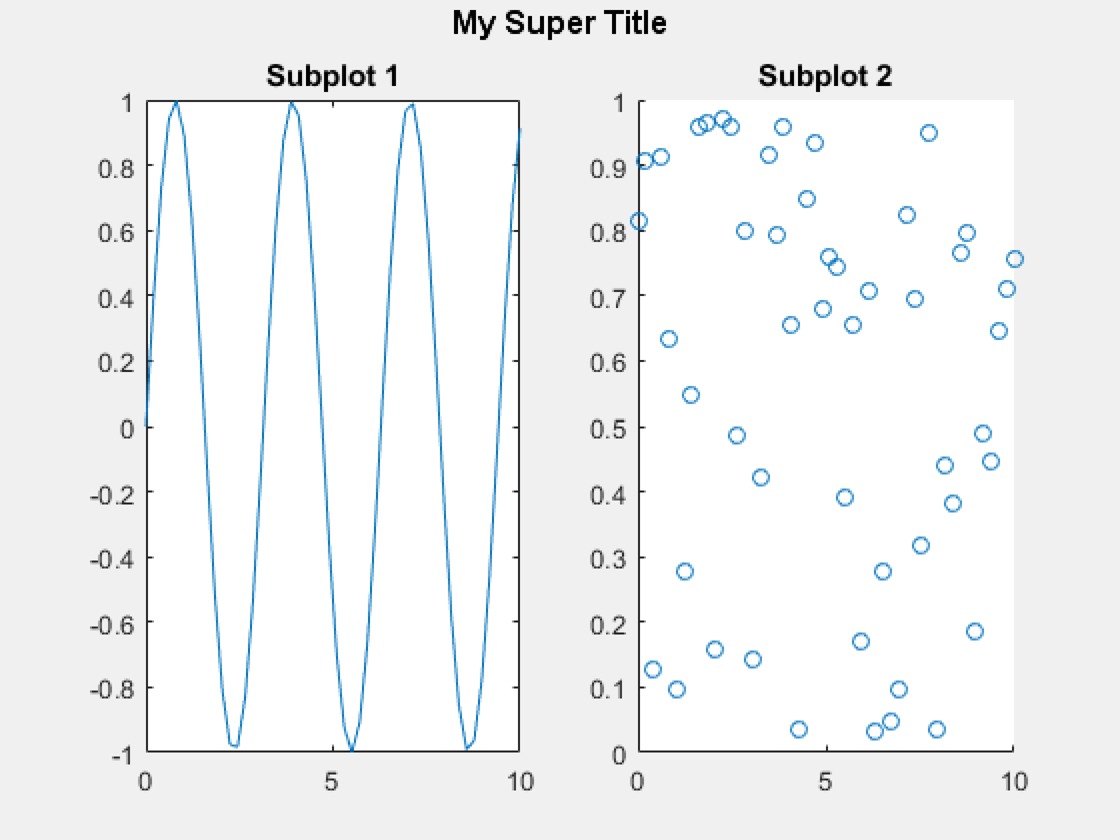
```
p.TitlePosition = 'centertop';
p.FontSize = 12;
p.FontWeight = 'bold';
```



Create two subplots in the panel by setting the `Parent` parameter to the panel object. Title each individual subplot.

```
subplot(1,2,1,'Parent',p)
x = linspace(0,10,50);
y1 = sin(2*x);
plot(x,y1)
title('Subplot 1')

subplot(1,2,2,'Parent',p)
```

```
y2 = rand(50,1);
scatter(x,y2)
title('Subplot 2')
```



See Also

Functions

`figure` | `hold` | `subplot` | `title` | `uipanel`

Related Examples

- "Create Chart with Two *y*-Axes" on page 2-55

- “Specify Axis Tick Values and Labels” on page 2-28

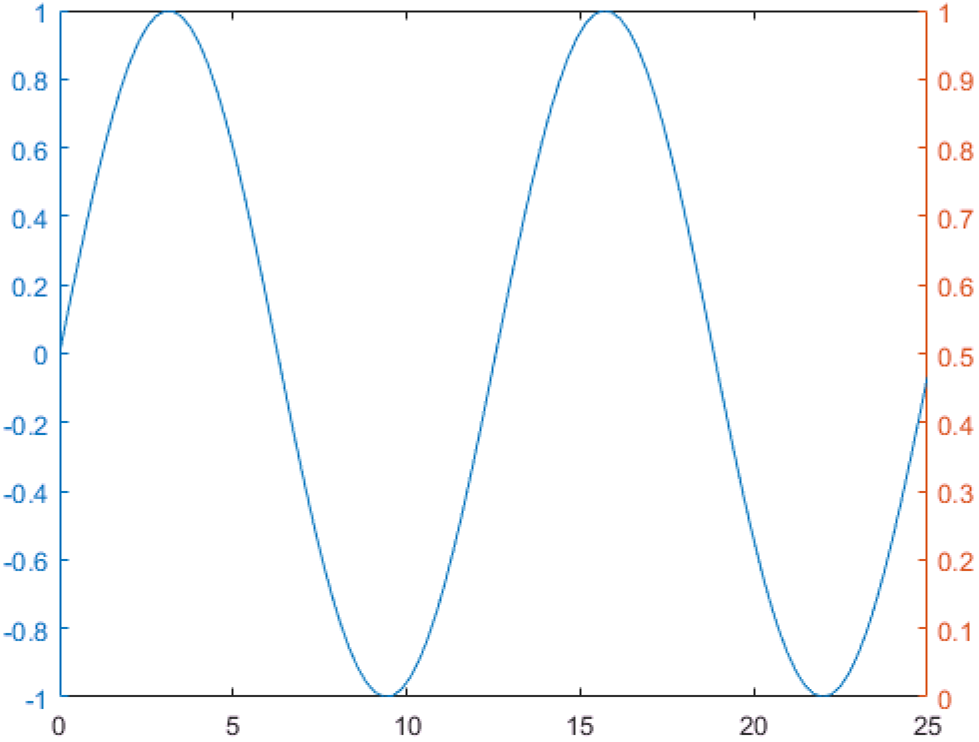
Create Chart with Two *y*-Axes

This example shows how to create a chart with *y*-axes on the left and right sides using the `yyaxis` function. It also shows how to label each axis, combine multiple plots, and clear the plots associated with one or both of the sides.

Plot Data Against Left *y*-Axis

Create axes with a *y*-axis on the left and right sides. The `yyaxis left` command creates the axes and activates the left side. Subsequent graphics functions, such as `plot`, target the active side. Plot data against the left *y*-axis.

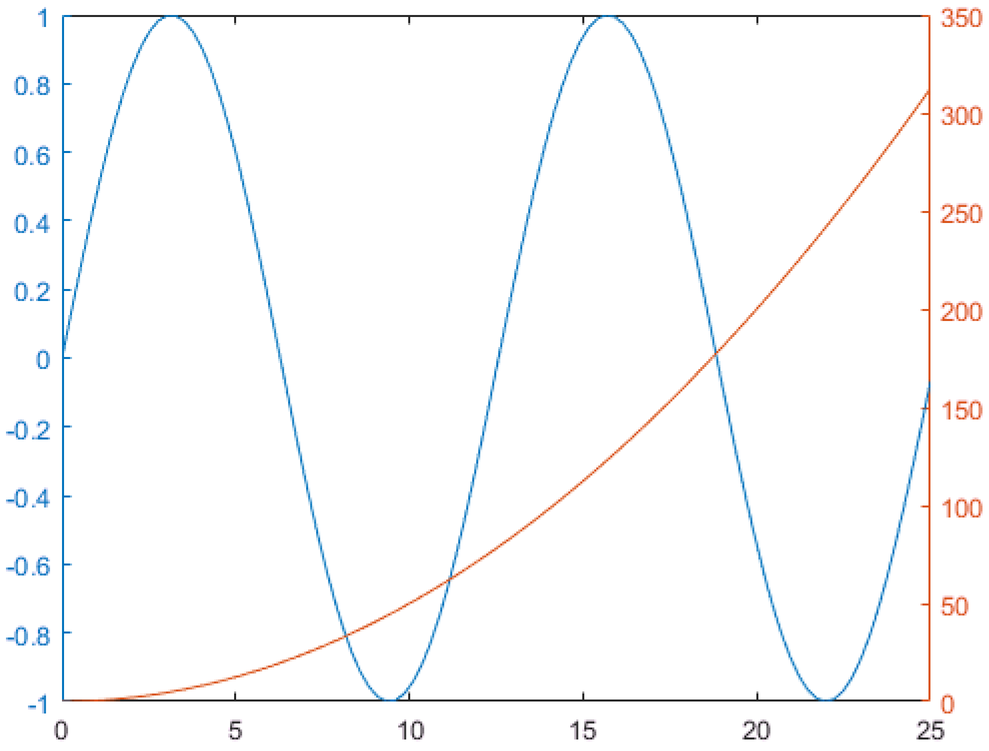
```
x = linspace(0,25);
y = sin(x/2);
yyaxis left
plot(x,y);
```



Plot Data Against Right *y*-Axis

Activate the right side using `yyaxis right`. Then plot a set of data against the right *y*-axis.

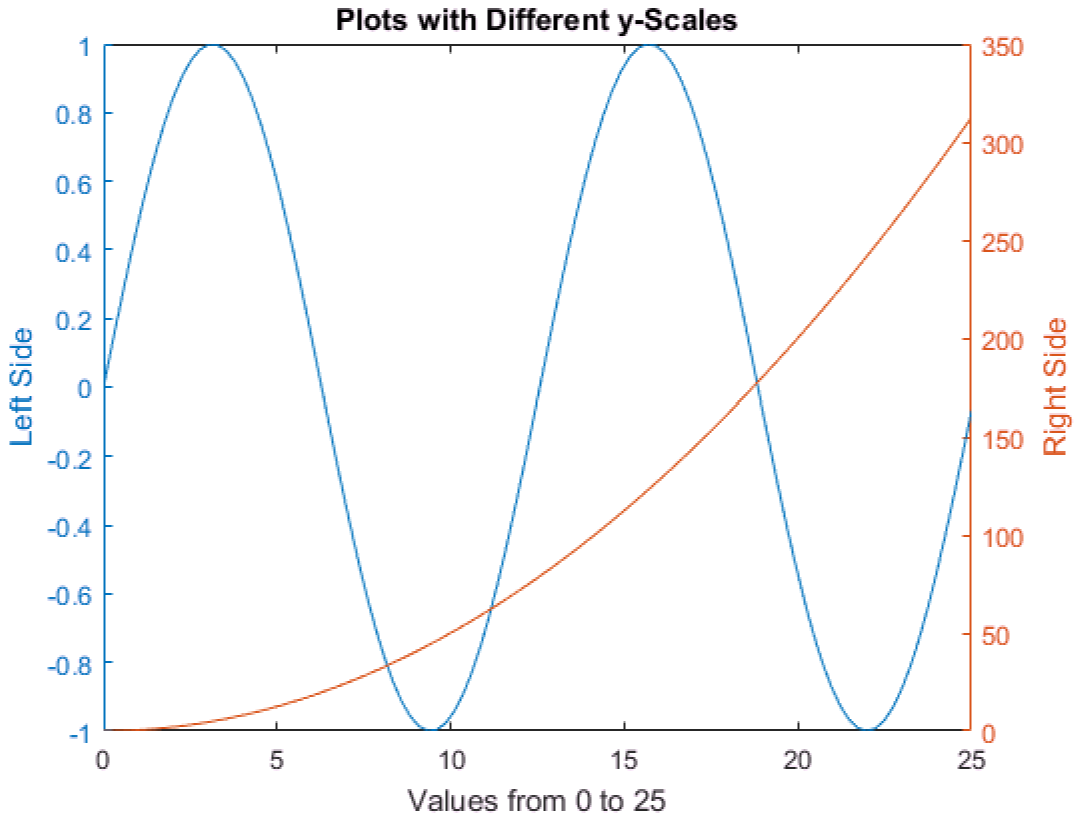
```
r = x.^2/2;
yyaxis right
plot(x,r);
```



Add Title and Axis Labels

Control which side of the axes is active using the `yyaxis left` and `yyaxis right` commands. Then, add a title and axis labels.

```
yyaxis left
title('Plots with Different y-Scales')
xlabel('Values from 0 to 25')
ylabel('Left Side')

yyaxis right
ylabel('Right Side')
```

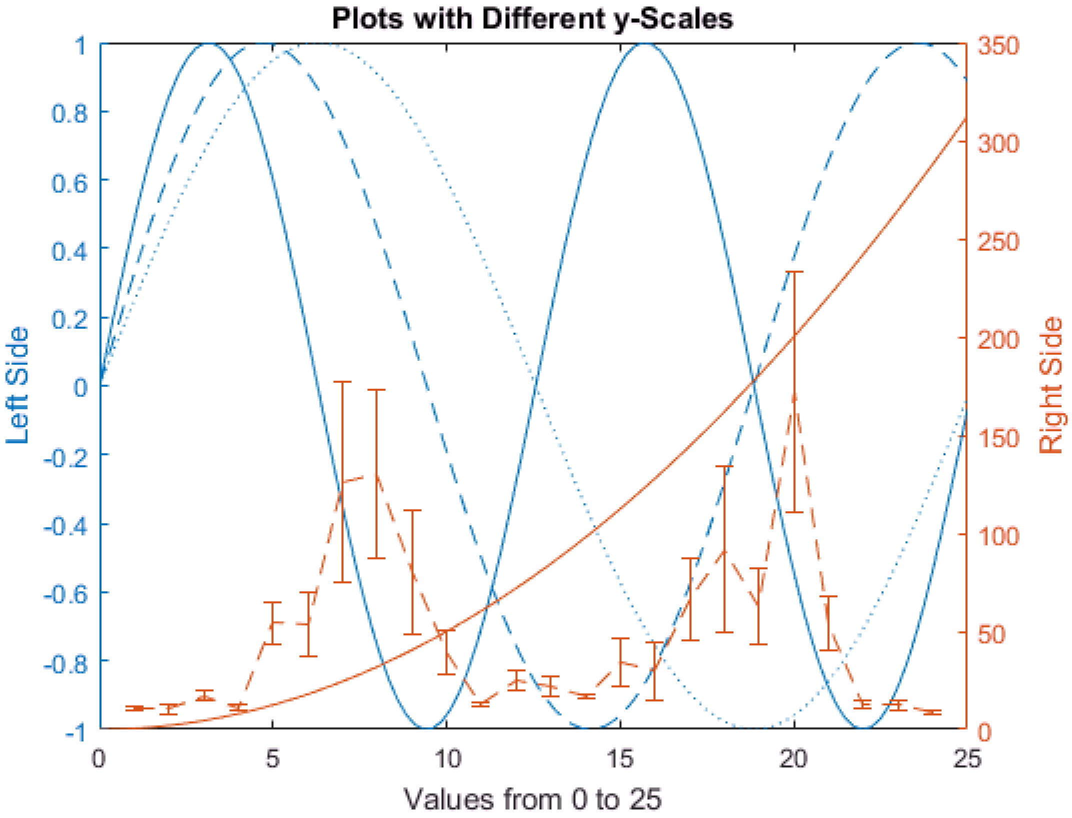


Plot Additional Data Against Each Side

Add two more lines to the left side using the `hold on` command. Add an errorbar to the right side. The new plots use the same color as the corresponding *y*-axis and cycle through the line style order. The `hold on` command affects both the left and right sides.

```
hold on

yyaxis left
y2 = sin(x/3);
plot(x,y2);
y3 = sin(x/4);
plot(x,y3);
```

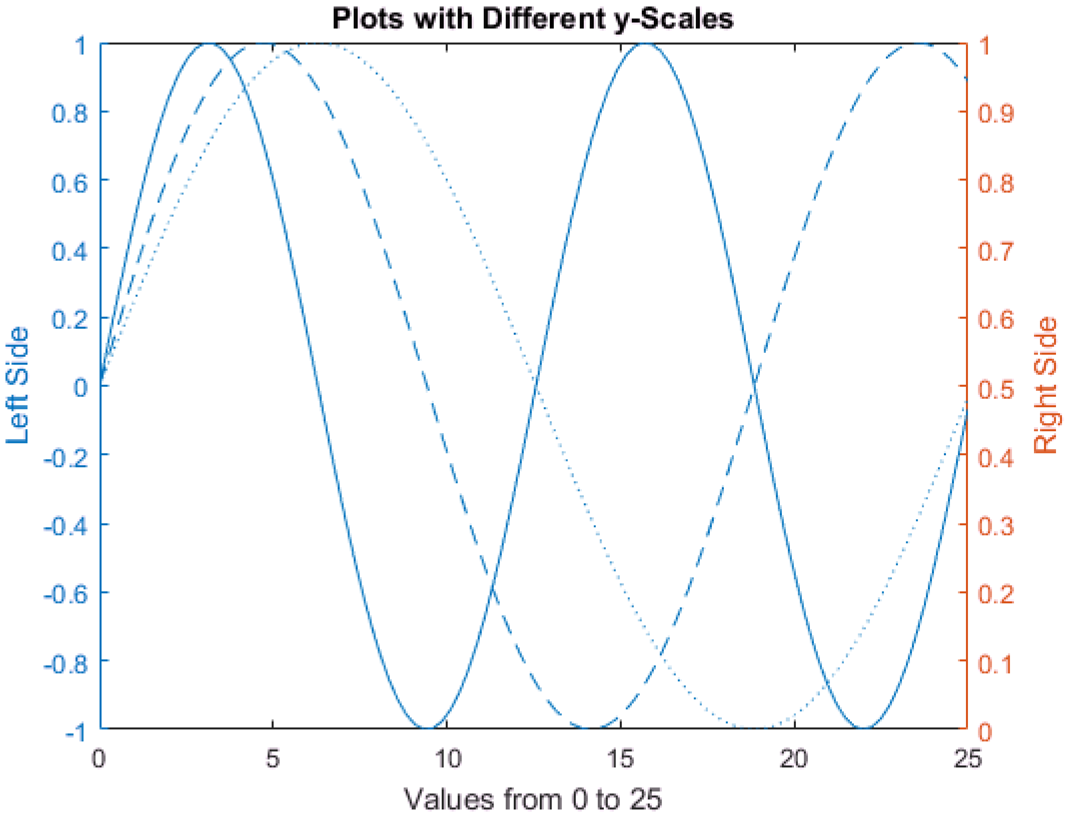
```
yyaxis right
load count.dat;
m = mean(count,2);
e = std(count,1,2);
errorbar(m,e)

hold off
```



Clear One Side of Axes

Clear the data from the right side of the axes by first making it active, and then using the `cla` command.

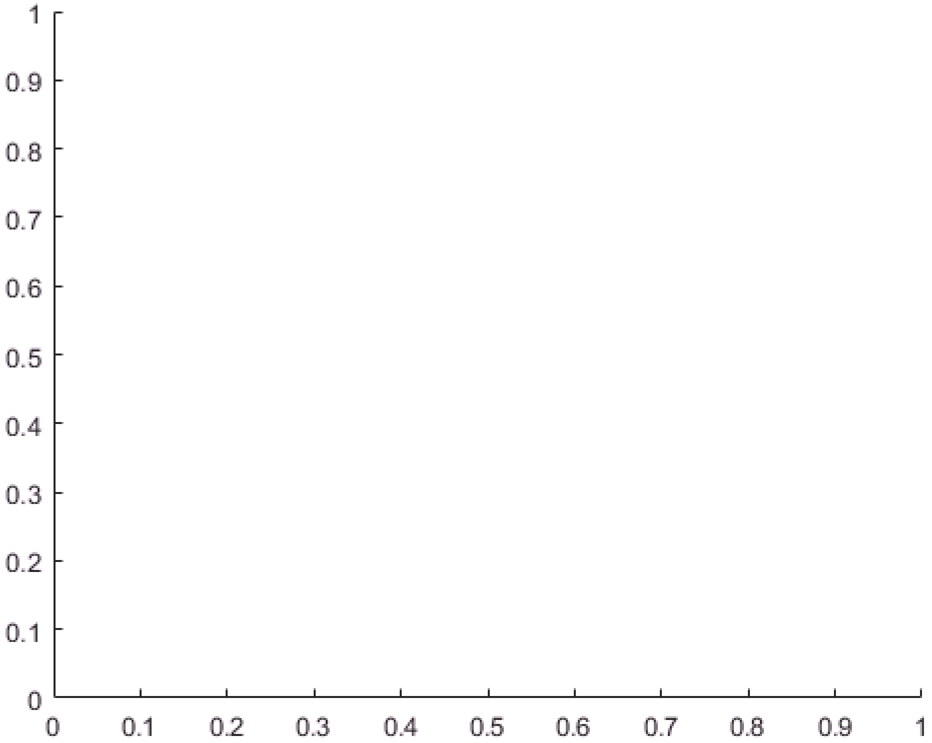
```
yyaxis right
cla
```



Clear Axes and Remove Right *y*-Axis

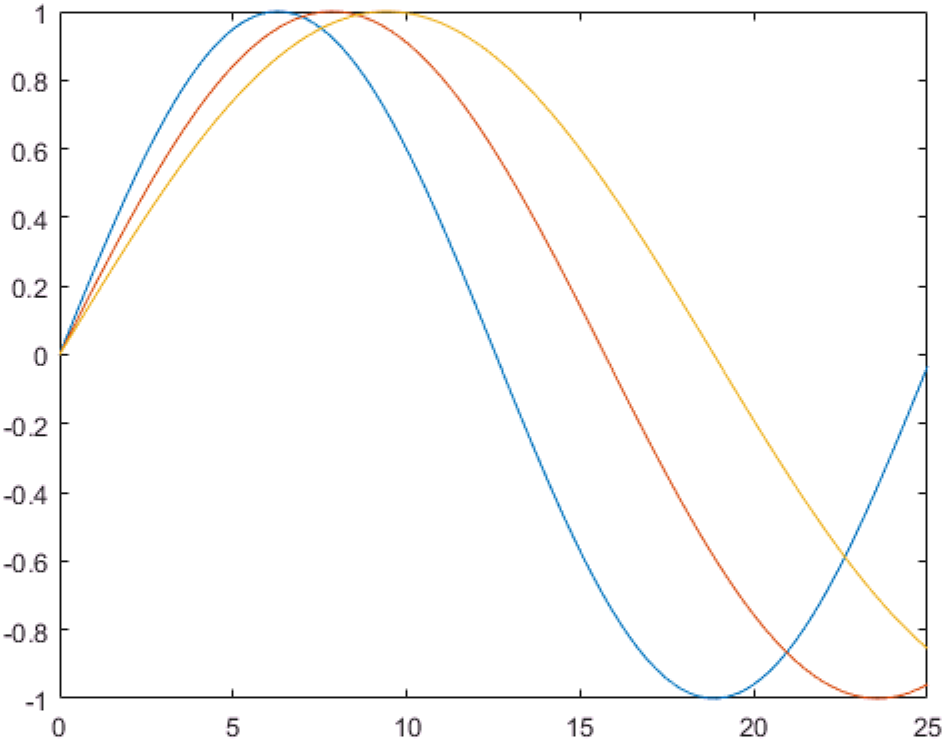
Clear the entire axes and remove the right *y*-axis using `cla reset`.

```
cla reset
```



Now when you create a plot, it only has one *y*-axis. For example, plot three lines against the single y-axis.

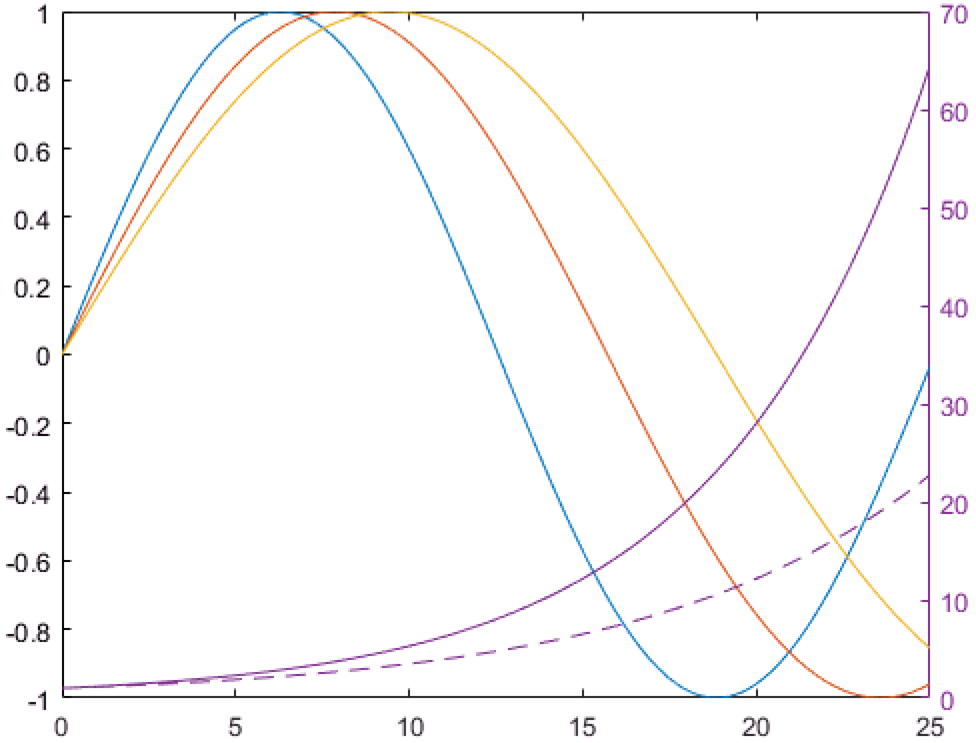
```
xx = linspace(0,25);
yy1 = sin(x/4);
yy2 = sin(x/5);
yy3 = sin(x/6);
plot(xx,yy1,xx,yy2,xx,yy3)
```



Add Second *y*-Axis to Existing Chart

Add a second *y*-axis to an existing chart using `yyaxis`. The existing plots and the left *y*-axis do not change colors. The right *y*-axis uses the next color in the axes color order. New plots added to the axes use the same color as the corresponding *y*-axis.

```
yyaxis right
rr1 = exp(xx/6);
rr2 = exp(xx/8);
plot(xx,rr1,xx,rr2)
```



See Also

Functions

`cla` | `hold` | `plot` | `title` | `xlabel` | `ylabel` | `yyaxis`

Related Examples

- "Modify Properties of Charts with Two *y*-Axes" on page 2-64
- "Combine Multiple Plots" on page 2-48

Modify Properties of Charts with Two *y*-Axes

In this section...

"Change Axes Properties" on page 2-64
"Change Ruler Properties" on page 2-66
"Specify Colors Using Default Color Order" on page 2-68

The `yyaxis` function creates an `Axes` object with a *y*-axis on the left and right sides. Axes properties related to the *y*-axis have two values. However, MATLAB gives access only to the value for the active side. For example, if the left side is active, then the `YDir` property of the `Axes` object contains the direction for the left *y*-axis. Similarly, if the right side is active, then the `YDir` property contains the direction for the right *y*-axis. An exception is that the `YAxis` property contains an array of two ruler objects (one for each *y*-axis).

You can change the appearance and behavior of a particular *y*-axis in either of these ways:

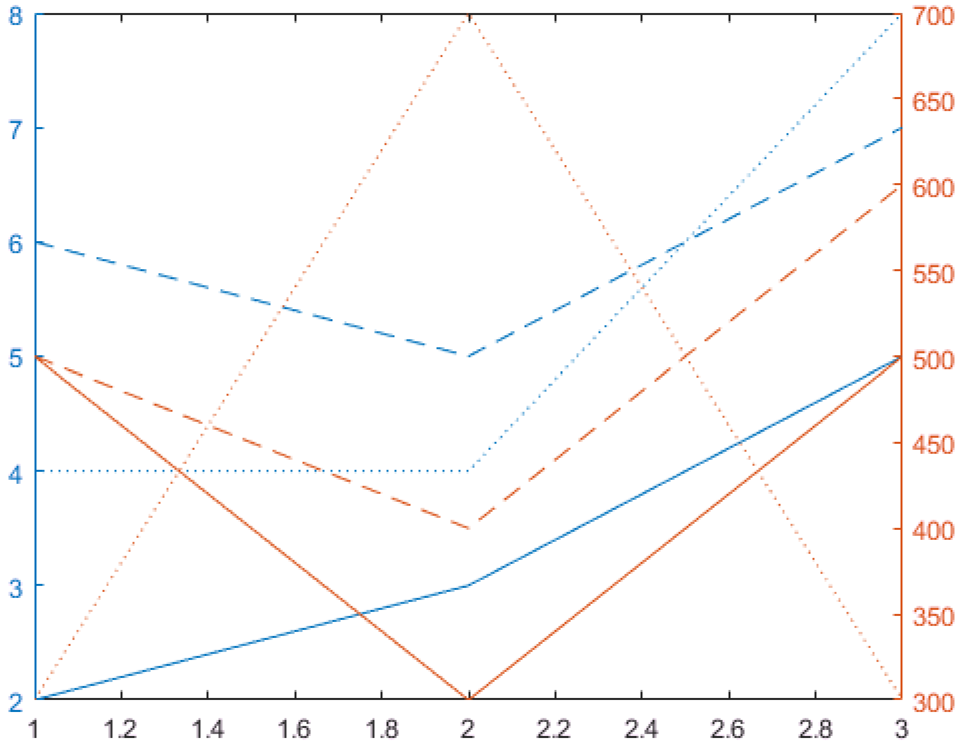
- Set the active side, and then change property values for the `Axes` object.
- Access the ruler objects through the `YAxis` property of the `Axes` object, and then change property values for the ruler object.

Change Axes Properties

Modify properties of a chart with two *y*-axes by setting `Axes` properties.

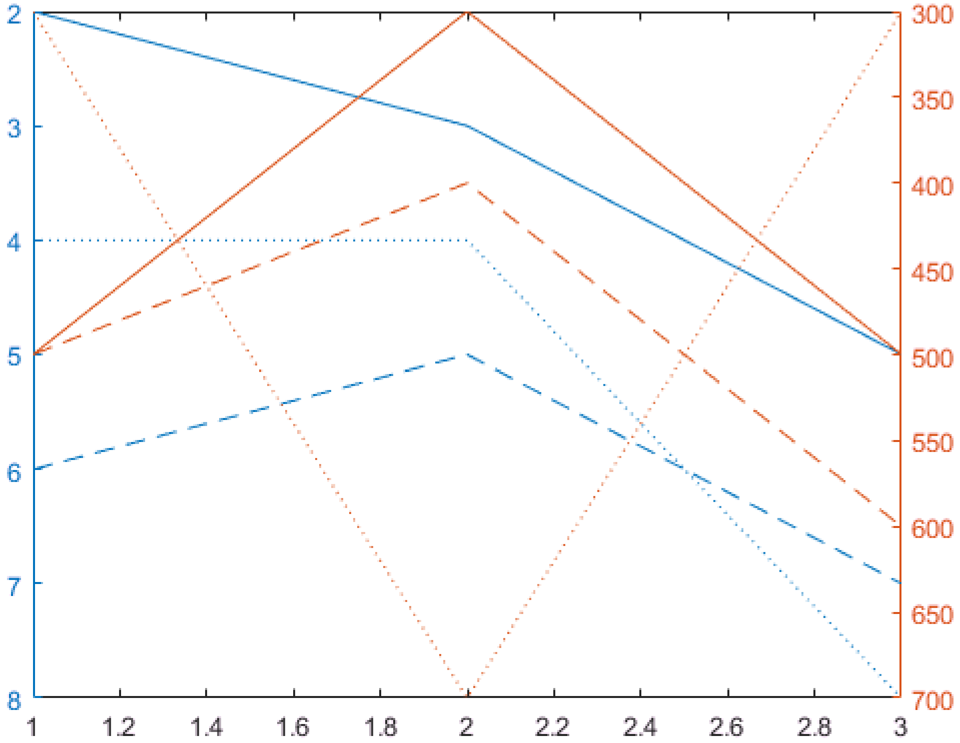
Create a chart with two *y*-axes and plot data.

```
x = [1 2 3];
y1 = [2 6 4; 3 5 4; 5 7 8];
y2 = 100*[5 5 3; 3 4 7; 5 6 3];
figure
yyaxis left
plot(x,y1)
yyaxis right
plot(x,y2)
```



Reverse the direction of increasing values along each *y*-axis. Use `yyaxis left` to activate the left side and set the direction for the left *y*-axis. Similarly, use `yyaxis right` to activate the right side. Then, set the direction for the right *y*-axis.

```
ax = gca;
yyaxis left
ax.YDir = 'reverse';
yyaxis right
ax.YDir = 'reverse';
```



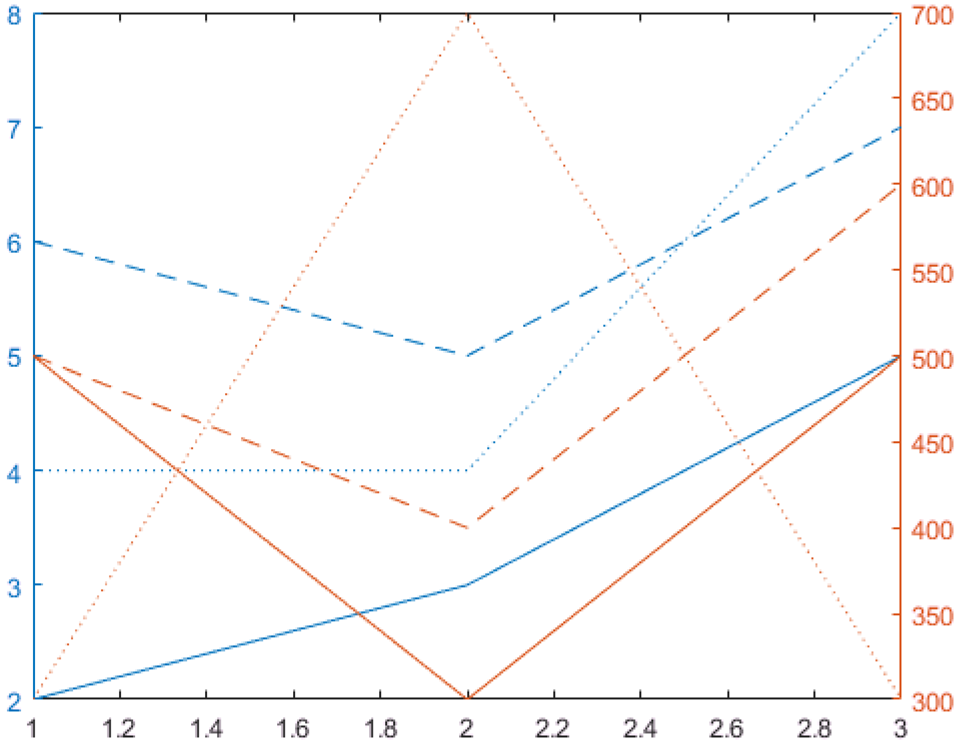
Change Ruler Properties

Modify properties of a chart with two *y*-axes by setting ruler properties.

Create a chart with two *y*-axes and plot data.

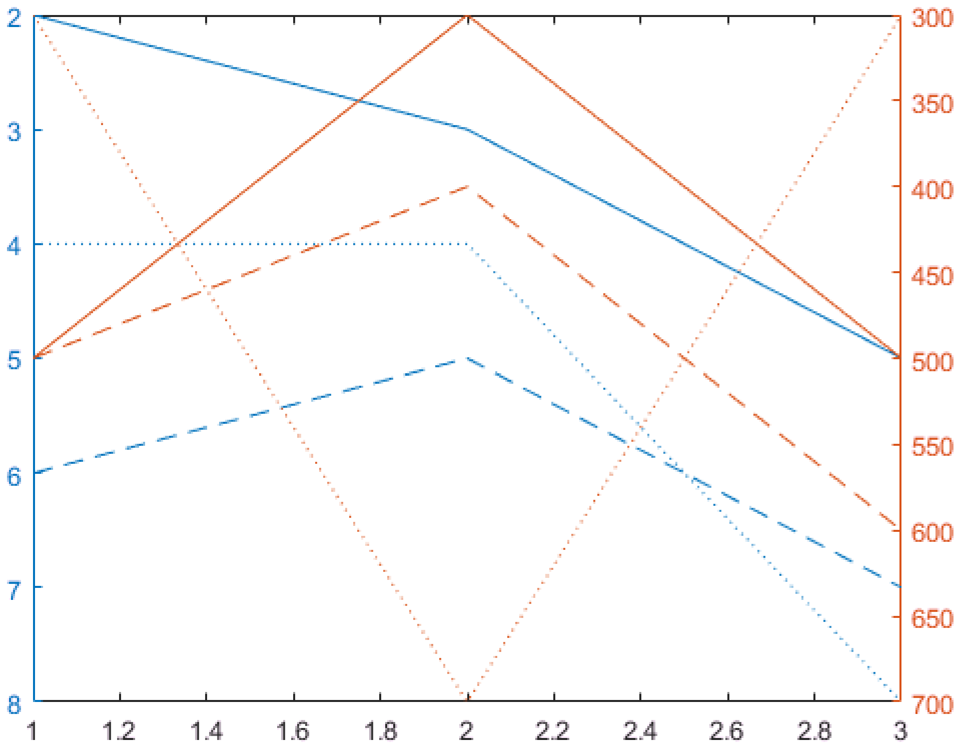
```
x = [1 2 3];
y1 = [2 6 4; 3 5 4; 5 7 8];
y2 = 100*[5 5 3; 3 4 7; 5 6 3];
figure
yyaxis left
```

```
plot(x,y1)
yyaxis right
plot(x,y2)
```



Reverse the direction of increasing values along each *y*-axis by setting properties of the ruler object associated with each axis. Use `ax.YAxis(1)` to refer to the ruler for the left side and `ax.YAxis(2)` to refer to the ruler for the right side.

```
ax = gca;
ax.YAxis(1).Direction = 'reverse';
ax.YAxis(2).Direction = 'reverse';
```



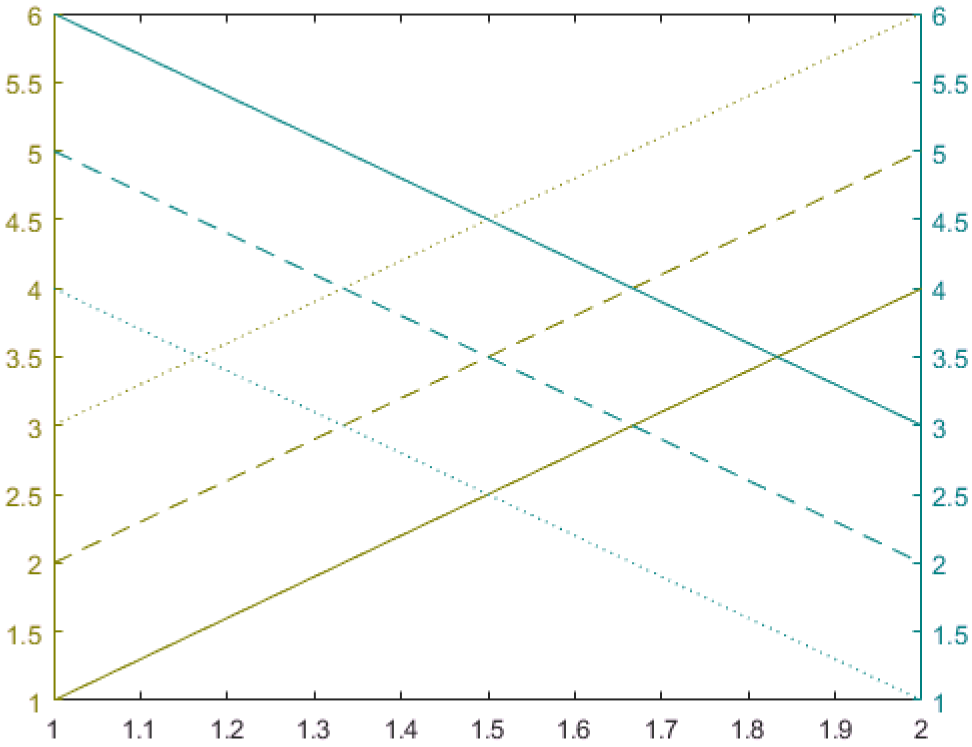
Specify Colors Using Default Color Order

Specify the colors for a chart with two *y*-axes by changing the default axes color order.

Create a figure. Define two RGB color values, one for the left side and one for the right side. Change the default axes color order to these two colors before creating the axes. Set the default value at the figure level so that the new colors affect only axes that are children of the figure `fig`. The new colors do not affect axes in other figures. Then create the chart.

```
fig = figure;
```

```
left_color = [.5 .5 0];
right_color = [0 .5 .5];
set(fig,'defaultAxesColorOrder',[left_color; right_color]);

y = [1 2 3; 4 5 6];
yyaxis left
plot(y)

z = [6 5 4; 3 2 1];
yyaxis right
plot(z)
```



See Also

Functions

`plot` | `yyaxis`

Properties

Axes Properties | Numeric Ruler Properties

Related Examples

- “Create Chart with Two *y*-Axes” on page 2-55
- “Default Property Values” on page 11-2

Plot Imaginary and Complex Data

Plot One Complex Input

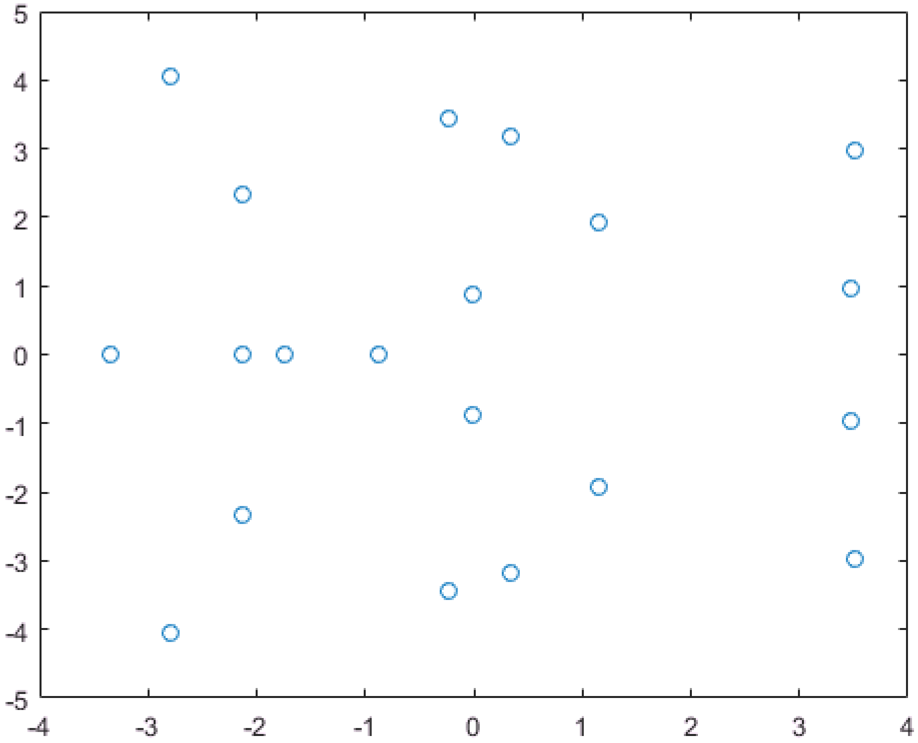
This example shows how to plot the imaginary part versus the real part of a complex vector, `z`. With complex inputs, `plot(z)` is equivalent to `plot(real(z),imag(z))`, where `real(z)` is the real part of `z` and `imag(z)` is the imaginary part of `z`.

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

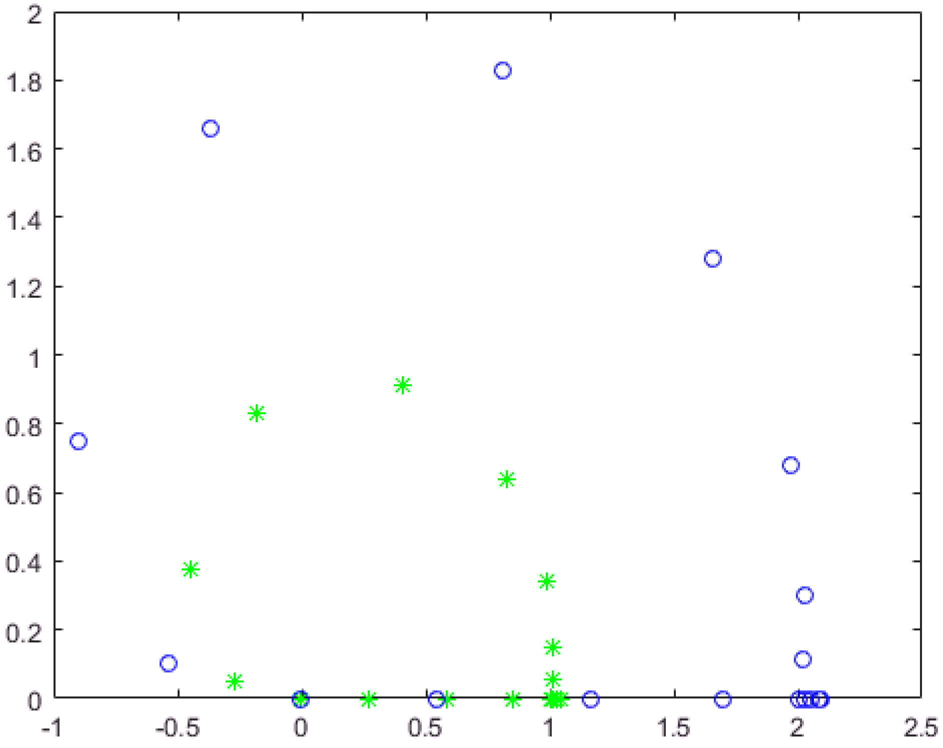
This example shows how to plot the imaginary part versus the real part of two complex vectors, `z1` and `z2`. If you pass multiple complex arguments to `plot`, such as `plot(z1,z2)`, then MATLAB® ignores the imaginary parts of the inputs and plots the real parts. To plot the real part versus the imaginary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

Create Heatmap from Tabular Data

Heatmaps are a way to visualize data using color. This example shows how to import a file into MATLAB® as a table and create a heatmap from the table columns. It also shows how to modify the appearance of the heatmap, such as setting the title and axis labels.

Import File as Table

Load the sample file `TemperatureData.csv`, which contains average daily temperatures from January 2015 through July 2016. Read the file into a table and display the first five rows.

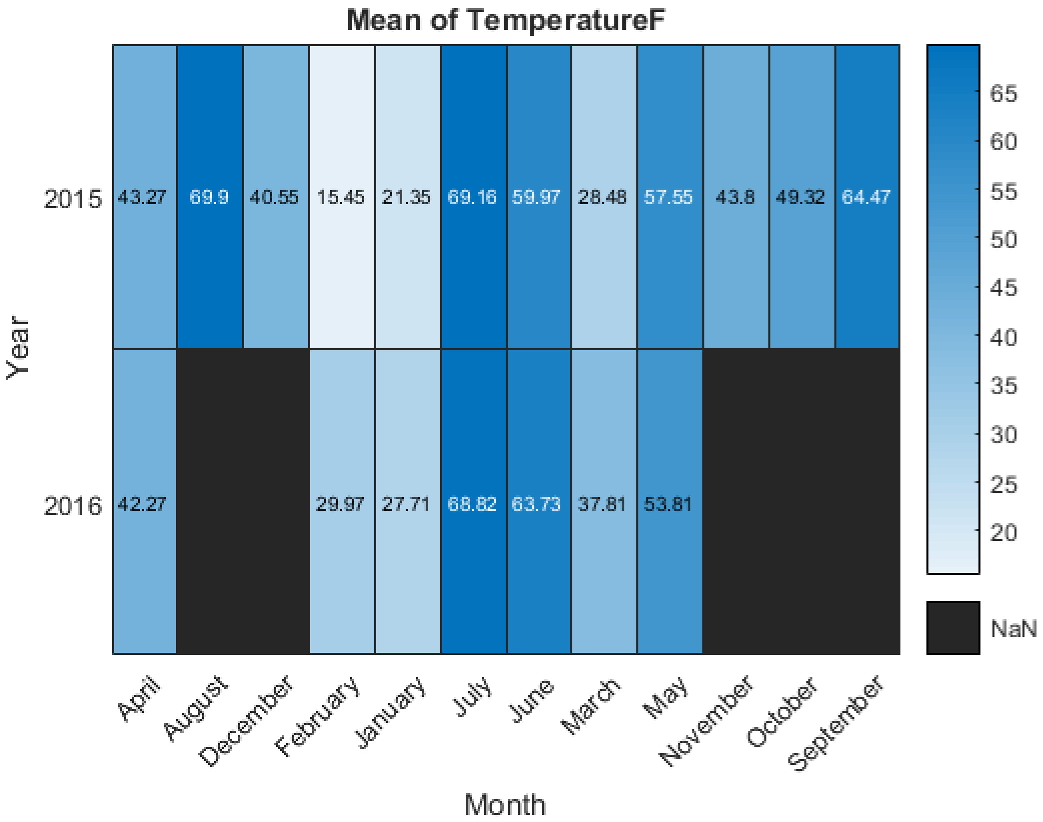
```
tbl = readtable(fullfile(matlabroot,'examples','graphics','TemperatureData.csv'));
head(tbl,5)
```

```
ans =

  5×4 table

    Year       Month        Day    TemperatureF
    ____    ___________    ___    ____________

    2015     'January'      1          23
    2015     'January'      2          31
    2015     'January'      3          25
    2015     'January'      4          39
    2015     'January'      5          29
```

Create Basic Heatmap

Create a heatmap that shows the months along the *x*-axis and years along the *y*-axis. Color the heatmap cells using the temperature data by setting the `ColorVariable` property. Assign the `HeatmapChart` object to the variable `h`. Use `h` to modify the chart after it is created.

```
h = heatmap(tbl,'Month','Year','ColorVariable','TemperatureF');
```



By default, MATLAB calculates the color data as the average temperature for each month. However, you can change the calculation method by setting the `ColorMethod` property.

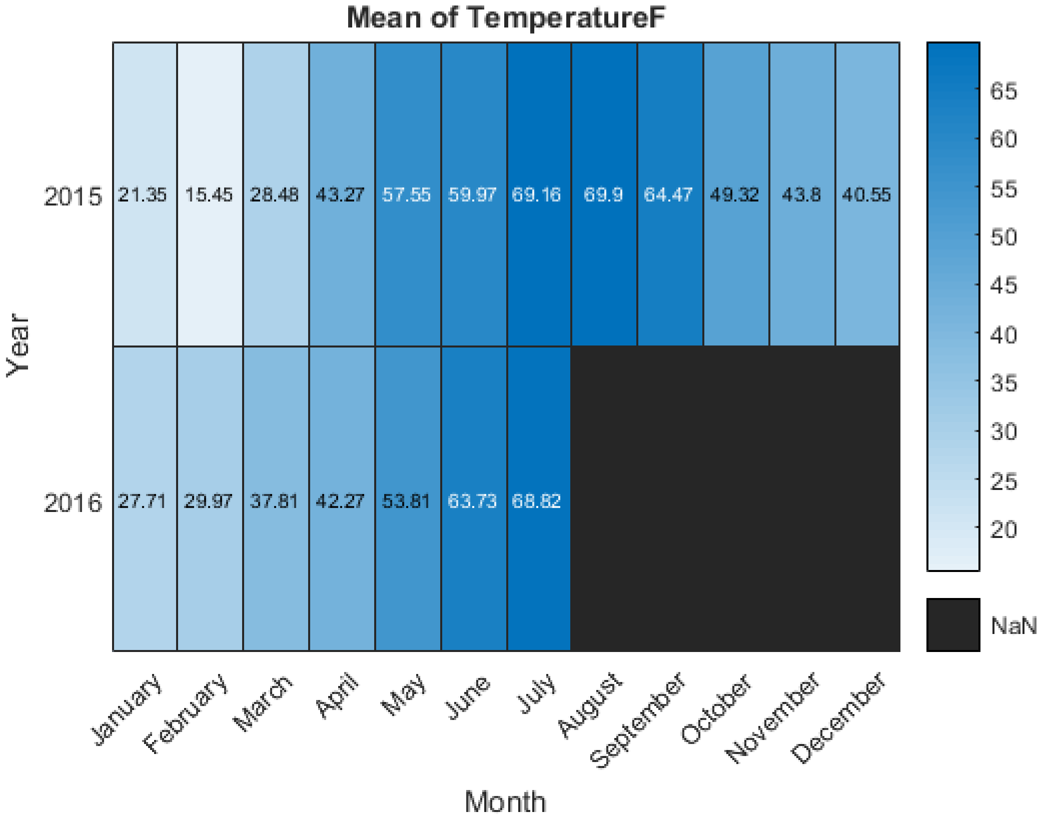
Reorder Values Along Axis

The values along an axis appear in alphabetical order. Reorder the months so that they appear in chronological order. You can customize the labels using categorical arrays.

First change the data in the `Month` column of the table from a cell array to a categorical array using the `categorical` function. Then use the `reordercats` function to reorder the categories.

```
h.SourceTable.Month = categorical(h.SourceTable.Month);
```

```
neworder = {'January','February','March','April','May','June','July',...
    'August','September','October','November','December'};
h.SourceTable.Month = reordercats(h.SourceTable.Month,neworder);
```



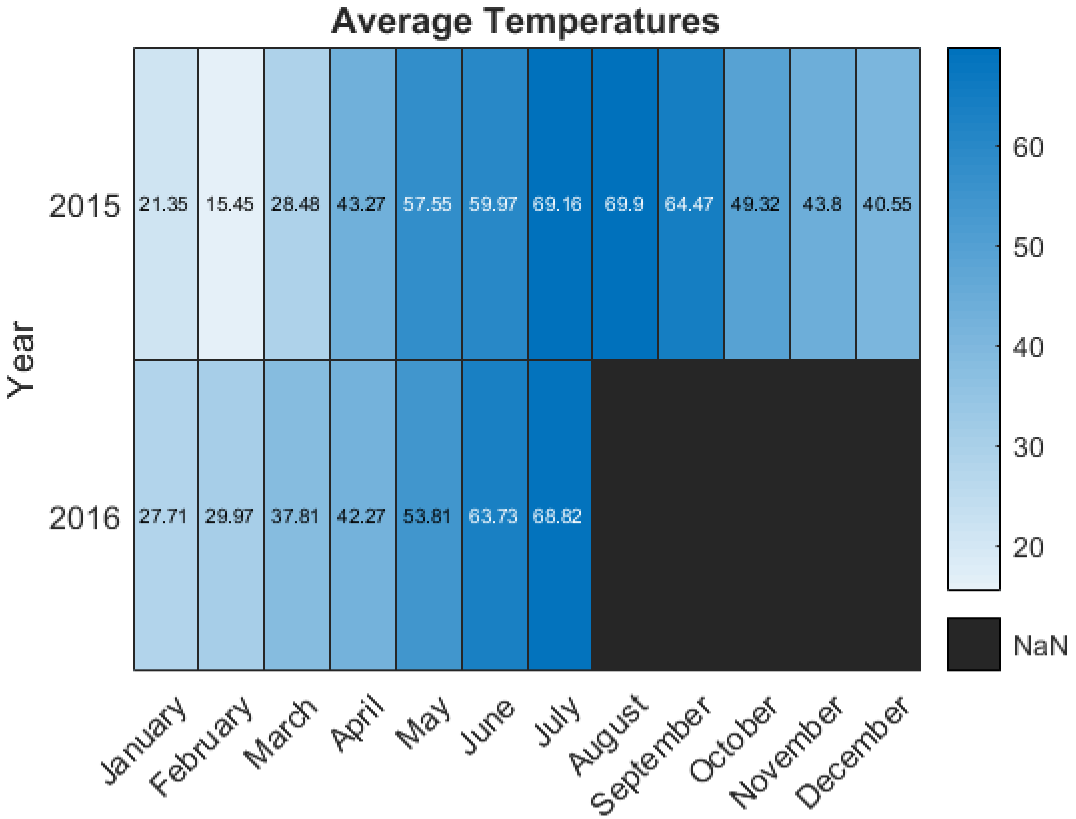
Similarly, you can add, remove, or rename the heatmap labels using the `addcats`, `removecats`, or `renamecats` functions for categorical arrays. View the current labels using the `categories` function.

You can apply these functions to the table in the workspace (`tbl`). Alternatively, you can apply them to the table stored in the `SourceTable` property of the `HeatmapChart` object (`h.SourceTable`). Applying them to the table stored in the `HeatmapChart` object avoids affecting the original data.

Modify Title and Axis Labels

When you create a heatmap using tabular data, the heatmap automatically generates a title and axis labels. Customize the title and axis labels by setting the `Title`, `XLabel`, and `YLabel` properties of the `HeatmapChart` object. For example, change the title and remove the *x*-axis label. Also, change the font size.

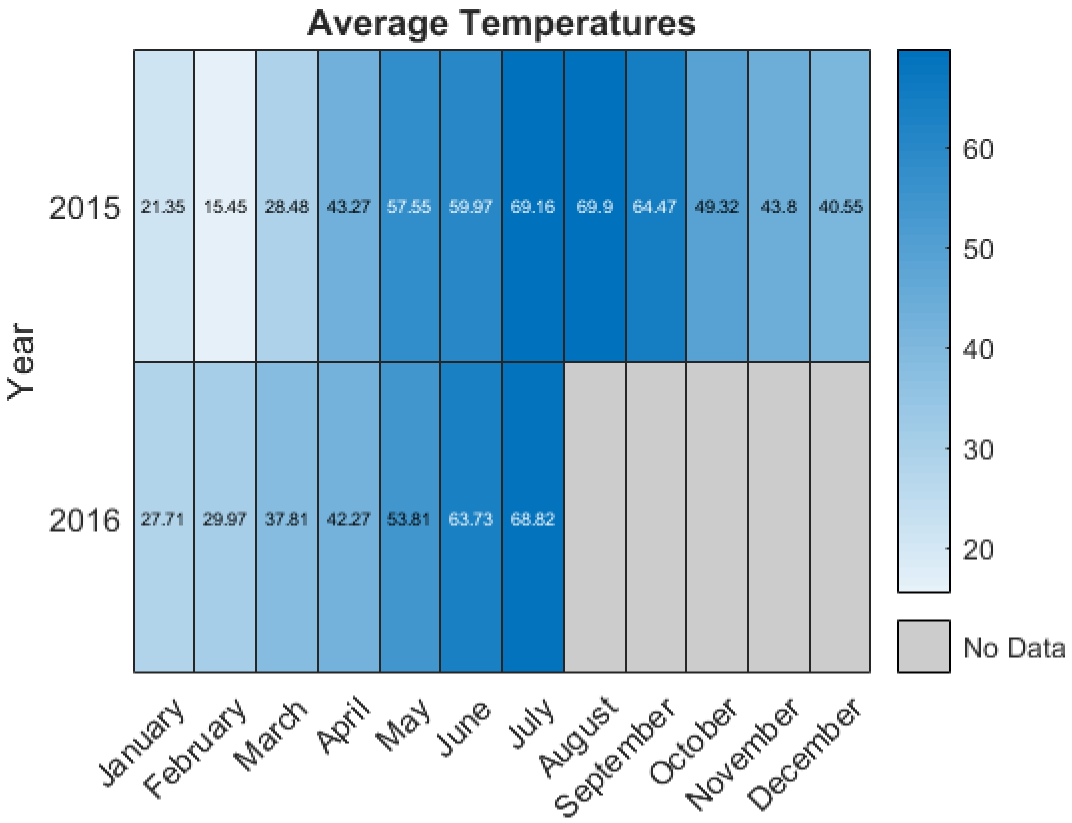
```
h.Title = 'Average Temperatures';
h.XLabel = '';
h.FontSize = 12;
```



Modify Appearance of Missing Data Cells

Since there is no data for August 2016 through December 2016, those cells appear as missing data. Modify the appearance of the missing data cells using the `MissingDataColor` and `MissingDataLabel` properties.

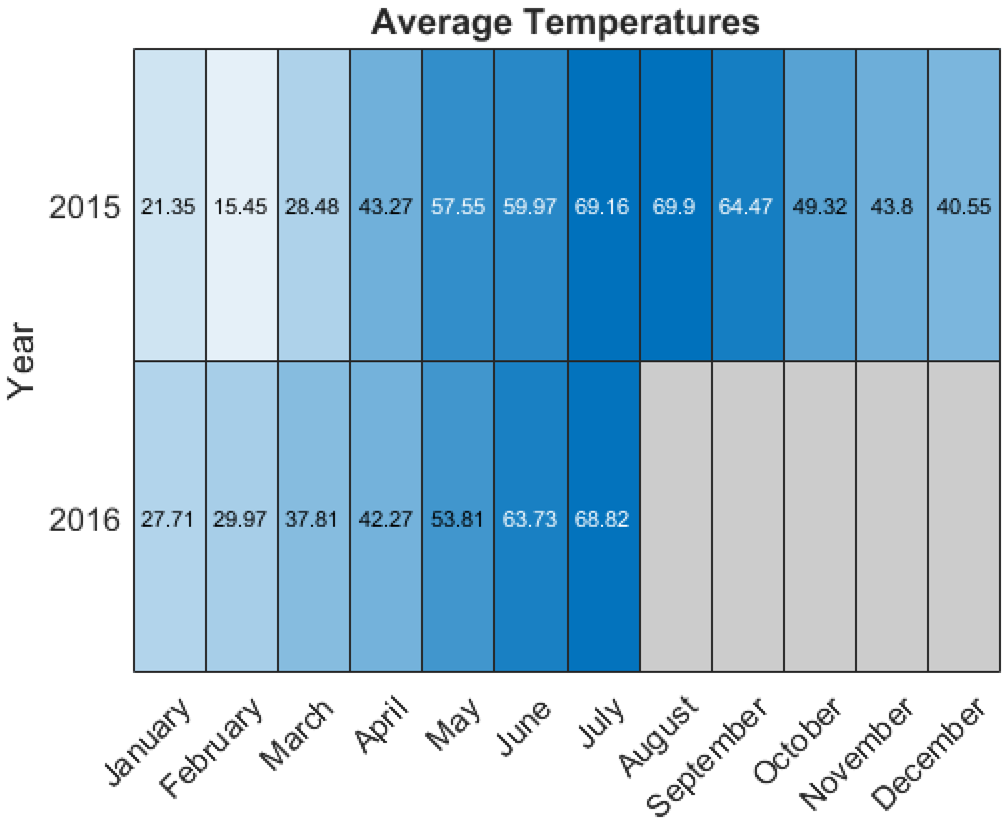
```
h.MissingDataColor = [0.8 0.8 0.8];
h.MissingDataLabel = 'No Data';
```



Remove Colorbar

Remove the colorbar by setting the `ColorbarVisible` property.

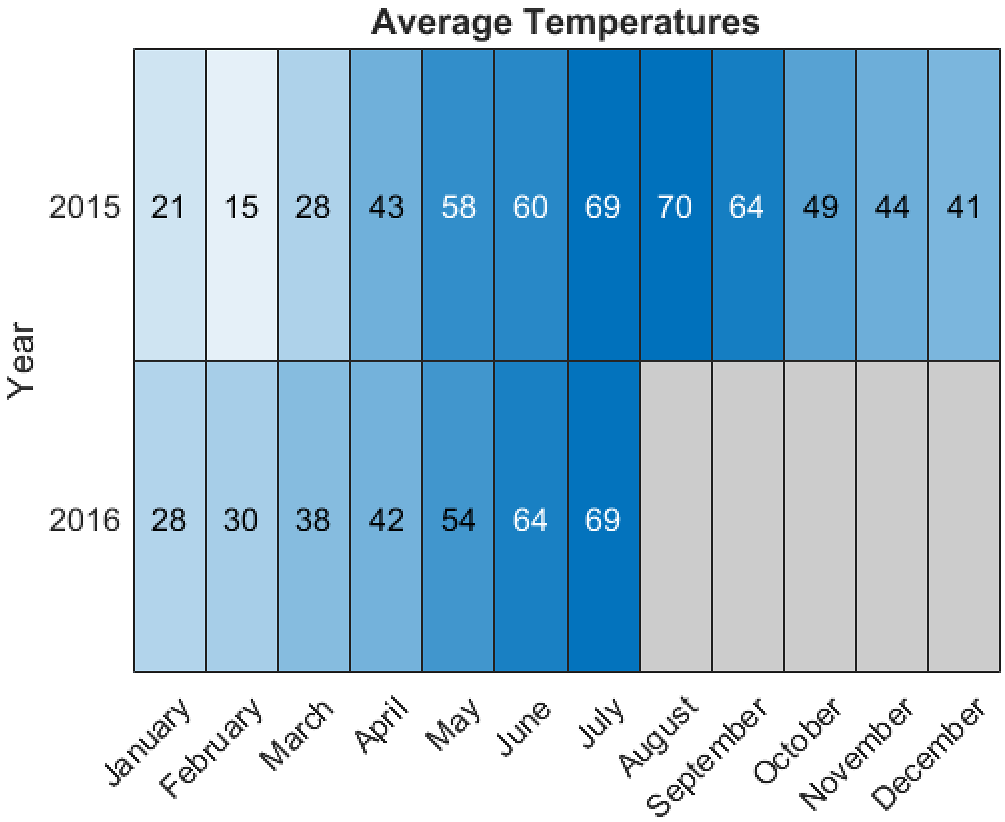
```
h.ColorbarVisible = 'off';
```



Format Cell Text

Customize the format of the text that appears in each cell by setting the `CellLabelFormat` property. For example, display the text with no decimal values.

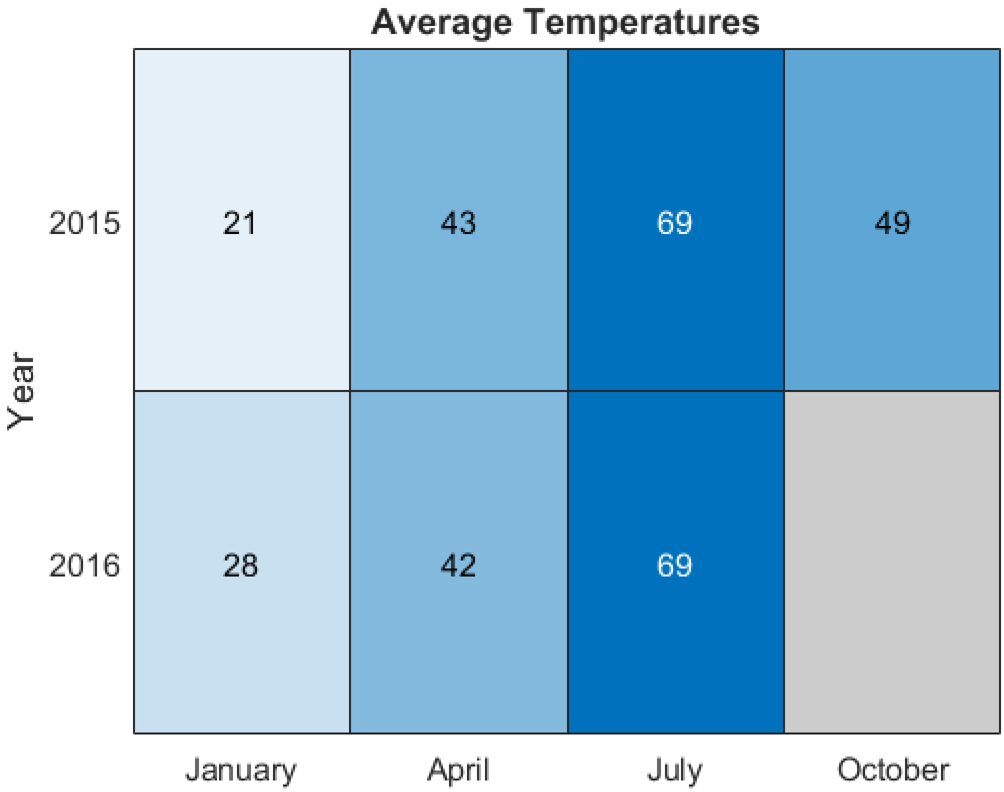
```
h.CellLabelFormat = '%.0f';
```



Add or Remove Values Along Axis

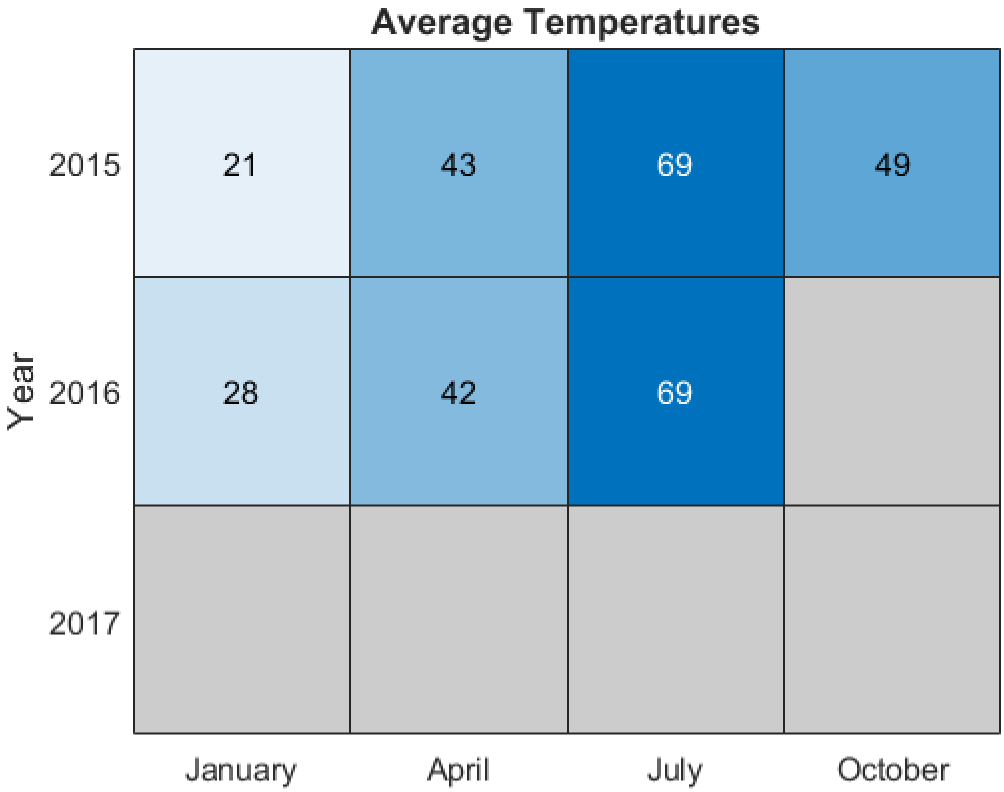
Show only the first month of each quarter. Use the `removecats` function to remove the unwanted categories. The `removecats` function replaces the removed months with `undefined` in the table stored in the `SourceTable` property.

```
rmv = {'February','March','May','June','August','September',...
    'November','December'};
h.SourceTable.Month = removecats(h.SourceTable.Month,rmv);
```



Add the year 2017 along the *y*-axis. First, convert the data in the `Year` column from a cell array to a categorical array. Use the `addcats` function to add the additional category. Since there is no data associated with this year, the heatmap cells use the missing data color.

```
h.SourceTable.Year = categorical(h.SourceTable.Year);
h.SourceTable.Year = addcats(h.SourceTable.Year,'2017');
```



See Also

Functions
`addcats` | `categorical` | `heatmap` | `readtable` | `removecats` | `renamecats` | `reordercats` | `table`

Properties
HeatmapChart Properties

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 "Save Figure to Reopen in MATLAB Later" on page 8-33 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”.

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

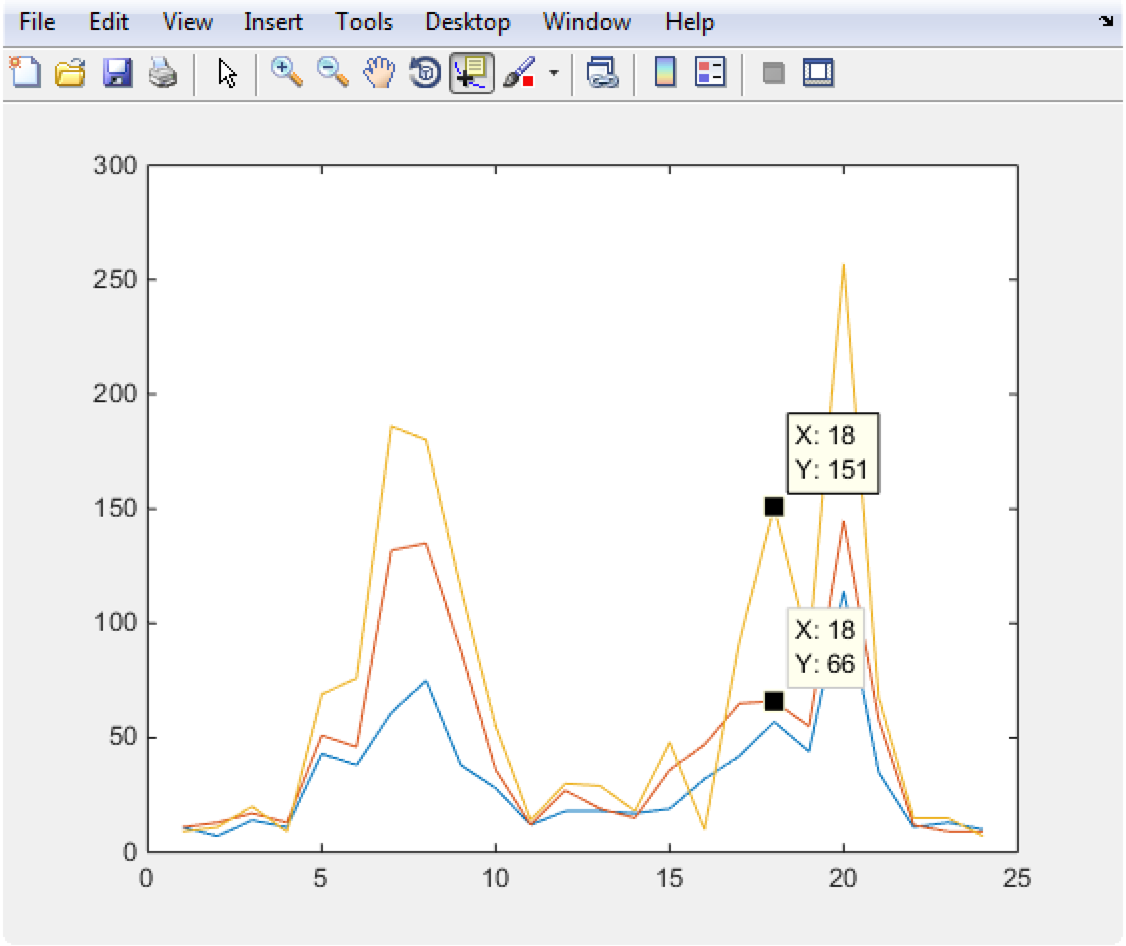
Enabling Data Cursor Mode

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.

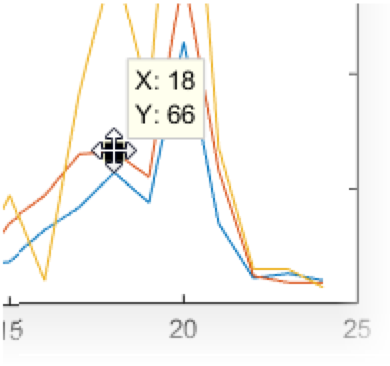
The illustrations below use traffic count data stored in `count.dat`:

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



Moving the Marker

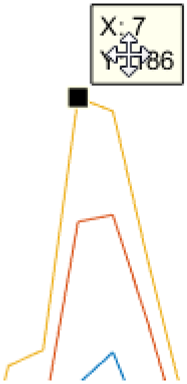
You can move the marker using the arrow keys and the mouse. The up and right arrows move the marker to data points having greater index values in the data arrays. The down and left arrow keys move the marker to data points having lesser index values. When you set **Selection Style** to **Mouse Position** 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.

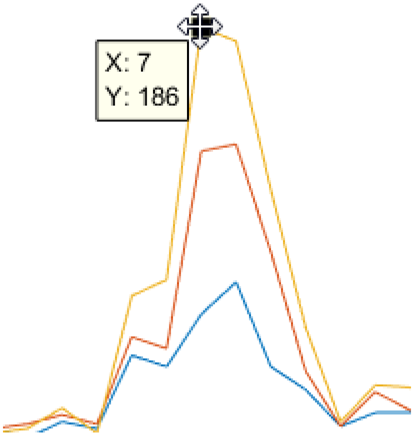
To position the datatip, press, but do not release the mouse button while over the datatip text box and drag it to one of the four positions, as shown below:



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

Dragging the Datatip to Different Locations

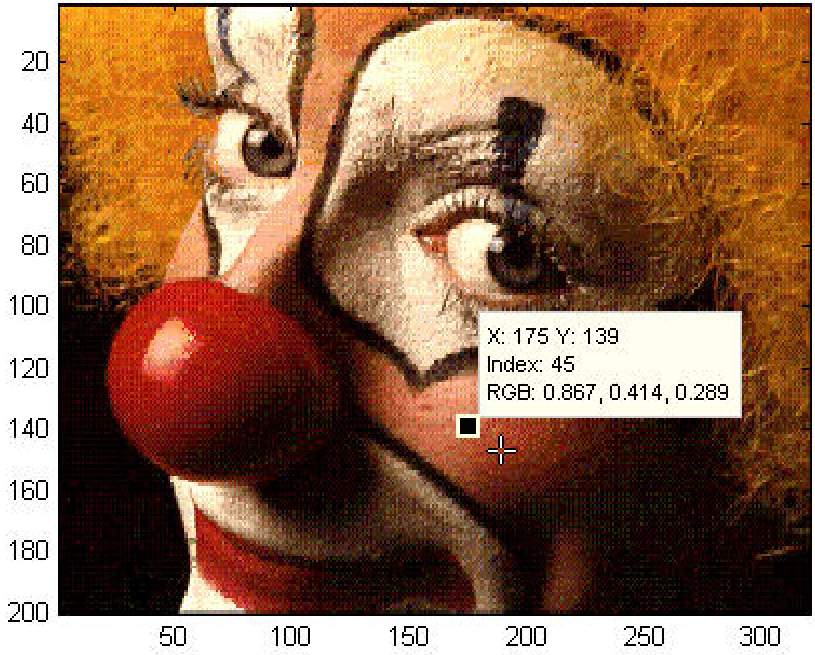
You can drag the datatip to different locations on the graph object by clicking down on the datatip and dragging the mouse. You can also use the arrow keys to move the datatip.



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.

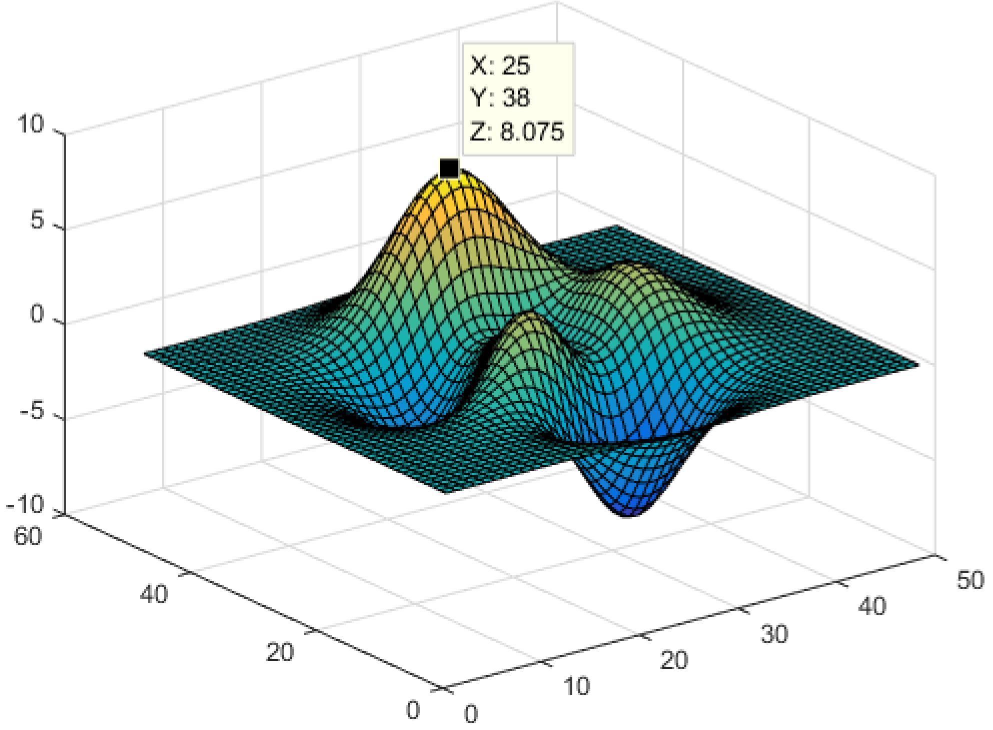
Datatips on Image Objects

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 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 (character vector or cell array of character vectors).

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 `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”.

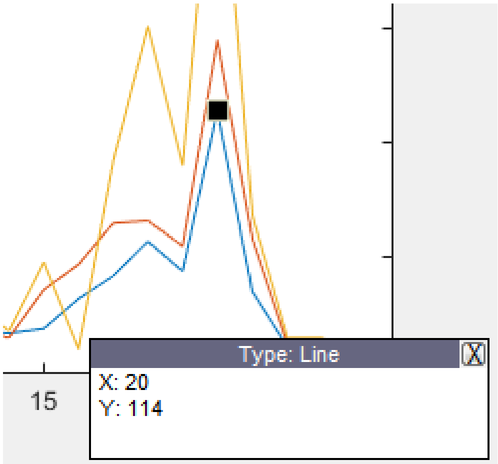
Display Style — Datatip or Cursor Window

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**.



Type: Line

X: 20
Y: 114

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
```

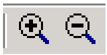
Zooming in Graphs

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 3-D Views" on page 3-16

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.

Zooming shifts the view of the data by modifying the axis limits. For axes in a 3-D view, if you want zooming to modify camera properties of the axes instead, then right-click the axes when in zoom mode and select **3D Options** > **Camera Pan and Zoom**.

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.

Panning shifts the view of the data by modifying the axis limits. For axes in a 3-D view, if you want panning to modify camera properties of the axes instead, then right-click the axes when in pan mode and select **3D Options** > **Camera Pan and Zoom**.

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:

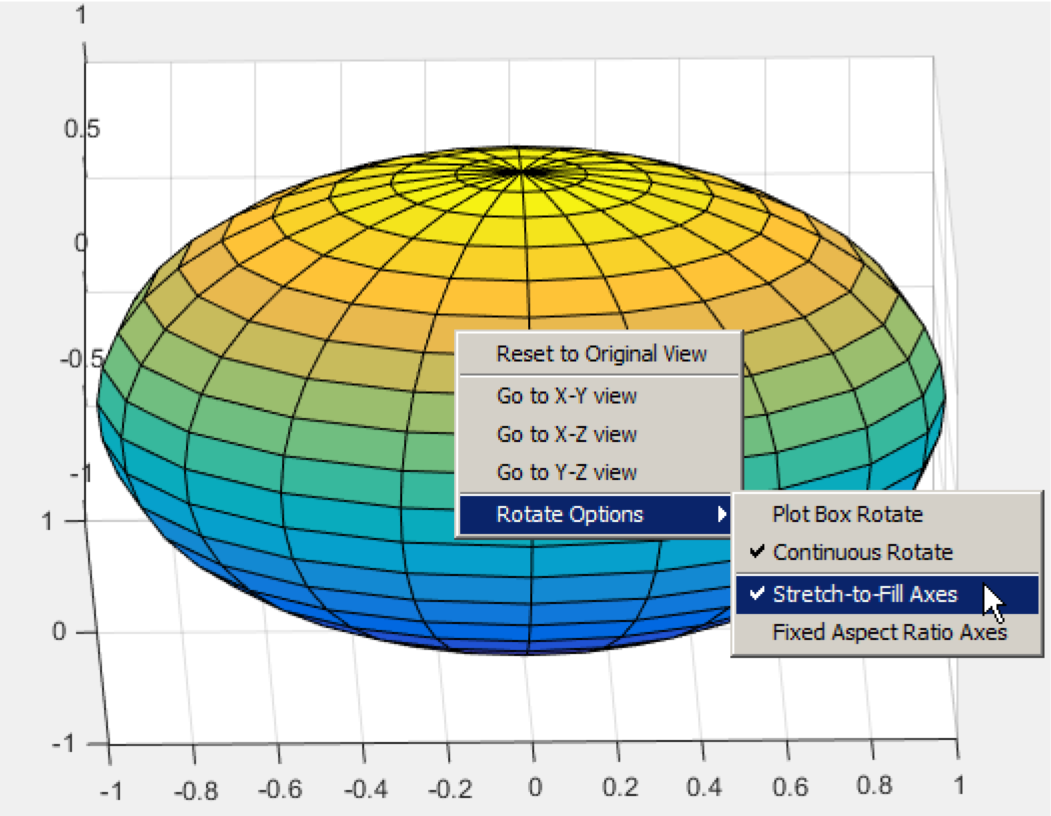
- **Plot Box Rotate** — Display only the axes bounding box for faster rotation of complex objects. Use this option if the default **Continuous Rotate** style is unacceptably slow.
- **Continuous Rotate** — Display all graphics during rotation.

Axes Behavior During Rotation

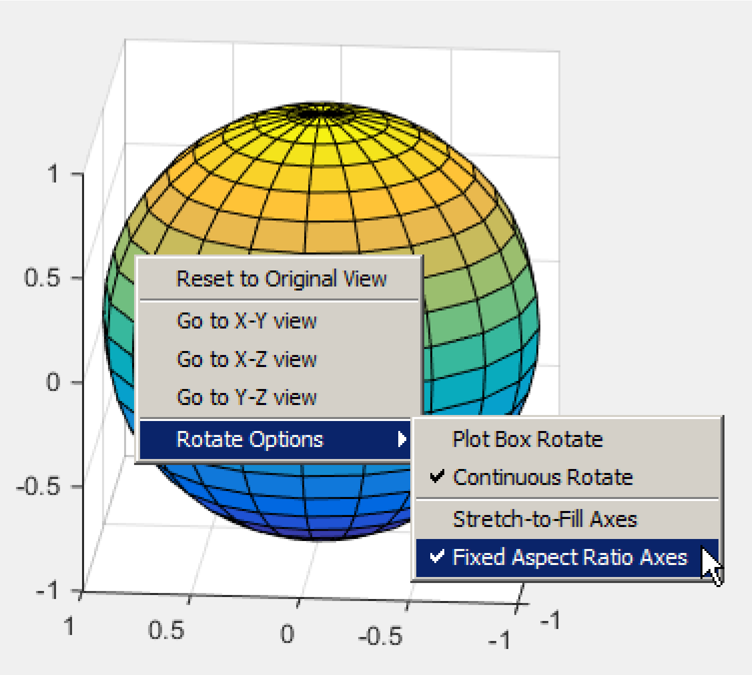
You can select two types of behavior with respect to the aspect ratio of axes during rotation:

- **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

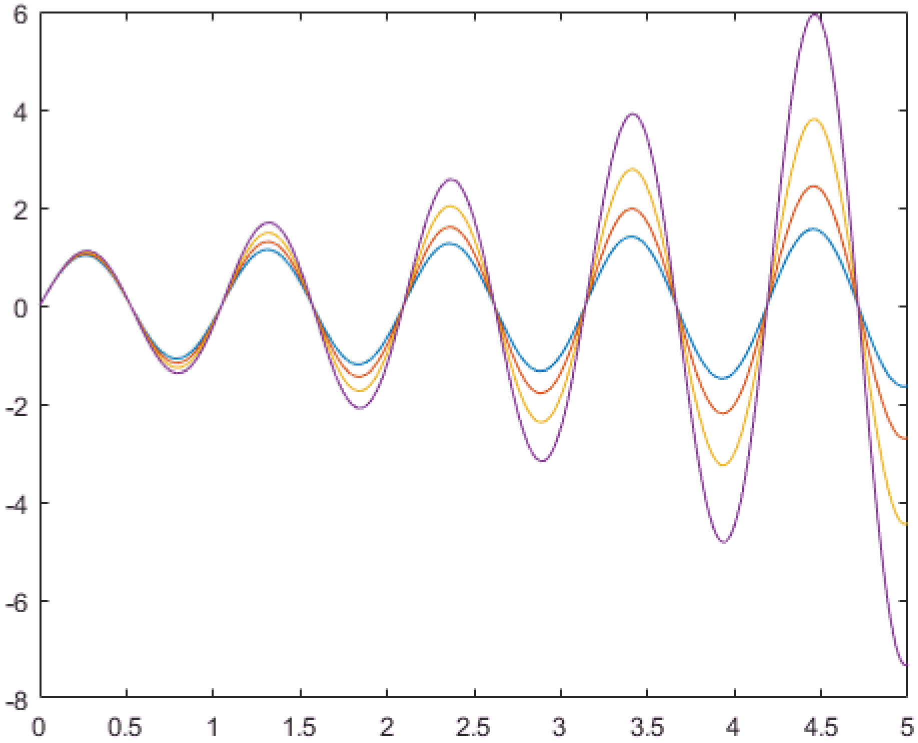
- “Exclude Objects from Graph Legend” on page 4-2
- “Specify Legend Descriptions During Line Creation” on page 4-5
- “Add Text to Specific Points on Graph” on page 4-8
- “Include Variable Values in Graph Text” on page 4-15
- “Text with Mathematical Expression Using LaTeX” on page 4-18
- “Greek Letters and Special Characters in Graph Text” on page 4-23
- “Add Annotations to Graph Interactively” on page 4-30
- “Add Text to Graph Interactively” on page 4-33
- “Align Objects in Graph Using Alignment Tools” on page 4-39

Exclude Objects from Graph Legend

This example shows how to plot several lines and create a legend that includes descriptions for a subset of the lines.

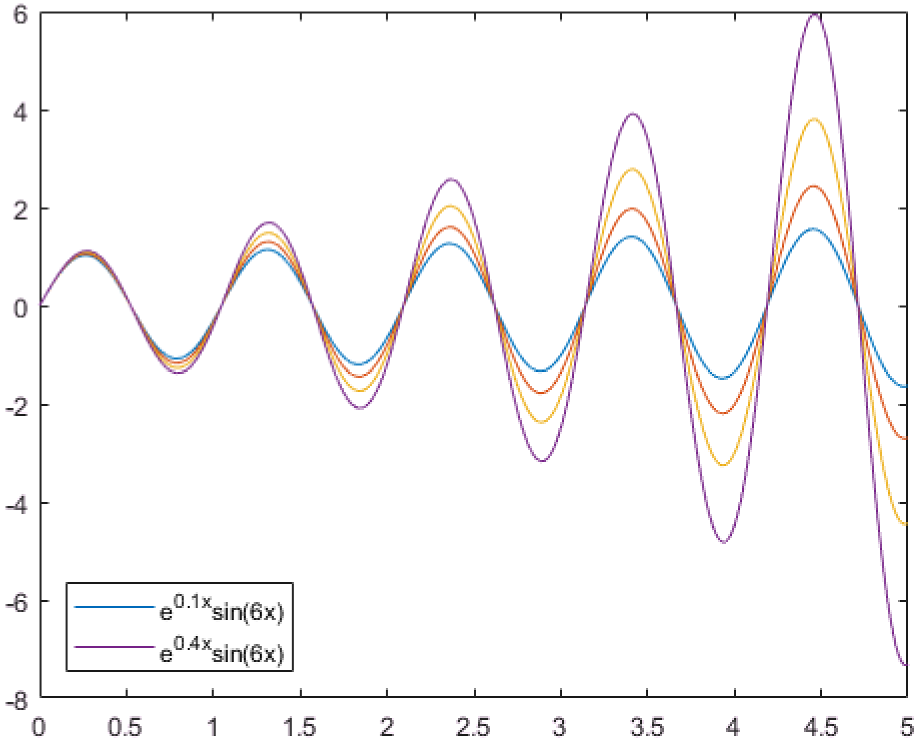
Plot four lines and return each chart line object as an output argument from the `plot` function.

```
x = linspace(0,5,500);
y1 = exp(0.1*x).*sin(6*x);
p1 = plot(x,y1);

hold on
y2 = exp(0.2*x).*sin(6*x);
p2 = plot(x,y2);

y3 = exp(0.3*x).*sin(6*x);
p3 = plot(x,y3);

y4 = exp(0.4*x).*sin(6*x);
p4 = plot(x,y4);
```



Create a legend in the lower left corner of the axes. Include descriptions for only the first and last lines in the legend by specifying `p1` and `p4` as the first input argument.

```
hold off
legend([p1 p4],'e^{0.1x}sin(6x)','e^{0.4x}sin(6x)','Location','southwest')
```



See Also

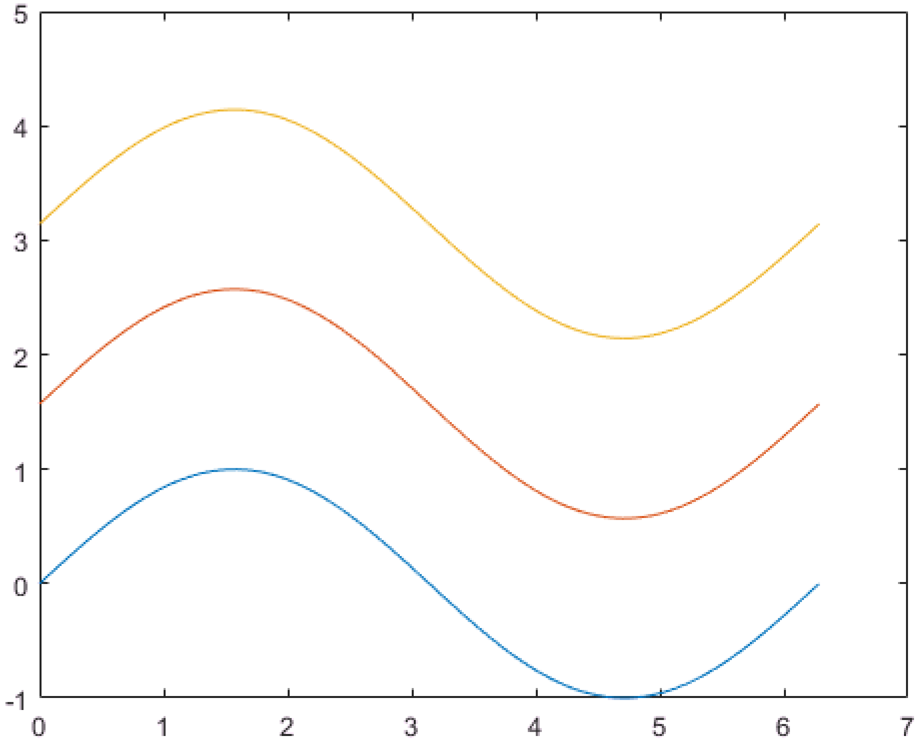
`legend` | `line` | `mean` | `plot`

Specify Legend Descriptions During Line Creation

This example shows how to plot multiple lines and specify the associated legend labels during the plotting command.

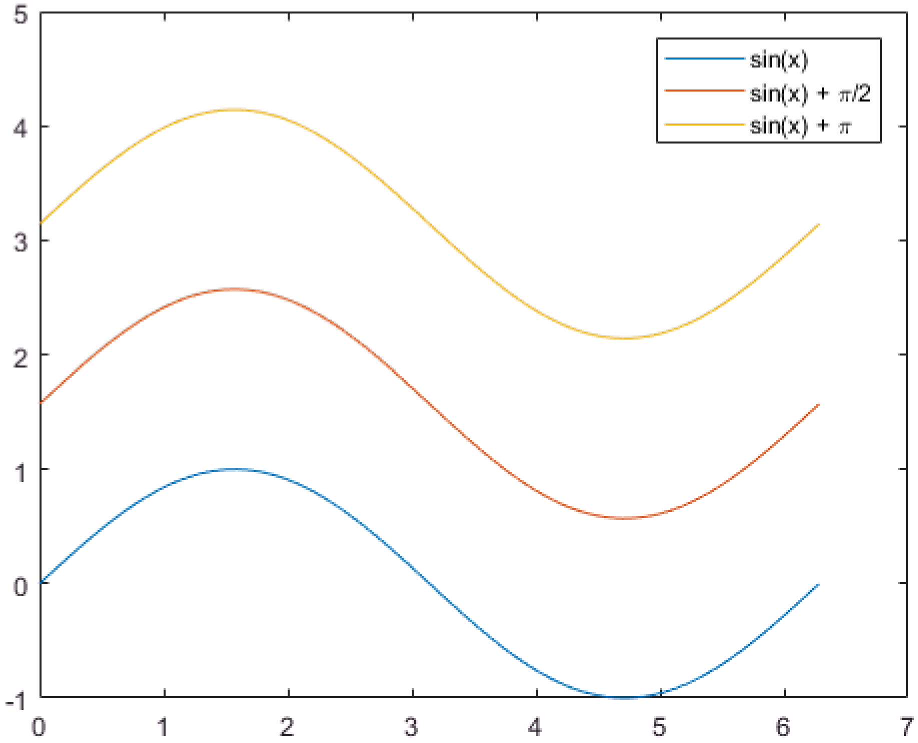
Plot three sine curves. For each line, set the `DisplayName` property.

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

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

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



Create the legend. The legend uses the `DisplayName` property for the labels.

```
legend('show')
```



See Also

`hold` | `legend` | `plot`

Add Text to Specific Points on Graph

In this section...

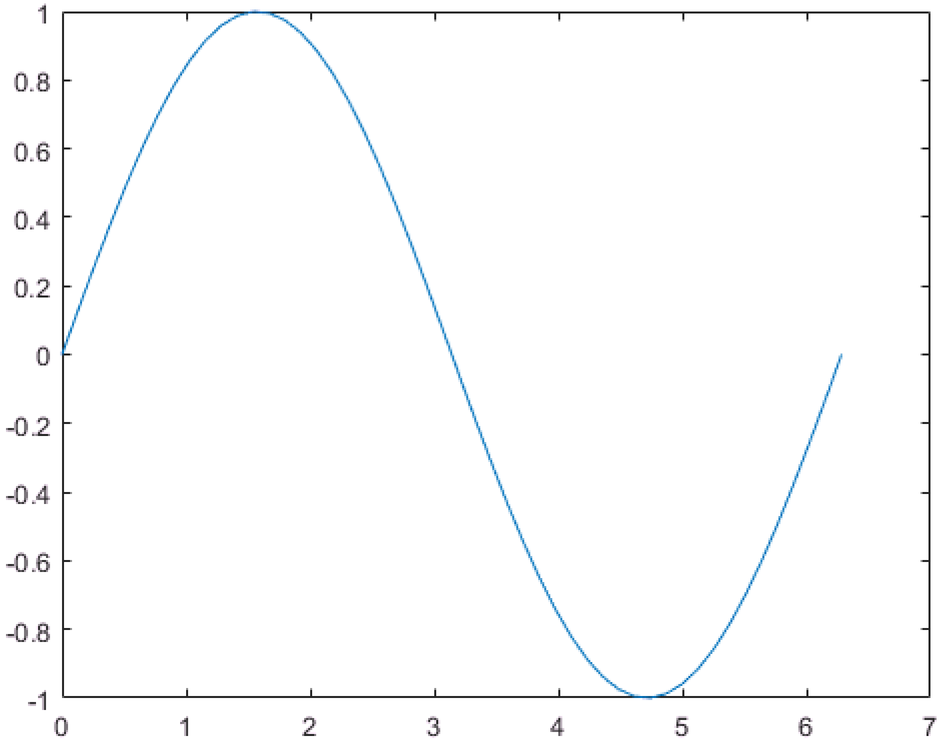
"Add Text to Three Data Points on Graph" on page 4-8
"Determine Minimum and Maximum Points and Add Text" on page 4-11

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.

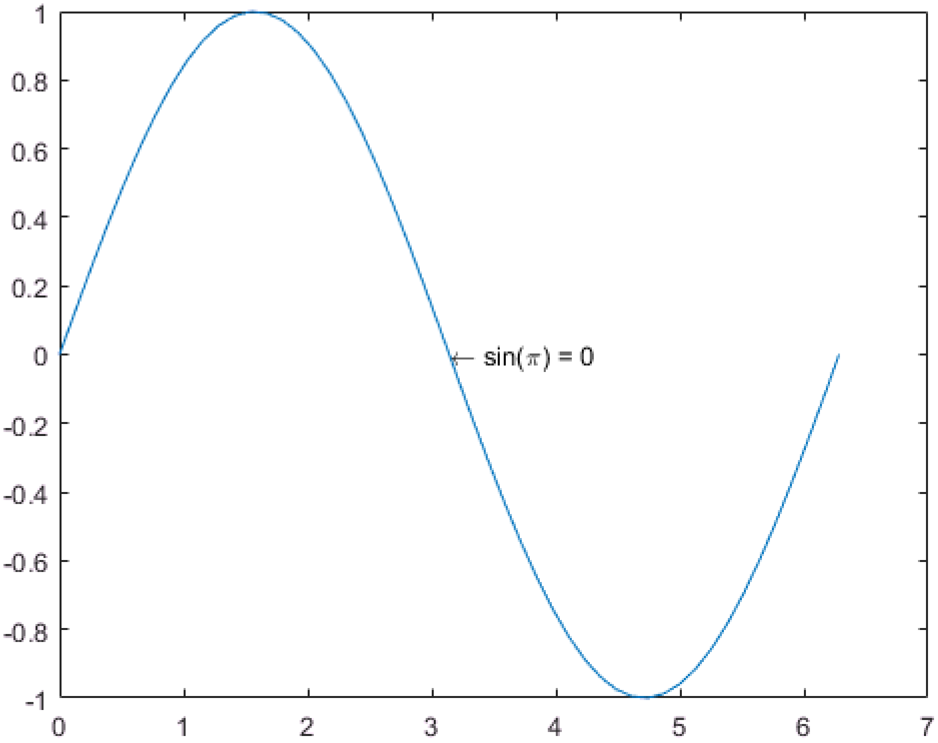
Use the `linspace` function to create `t` as a vector of 50 values between 0 and 2π. Create `y` as sine values. Plot the data.

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



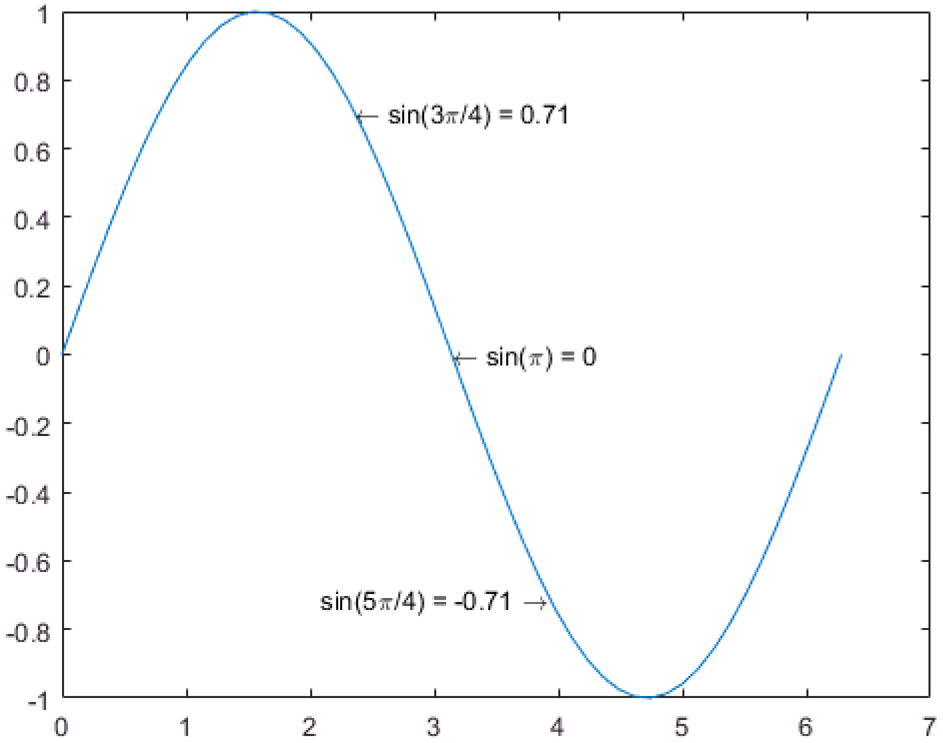
Use the `text` function to add a text description to the graph at the point $(\pi, \sin(\pi))$. The first two input arguments to this function specify the text position. The third argument specifies the text. Display an arrow pointing to the left by including the TeX markup `\leftarrow` in the text.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

```
x1 = pi;
y1 = sin(pi);
txt1 = '\leftarrow sin(\pi) = 0';
text(x1,y1,txt1)
```



Add text descriptions to two more data points on the graph. By default, the data point is to the left of the text. To show the data point to the right of the text, specify the `HorizontalAlignment` property as `'right'`. Use the TeX markup `\rightarrow` to diplay an arrow pointing to the right.

```
x2 = 3*pi/4;
y2 = sin(3*pi/4);
txt2 = '\leftarrow sin(3\pi/4) = 0.71';
text(x2,y2,txt2)

x3 = 5*pi/4;
y3 = sin(5*pi/4);
txt3 = 'sin(5\pi/4) = -0.71 \rightarrow';
text(x3,y3,txt3,'HorizontalAlignment','right')
```

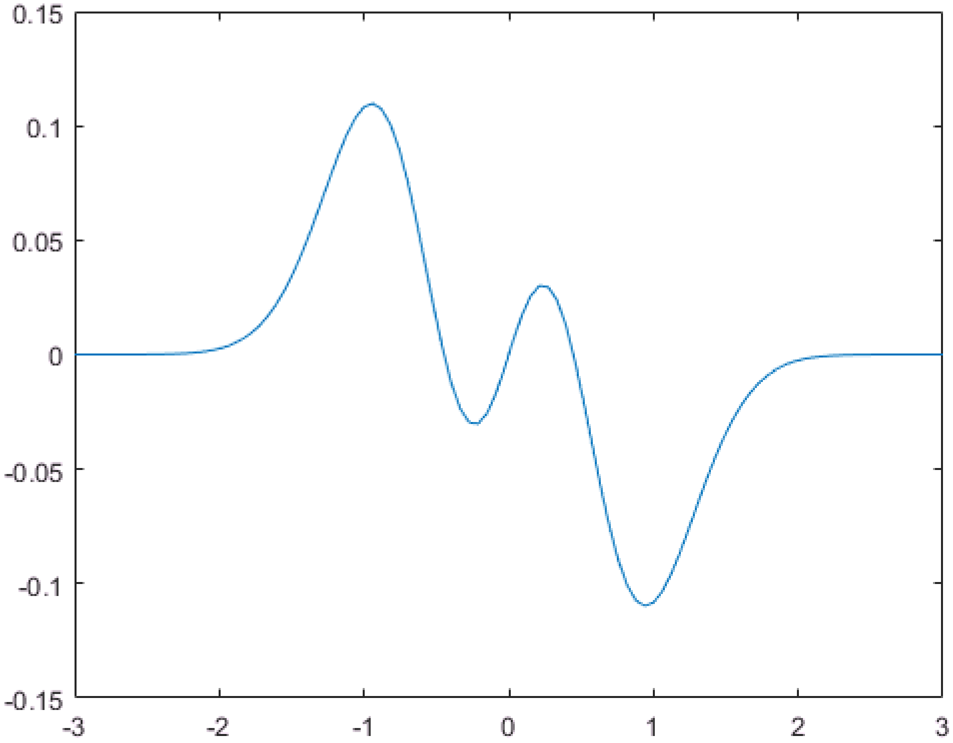


Determine Minimum and Maximum Points and Add Text

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)
```

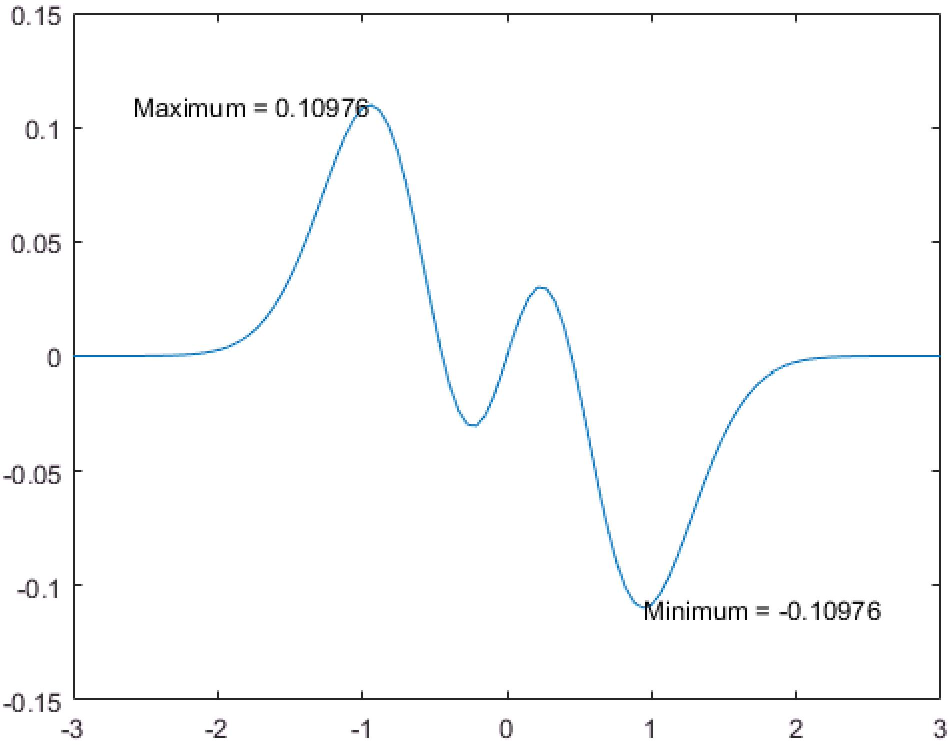


Find the indices of the minimum and maximum values in `y`. Use the indices to determine the (x,y) values at the minimum and maximum points.

```
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 text. Specify the text alignment in relation to the data 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 4-15

- "Greek Letters and Special Characters in Graph Text" on page 4-23

Include Variable Values in Graph Text

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

In this section...

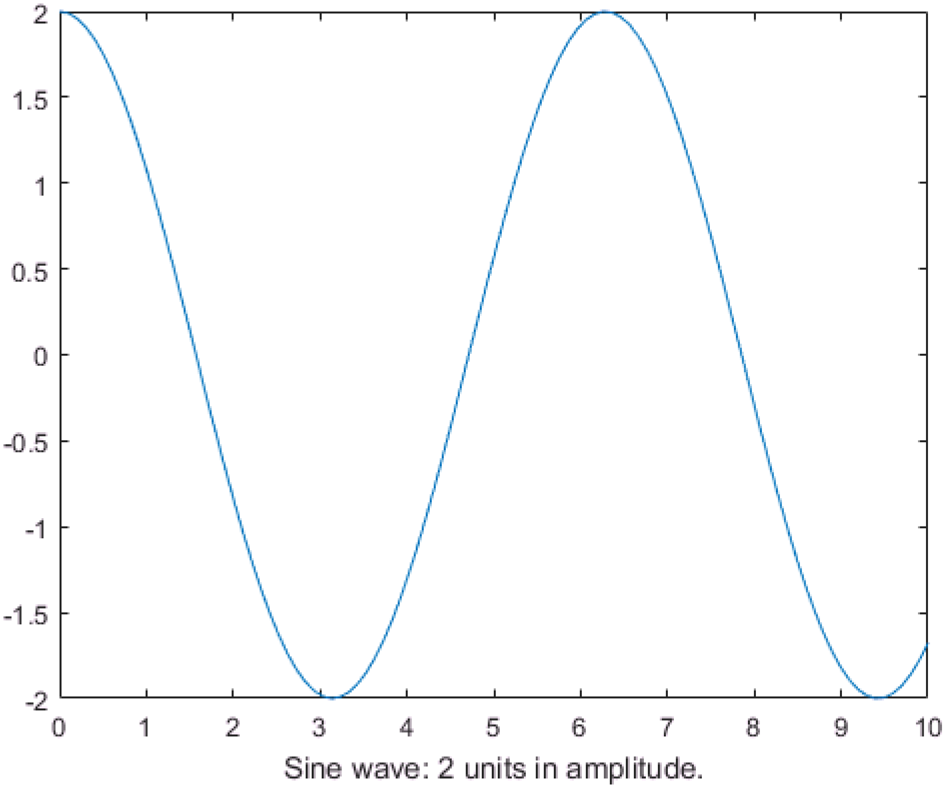
“Include Variable Value in Axis Label” on page 4-15

“Include Loop Variable Value in Graph Title” on page 4-16

Include Variable Value in Axis Label

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

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

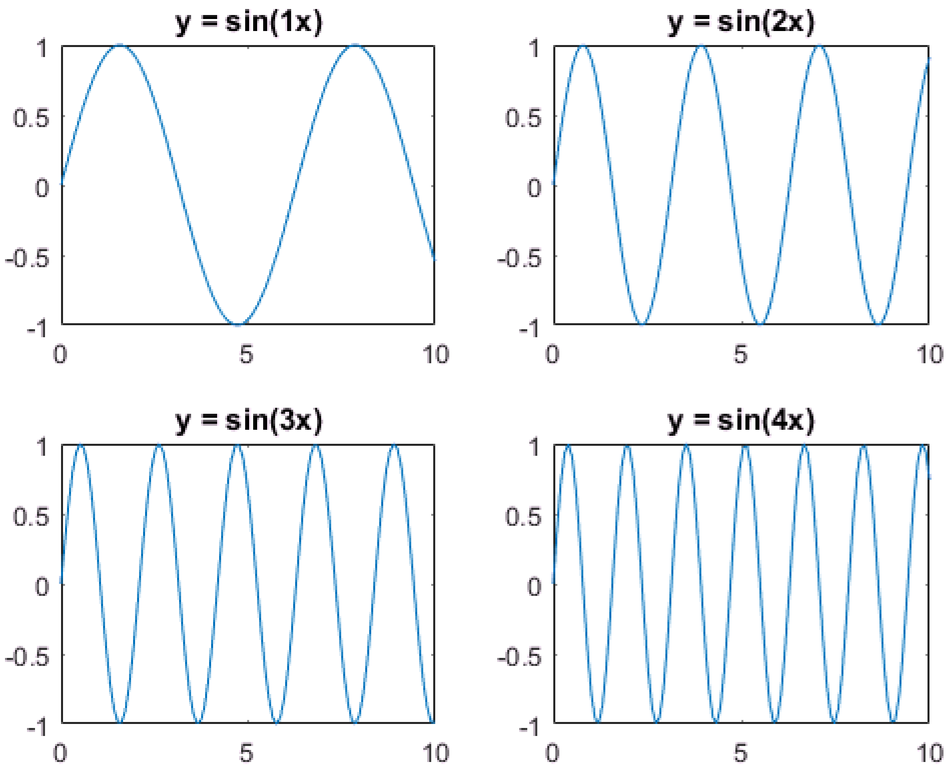


Include Loop Variable Value in Graph Title

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 4-8
- "Greek Letters and Special Characters in Graph Text" on page 4-23

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...

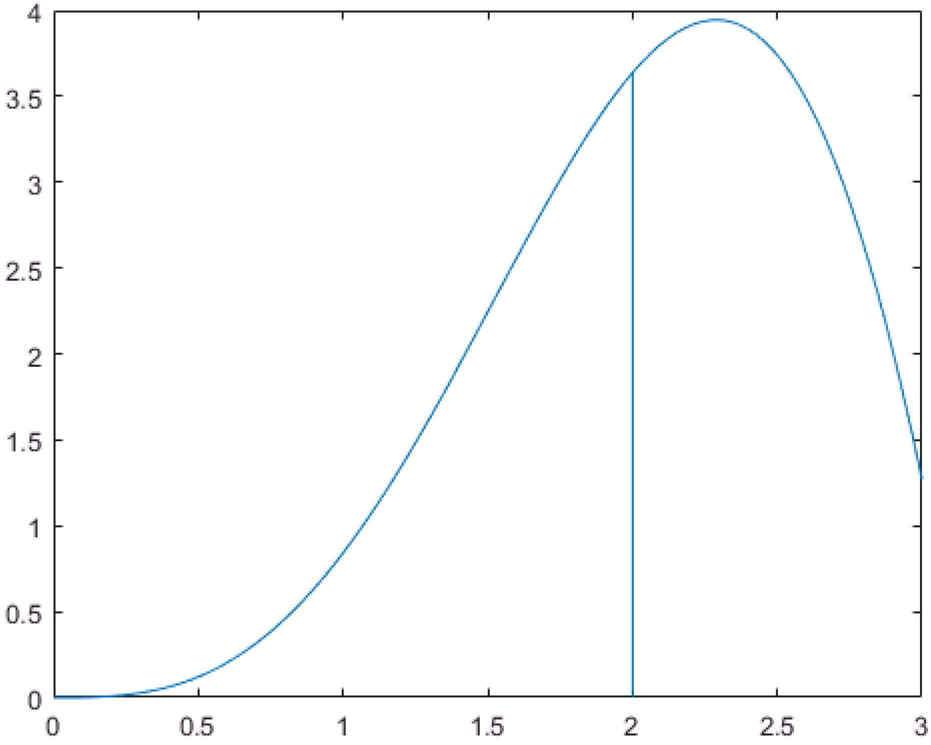
"Add Text with Integral Expression to Graph" on page 4-18

"Add Text with Summation Symbol to Graph" on page 4-20

Add Text with Integral Expression to Graph

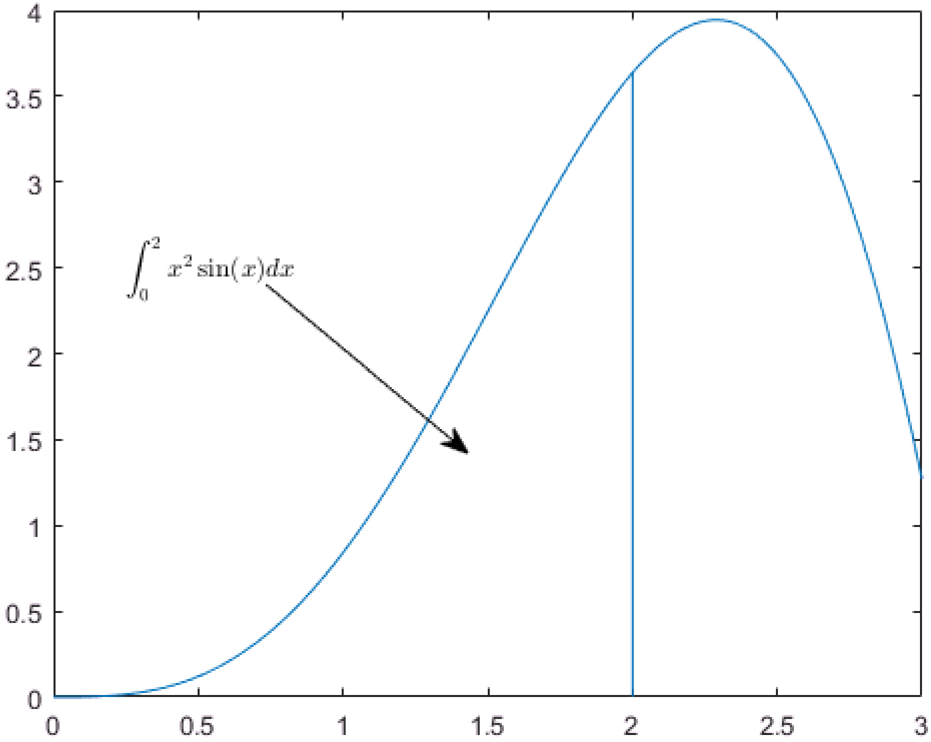
Plot $y = x^2 \sin(x)$. Draw a vertical line at $x = 2$ from the *x*-axis to the plotted line.

```
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])
```

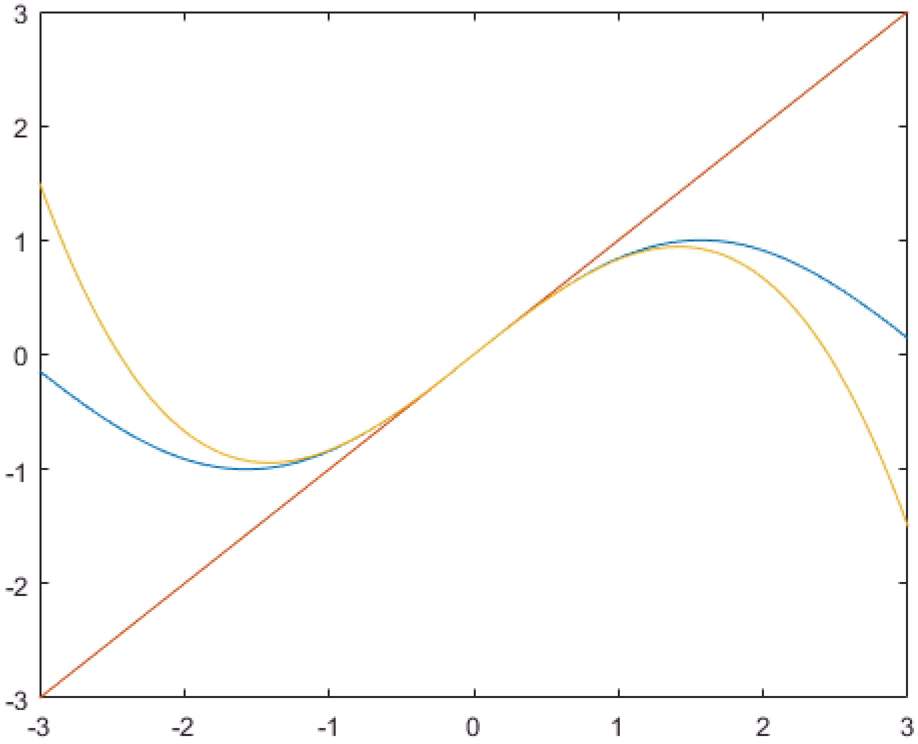


Add Text with Summation Symbol to Graph

Plot the sine function and plot two polynomials.

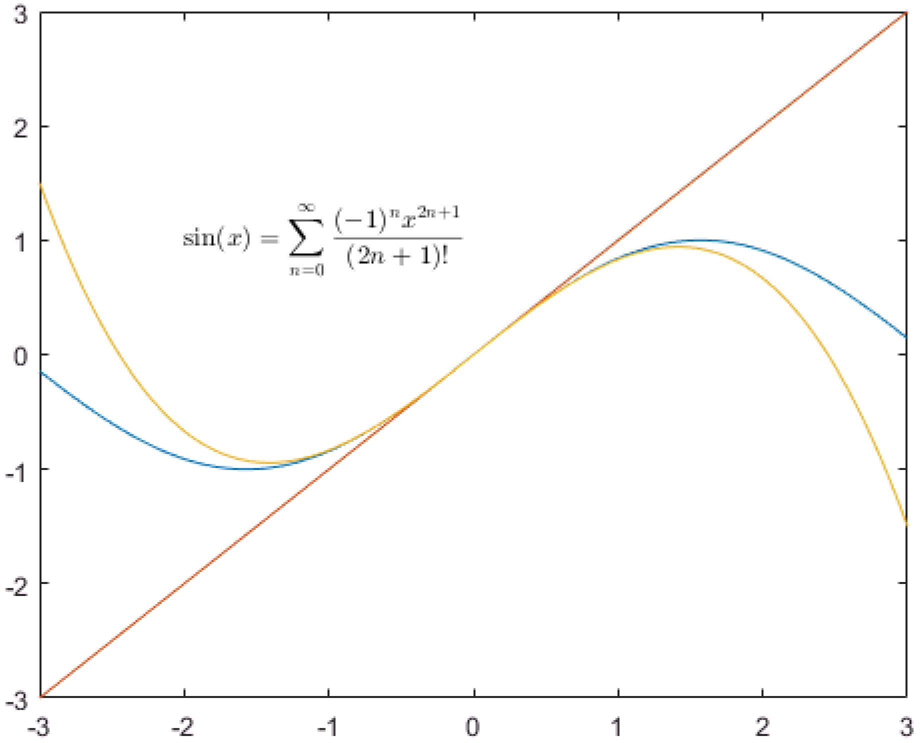
```
x = linspace(-3,3);
y = sin(x);
plot(x,y)

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 4-8
- “Greek Letters and Special Characters in Graph Text” on page 4-23

Greek Letters and Special Characters in Graph Text

You can add text to a graph that includes Greek letters and special characters using TeX markup. You also can use TeX markup to add superscripts, subscripts, and modify the text type and color.

In this section...

“Include Greek Letters in Graph Text” on page 4-23

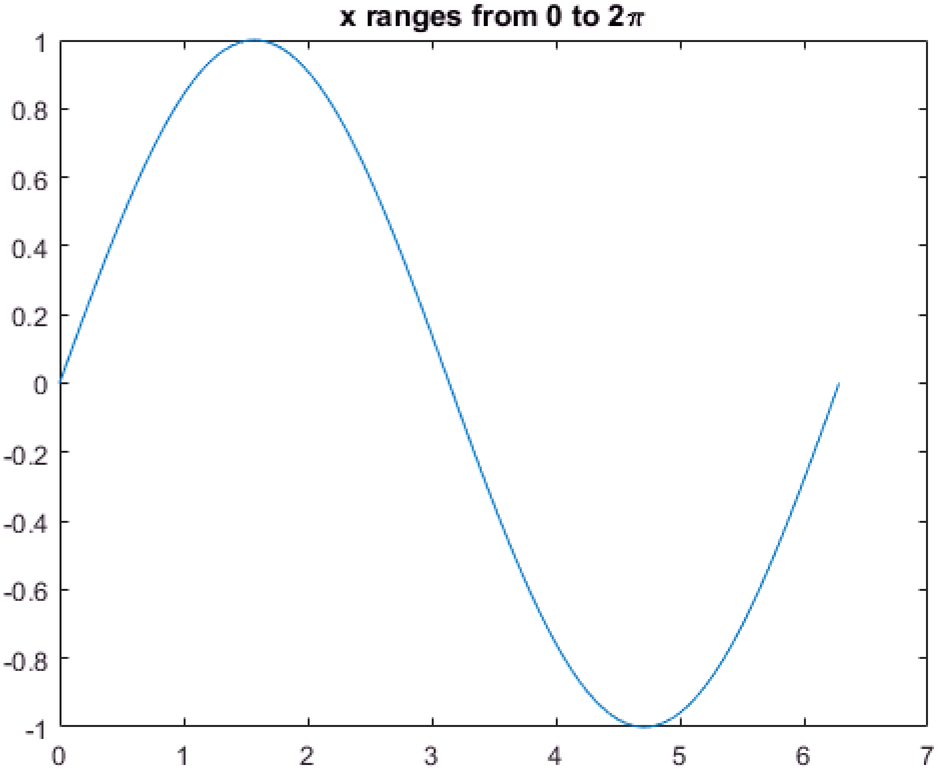
“Include Superscripts and Annotations in Graph Text” on page 4-24

“TeX Markup Options” on page 4-26

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')
```

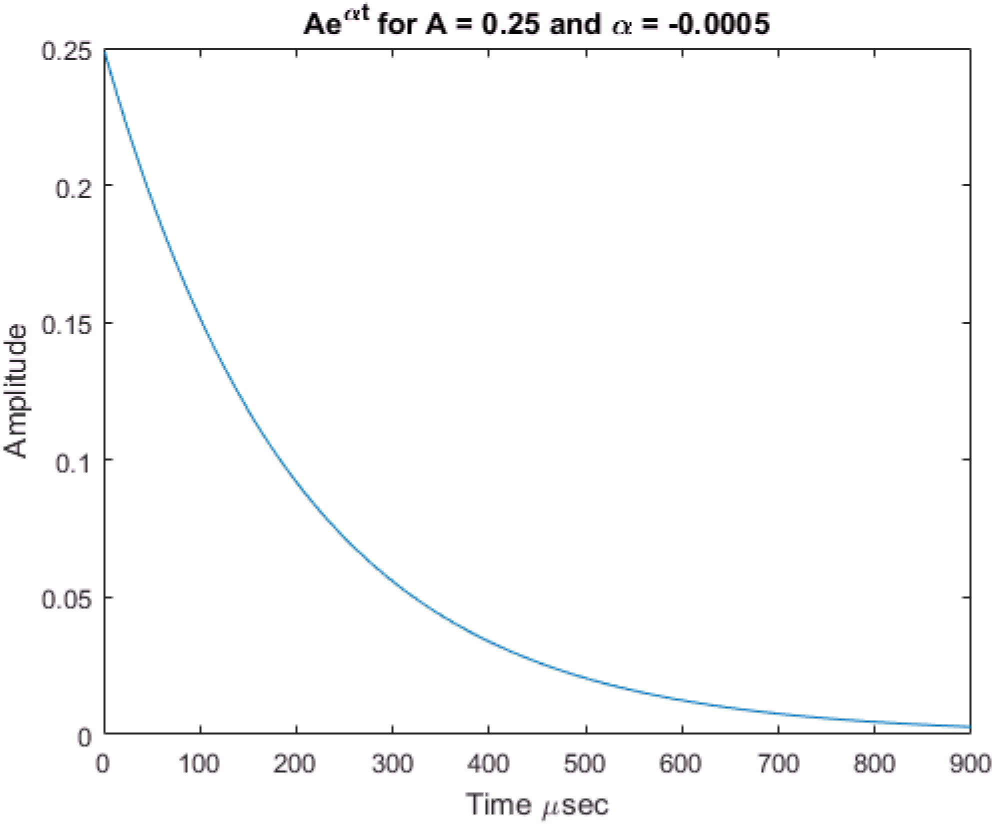


Include Superscripts and Annotations in Graph Text

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 α and μ in the text using the TeX markups `\alpha` and `\mu`, respectively.

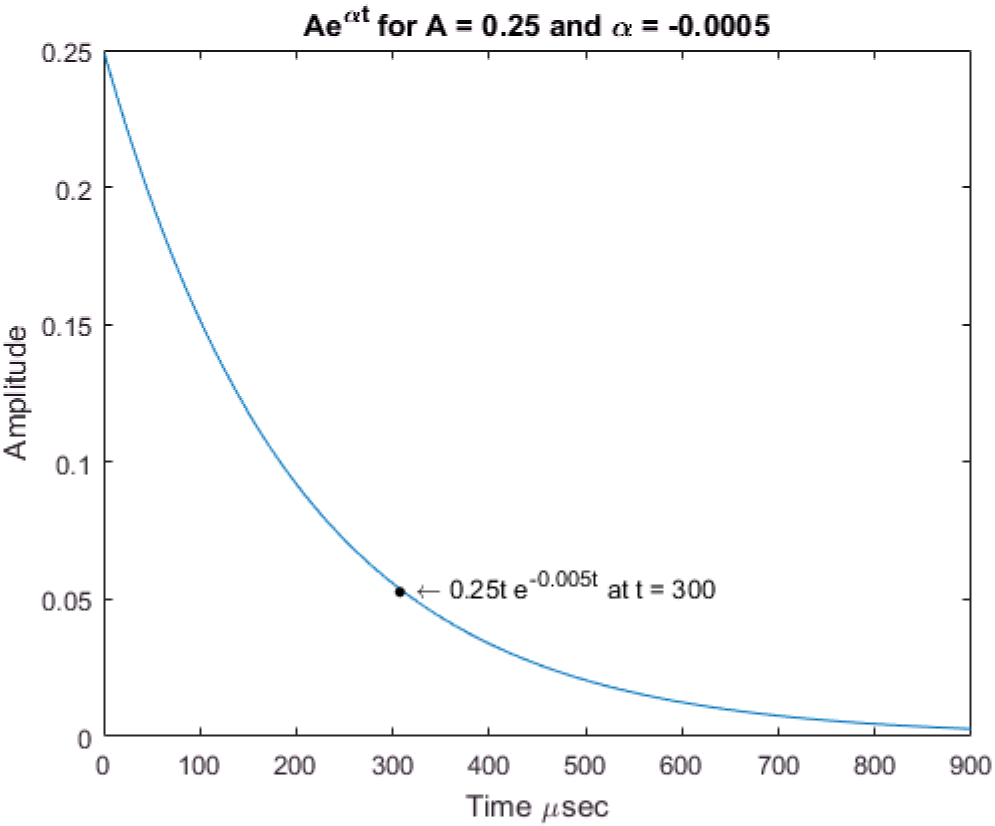
```
t = 1:900;
y = 0.25*exp(-0.005*t);
```

```
figure
plot(t,y)
title('Ae^{\alphat} 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 specified data point is to the left of the text.

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



TeX Markup Options

MATLAB supports a subset of TeX markup. Use TeX markup to add superscripts and subscripts, modify the text type and color, and include special characters. MATLAB interprets the text using TeX markup as long as the `Interpreter` property of the text object is set to `'tex'` (the default).

This table lists the supported modifiers with the `Interpreter` property set to `'tex'`. Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they only modify the next character or the characters within the curly braces.

Modifier	Description	Example
`^{ }`	Superscript	`'text^{superscript}'`
`_{ }`	Subscript	`'text_{subscript}'`
`\bf`	Bold font	`'\bf text'`
`\it`	Italic font	`'\it text'`
`\sl`	Oblique font (usually the same as italic font)	`'\sl text'`
`\rm`	Normal font	`'\rm text'`
`\fontname{specifier}`	Font name — Set `specifier` as the name of a font family. You can use this in combination with other modifiers.	`'\fontname{Courier} text'`
`\fontsize{specifier}`	Font size — Set `specifier` as a numeric scalar value in point units to change the font size.	`'\fontsize{15} text'`
`\color{specifier}`	Font color — Set `specifer` as one of these colors: `red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`, `gray`, `darkGreen`, `orange`, or `lightBlue`.	`'\color{magenta} text'`
`\color[rgb]{specifier}`	Custom font color — Set `specifier` as a three-element RGB triplet.	`'\color[rgb]{0,0.5,0.5} text'`

This table lists the supported special characters with the `Interpreter` property set to `'tex'`.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\alpha`	α	`\upsilon`	υ	`\sim`	~
`\angle`	∠	`\phi`	ϕ	`\leq`	≤

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsui	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrighta	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta	ϑ	\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon	ϒ	\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∍	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	ℵ
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in	∈	\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	ο
\rfloor	⌋	\neg	¬	\nabla	∇

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
`\lfloor`	⌊	`\times`	x	`\ldots`	...
`\perp`	⊥	`\surd`	√	`\prime`	´
`\wedge`	∧	`\varpi`	ϖ	`\0`	∅
`\rceil`	⌉	`\rangle`	〉	`\mid`	|
`\vee`	∨	`\langle`	〈	`\copyright`	©

See Also

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

More About

- "Add Title, Axis Labels, and Legend to Graph" on page 2-13
- "Include Variable Values in Graph Text" on page 4-15
- "Add Text to Specific Points on Graph" on page 4-8

Add Annotations to Graph Interactively

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

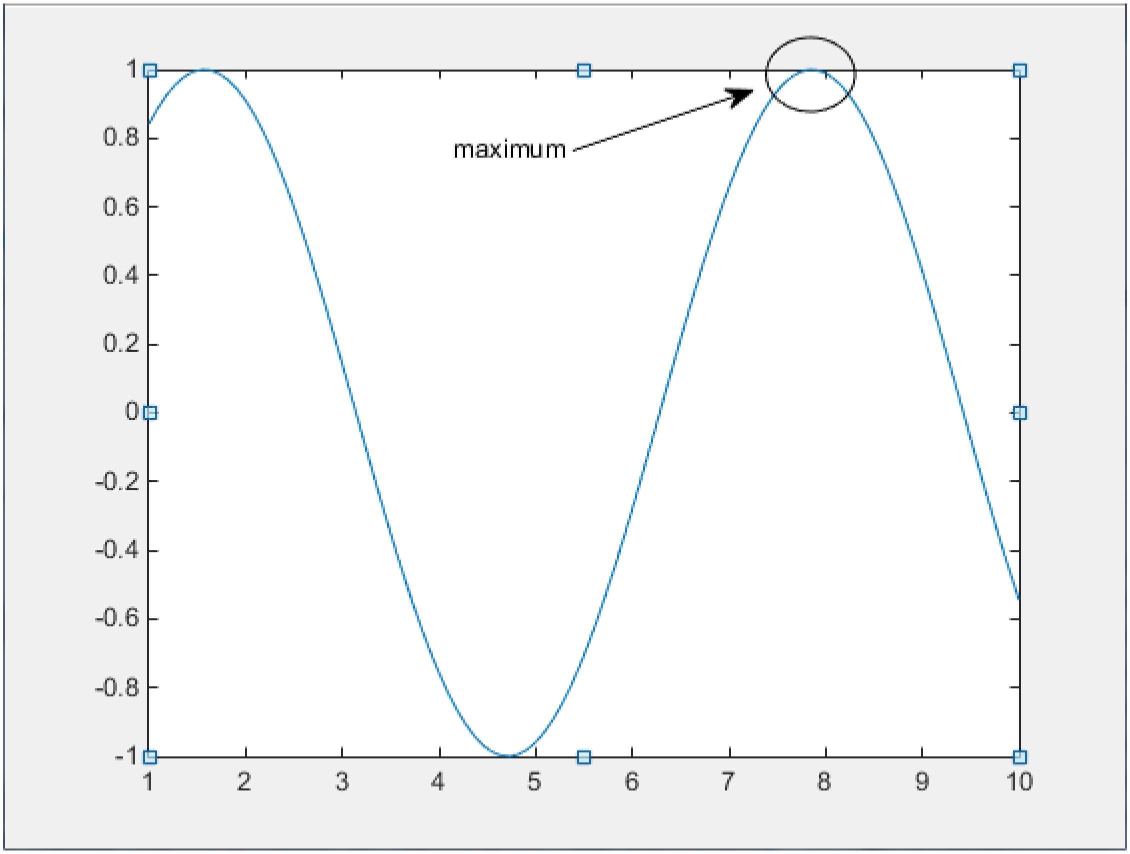
In this section...
"Add Annotations" on page 4-30 "Pin Annotations to Points in Graph" on page 4-31

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.

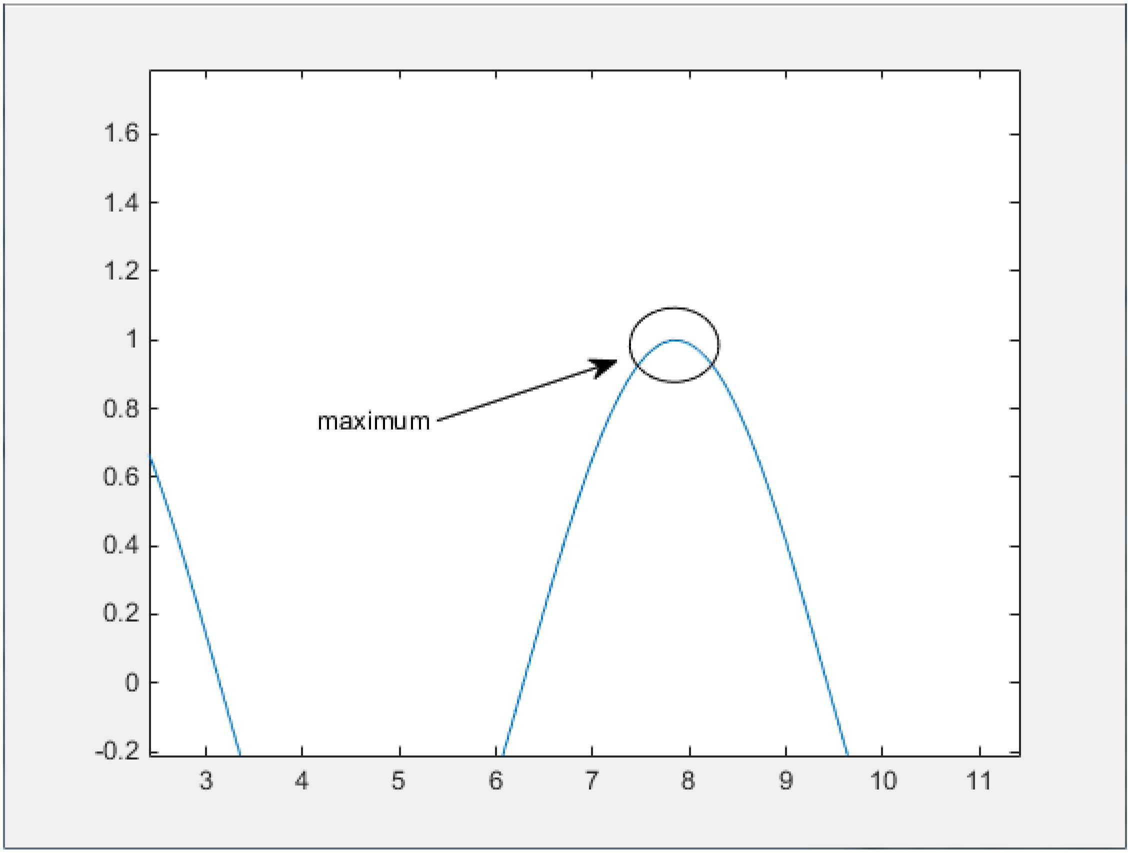


To change the location of an annotation, drag it. To modify the appearance of an annotation, right-click it and use the context menu. To view additional properties, open the Property Editor select **Show Property Editor** from the context menu.

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 4-33

Add Text to Graph Interactively

In this section...

"Add Title and Axis Labels" on page 4-33

"Add Legend" on page 4-35

"Add Annotations to Graph" on page 4-37

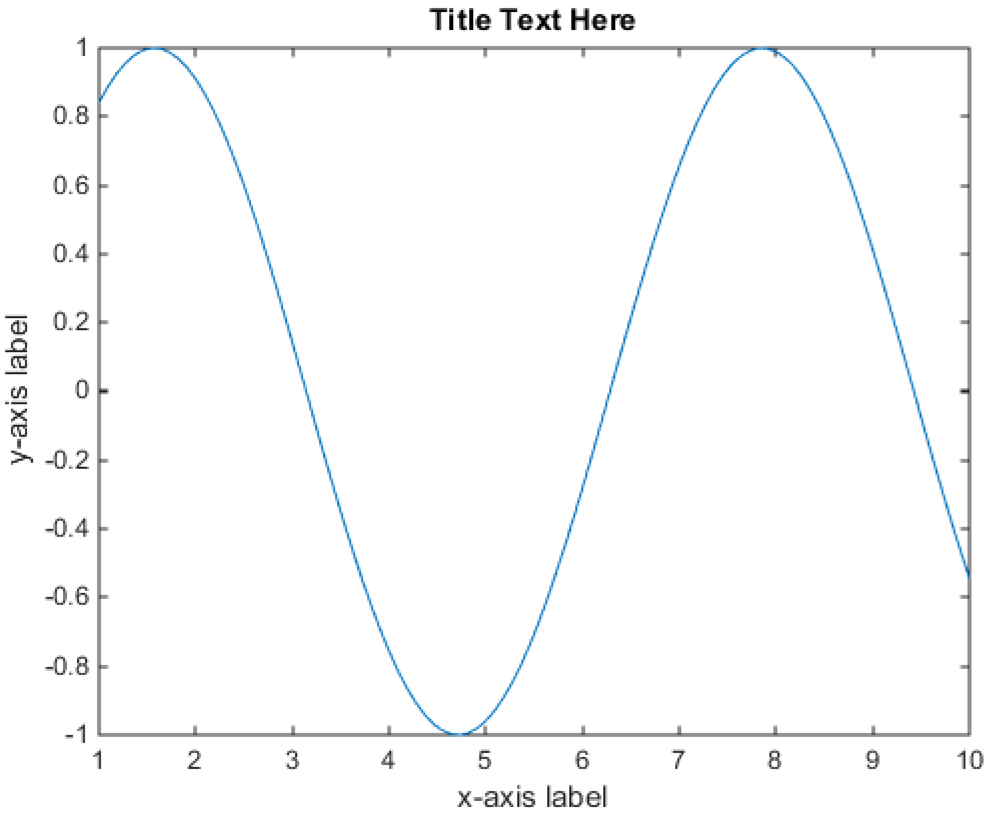
This example shows how to interactively add a title, legend, axis labels, and other text to a graph using the figure menus and plot tools.

Add Title and Axis Labels

Create a simple line plot.

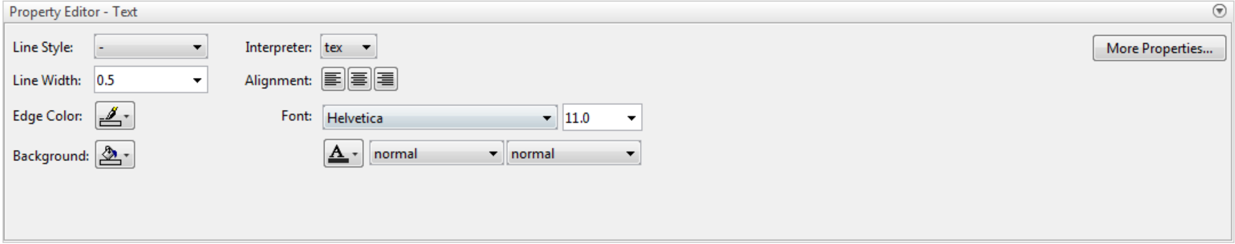
```
x = linspace(1,10);
plot(x,sin(x))
```

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.



Property Editor - Text

Line Style: -

Line Width: 0.5

Edge Color:

Background:

Interpreter: tex

Alignment:

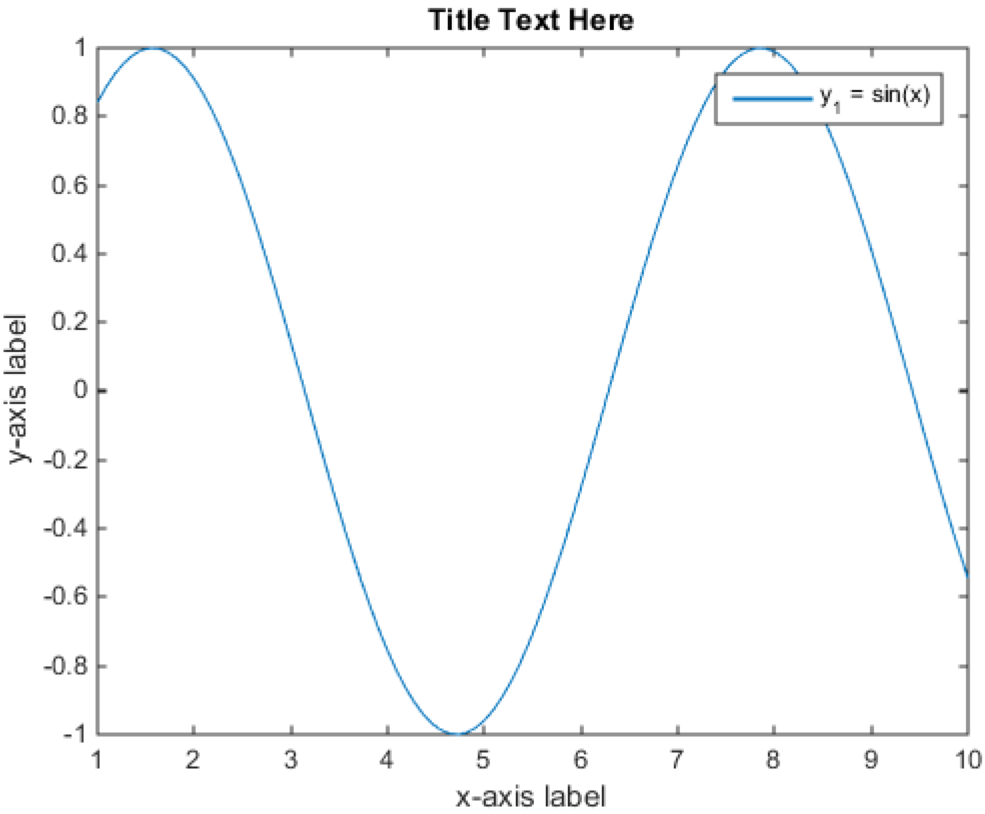
Font: Helvetica 11.0

normal normal

Add Legend

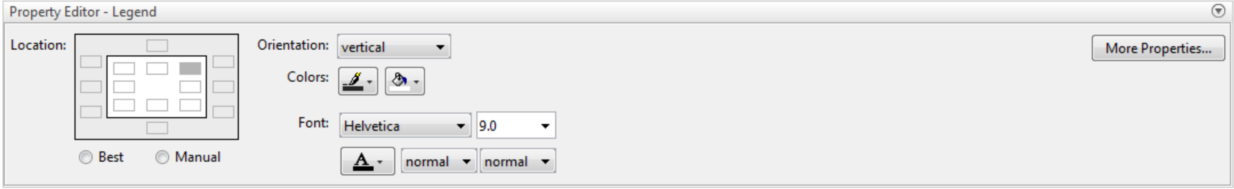
Add a legend to the graph. In the figure, select **Insert** > **Legend**.

By default, the legend labels each plotted object with `data1`, `data2`, and so on. Change the legend label by double-clicking the label and retyping a new label. 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.



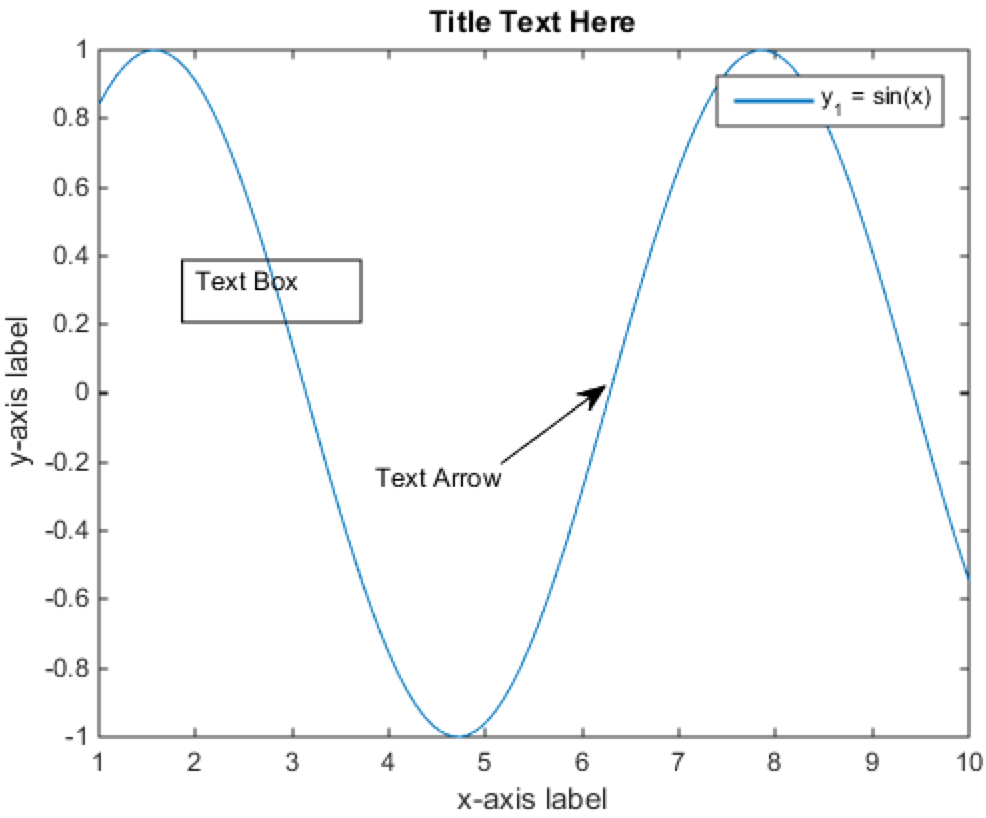
Note: To display a legend with more than 50 items, use the `legend` function.

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 **Text Arrow** 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 4-30
- "Add Text to Specific Points on Graph" on page 4-8
- "Add Title, Axis Labels, and Legend to Graph" on page 2-13

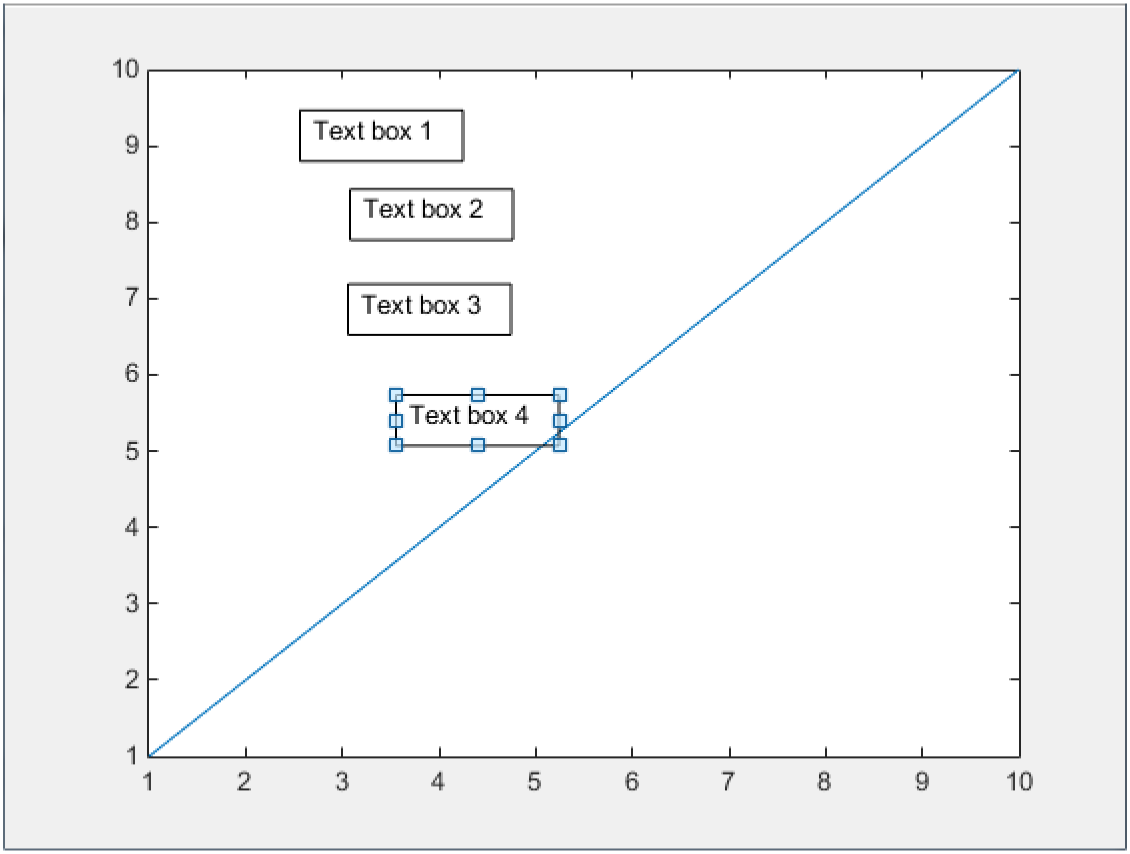
Align Objects in Graph Using Alignment Tools

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

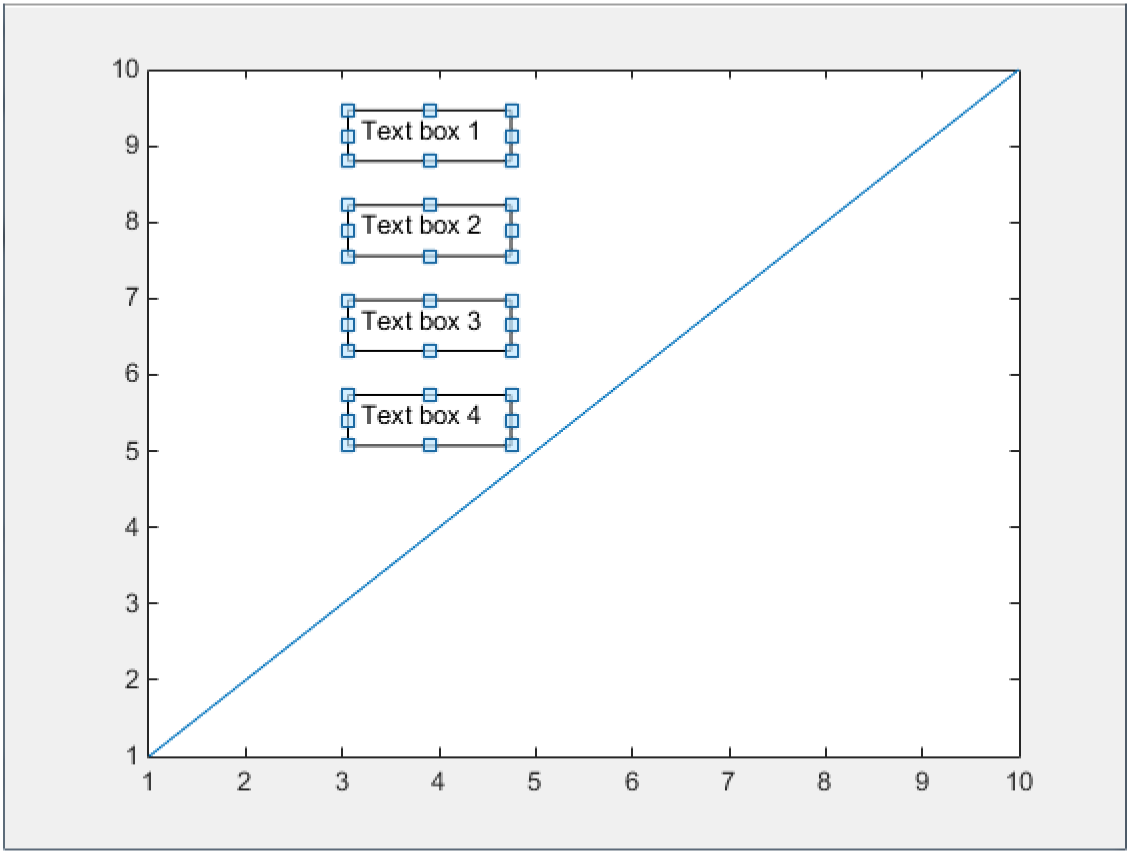
Plot a line.

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

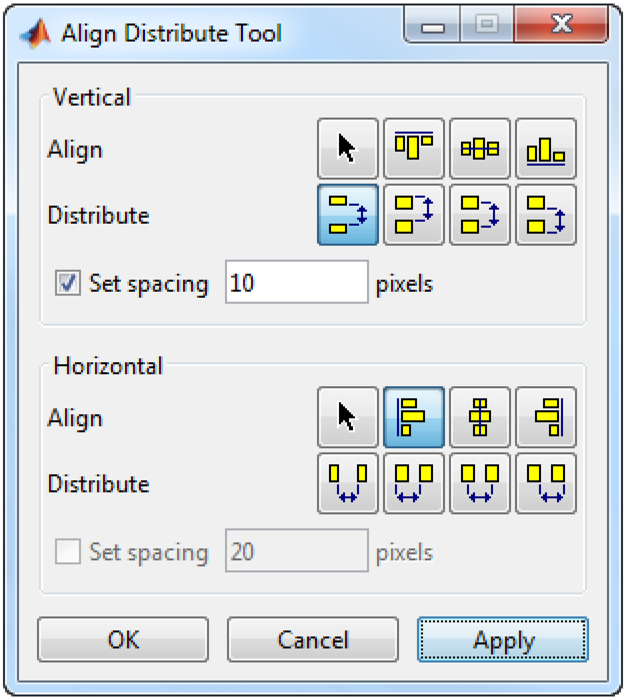
Add four text box annotations to the graph. In the figure, select **Insert** > **TextBox**. Approximately align the text boxes in a vertical column.



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.



Vertical

Align

Distribute

☑ Set spacing 10 pixels

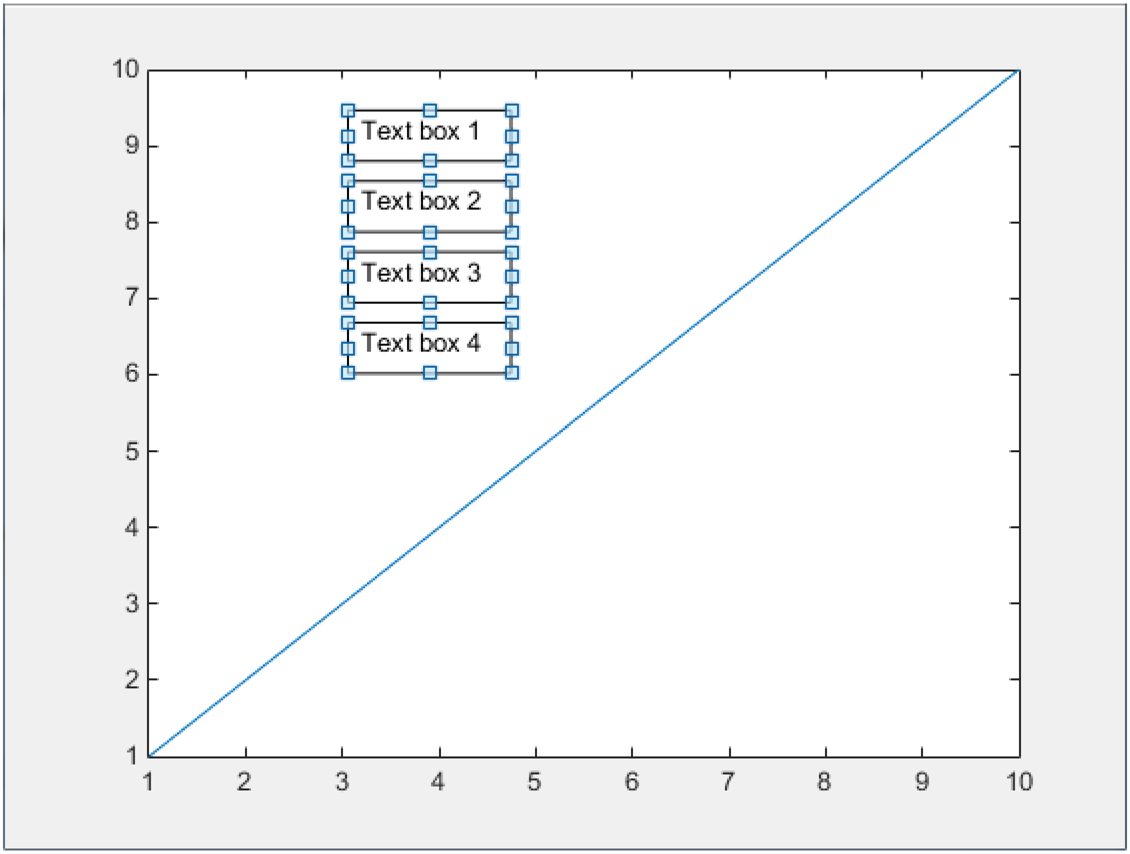
Horizontal

Align

Distribute

□ Set spacing 20 pixels

The text boxes align according to your alignment settings.



See Also

`annotation`

More About

- "Add Text to Graph Interactively" on page 4-33
- "Add Annotations to Graph Interactively" on page 4-30

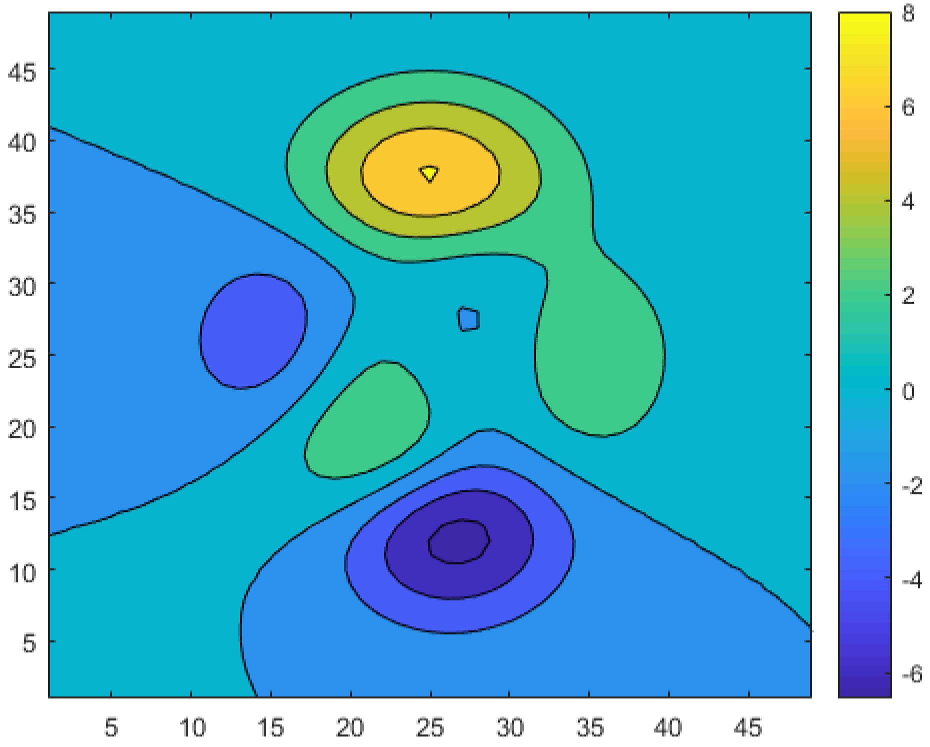
Coloring Graphs

- “Creating Colorbars” on page 5-2
- “Change Color Scheme Using a Colormap” on page 5-10
- “How Surface Plot Data Relates to a Colormap” on page 5-16
- “How Image Data Relates to a Colormap” on page 5-22
- “How Patch Data Relates to a Colormap” on page 5-30
- “Control Colormap Limits” on page 5-39
- “Differences Between Colormaps and Truecolor” on page 5-45

Creating Colorbars

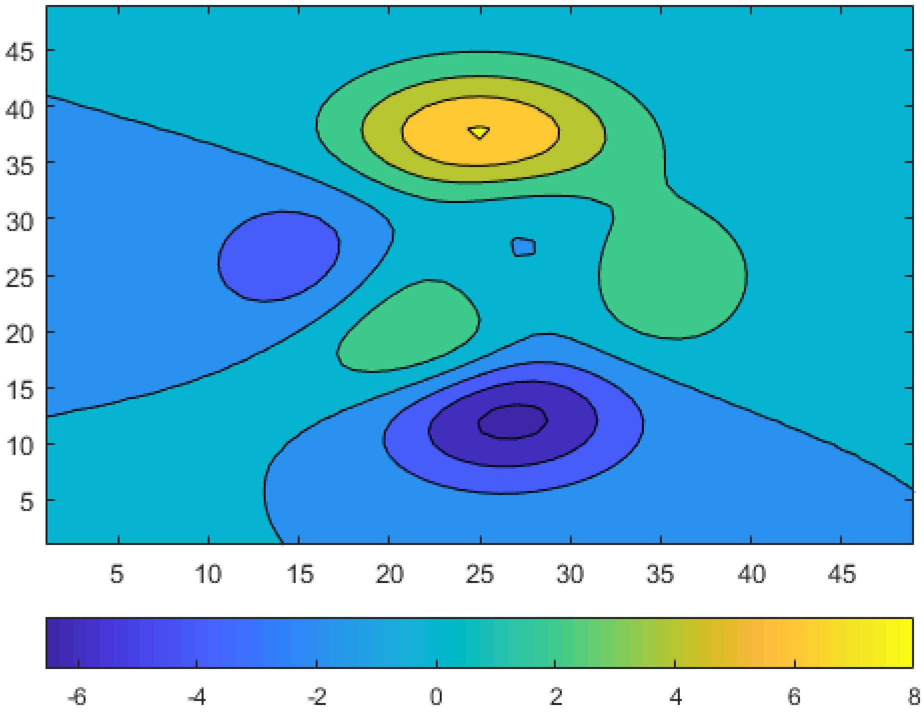
Colorbars allow you to see the relationship between your data and the colors displayed in your chart. After you have created a colorbar, you can customize different aspects of its appearance, such as its location, thickness, and tick labels. For example, this colorbar shows the relationship between the values of the peaks function and the colors shown in the plot next to it.

```
contourf(peaks)
c = colorbar;
```



The default location of the colorbar is on the right size of the axes. However, you can move the colorbar to a different location by setting the Location property. In this case, the `'southoutside'` option places the colorbar below the axes.

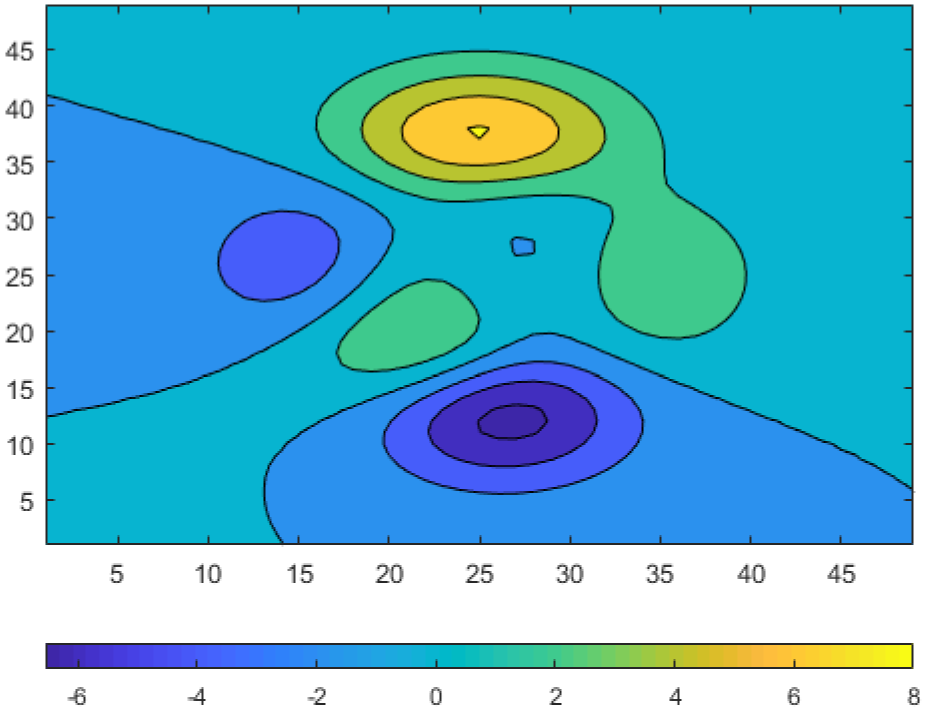
```
c.Location = 'southoutside';
```



You can also change the thickness of the colorbar. The Position property controls the location and size of most graphics objects, including axes and colorbars. Because this colorbar is horizontal, the fourth value in `c.Position` (which corresponds to height) controls its thickness. Here, the colorbar is narrowed and the axes position is reset so that there is no overlap with the colorbar.

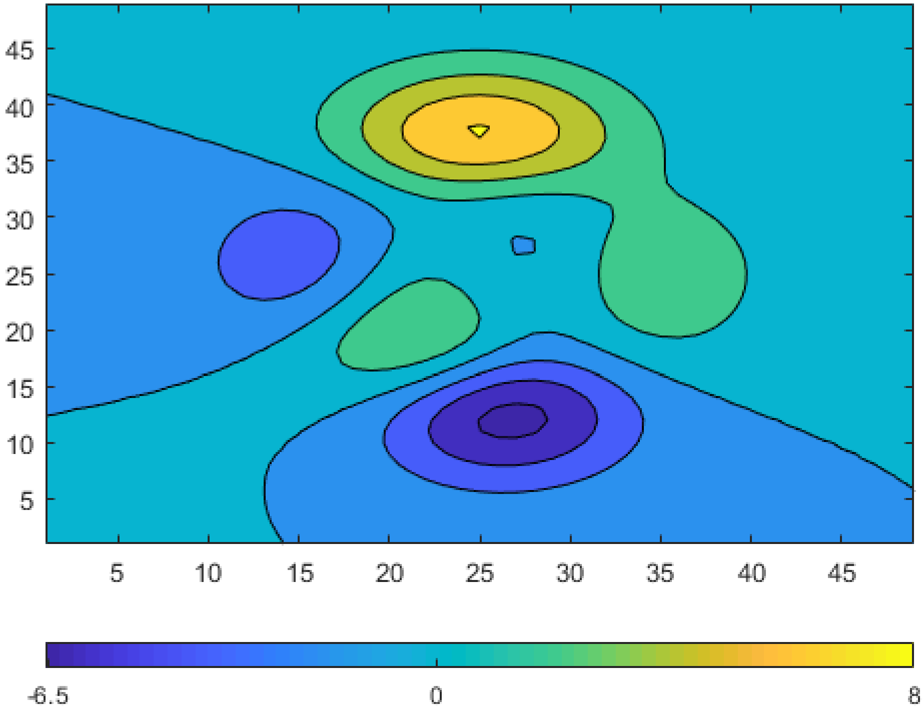
```
ax = gca;
```

```
axpos = ax.Position;
c.Position(4) = 0.5*c.Position(4);
ax.Position = axpos;
```



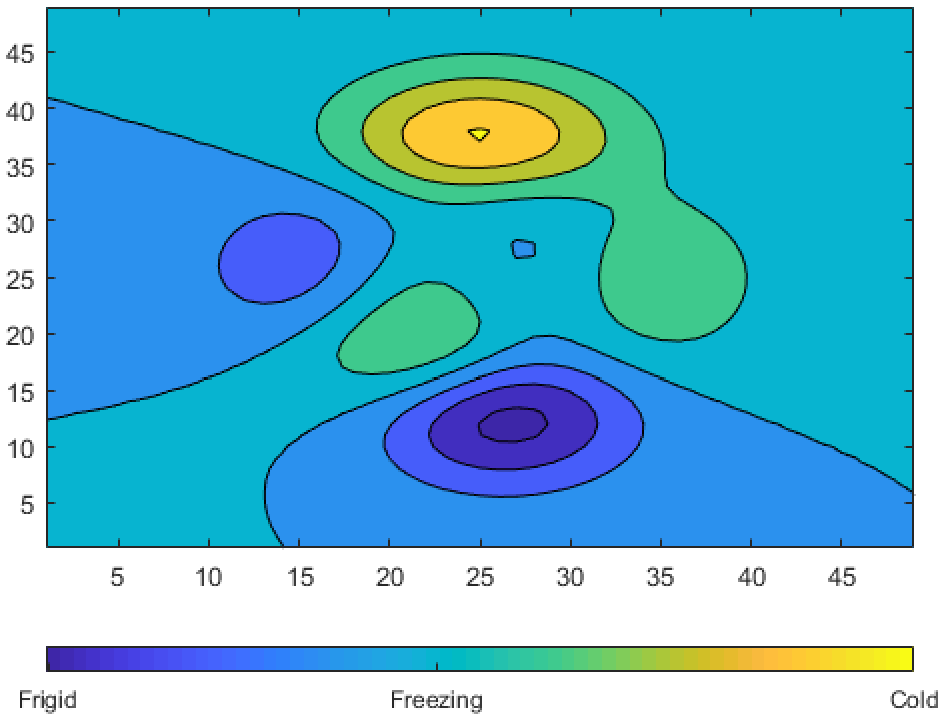
`Colorbar` objects have several properties for modifying the tick spacing and labels. For example, you can specify that the ticks occur in only three places: `–6.5`, `0`, and `8`.

```
c.Ticks = [-6.5 0 8];
```



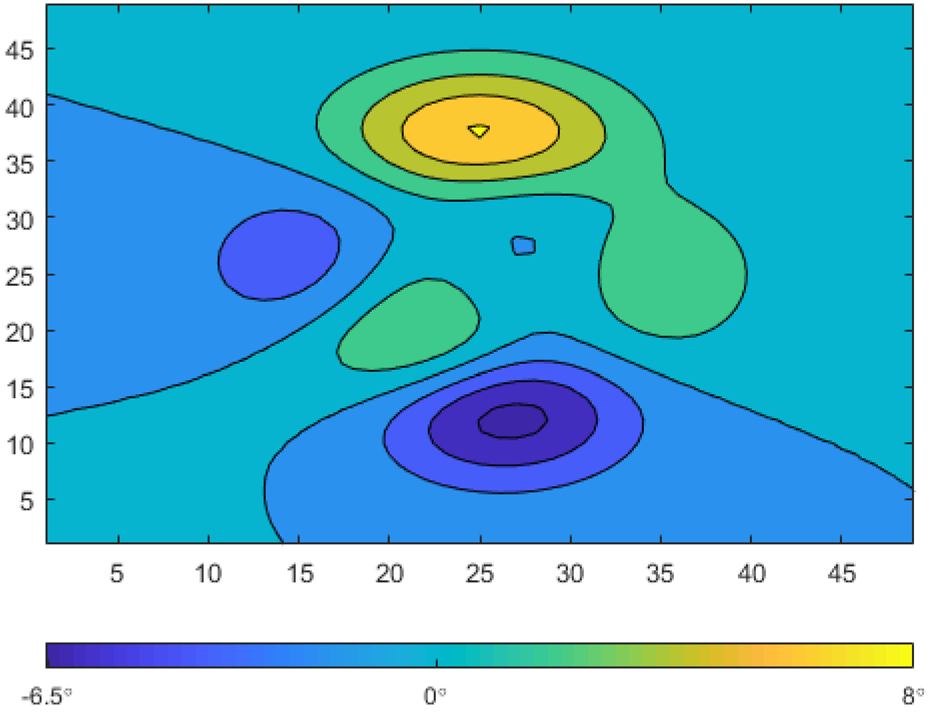
You can change the tick labels to any values. Use a cell array to specify the tick labels.

```
c.TickLabels = {'Frigid','Freezing','Cold'};
```



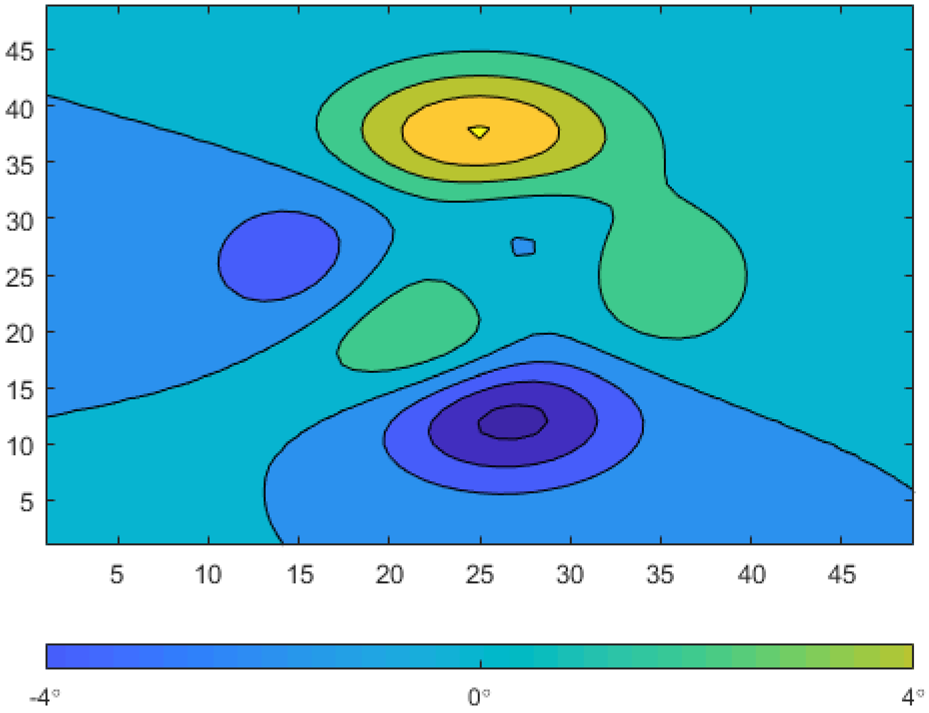
You can also use TeX or LaTeX markup in the tick labels. Use the TickLabelInterpreter property to set the interpreter when you use TeX or LaTeX.

```
c.TickLabelInterpreter = 'tex';
c.TickLabels = {'-6.5\circ','0\circ','8\circ'};
```



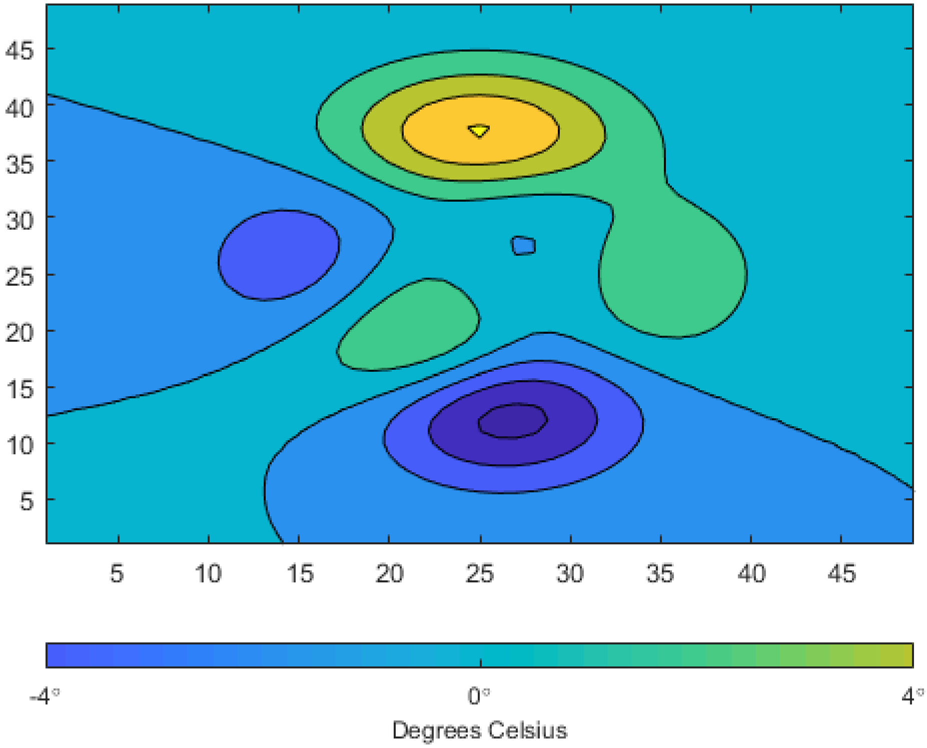
You can change the limits of the colorbar to focus on a specific region of color. For example, you can narrow the limits and adjust the tick labels to reflect the new limits. The resulting colorbar excludes the dark blue shades that used to be on the left and the yellow shades that used to be on the right.

```
c.Limits = [-4 4];
c.Ticks = [-4 0 4];
c.TickLabels = {'-4\circ','0\circ','4\circ'};
```



Add a descriptive label to the colorbar using the Label property. Because the `Label` property must be specified as a `Text` object, you must set the `String` property of the `Text` object first. Then you can assign that `Text` object to the `Label` property. The following command accomplishes both tasks in one step.

```
c.Label.String = 'Degrees Celsius';
```



See Also

Functions

`colorbar` | `pcolor`

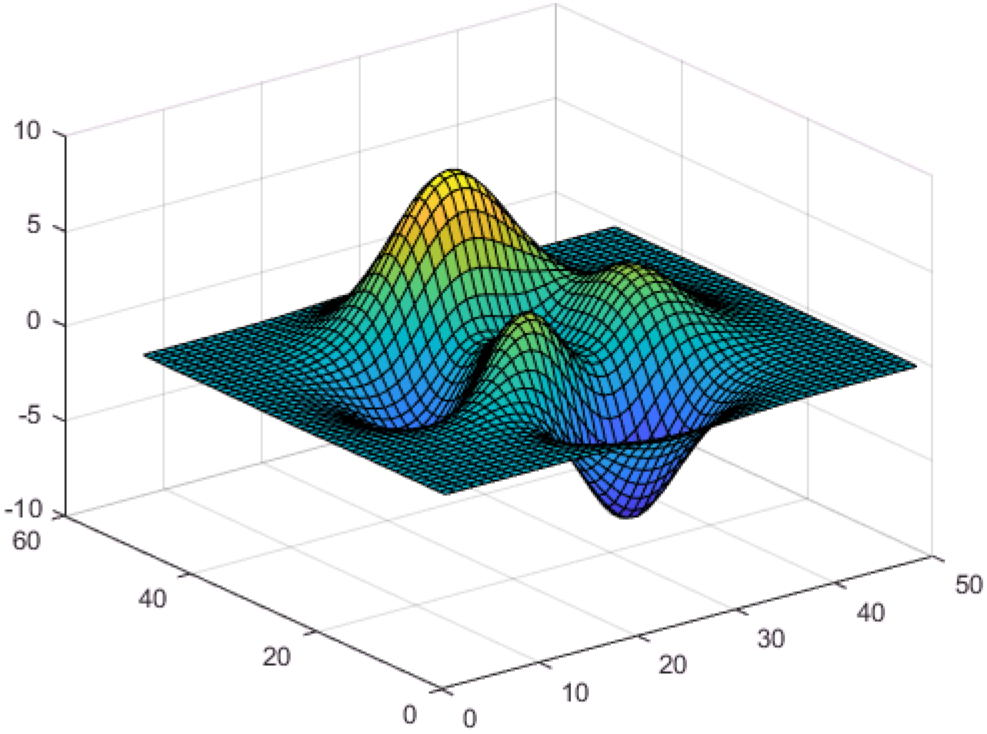
Properties

Colorbar Properties

Change Color Scheme Using a Colormap

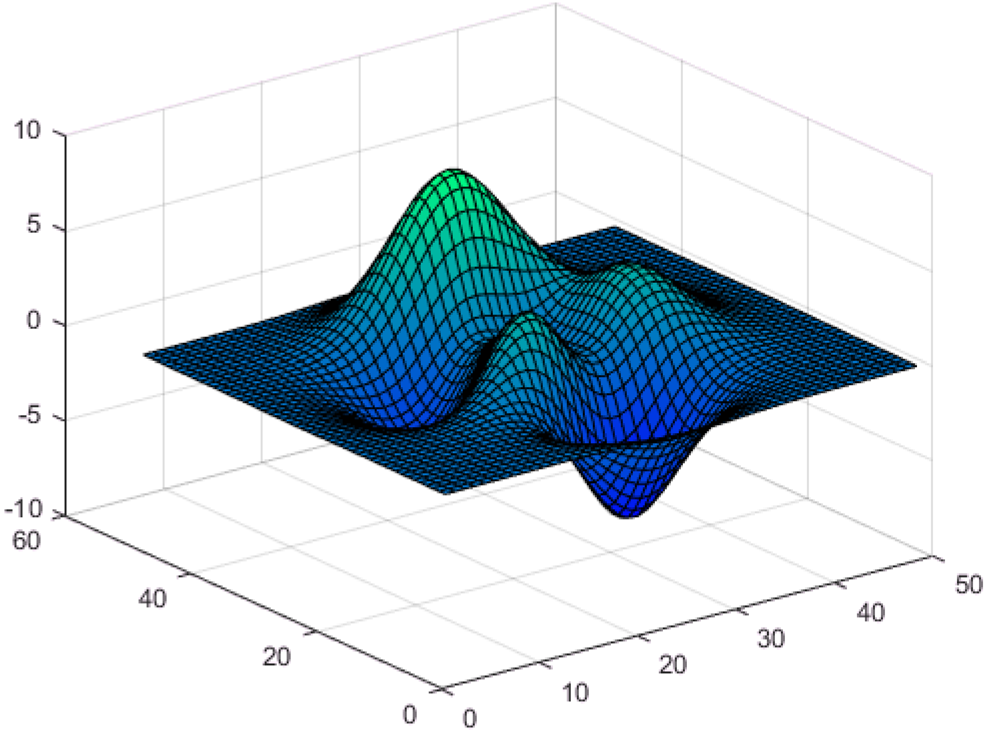
MATLAB® uses a default color scheme when it displays visualizations such as surface plots. You can change the color scheme by specifying a colormap. For example, here is a surface plot with the default color scheme.

```
f = figure;
surf(peaks);
```



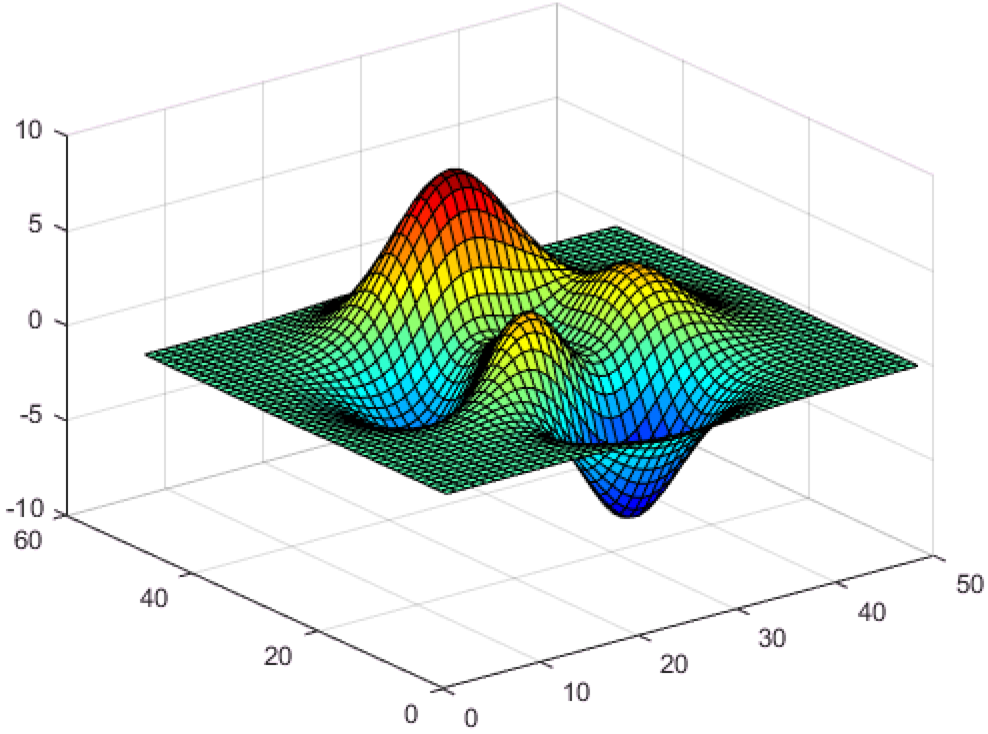
The following command changes the colormap of the current figure to winter, one of several predefined colormaps (see “Colormaps” for a full list).

```
colormap winter;
```



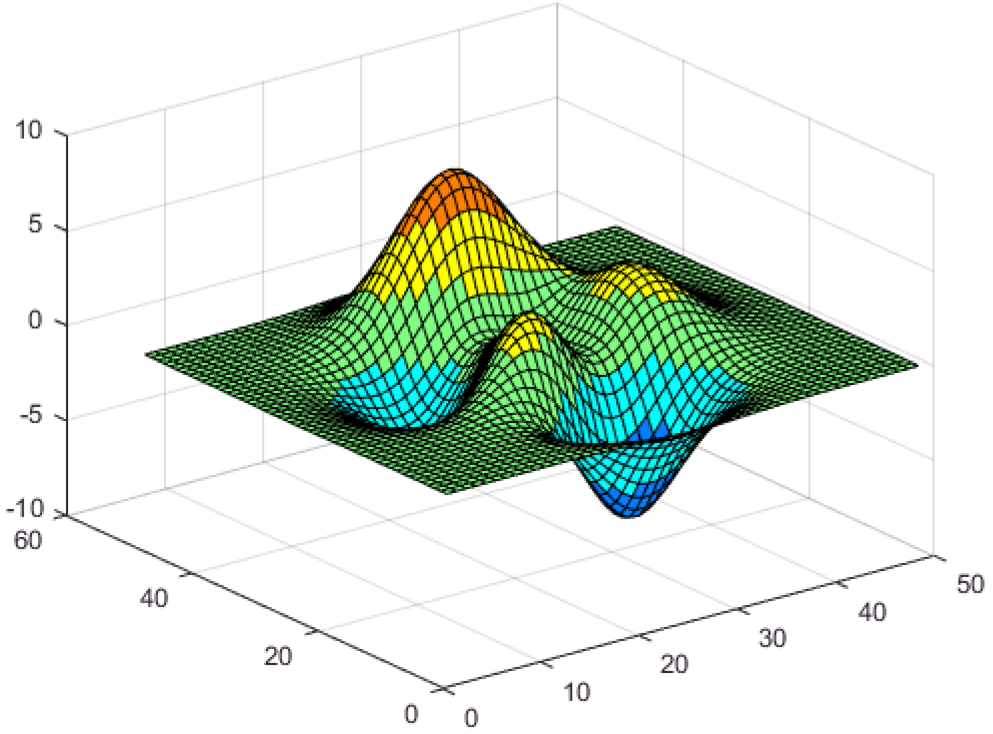
If you have multiple figures open, pass the `Figure` object as the first argument to the `colormap` function.

```
colormap(f,jet);
```



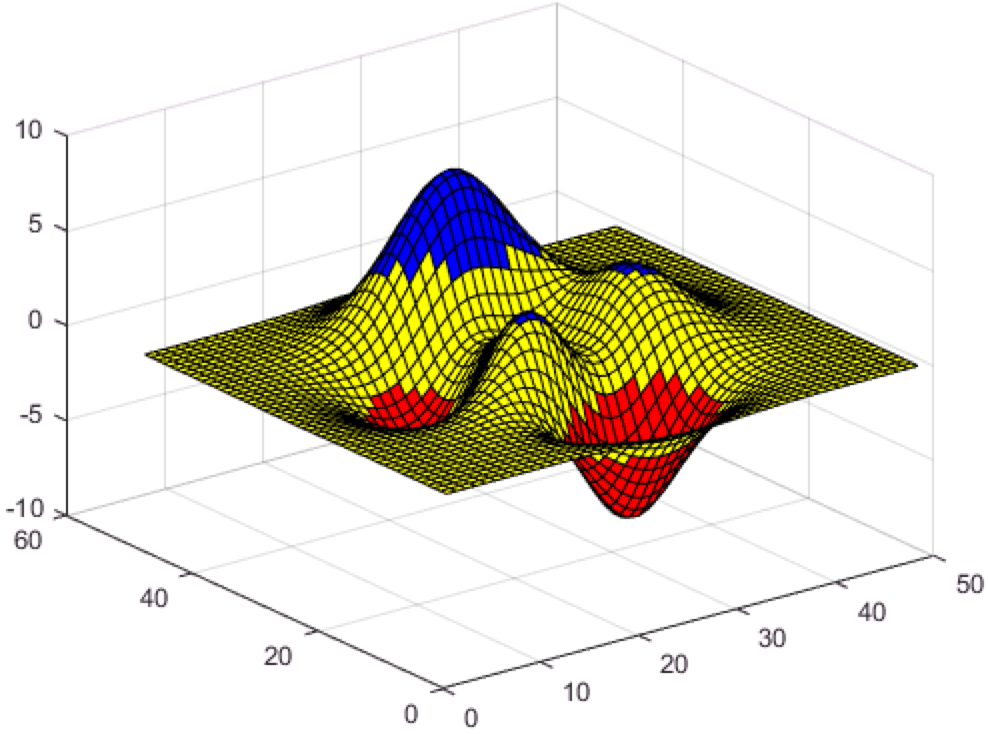
Each predefined colormap provides a palette of 64 colors by default. However, you can specify any number of colors by passing a whole number to the predefined colormap function. For example, here is the jet colormap with five entries.

```
c = jet(5);
colormap(c);
```



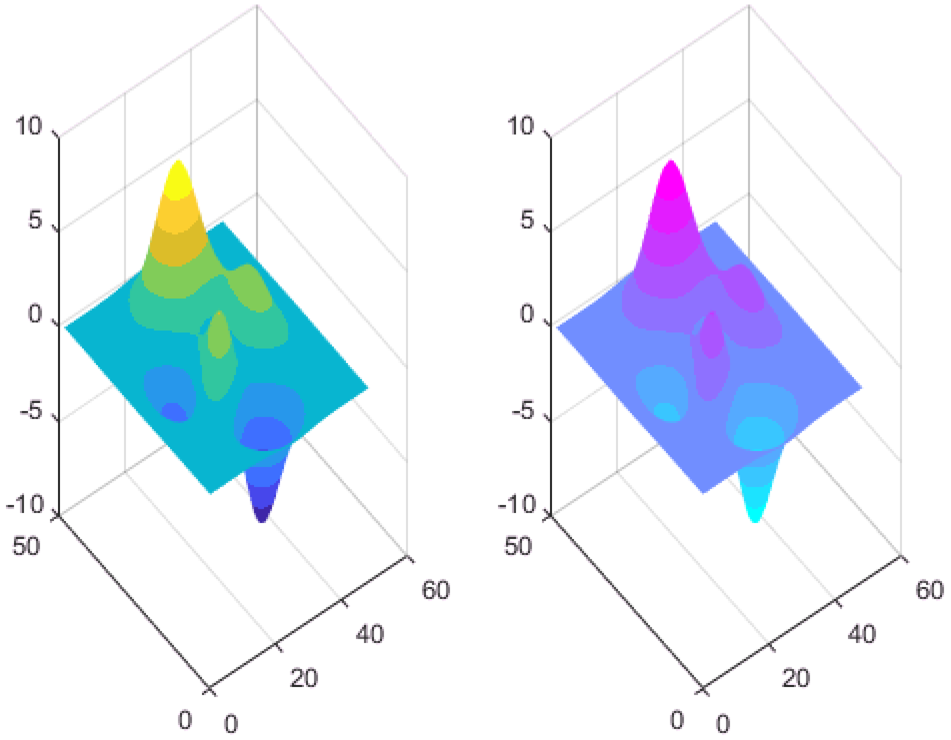
You can also create your own colormap as an m-by-3 array. Each row in the array contains the red, green, and blue intensities of a different color. The intensities are in the range [0,1]. Here is a simple colormap that contains three entries.

```
mycolors = [1 0 0; 1 1 0; 0 0 1];
colormap(mycolors);
```



If you are working with subplots, you can assign a different colormap to each subplot by passing the axes to the `colormap` function.

```
ax1 = subplot(1,2,1);
surf(peaks);
shading interp;
colormap(parula(10));
ax2 = subplot(1,2,2);
surf(peaks);
shading interp;
colormap(ax2,cool(10));
```



Related Examples

- "How Surface Plot Data Relates to a Colormap" on page 5-16

How Surface Plot Data Relates to a Colormap

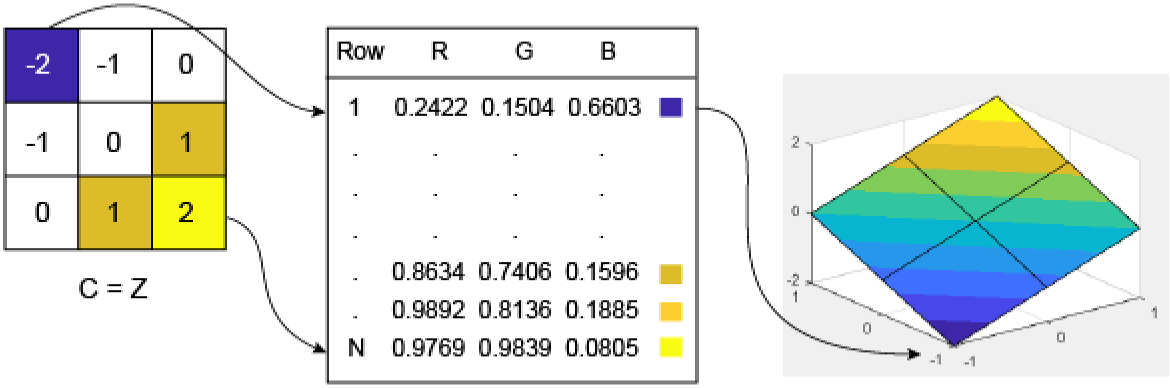
When you create surface plots using functions such as `surf` or `mesh`, you can easily change the color scheme by calling the `colormap` function. After you have selected a color scheme, you can change the direction or pattern of the colors across the surface.

Relationship Between the Surface and the Colormap

The `CData` property of a `Surface` object contains an indexing array `C` that associates specific locations in your plot with colors in the colormap. `C` has the following relationship to the surface $z = f(x,y)$:

- `C` is the same size as `Z`, where `Z` is the array containing the values of $f(x,y)$ at each grid point on the surface.
- The value at `C(i,j)` controls the color at the grid location `(i,j)` on the surface.
- By default, `C` is equal to `Z`, which corresponds to colors varying with altitude.
- By default, the range of `C` maps linearly to the number of rows in the colormap array.

For example, a 3-by-3 sampling of `Z = X + Y` has the following relationship to a colormap containing `N` entries.



-2	-1	0
-1	0	1
0	1	2

C = Z

Row	R	G	B	
1	0.2422	0.1504	0.6603	■
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	■
.	0.9892	0.8136	0.1885	■
N	0.9769	0.9839	0.0805	■

Colormap

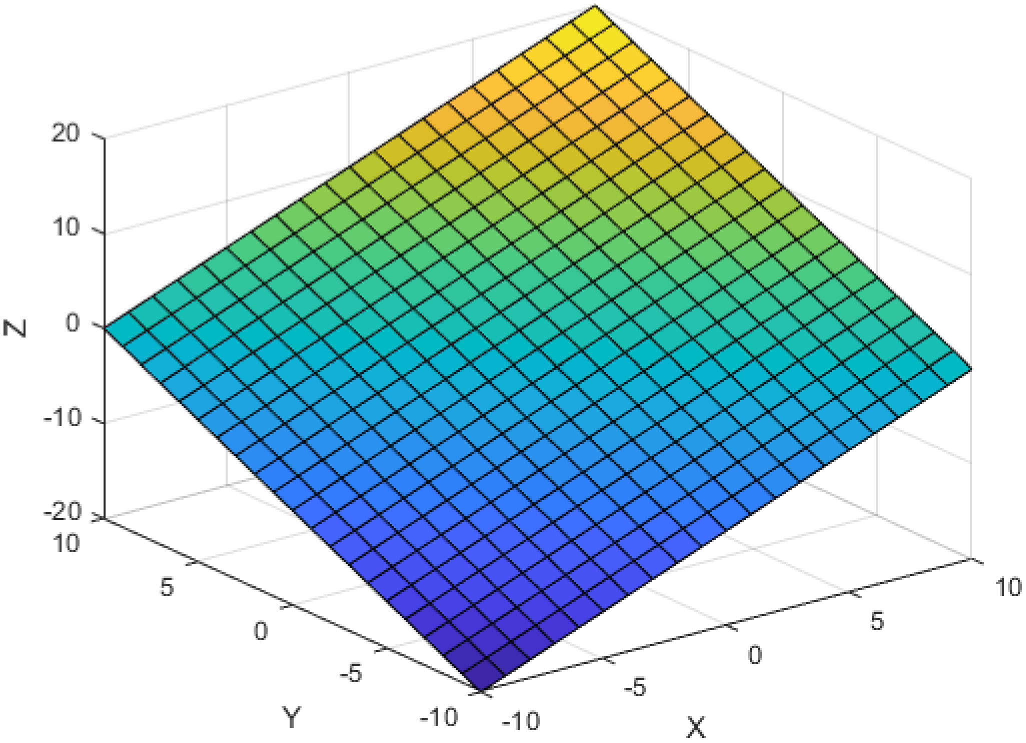
Notice that the smallest value (-2) maps to the first row in the colormap. The largest value (2) maps to the last row in the colormap. The intermediate values in C map linearly to the intermediate rows in the colormap.

Note: The preceding surface plot shows how colors are assigned to vertices on the surface. However, the default behavior is to fill the patch faces with solid color. That solid color is based on the colors assigned to the surrounding vertices. For more information, see the FaceColor property description.

Change the Direction or Pattern of Colors

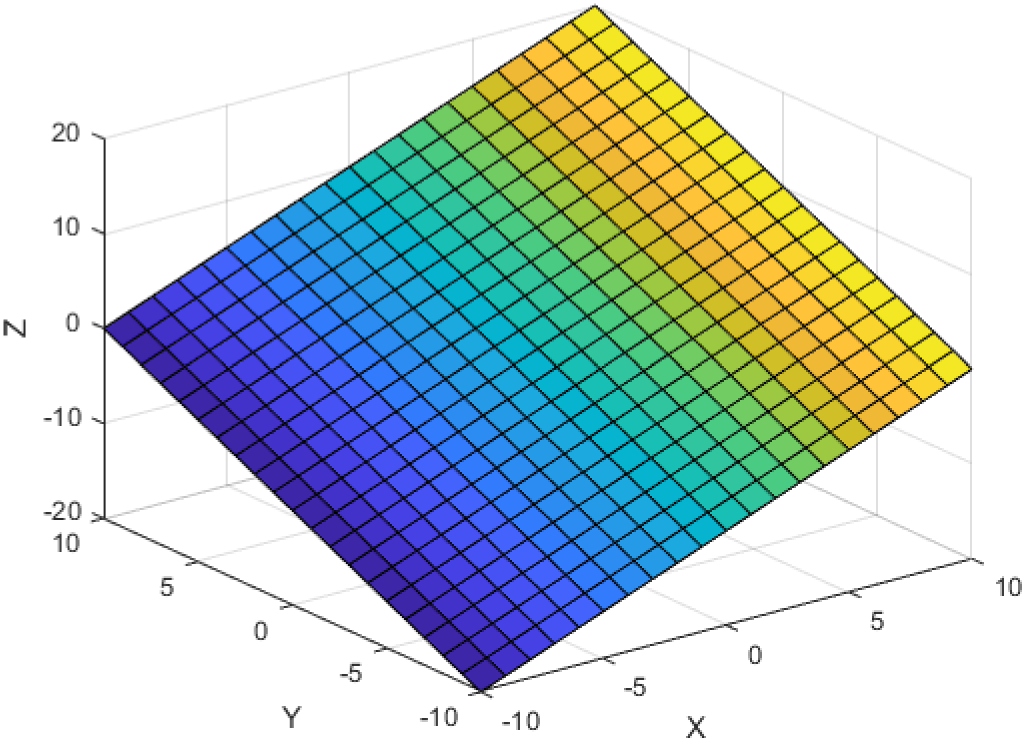
When using the default value of C=Z, the colors vary with changes in Z.

```
[X,Y] = meshgrid(-10:10);
Z = X + Y;
s = surf(X,Y,Z);
xlabel('X');
ylabel('Y');
zlabel('Z');
```



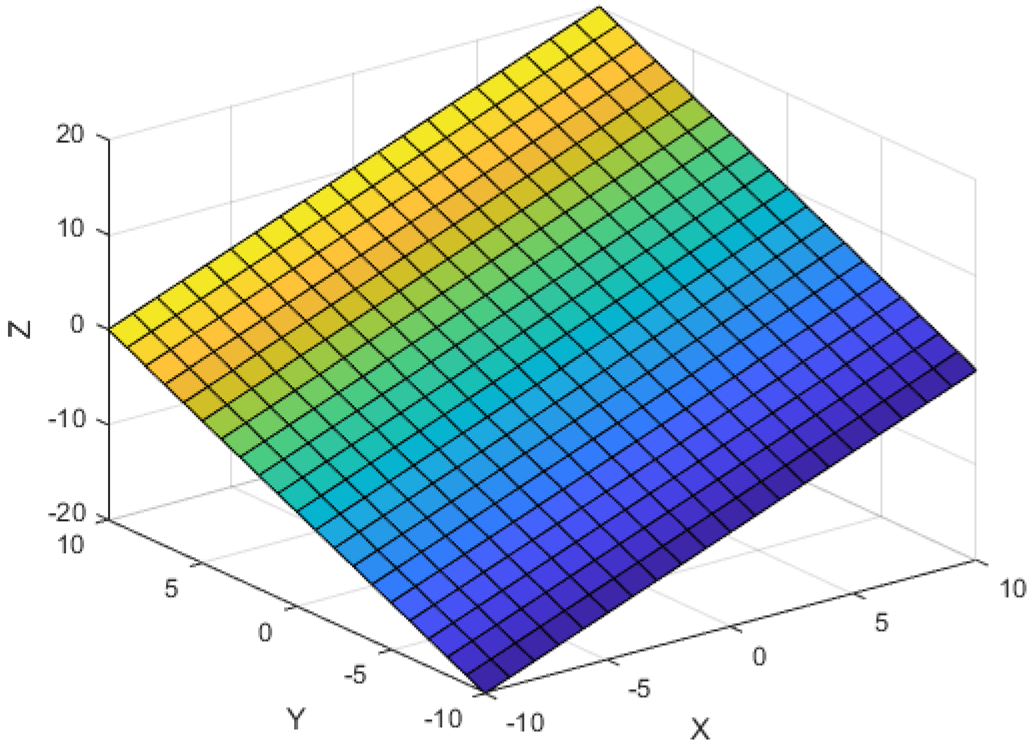
You can change this behavior by specifying `C` when you create the surface. For example, the colors on this surface vary with `X`.

```
C = X;
s = surf(X,Y,Z,C);
xlabel('X');
ylabel('Y');
zlabel('Z');
```



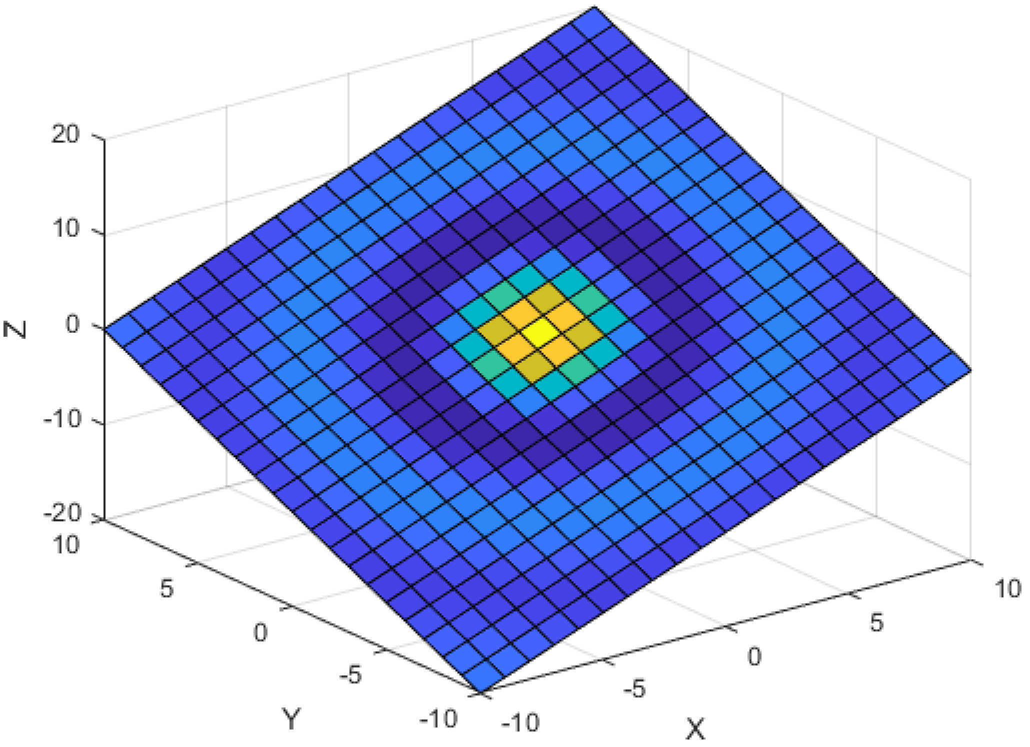
Alternatively, you can set the `CData` property directly. This command makes the colors vary with `Y`.

```
s.CData = Y;
```



The colors do not need to follow changes in a single dimension. In fact, `CData` can be *any* array that is the same size as `Z`. For example, the colors on this plane follow the shape of a sinc function.

```
R = sqrt(X.^2 + Y.^2) + eps;
s.CData = sin(R)./(R);
```



See Also

Functions

`mesh` | `surf`

Properties

Chart Surface Properties

Related Examples

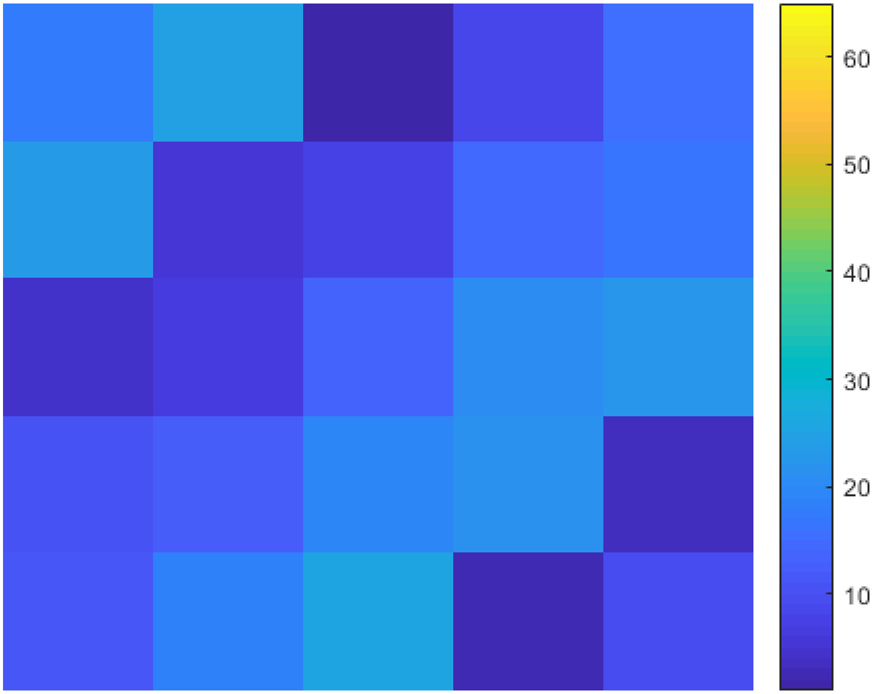
- “Change Color Scheme Using a Colormap” on page 5-10
- “Differences Between Colormaps and Truecolor” on page 5-45

How Image Data Relates to a Colormap

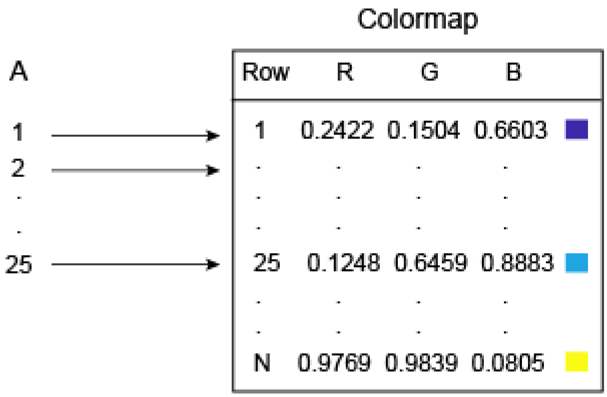
When you display images using the `image` function, you can control how the range of pixel values maps to the range of the colormap. For example, here is a 5-by-5 magic square displayed as an image using the default colormap.

```
A = magic(5)
im = image(A);
axis off
colorbar
```

```
A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```



A contains values between 1 and 25. MATLAB treats those values as indices into the colormap, which has 64 entries. Thus, all the pixels in the preceding image map to the first 25 entries in the colormap (roughly the blue region of the colorbar).

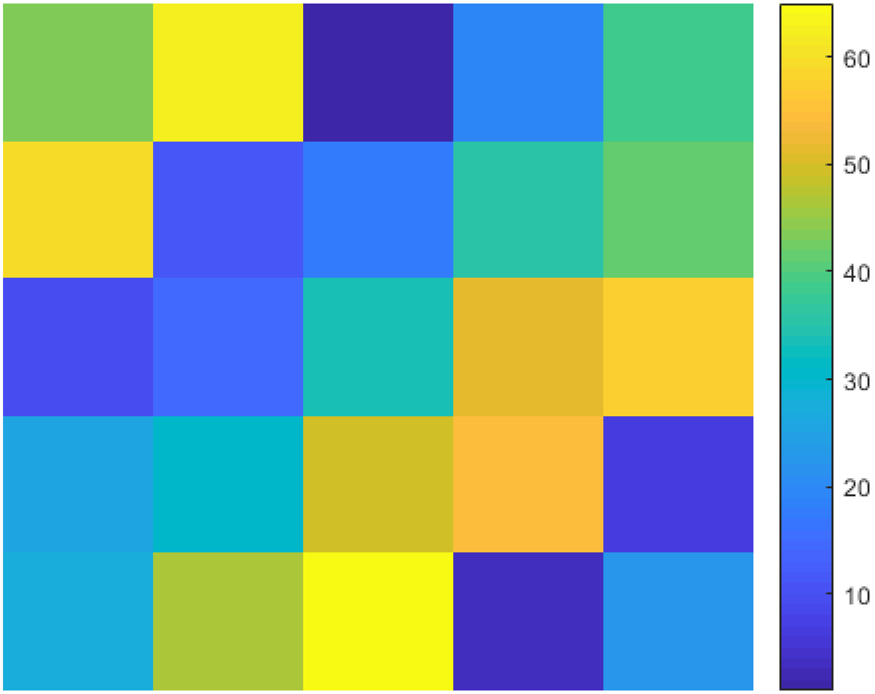


Colormap

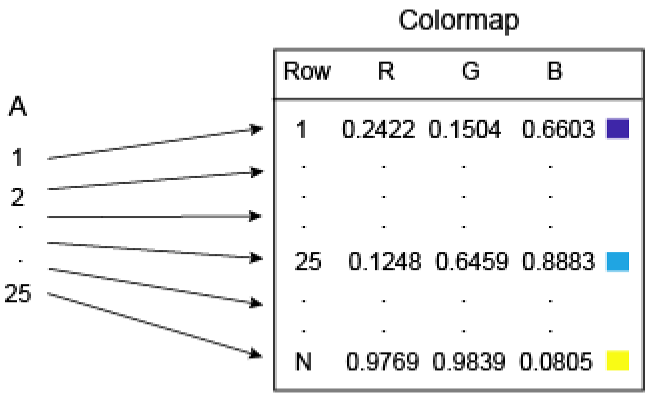
Row	R	G	B	
1	0.2422	0.1504	0.6603	🟪
.	.	.	.	
.	.	.	.	
.	.	.	.	
25	0.1248	0.6459	0.8883	🟦
.	.	.	.	
.	.	.	.	
N	0.9769	0.9839	0.0805	🟨

You can control this mapping with the `CDataMapping` property of the `Image` object. The default behavior shown in the preceding diagram corresponds to the `'direct'` option for this property. Direct mapping is useful when you are displaying images (such as GIF images) that contain their own colormap. However, if your image represents measurements of some physical unit (e.g., meters or degrees) then set the `CDataMapping` property to `'scaled'`. Scaled mapping uses the full range of colors, and it allows you to visualize the relative differences in your data.

```
im.CDataMapping = 'scaled';
```



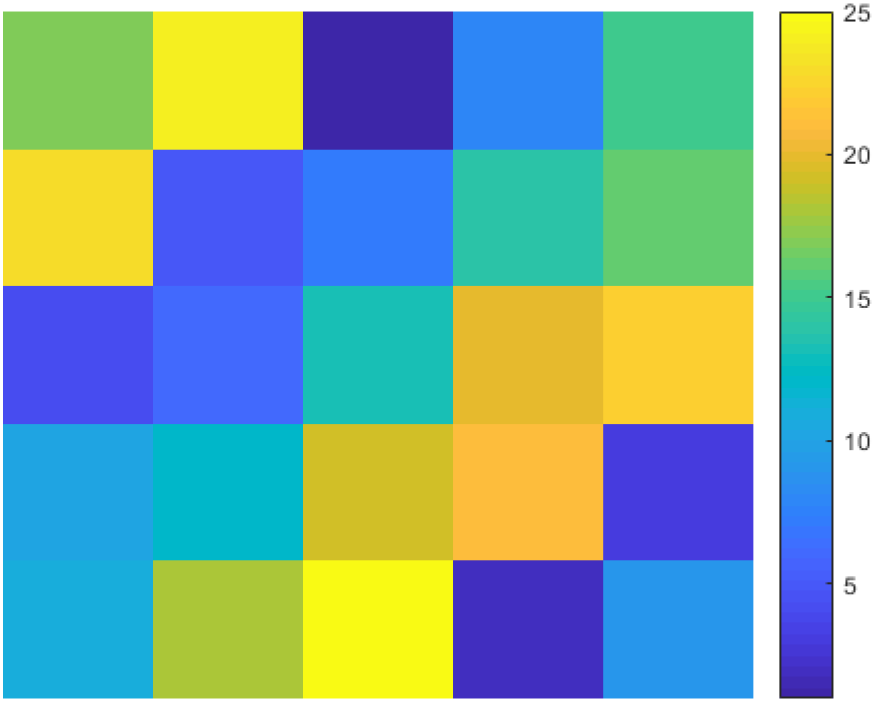
The `'scaled'` option maps the smallest value in `A` to the first entry in the colormap, and maps largest value in `A` maps to the last entry in the colormap. All intermediate values of `A` are linearly scaled to the colormap.



Row	R	G	B	
1	0.2422	0.1504	0.6603	🟪
.	.	.	.	
.	.	.	.	
.	.	.	.	
25	0.1248	0.6459	0.8883	🟦
.	.	.	.	
.	.	.	.	
N	0.9769	0.9839	0.0805	🟨

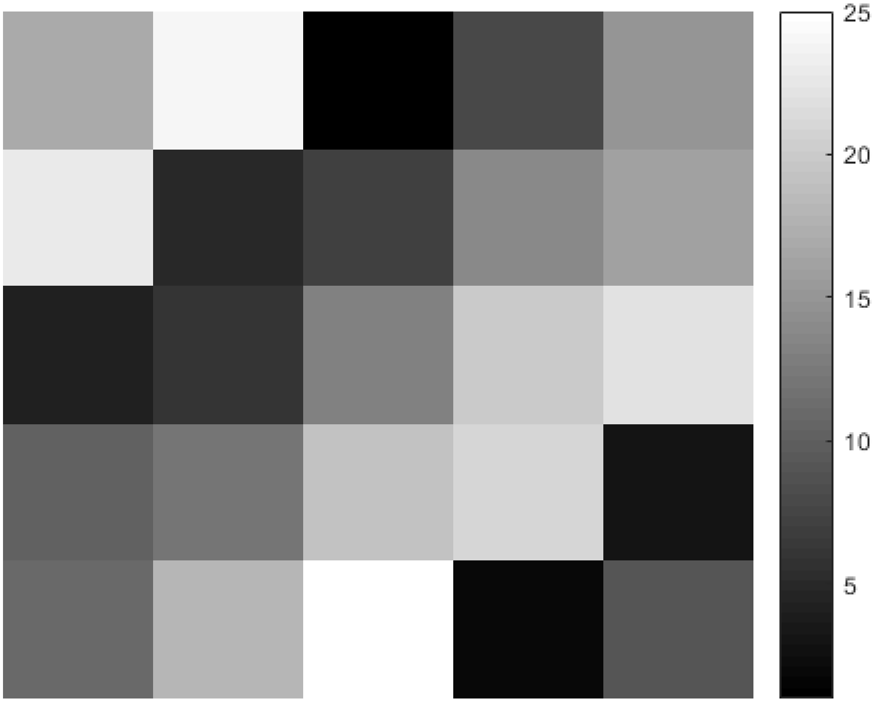
As an alternative to setting the `CDataMapping` property to `'scaled'`, you can call the `imagesc` function to get the same effect.

```
imagesc(A);
axis off
colorbar
```



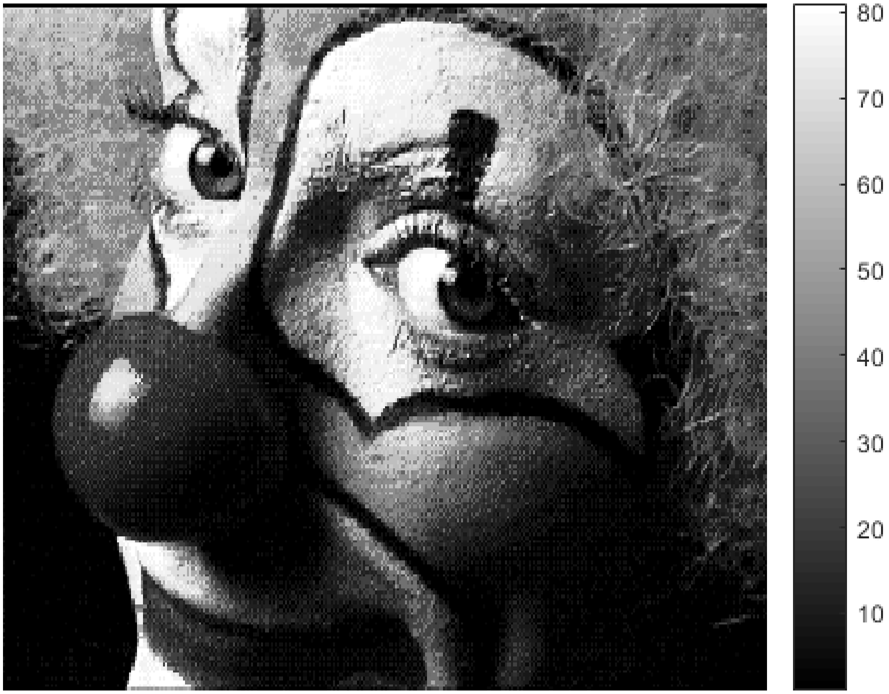
If you change the colormap, the values in A are scaled to the new colormap.

```
colormap(gray);
```



Scaled mapping is also useful for displaying pictorial images that have no colormap, or when you want to change the colormap for a pictorial image. The following commands display an image using the `gray` colormap, which is different than the original colormap that is stored with this image.

```
load clown
image(X,'CDataMapping','scaled');
colormap(gray);
axis off
colorbar
```



See Also

Functions

`image` | `imagesc`

Properties

Image Properties

Related Examples

- "Image Types" on page 7-5
- "Differences Between Colormaps and Truecolor" on page 5-45

How Patch Data Relates to a Colormap

When you create graphics that use `Patch` objects, you can control the overall color scheme by calling the `colormap` function. You can also control the relationship between the colormap and your patch by:

- Assigning specific colors to the faces
- Assigning specific colors to the vertices surrounding each face

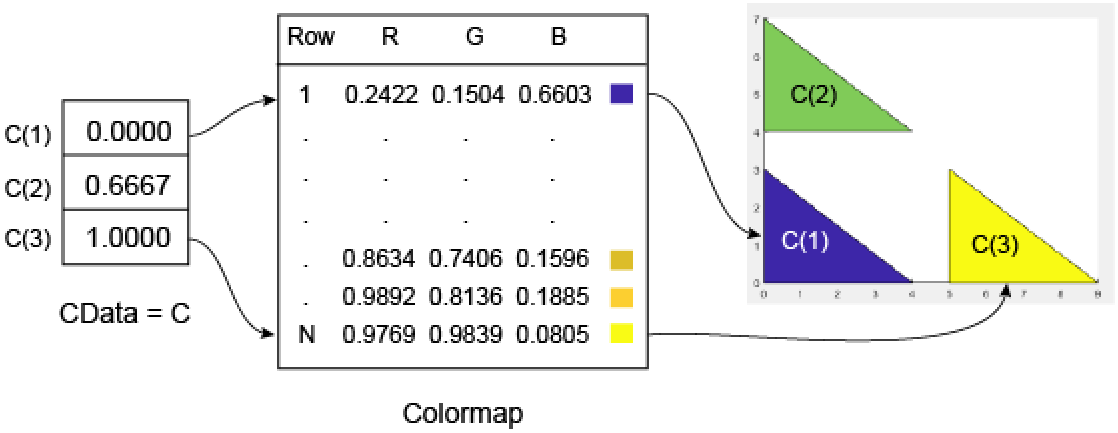
The way you control these relationships depends on how you specify your patches: as x-, y-, and z-coordinates, or as face-vertex data.

Relationship of the Colormap to *x*-, *y*-, and *z*-Coordinate Arrays

If you create a `Patch` object using *x*-, *y*-, and *z*-coordinate arrays, the `CData` property of the `Patch` object contains an indexing array `C`. This array controls the relationship between the colormap and your patch. To assign colors to the faces, specify `C` as an array with these characteristics:

- `C` is an n-by-1 array, where n is the number of faces.
- The value at `C(i)` controls the color for face `i`.

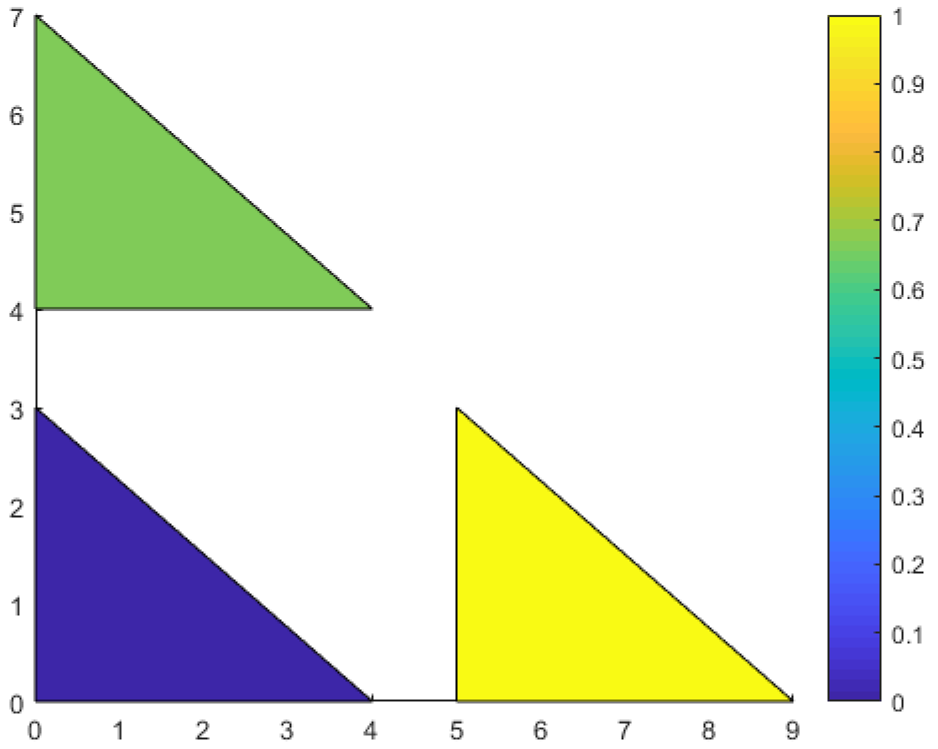
Here is an example of `C` and its relationship to the colormap and three faces. The value of `C(i)` controls the color for the face defined by vertices (`X(i,:)`, `Y(i,:)`).



Row	R	G	B	
1	0.2422	0.1504	0.6603	🟪
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	🟨
.	0.9892	0.8136	0.1885	🟧
N	0.9769	0.9839	0.0805	🟨

The smallest value in C is 0. It maps to the first row in the colormap. The largest value in C is 1, and it maps to the last row in the colormap. Intermediate values of C map linearly to the intermediate rows in the colormap. In this case, C(2) maps to the color located about two-thirds from the beginning of the colormap. This code creates the Patch object described in the preceding illustration.

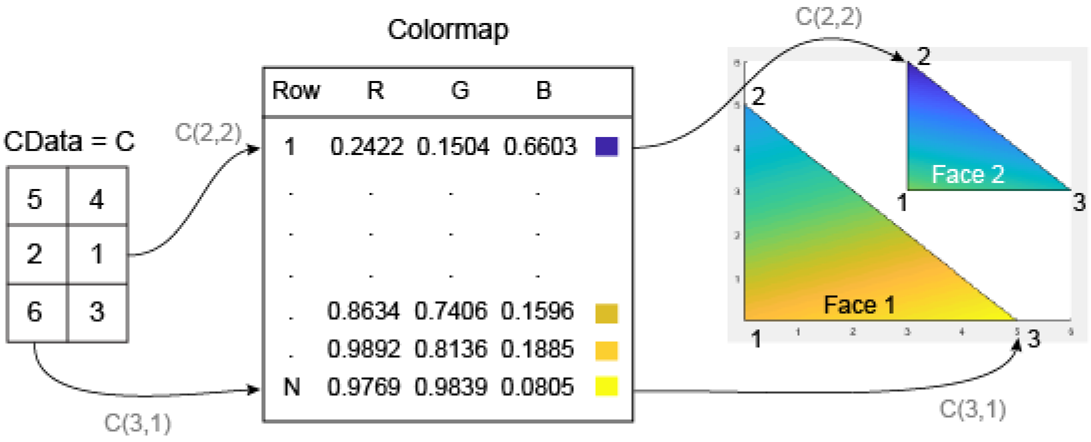
```
X = [0 0 5; 0 0 5; 4 4 9];
Y = [0 4 0; 3 7 3; 0 4 0];
C = [0; .6667; 1];
p = patch(X,Y,C);
colorbar
```



To assign colors to the vertices, specify `C` as an array with these characteristics:

- `C` is an m-by-n array, where m is the number of vertices per face, and n is the number of faces.
- The value at `C(i,j)` controls the color at vertex `i` of face `j`.

Here is an example of `C` and its relationship to the colormap and six vertices. The value of `C(i,j)` controls the color for the vertex at (`X(i,j)`, `Y(i,j)`).



CData = C

5	4
2	1
6	3

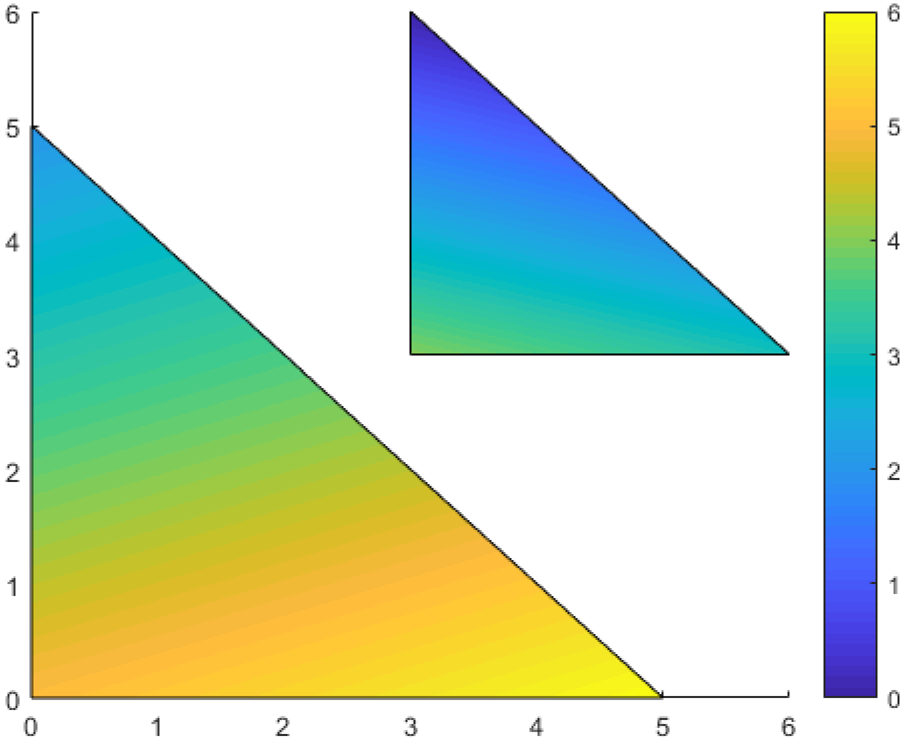
Colormap

Row	R	G	B	
1	0.2422	0.1504	0.6603	🟦
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	🟨
.	0.9892	0.8136	0.1885	🟨
N	0.9769	0.9839	0.0805	🟨

As with patch faces, MATLAB scales the values in `C` to the number of rows in the colormap. In this case, the smallest value is `C(2,2)=1`, and it maps to the first row in the colormap. The largest value is `C(3,1)=6`, and it maps to the last row in the colormap.

This code creates the `Patch` object described in the preceding illustration. The `FaceColor` property is set to `'interp'` to make the vertex colors blend across each face.

```
clf
X = [0 3; 0 3; 5 6];
Y = [0 3; 5 6; 0 3];
C = [5 4; 2 0; 6 3];
p = patch(X,Y,C,'FaceColor','interp');
colorbar
```



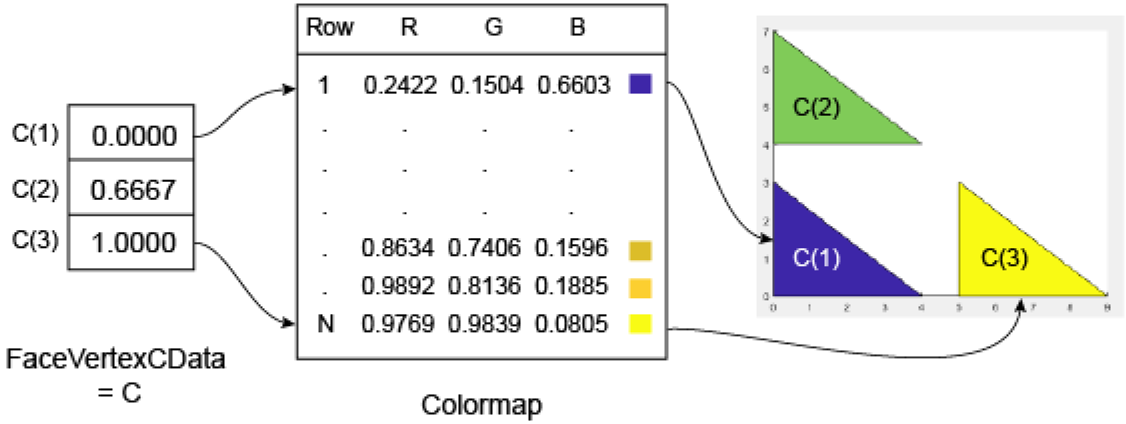
Relationship of the Colormap to Face-Vertex Data

If you create patches using face-vertex data, the `FaceVertexCData` property of the `Patch` object contains an indexing array `C`. This array controls the relationship between the colormap and your patch.

To assign colors to the faces, specify `C` as an array with these characteristics:

- `C` is an n-by-1 array, where n is the number of faces.
- The value at `C(i)` controls the color for face `i`.

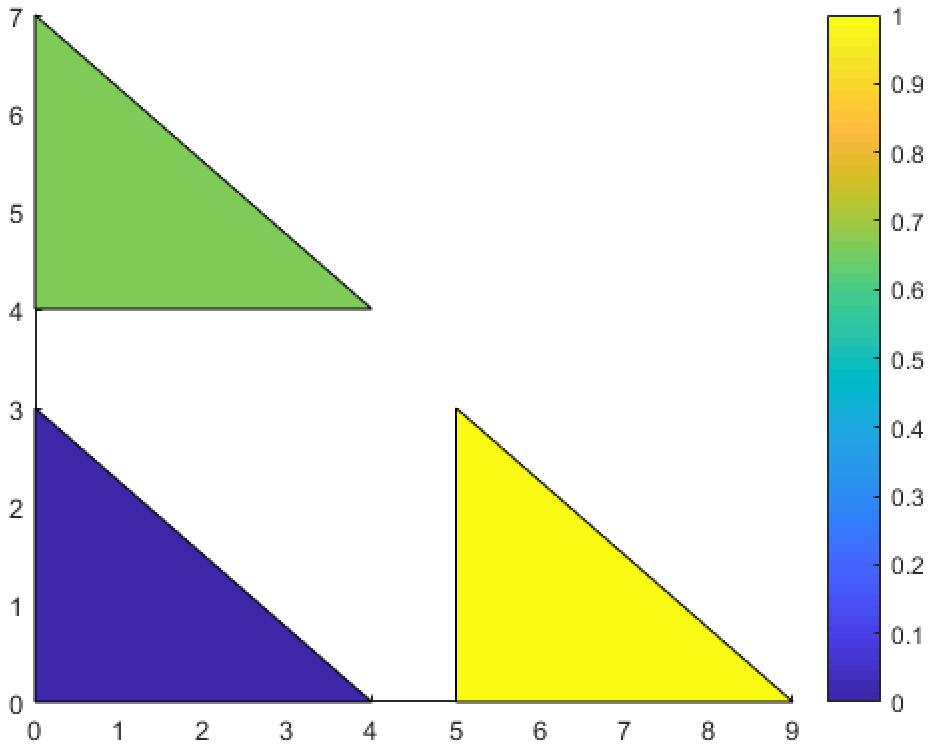
Here is an example of `C` and its relationship to the colormap and three faces.



The smallest value in C is 0, and it maps to the first row in the colormap. The largest value in C is 1, and it maps to the last value in the colormap. Intermediate values of C map linearly to the intermediate rows in the colormap. In this case, C(2) maps to the color located about two-thirds from the bottom of the colormap.

This code creates the Patch object described in the preceding illustration. The FaceColor property is set to 'flat' to display the colormap colors instead of the default color, which is black.

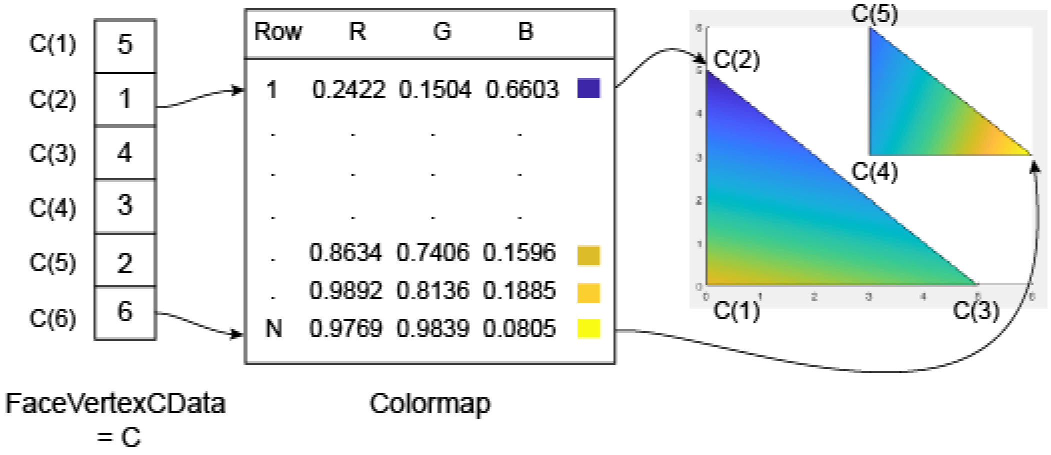
```
clf
vertices = [0 0; 0 3; 4 0; 0 4; 0 7; 4 4; 5 0; 5 3; 9 0];
faces = [1 2 3; 4 5 6; 7 8 9];
C = [0; 0.6667; 1];
p = patch('Faces',faces,'Vertices',vertices,'FaceVertexCData',C);
p.FaceColor = 'flat';
colorbar
```



To assign colors to the vertices, specify the `FaceVertexCData` property of the `Patch` object as array `C` with these characteristics:

- `C` is an n-by-1 array, where n is the number of vertices.
- The value at `C(i)` controls the color at vertex `i`.

Here is an example of `C` and its relationship to the colormap and six vertices.



C(1)	5
C(2)	1
C(3)	4
C(4)	3
C(5)	2
C(6)	6

FaceVertexCData = C

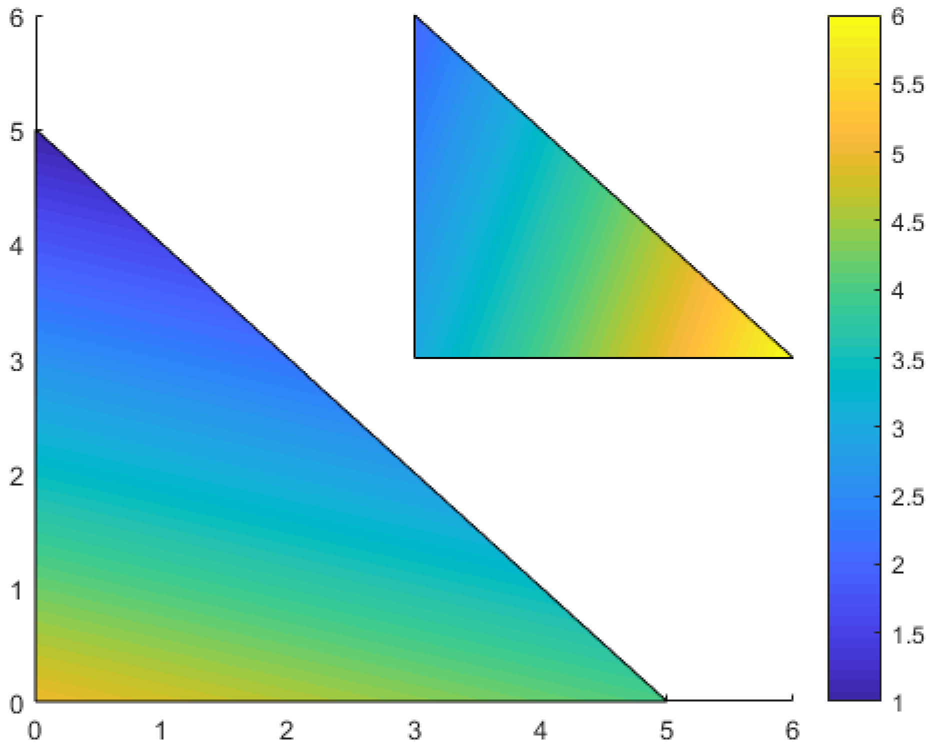
Row	R	G	B	
1	0.2422	0.1504	0.6603	🟦
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	🟨
.	0.9892	0.8136	0.1885	🟨
N	0.9769	0.9839	0.0805	🟨

Colormap

As with patch faces, MATLAB scales the values in `C` to the number of rows in the colormap. In this case, the smallest value is `C(2)=1`, and it maps to the first row in the colormap. The largest value is `C(6)=6`, and it maps to the last row in the colormap.

This code creates the `Patch` object described in the preceding illustration. The `FaceColor` property is set to `'interp'` to make the vertex colors blend across each face.

```
clf
vertices = [0 0; 0 5; 5 0; 3 3; 3 6; 6 3];
faces = [1 2 3; 4 5 6];
C = [5; 1; 4; 3; 2; 6];
p = patch('Faces',faces,'Vertices',vertices,'FaceVertexCData',C);
p.FaceColor = 'interp';
colorbar
```



See Also

Functions

`patch`

Properties

Patch Properties

Related Examples

- “Change Color Scheme Using a Colormap” on page 5-10
- “Differences Between Colormaps and Truecolor” on page 5-45

Control Colormap Limits

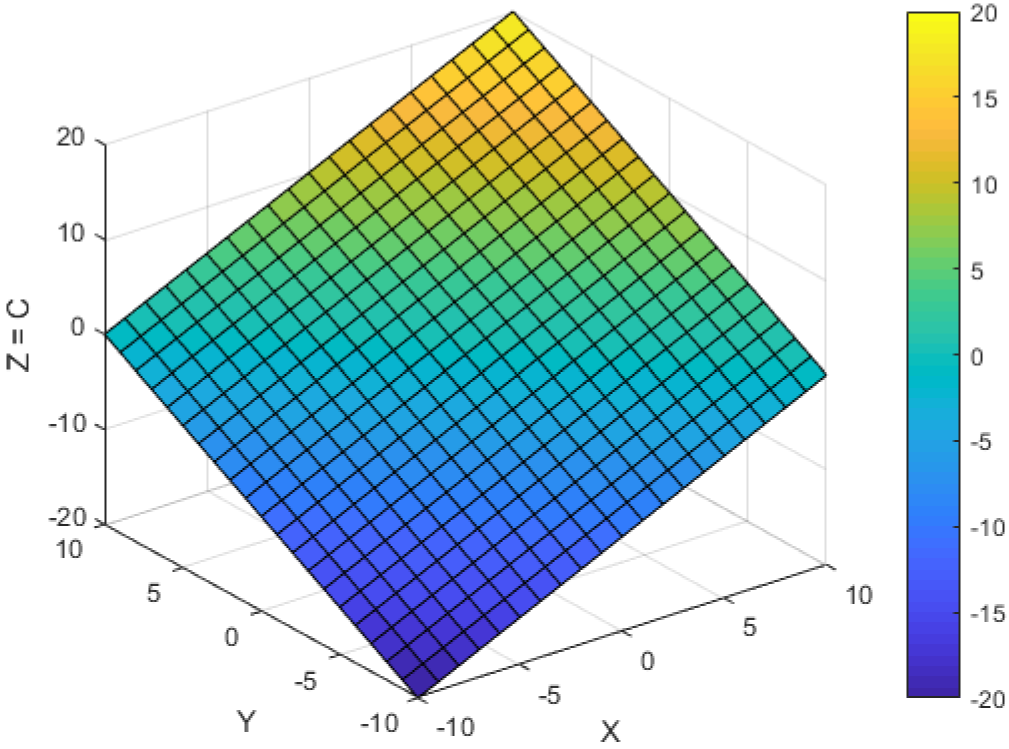
For many types of visualizations you create, MATLAB maps the full range of your data to the colormap by default. The smallest value in your data maps to the first row in the colormap, and the largest value maps to the last row in the colormap. All intermediate values map linearly to the intermediate rows of the colormap.

This default mapping is useful in most cases, but you can perform the mapping over any range you choose, even if the range you choose is different than the range of your data. Choosing a different mapping range allows you to:

- See where your data is at or beyond the limits of that range.
- See where your data lies within that range.

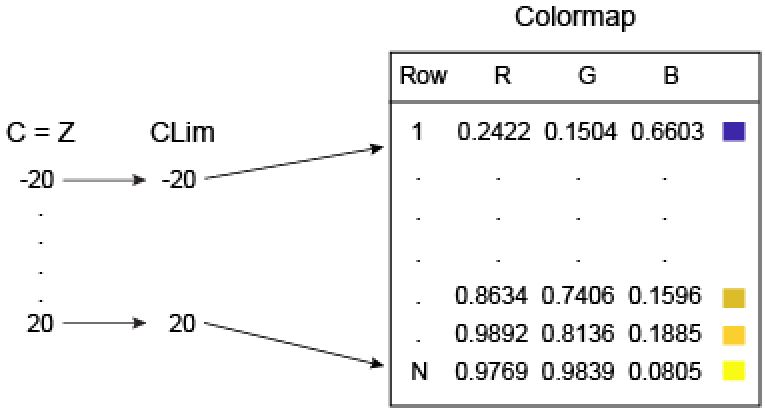
Consider the surface $Z = X + Y$, where $-10 \le x \le 10$ and $-10 \le y \le 10$.

```
[X,Y] = meshgrid(-10:10);
Z = X + Y;
s = surf(X,Y,Z);
xlabel('X');
ylabel('Y');
zlabel('Z = C');
colorbar
```



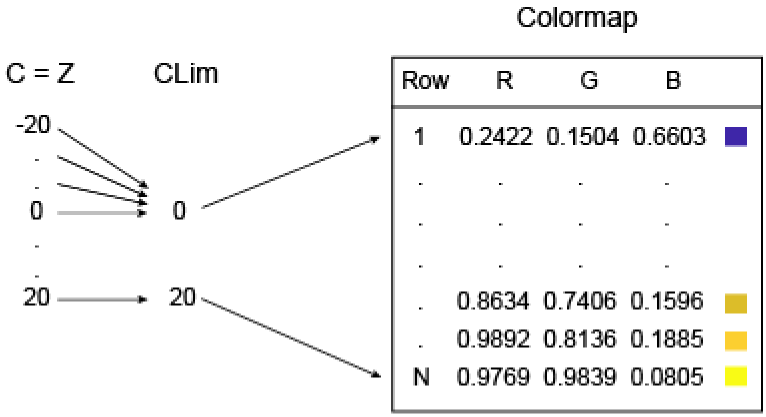
"How Surface Plot Data Relates to a Colormap" on page 5-16 describes the properties that control the color in this presentation. Essentially, the `CData` property of the `Surface` object contains an array `C` that associates each grid point on the surface to a color in the colormap. By default, `C` is equal to `Z`, where `Z` is the array containing the values of $z = f(x,y)$ at the grid points. Thus, the colors vary with changes in `Z`.

The mapping range is controlled by the `CLim` property of the `Axes` object. This property contains a two-element vector of the form `[cmin cmax]`. The default value of `cmin` is equal to the smallest value of `C`, and the default value of `cmax` is the largest value of `C`. In this case, `CLim` is `[-20 20]` because the range of `C` reflects the range of `Z`.



Row	R	G	B	
1	0.2422	0.1504	0.6603	🟦
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	🟨
.	0.9892	0.8136	0.1885	🟨
N	0.9769	0.9839	0.0805	🟨

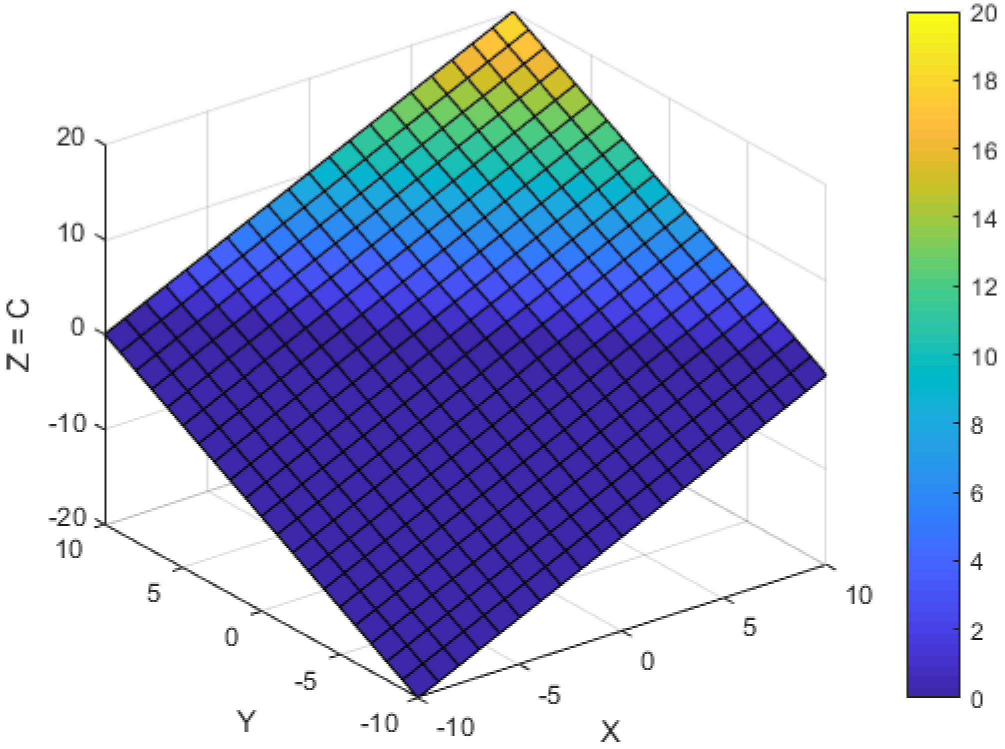
Changing `CLim` to `[0 20]` clips all the values at or below `0` to the first color in the colormap.



Row	R	G	B	
1	0.2422	0.1504	0.6603	🟦
.	.	.	.	
.	.	.	.	
.	.	.	.	
.	0.8634	0.7406	0.1596	🟨
.	0.9892	0.8136	0.1885	🟨
N	0.9769	0.9839	0.0805	🟨

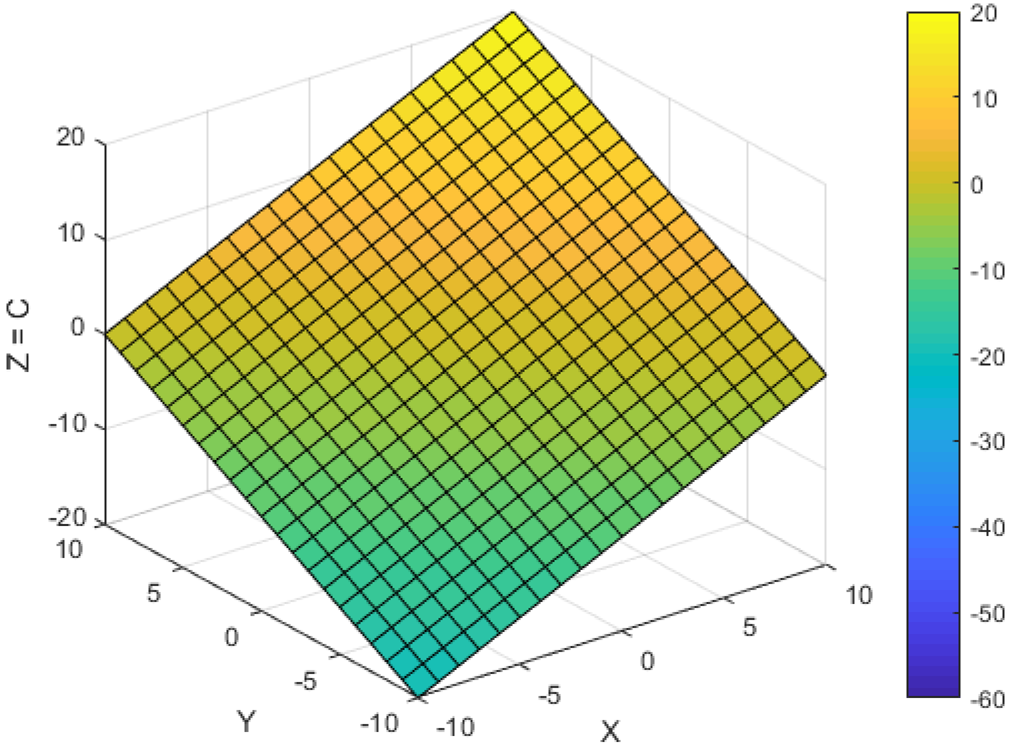
This command changes the `CLim` property to `[0 20]`. Notice that the lower half of the surface maps to the first color in the colormap (dark blue). This clipping occurs because `C` (which is equal to `Z`) is less than or equal to zero at those points.

```
caxis([0 20]);
```



You can also widen the mapping range to see where your data lies within that range. For example, changing the range to `[-60 20]` results in a surface that only uses half of the colors. The lower half of the colormap corresponds to values that are outside the range of `C`, so those colors are not represented on the surface.

```
caxis([-60 20]);
```



Note: You can set the `CLim` property for surface plots, patches, images, or any graphics object that uses a colormap. However, this property only affects graphics objects that have the `CDataMapping` property set to `'scaled'`. If the `CDataMapping` property is set to `'direct'`, then all values of `C` index directly into the colormap without any scaling. Any values of `C` that are less than 1 are clipped to the first color in the colormap. Any values of `C` that are greater than the length of the colormap are clipped to the last color in the colormap.

See Also

`caxis` | `colorbar` | `colormap` | `surf`

Related Examples

- “Change Color Scheme Using a Colormap” on page 5-10
- “How Surface Plot Data Relates to a Colormap” on page 5-16
- “Creating Colorbars” on page 5-2

Differences Between Colormaps and Truecolor

Many graphics objects, such as surfaces, patches, and images, support two different techniques for specifying color: colormaps (which use indexed color) and truecolor. Each technique involves a different workflow and has a different impact on your visual presentation.

Differences in Workflow

A colormap is an m-by-3 array in which each row specifies an RGB triplet. To use a colormap in a graphical presentation, you assign an index to each location in your graphic. Each index addresses a row in the colormap to display a color at the specified location in the graphic. By contrast, using truecolor involves specifying an RGB triplet at every location in your graphic.

Here are some points to consider when deciding which to technique to use:

- Truecolor is more direct. If you want to assign specific red, green, and blue values to specific locations in your graphic, it is usually easier to do it using truecolor.
- Making changes in a region of the color palette is easier to do in a colormap. For example, if you want to brighten the transition from blue to green in a gradient, it is easier to edit those rows in the colormap than it is to edit the colors at the individual locations in your graphic.
- The format of your data might be more appropriate for one of the workflows. For example, many compressed GIF images are stored using indexed color.

Both coloring techniques use a color array `C` to specify the color information. The shape of `C` depends on the type of graphics object and the coloring method you choose. This table summarizes the differences.

Type of Graphics Object	Property that Contains Color Array `C`	Shape of `C` for Indexed Color	Shape of `C` for Truecolor
`Surface`	`CData`	`C` is an m-by-n array that is the same size as the z-coordinate array. The value at `C(i,j)` specifies the colormap index for `Z(i,j)`.	`C` is an m-by-n-by-3 array, where `C(:,:,i)` the same size as the z-coordinate array.

Type of Graphics Object	Property that Contains Color Array C	Shape of C for Indexed Color	Shape of C for Truecolor
			C(i,j,1) specifies the red component for Z(i,j). C(i,j,2) specifies the green component for Z(i,j). C(i,j,3) specifies the blue component for Z(i,j).
Image	CData	C is an m-by-n array for an m-by-n image. The value at C(i,j) specifies the colormap index for pixel (i,j).	C is an m-by-n-by-3 array for an m-by-n image. C(i,j,1) specifies the red component for pixel (i,j). C(i,j,2) specifies the green component for pixel (i,j). C(i,j,3) specifies the blue component for pixel (i,j).

Type of Graphics Object	Property that Contains Color Array `C`	Shape of `C` for Indexed Color	Shape of `C` for Truecolor
`Patch` (x, y, z)	`CData`	To color patch faces, `C` is a 1-by-m array for m patch faces. `C(i)` specifies the colormap index for face `i`. To color patch vertices, `C` is an m-by-n array, where m is the number of vertices per face, and n is the number of faces. `C(i,j)` specifies the colormap index for vertex `i` of face `j`.	To color patch faces, `C` is an m-by-3 array for m patch faces. `C(i,:)` specifies the red, green, and blue values for face `i`. To color patch vertices, `C` is an n-by-3 array, where n is the total number of vertices. `C(i,:)` specifies the red, green, and blue values for vertex `i`.
`Patch` (face-vertex data)	`FaceVertexCData`	To color patch faces, `C` is a 1-by-m array for m patch faces. `C(i)` specifies the colormap index for face `i`. To color patch vertices, `C` is a 1-by-n array, where n is the total number of vertices. `C(i)` specifies the colormap index for vertex `i`.	To color patch faces, `C` is an m-by-3 array for m patch faces. `C(i,:)` specifies the red, green, and blue values for face `i`. To color patch vertices, `C` is an n-by-3 array, where n is the total number of vertices. `C(i,:)` specifies the red, green, and blue values for vertex `i`.

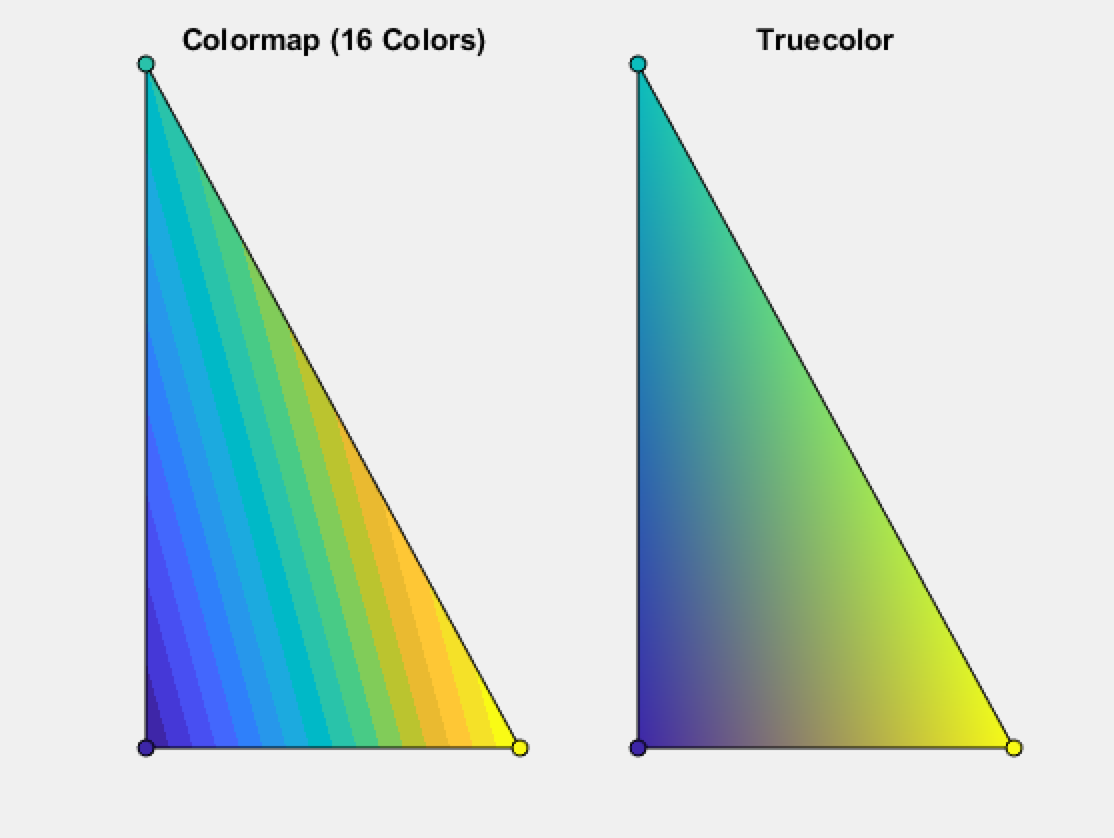
Differences in Visual Presentation

Colormaps offer a palette of m colors, where m is the length of the colormap. By contrast, truecolor offers a palette of 256 × 256 × 256 ≈ 1.68 million colors.

Consider these factors as you decide how large your color palette needs to be:

- Smaller color palettes are the most economical way to fill large regions with solid color. They are also useful in visualizing contours of surfaces.
- Larger color palettes are better for showing subtle transitions and smooth color gradients.

Interpolating vertex colors across a patch face is one situation in which the differences between indexed color and truecolor are more noticeable. The following two patches illustrate an extreme case. The patch on the left uses a small colormap, whereas the patch on the right uses truecolor.



If you are concerned about the limited palette of a colormap, you can add more colors to it. “Change Color Scheme Using a Colormap” on page 5-10 shows how to use a colormap with a specific number of colors.

Related Examples

- “Image Types” on page 7-5
- “Change Color Scheme Using a Colormap” on page 5-10
- “How Surface Plot Data Relates to a Colormap” on page 5-16
- “How Image Data Relates to a Colormap” on page 5-22
- “How Patch Data Relates to a Colormap” on page 5-30

Creating Specialized Plots

- “Types of Bar Graphs” on page 6-3
- “Modify Baseline of Bar Graph” on page 6-10
- “Overlay Bar Graphs” on page 6-13
- “Combine Line and Bar Charts Using Two y-Axes” on page 6-17
- “Color 3-D Bars by Height” on page 6-21
- “Compare Data Sets Using Overlayed Area Graphs” on page 6-24
- “Offset Pie Slice with Greatest Contribution” on page 6-29
- “Add Legend to Pie Chart” on page 6-31
- “Label Pie Chart With Text and Percentages” on page 6-34
- “Data Cursors with Histograms” on page 6-42
- “Color Analysis with Bivariate Histogram” on page 6-44
- “Control Categorical Histogram Display” on page 6-50
- “Replace Discouraged Instances of hist and histc” on page 6-58
- “Combine Line and Stem Plots” on page 6-67
- “Overlay Stairstep Plot and Line Plot” on page 6-70
- “Combine Contour Plot and Quiver Plot” on page 6-73
- “Projectile Path Over Time” on page 6-75
- “Label Contour Plot Levels” on page 6-77
- “Change Fill Colors for Contour Plot” on page 6-79
- “Highlight Specific Contour Levels” on page 6-81
- “Animation Techniques” on page 6-84
- “Trace Marker Along Line” on page 6-86
- “Move Group of Objects Along Line” on page 6-89
- “Animate Graphics Object” on page 6-93
- “Line Animations” on page 6-97

- "Record Animation for Playback" on page 6-100
- "Customize Polar Axes Grid Lines and Appearance" on page 6-104
- "Compass Labels on Polar Axes" on page 6-113
- "Create Line Plot with Markers" on page 6-115

Types of Bar Graphs

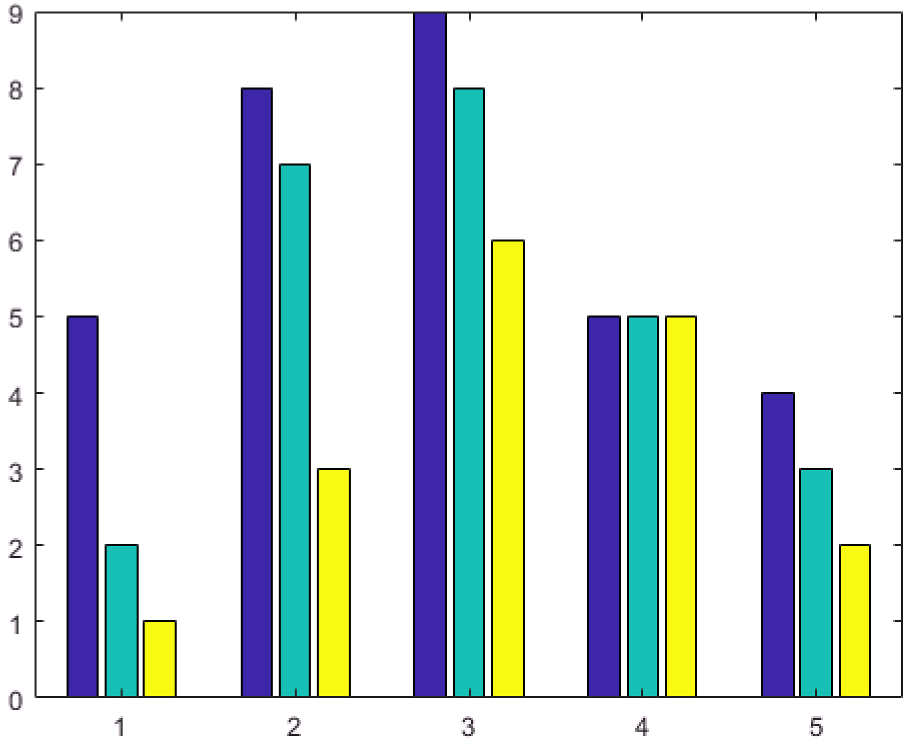
Bar graphs are useful for viewing results over a period of time, comparing results from different data sets, and showing how individual elements contribute to an aggregate amount.

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

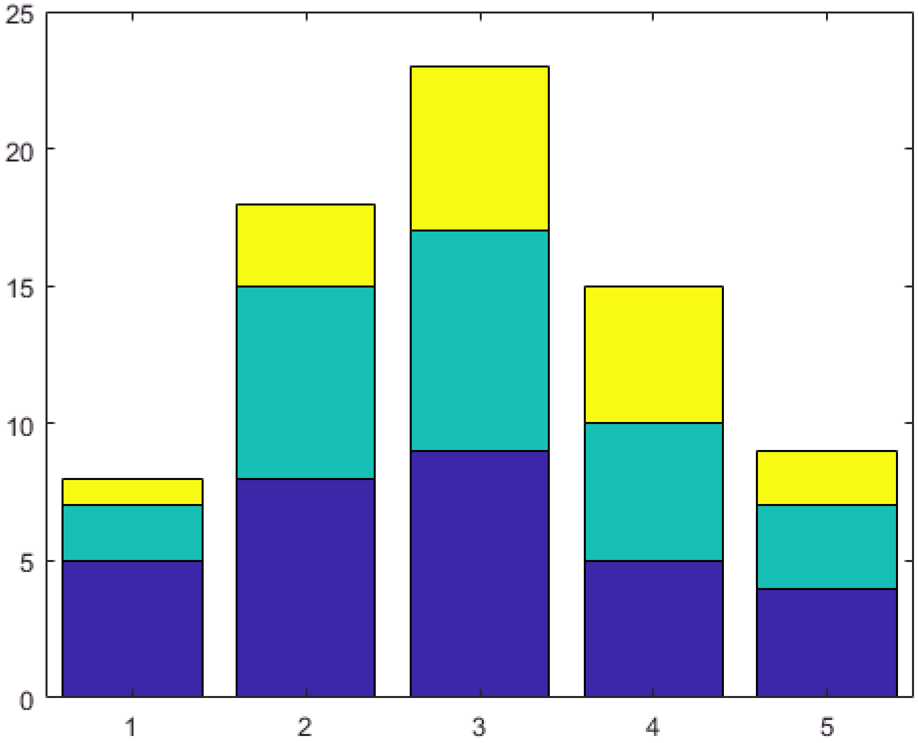
The `bar` function distributes bars along the *x*-axis. Elements in the same row of a matrix are grouped together. For example, if a matrix has five rows and three columns, then `bar` displays five groups of three bars along the *x*-axis. The first cluster of bars represents the elements in the first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

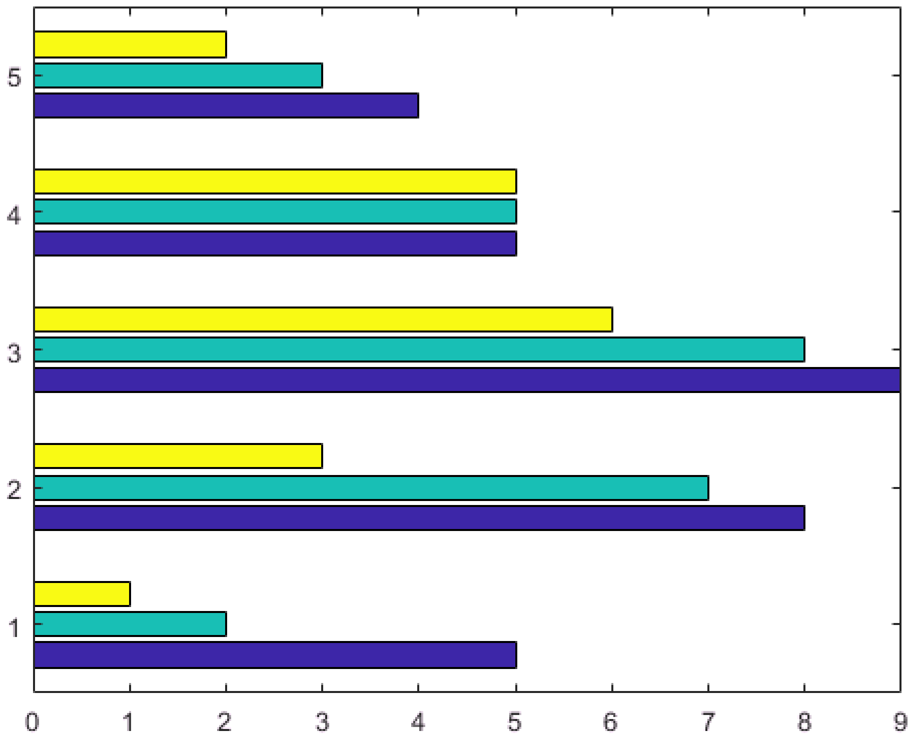
```
figure
bar(Y,'stacked')
```



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.

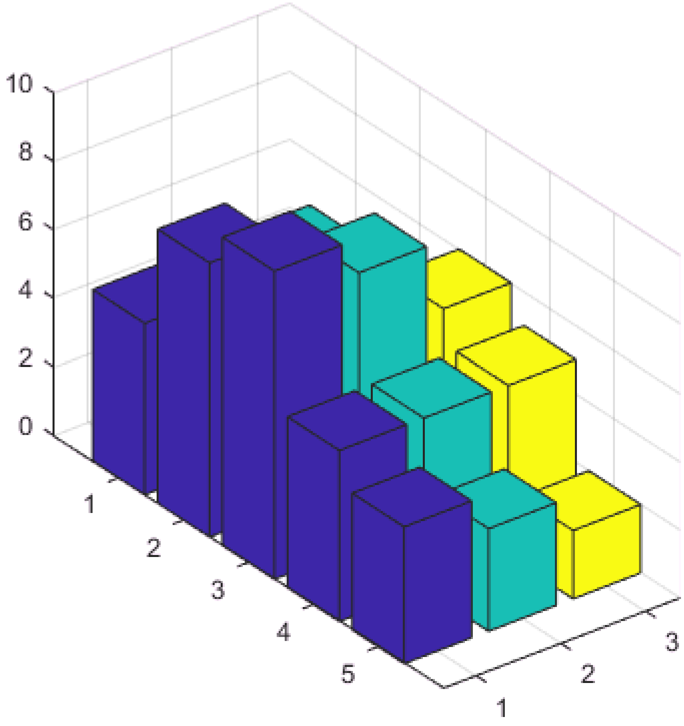
```
Y = [5,2,1
     8,7,3
     9,8,6
     5,5,5
     4,3,2];
figure
barh(Y)
```



3-D Bar Graph

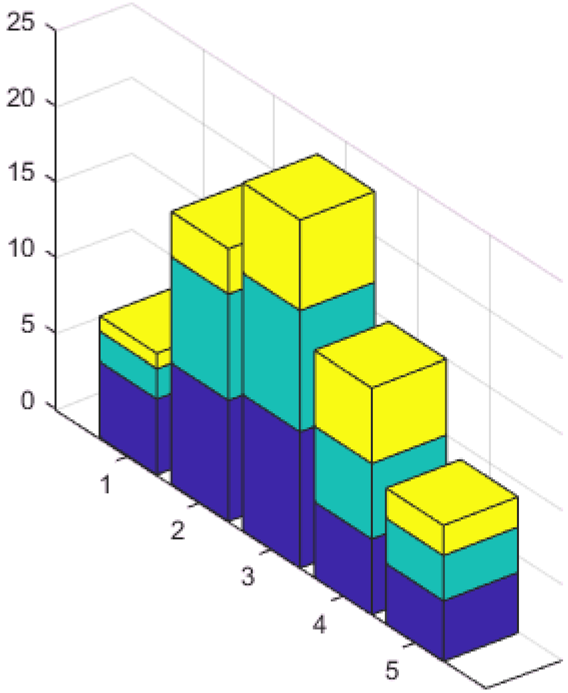
The `bar3` function draws each element as a separate 3-D block and distributes the elements of each column along the *y*-axis.

```
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.

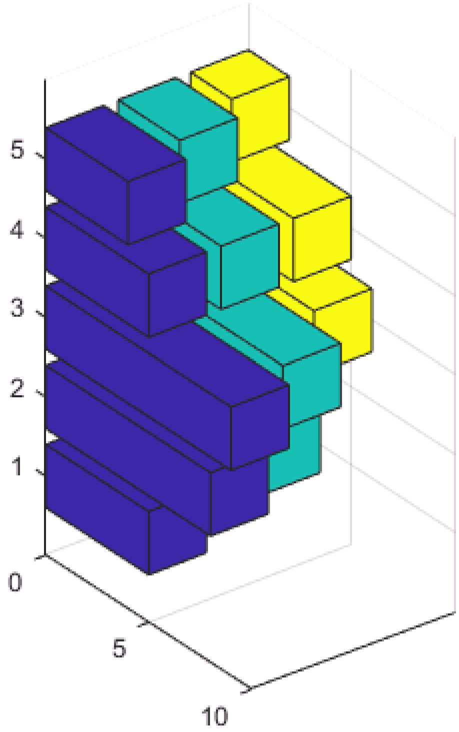
```
figure
bar3(Y,'stacked')
```



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

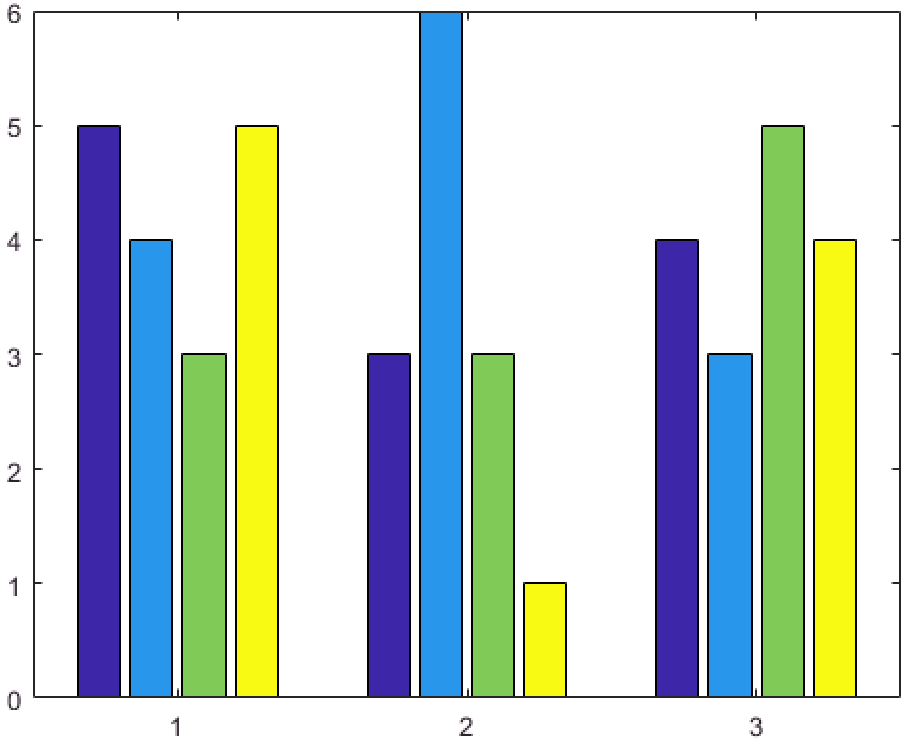
`bar` | `bar3` | `bar3h` | `barh`

Modify Baseline of Bar Graph

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

Create a bar graph of a four-column matrix. The `bar` function creates a bar series for each column of the matrix. Return the four bar series as `b`.

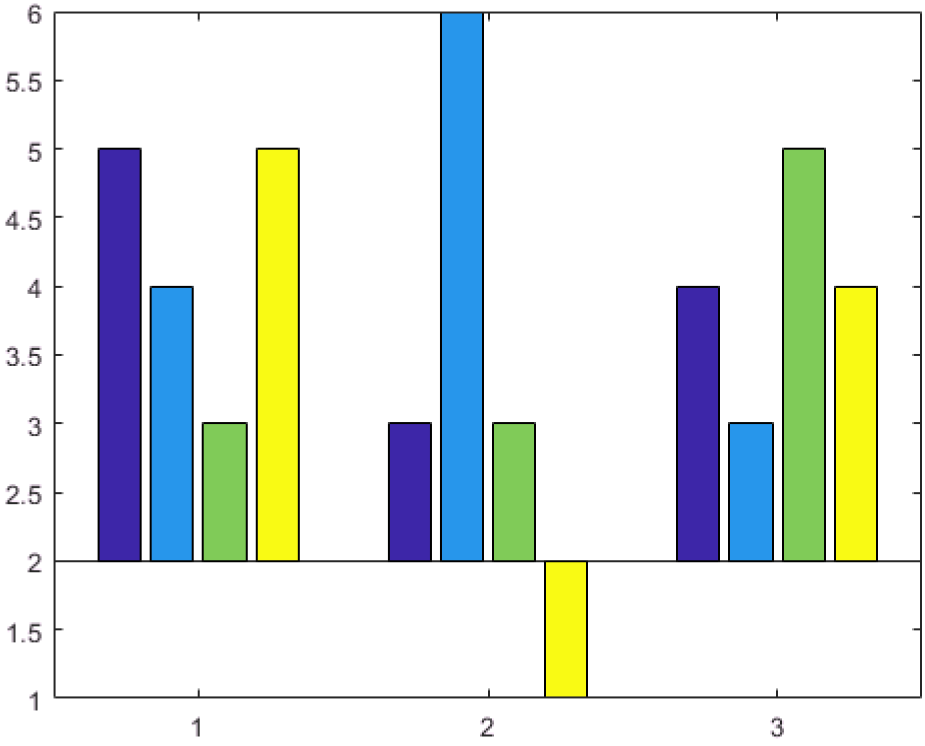
```
Y = [5, 4, 3, 5;
     3, 6, 3, 1;
     4, 3, 5, 4];
b = bar(Y);
```



All bar series in a graph share the same baseline. Change the value of the baseline to 2 by setting the `BaseValue` property for any of the bar series.

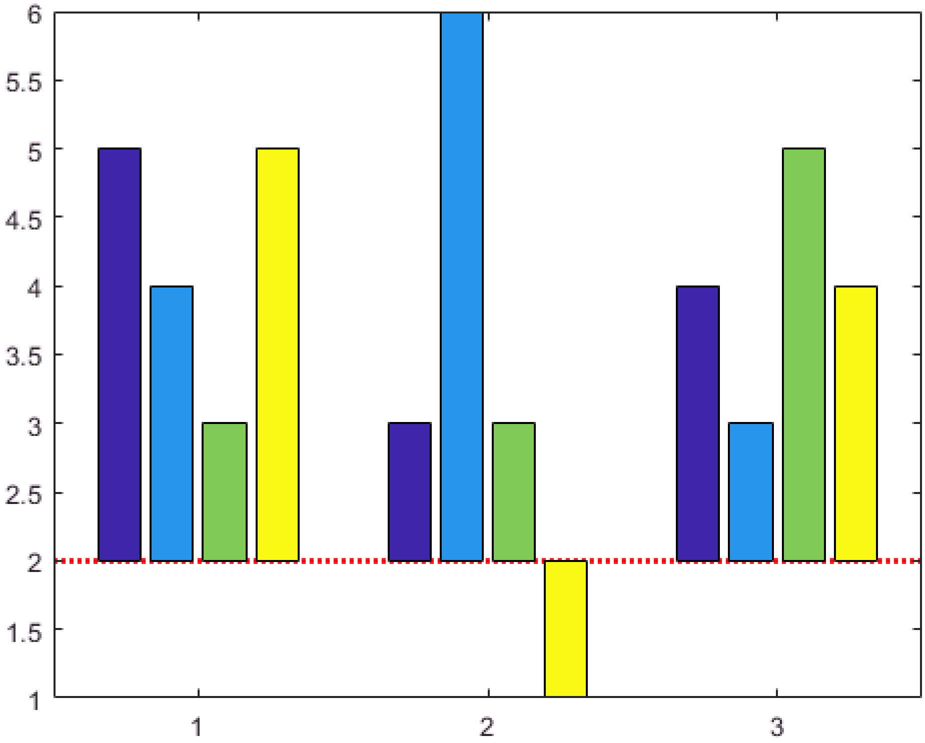
Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead, such as `set(b(1),'BaseValue',2)`.

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



Change the baseline to a thick, red dotted line.

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



See Also

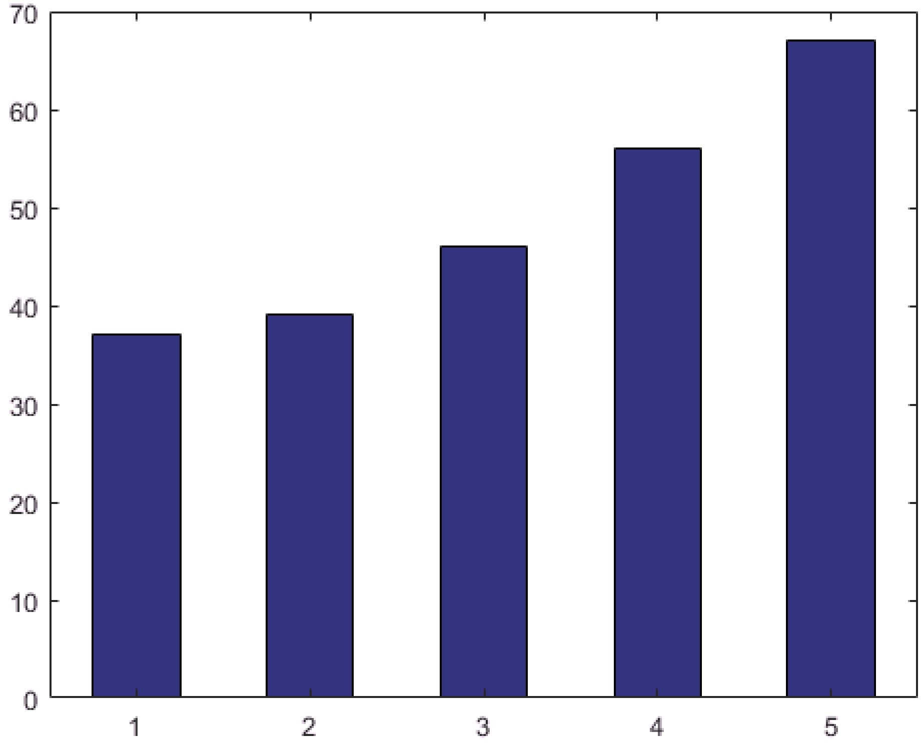
`bar` | `barh`

Overlay Bar Graphs

This example shows how to overlay two bar graphs and specify the bar colors and widths. Then, it shows how to add a legend, display the grid lines, and specify the tick labels.

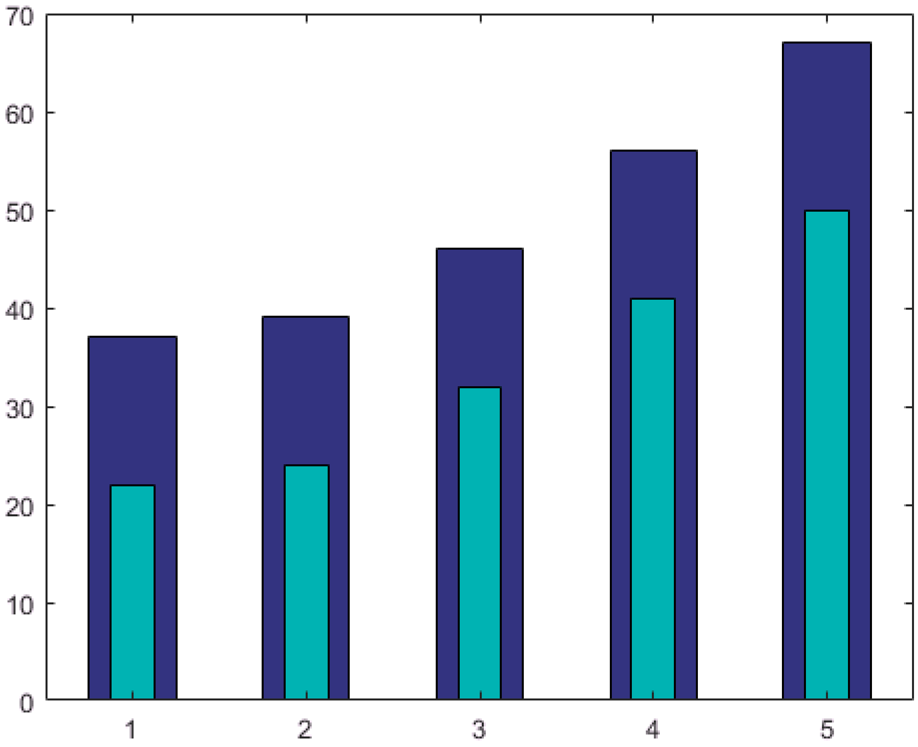
Create a bar graph. Set the bar width to 0.5 so that the bars use 50% of the available space. Specify the bar color by setting the `FaceColor` property to an RGB color value.

```
x = [1 2 3 4 5];
temp_high = [37 39 46 56 67];
w1 = 0.5;
bar(x,temp_high,w1,'FaceColor',[0.2 0.2 0.5])
```



Plot a second bar graph over the first bar graph. Use the `hold` function to retain the first graph. Set the bar width to .25 so that the bars use 25% of the available space. Specify a different RGB color value for the bar color.

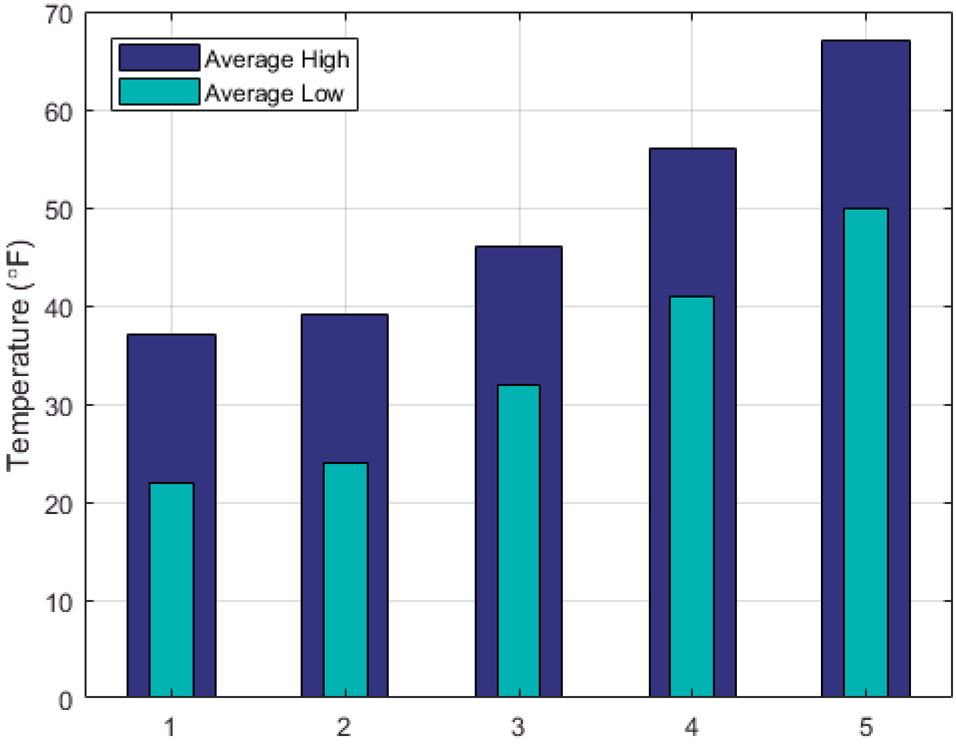
```
temp_low = [22 24 32 41 50];
w2 = .25;
hold on
bar(x,temp_low,w2,'FaceColor',[0 0.7 0.7])
hold off
```



Add grid lines, a *y*-axis label, and a legend in the upper left corner. Specify the legend descriptions in the order that you create the graphs.

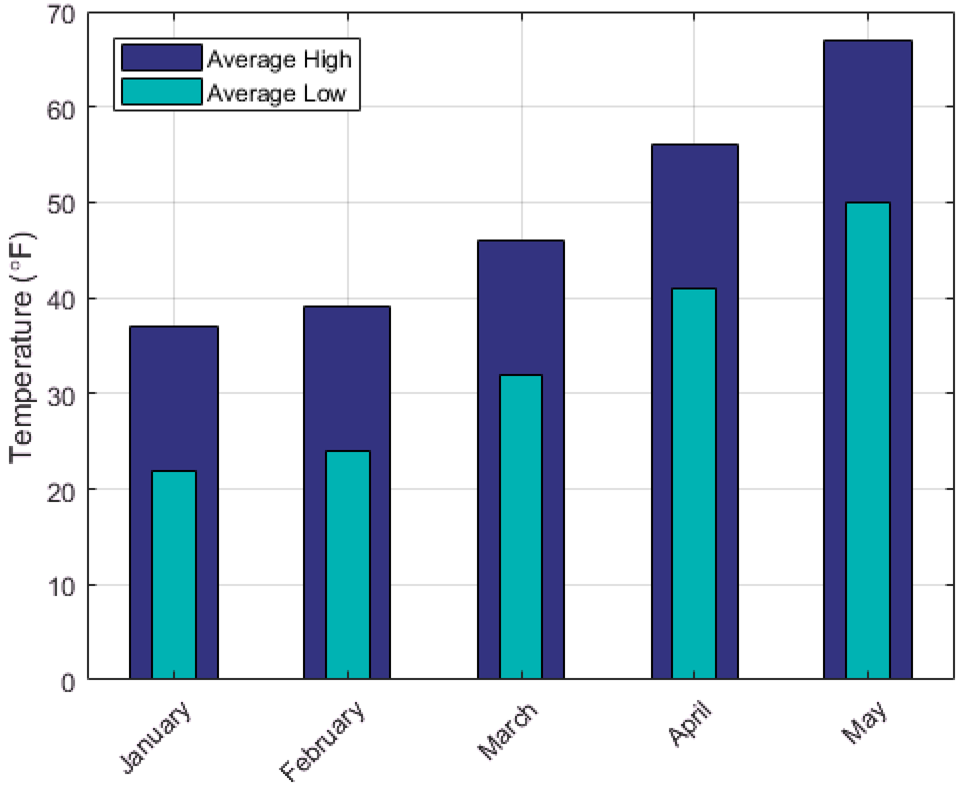
```
grid on
```

```
ylabel('Temperature (\circF)')
legend({'Average High','Average Low'},'Location','northwest')
```



Specify the *x*-axis tick labels by setting the `XTick` and `XTickLabel` properties of the axes object. The `XTick` property specifies tick value locations along the *x*-axis. The `XTickLabel` property specifies the text to use at each tick value. Rotate the labels using the `XTickLabelRotation` property. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set instead.

```
ax = gca;
ax.XTick = [1 2 3 4 5];
ax.XTickLabels = {'January','February','March','April','May'};
ax.XTickLabelRotation = 45;
```



See Also

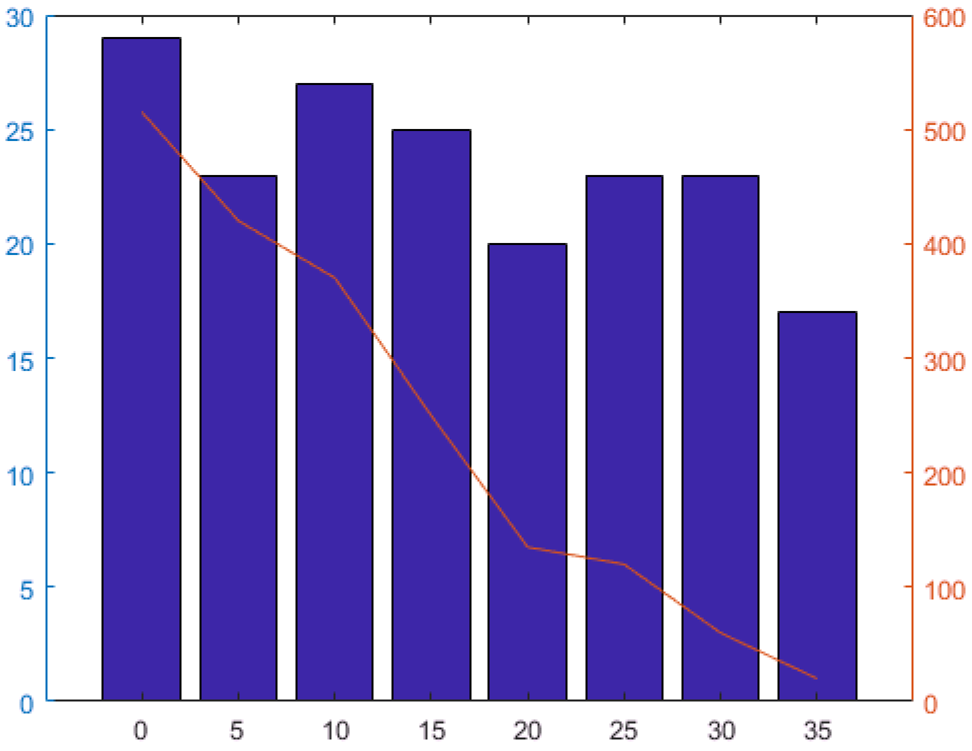
`bar` | `barh` | `hold`

Combine Line and Bar Charts Using Two y-Axes

This example shows how to combine a line chart and a bar chart using two different *y*-axes. It also shows how to customize the line and bars.

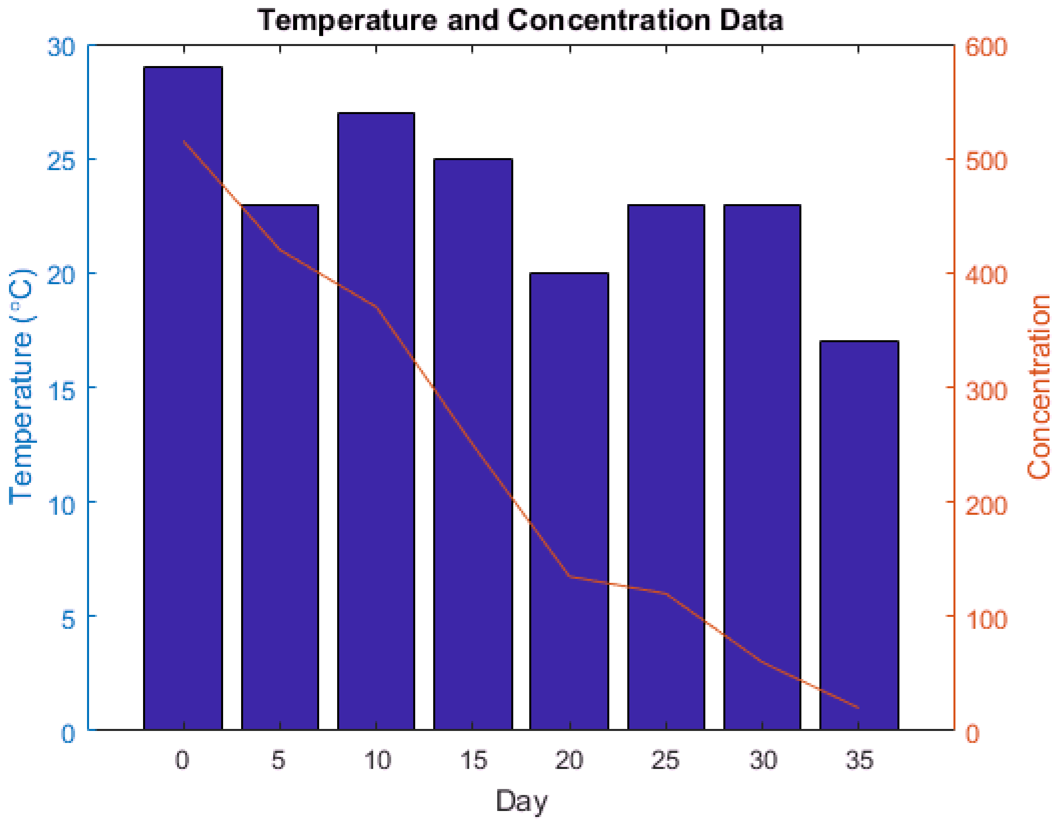
Create a chart that has two *y*-axes using `yyaxis`. Graphics functions target the active side of the chart. Control the active side using `yyaxis`. Plot a bar chart using the left *y*-axis. Plot a line chart using the right *y*-axis. Assign the bar series object and the chart line object to variables.

```
days = 0:5:35;
conc = [515 420 370 250 135 120 60 20];
temp = [29 23 27 25 20 23 23 17];

yyaxis left
b = bar(days,temp);
yyaxis right
p = plot(days,conc);
```



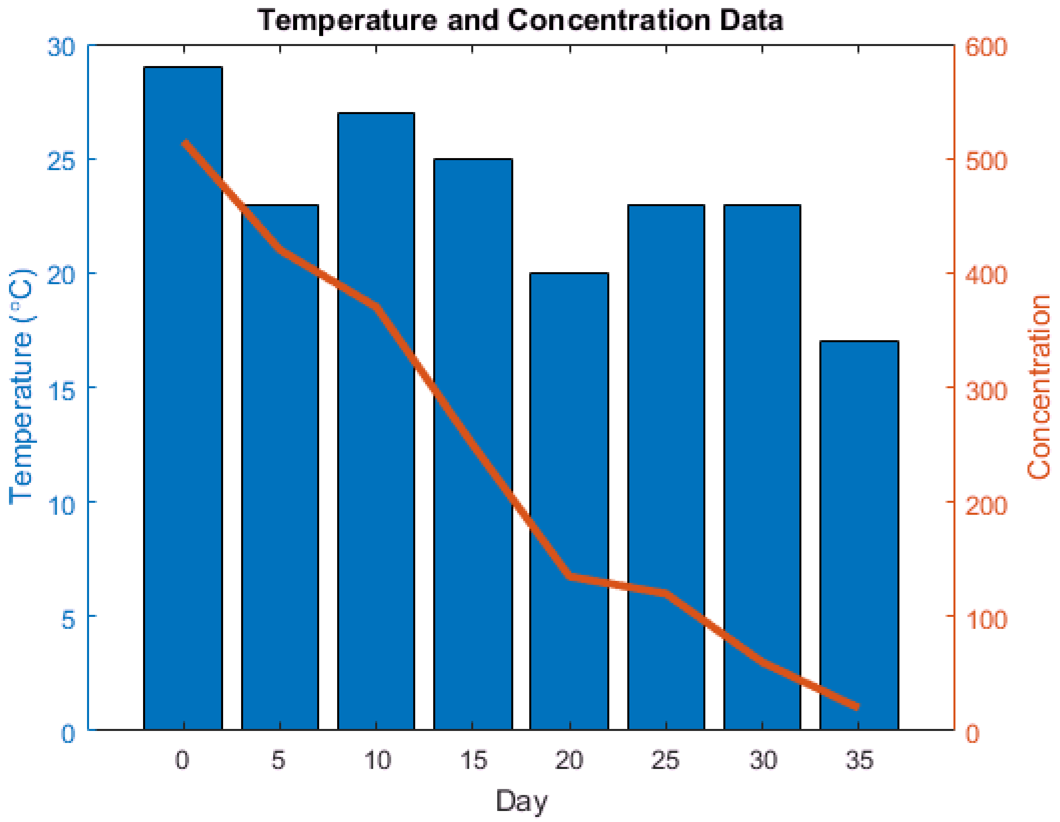
Add a title and axis labels to the chart.

```
title('Temperature and Concentration Data')
xlabel('Day')
yyaxis left
ylabel('Temperature (\circC)')
yyaxis right
ylabel('Concentration')
```



Change the width of the chart line and change the bar colors.

```
p.LineWidth = 3;
b.FaceColor = [ 0 0.447 0.741];
```



See Also

Functions
`bar` | `hold` | `plot` | `title` | `xlabel` | `ylabel` | `yyaxis`

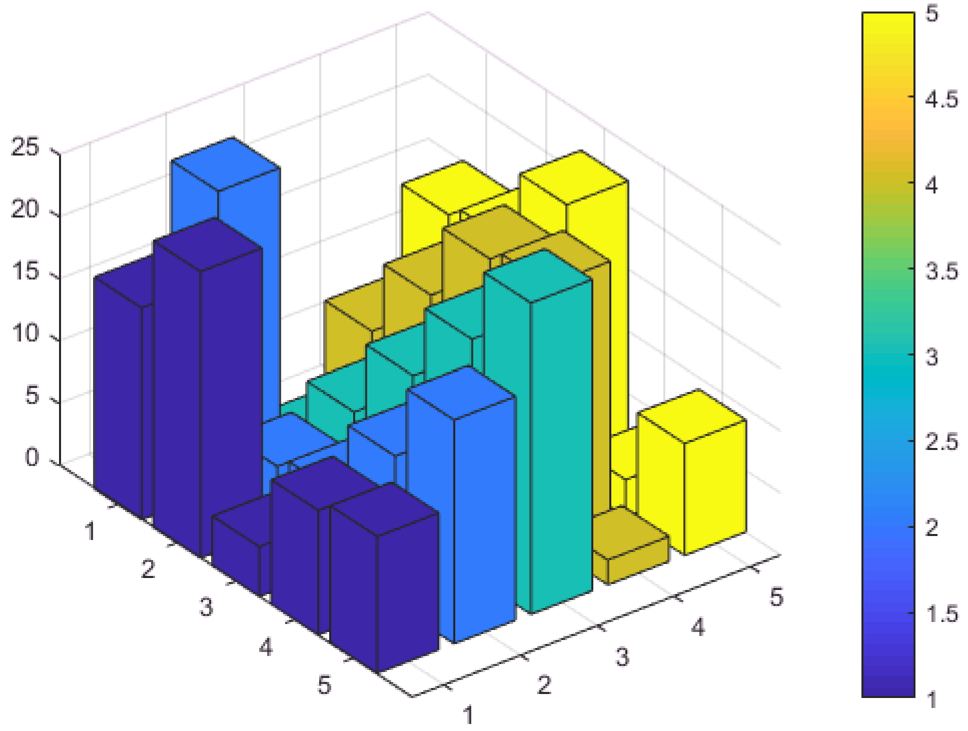
Properties
Bar Properties | Chart Line Properties

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.

Create a 3-D bar graph of data from the `magic` function. Return the surface objects used to create the bar graph in array `b`. Add a colorbar to the graph.

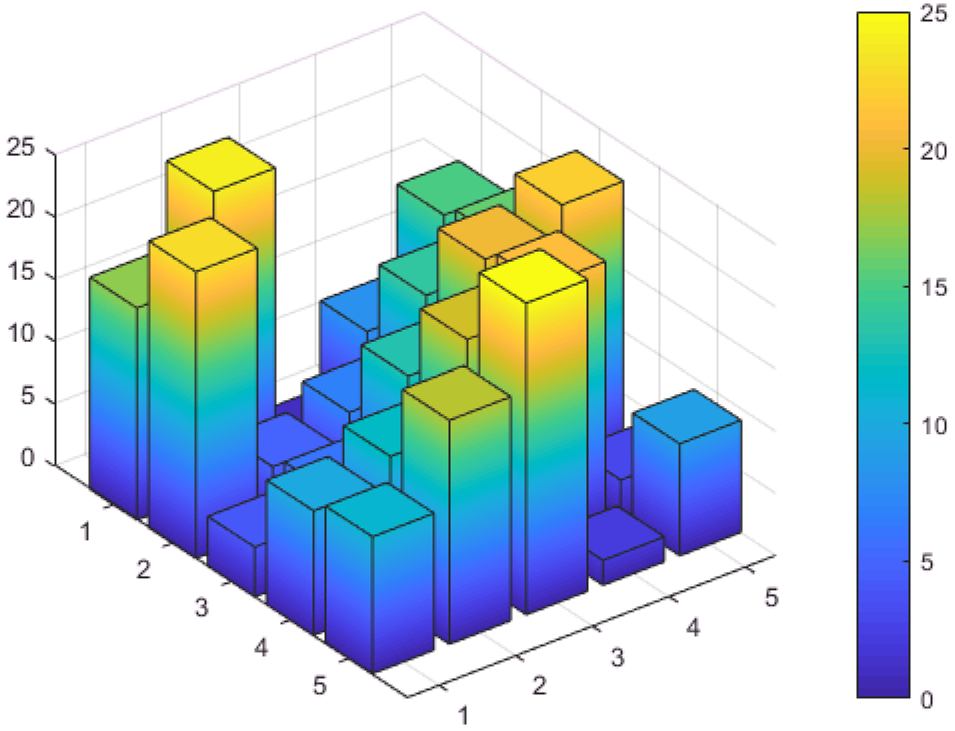
```
Z = magic(5);
b = bar3(Z);
colorbar
```



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'`.

Note: Starting in R2014b, you can use dot notation to query and set properties. If you are using an earlier release, use the get and set functions instead, such as `zdata = get(b(k),'ZData')`.

```
for k = 1:length(b)
    zdata = b(k).ZData;
    b(k).CData = zdata;
    b(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.

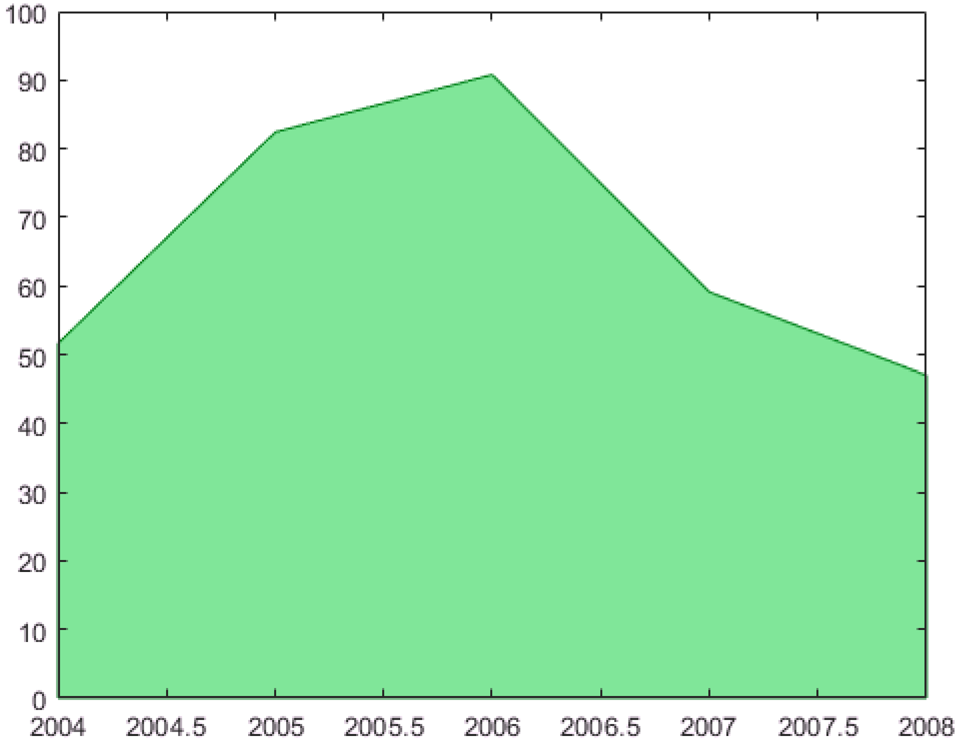
Overlay Two Area Graphs

Create 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];
```

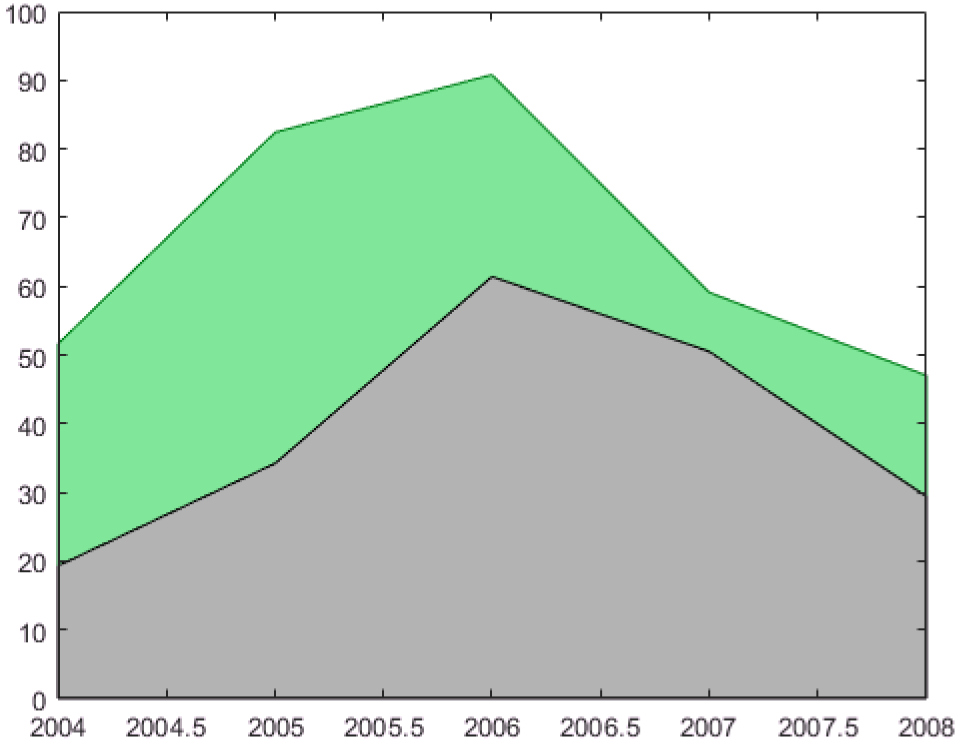
Display sales and expenses as two separate area graphs in the same axes. First, plot an area graph of `sales`. Change the color of the area graph by setting the `FaceColor` and `EdgeColor` properties using RGB triplet color values.

```
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 existing graph. Plot a second area graph of expenses. Then, set the hold state back to off.

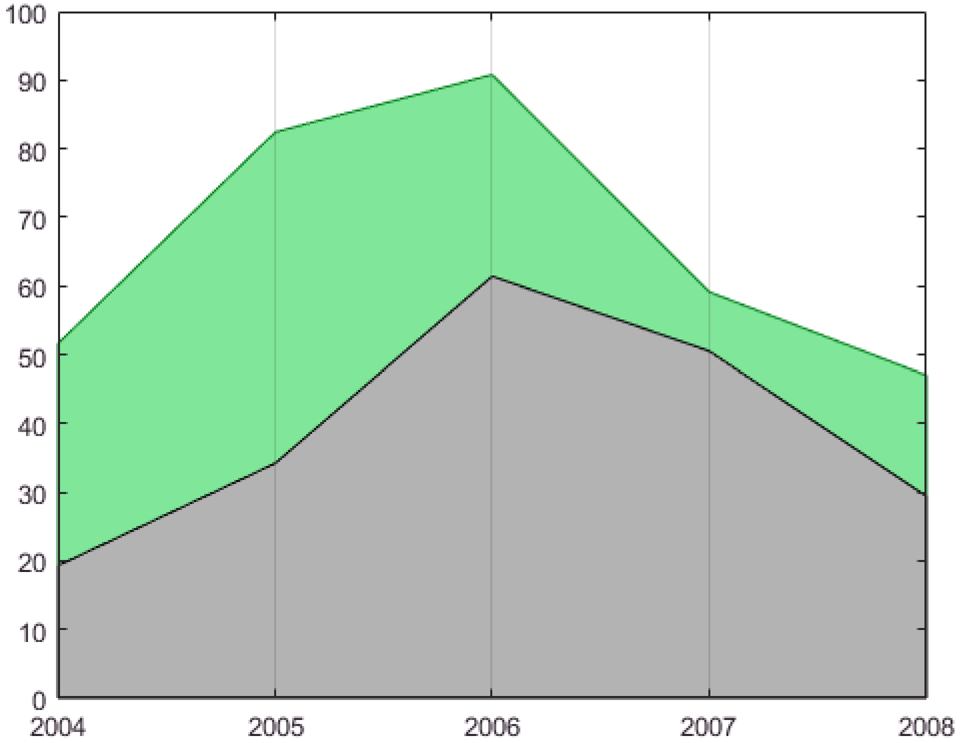
```
hold on
area(years,expenses,'FaceColor',[0.7 0.7 0.7],'EdgeColor','k')
hold off
```



Add Grid Lines

Set the tick marks along the *x*-axis to correspond to whole years. Draw a grid line for each tick mark. Display the grid lines on top of the area graphs by setting the `Layer` property. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead.

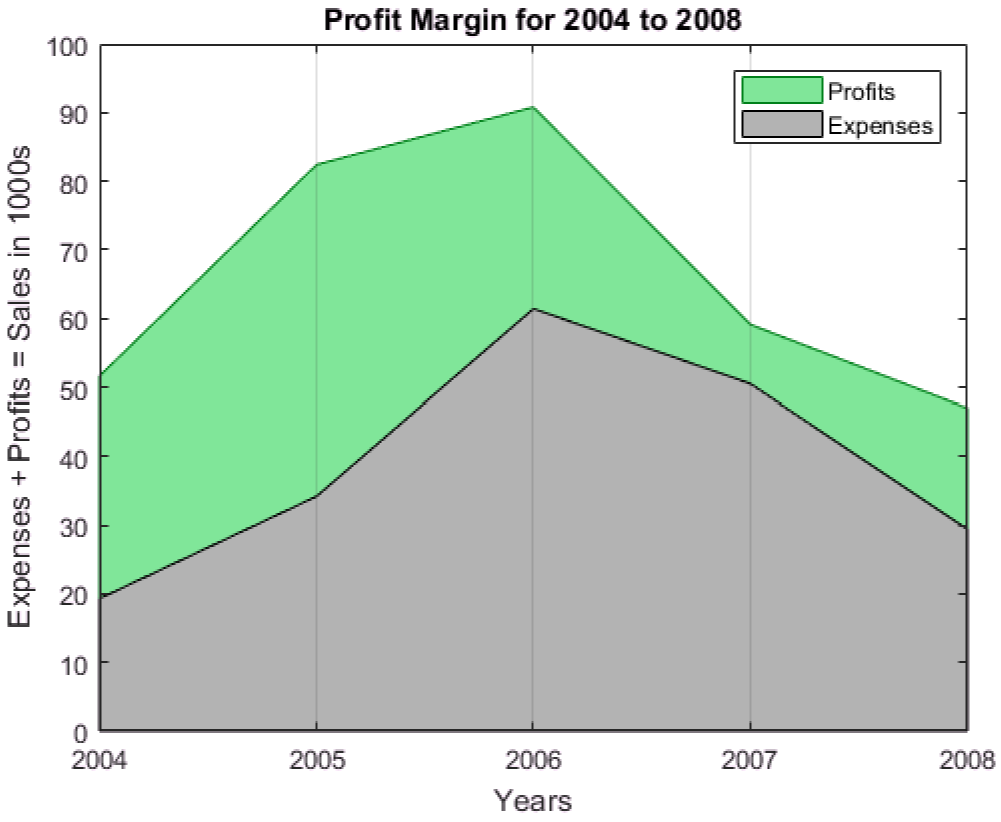
```
ax = gca; % current axes
ax.XTick = years;
ax.XGrid = 'on';
ax.Layer = 'top';
```



Add Title, Axis Labels, and Legend

Give the graph a title and add axis labels. 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.

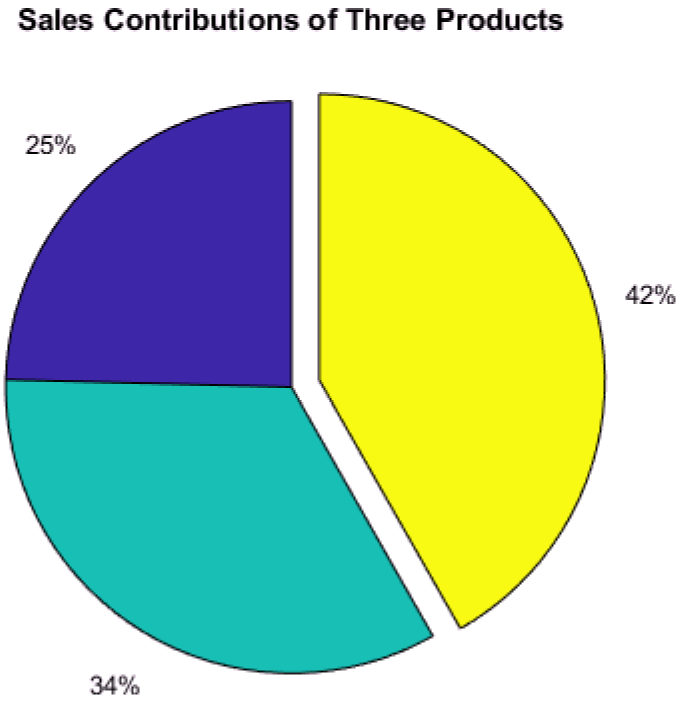
```
explode = zeros(1,3);
```

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')
```



See Also

`max` | `pie` | `zeros`

Related Examples

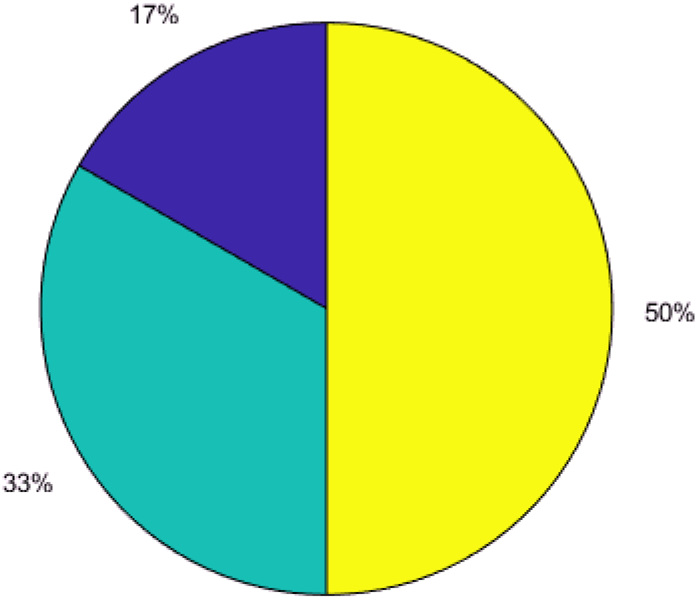
- "Add Legend to Pie Chart" on page 6-31

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)
```

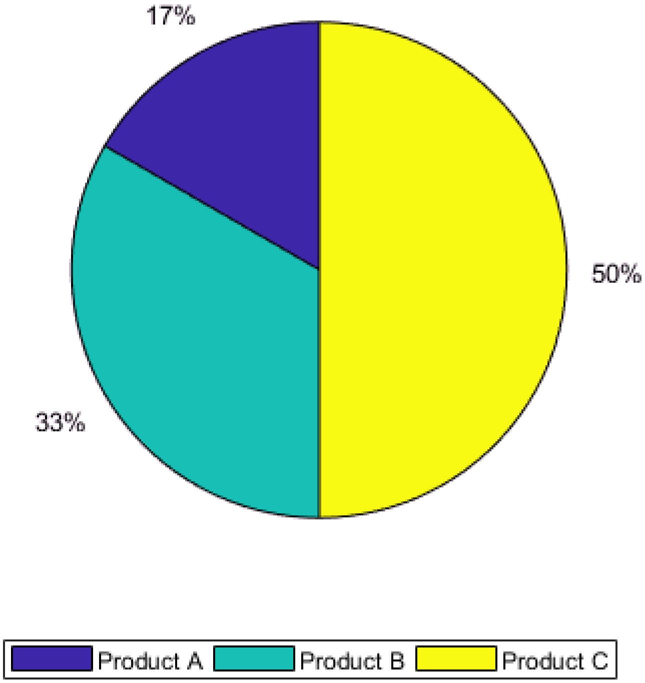


Specify the description for each pie slice in the cell array `labels`.

```
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" on page 6-29

Label Pie Chart With Text and Percentages

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.

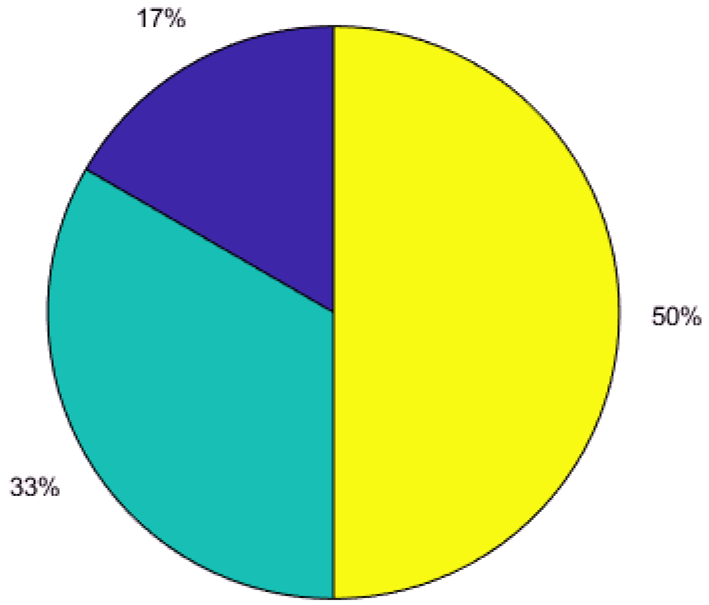
In this section...

"Create Pie Chart" on page 6-34
"Store Precalculated Percent Values" on page 6-35
"Combine Percent Values and Additional Text" on page 6-36
"Determine Horizontal Distance to Move Each Label" on page 6-38
"Position New Label" on page 6-39

Create Pie Chart

Create a pie chart. Specify an output argument, `h`, to contain the text and patch objects 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 labels directly to the `pie` function. For example, `pie(x,{'Item A','Item B','Item C'})`.

Store Precalculated Percent Values

Extract the three text objects 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
```

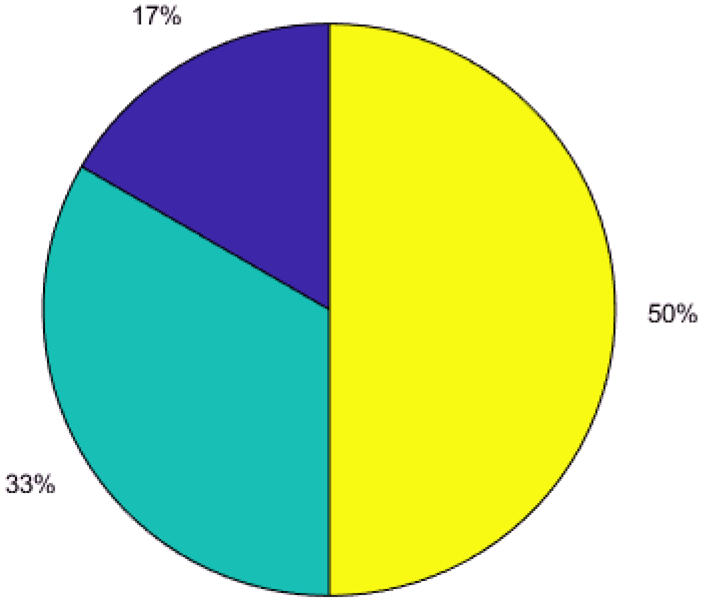
Combine Percent Values and Additional Text

Specify the text in the cell array `txt`. Then, concatenate the text with the associated percent values in the cell array `combinedtxt`.

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

Before updating the labels, store the text `Extent` property values for the current labels. The extent values give 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
```

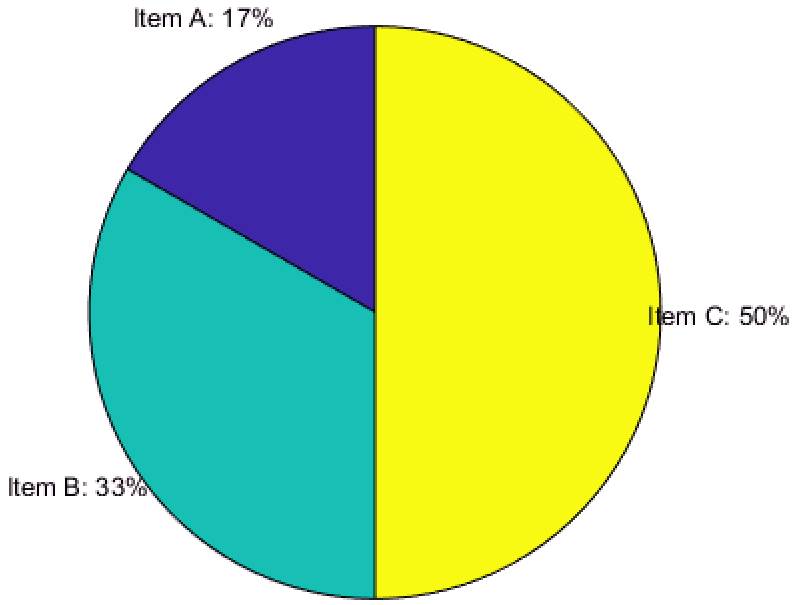


```
oldExtents = cell2mat(oldExtents_cell); % numeric array
```

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

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

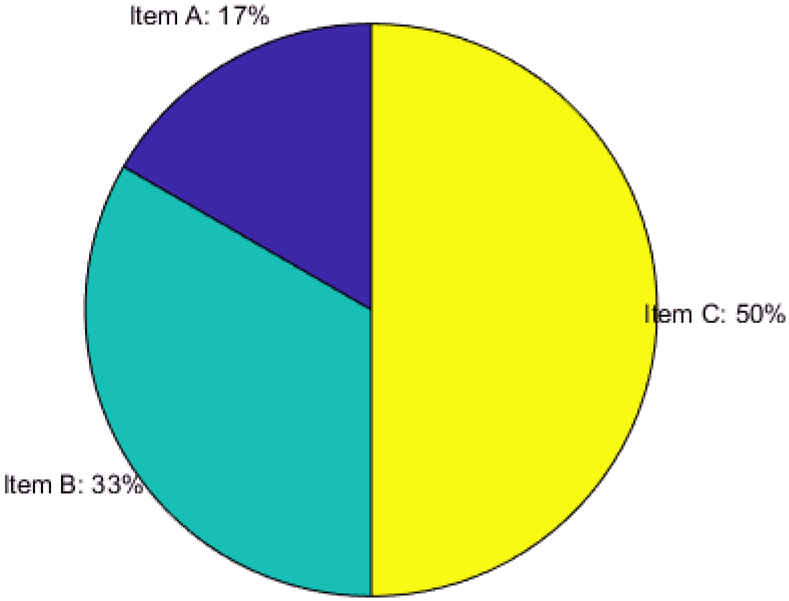
```
hText(1).String = combinedtxt(1);
hText(2).String = combinedtxt(2);
hText(3).String = combinedtxt(3);
```



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);
```

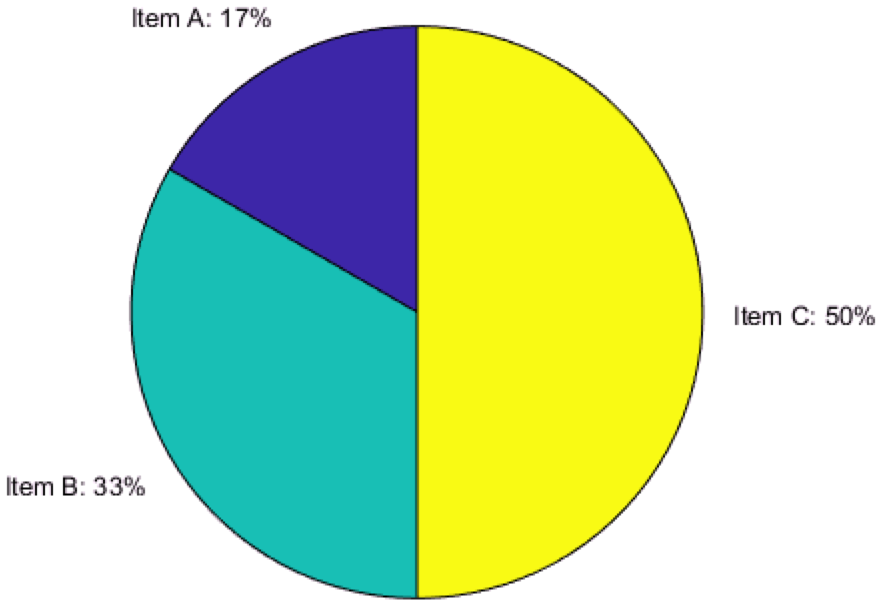
Use the change in width to calculate the horizontal distance to move each label. Store the calculated offsets in `offset`.

```
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" on page 6-31

Data Cursors with Histograms

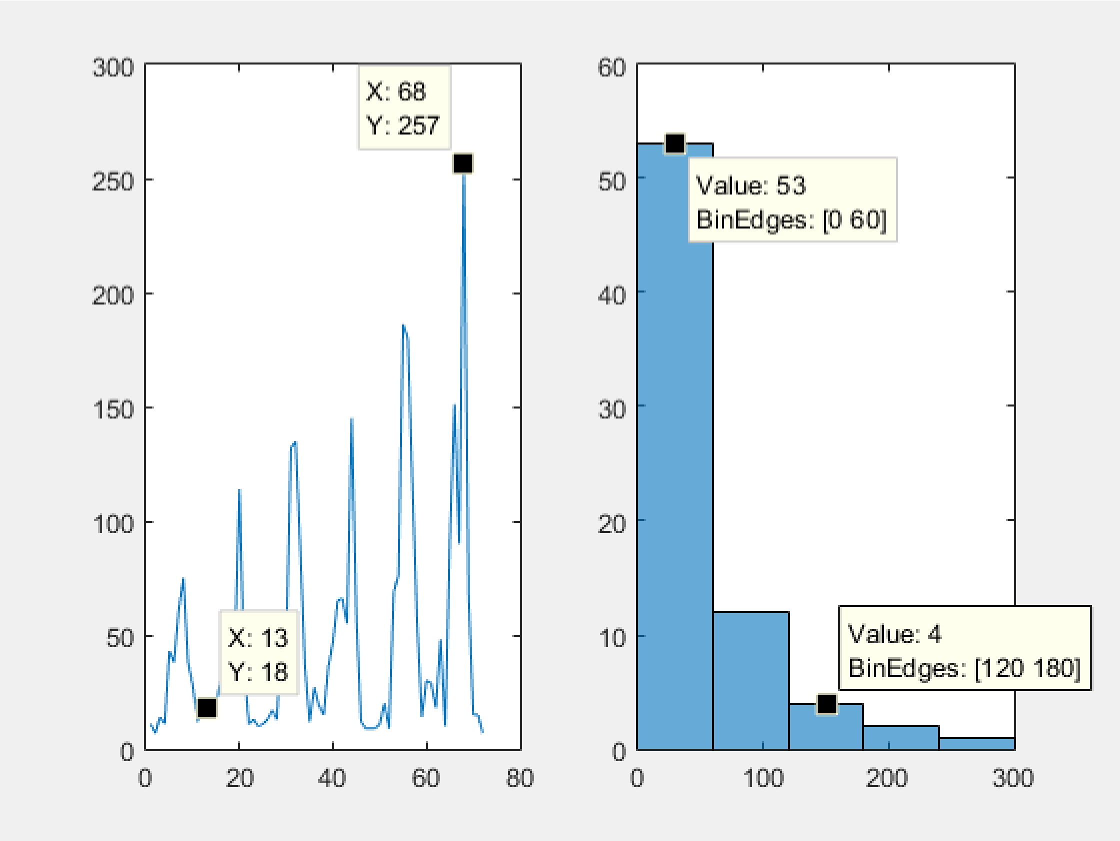
When you use the Data Cursor tool on a histogram plot, it customizes the data tips it displays in an appropriate way. Instead of providing *x*-, *y*-,*z*- coordinates, the datatips display the following information:

- Number of observations falling into the selected bin
- The 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
subplot(1,2,1);
plot(count(:))
subplot(1,2,2);
histogram(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 a datatip to a histogram, you can move the datatip to any other bar by clicking inside that bar. If you use the cursor keys to shift the datatip back or forth across the graph, the datatip moves to the preceding or succeeding bar.

Color Analysis with Bivariate Histogram

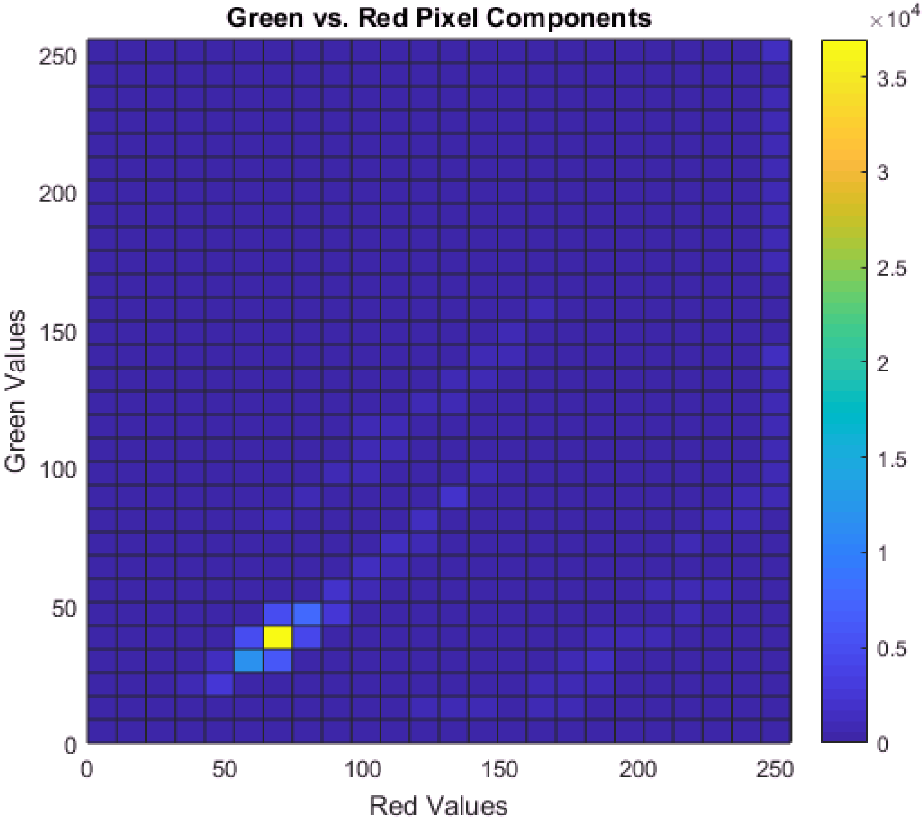
This example shows how to adjust the color scale of a bivariate histogram plot to reveal additional details about the bins.

Load the image `peppers.png`, which is a color photo of several types of peppers and other vegetables. The unsigned 8-bit integer array `rgb` contains the image data.

```
rgb = imread('peppers.png');
imshow(rgb)
```

Plot a bivariate histogram of the red and green RGB values for each pixel to visualize the color distribution.

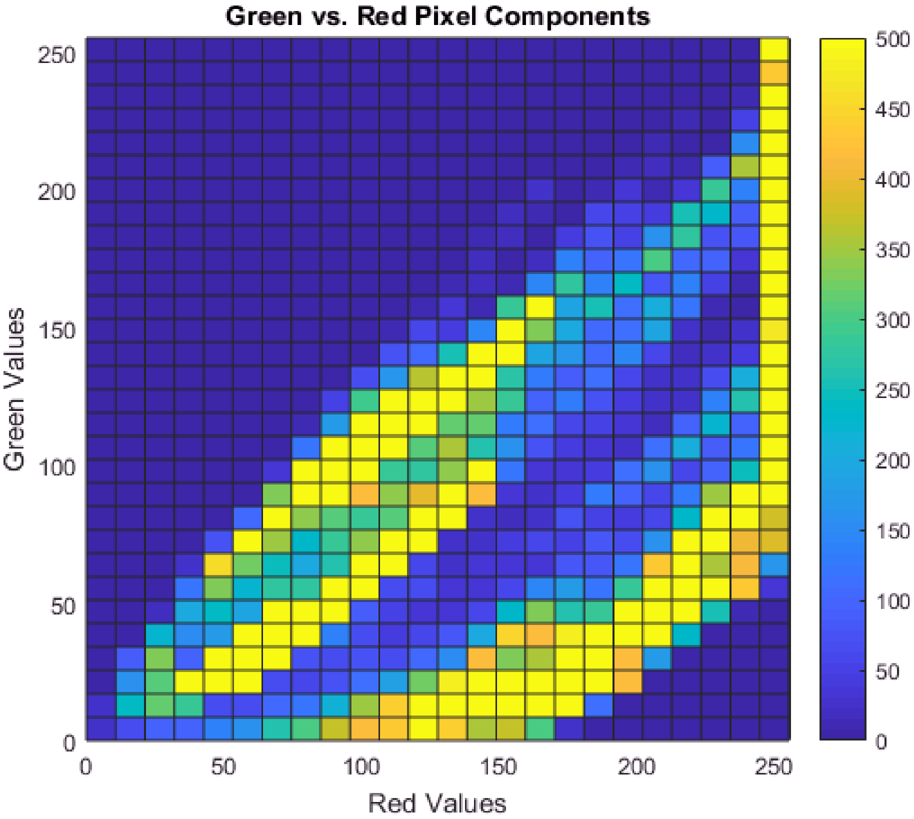
```
r = rgb(:,:,1);
g = rgb(:,:,2);
b = rgb(:,:,3);
histogram2(r,g,'DisplayStyle','tile','ShowEmptyBins','on', ...
    'XBinLimits',[0 255],'YBinLimits',[0 255]);
axis equal
colorbar
xlabel('Red Values')
ylabel('Green Values')
title('Green vs. Red Pixel Components')
```



The histogram is heavily weighted towards the bottom of the color scale because there are a few bins with very large counts. This results in most of the bins displaying as the first color in the colormap, blue. Without additional detail it is hard to draw any conclusions about which color is more dominant.

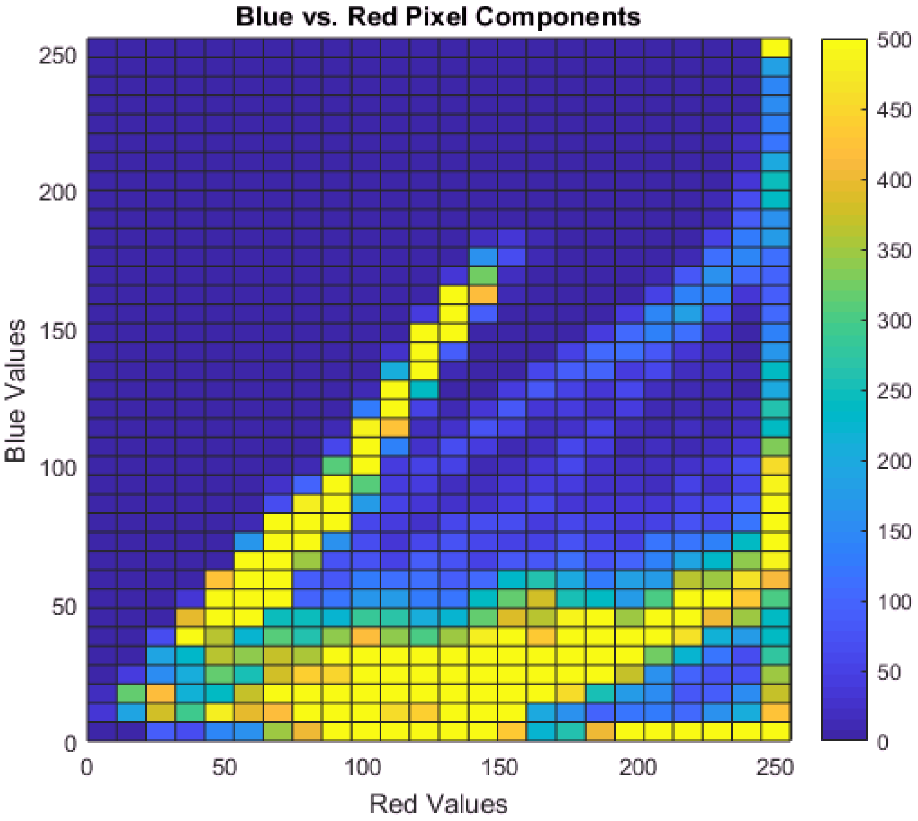
To view more detail, rescale the histogram color scale by setting the `CLim` property of the axes to have a range between 0 and 500. The result is that the histogram bins whose count is 500 or greater display as the last color in the colormap, yellow. Since most of the bin counts are within this smaller range, there is greater variation in the color of bins displayed.

```
ax = gca;
ax.CLim = [0 500];
```



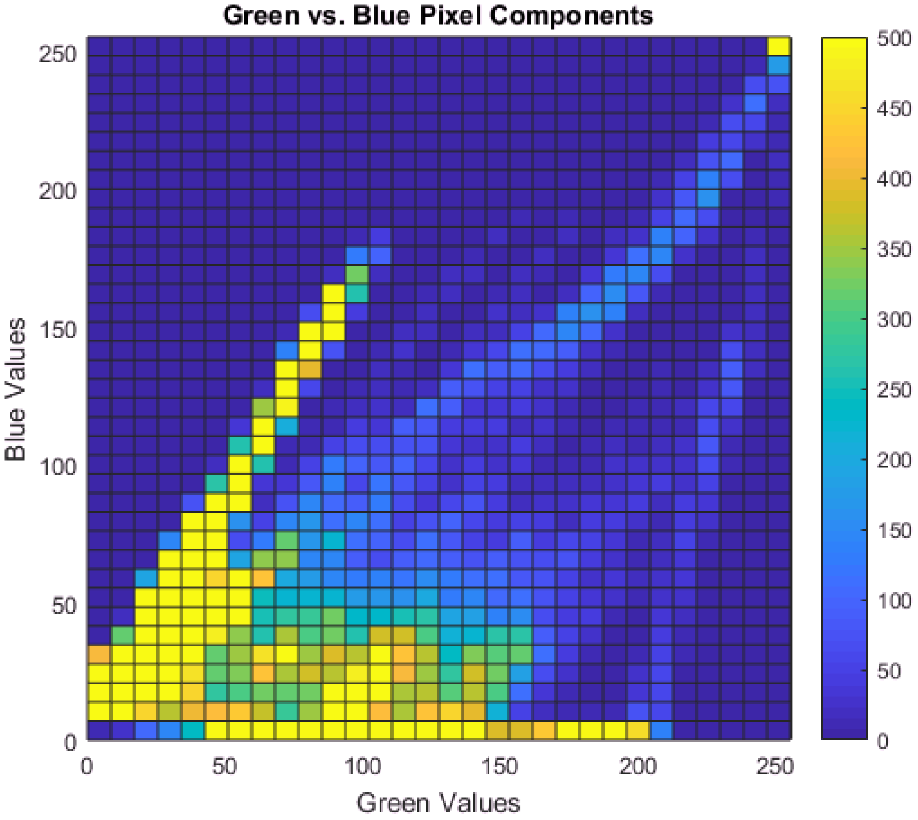
Use a similar method to compare the dominance of red vs. blue and green vs. blue.

```
histogram2(r,b,'DisplayStyle','tile','ShowEmptyBins','on',...
    'XBinLimits',[0 255],'YBinLimits',[0 255]);
axis equal
colorbar
xlabel('Red Values')
ylabel('Blue Values')
title('Blue vs. Red Pixel Components')
ax = gca;
ax.CLim = [0 500];
```



```
histogram2(g,b,'DisplayStyle','tile','ShowEmptyBins','on',...
    'XBinLimits',[0 255],'YBinLimits',[0 255]);
axis equal
```

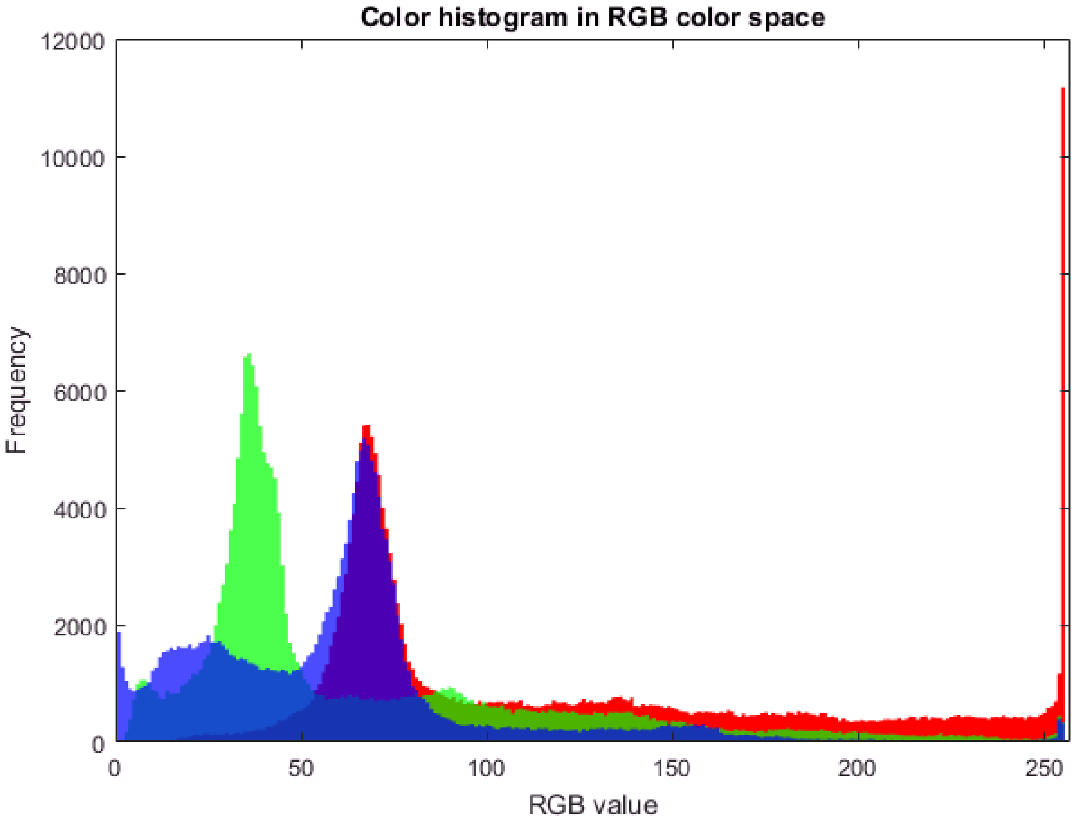
```
colorbar
xlabel('Green Values')
ylabel('Blue Values')
title('Green vs. Blue Pixel Components')
ax = gca;
ax.CLim = [0 500];
```



In each case, blue is the least dominant color signal. Looking at all three histograms, red appears to be the dominant color.

Confirm the results by creating a color histogram in the RGB color space. All three color components have spikes for smaller RGB values. However, the values above 100 occur more frequently in the red component than any other.

```
histogram(r,'BinMethod','integers','FaceColor','r','EdgeAlpha',0,'FaceAlpha',1)
hold on
histogram(g,'BinMethod','integers','FaceColor','g','EdgeAlpha',0,'FaceAlpha',0.7)
histogram(b,'BinMethod','integers','FaceColor','b','EdgeAlpha',0,'FaceAlpha',0.7)
xlabel('RGB value')
ylabel('Frequency')
title('Color histogram in RGB color space')
xlim([0 257])
```



See Also

histogram | histogram2

Control Categorical Histogram Display

This example shows how to use `histogram` to effectively view categorical data. You can use the name-value pairs `'NumDisplayBins'`, `'DisplayOrder'`, and `'ShowOthers'` to change the display of a categorical histogram. These options help you to better organize the data and reduce noise in the plot.

Create Categorical Histogram

The sample file `outages.csv` contains data representing electric utility outages in the United States. The file contains six columns: `Region`, `OutageTime`, `Loss`, `Customers`, `RestorationTime`, and `Cause`.

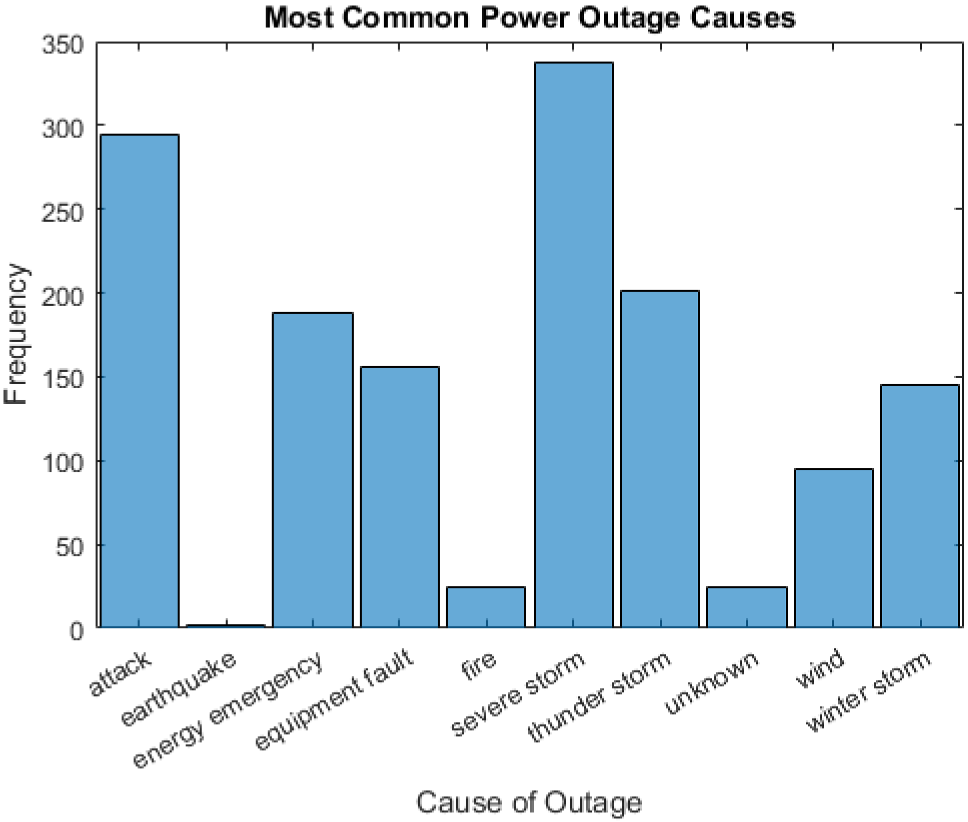
Read the `outages.csv` file as a table. Use the `'Format'` option to specify the kind of data each column contains: categorical (`'%C'`), floating-point numeric (`'%f'`), or datetime (`'%D'`). Index into the first few rows of data to see the variables.

```
data_formats = '%C%D%f%f%D%C';
C = readtable('outages.csv','Format',data_formats);
first_few_rows = C(1:10,:)
```

```
first_few_rows =

  10×6 table

     Region          OutageTime        Loss      Customers     RestorationTime          C
    _________    ________________    ______    __________    ________________    _____

    SouthWest    2002-02-01 12:18    458.98    1.8202e+06    2002-02-07 16:50    winter
    SouthEast    2003-01-23 00:49    530.14    2.1204e+05    NaT                 winter
    SouthEast    2003-02-07 21:15     289.4    1.4294e+05    2003-02-17 08:14    winter
    West         2004-04-06 05:44    434.81    3.4037e+05    2004-04-06 06:10    equipm
    MidWest      2002-03-16 06:18    186.44    2.1275e+05    2002-03-18 23:23    severe
    West         2003-06-18 02:49         0             0    2003-06-18 10:54    attack
    West         2004-06-20 14:39    231.29           NaN    2004-06-20 19:16    equipm
    West         2002-06-06 19:28    311.86           NaN    2002-06-07 00:51    equipm
    NorthEast    2003-07-16 16:23    239.93         49434    2003-07-17 01:12    fire
    MidWest      2004-09-27 11:09    286.72         66104    2004-09-27 16:37    equipm
```

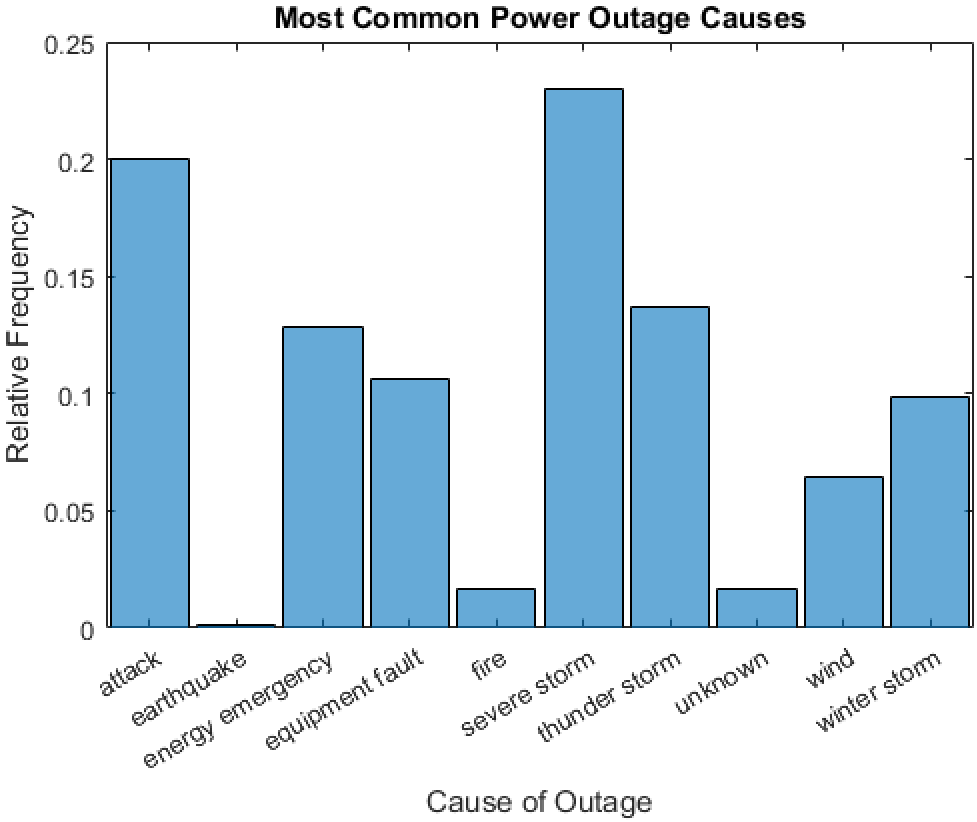
Plot a categorical histogram of the `Cause` variable. Specify an output argument to return a handle to the histogram object.

```
h = histogram(C.Cause);
xlabel('Cause of Outage')
ylabel('Frequency')
title('Most Common Power Outage Causes')
```



Change the normalization of the histogram to use the `'probability'` normalization, which displays the relative frequency of each outage cause.

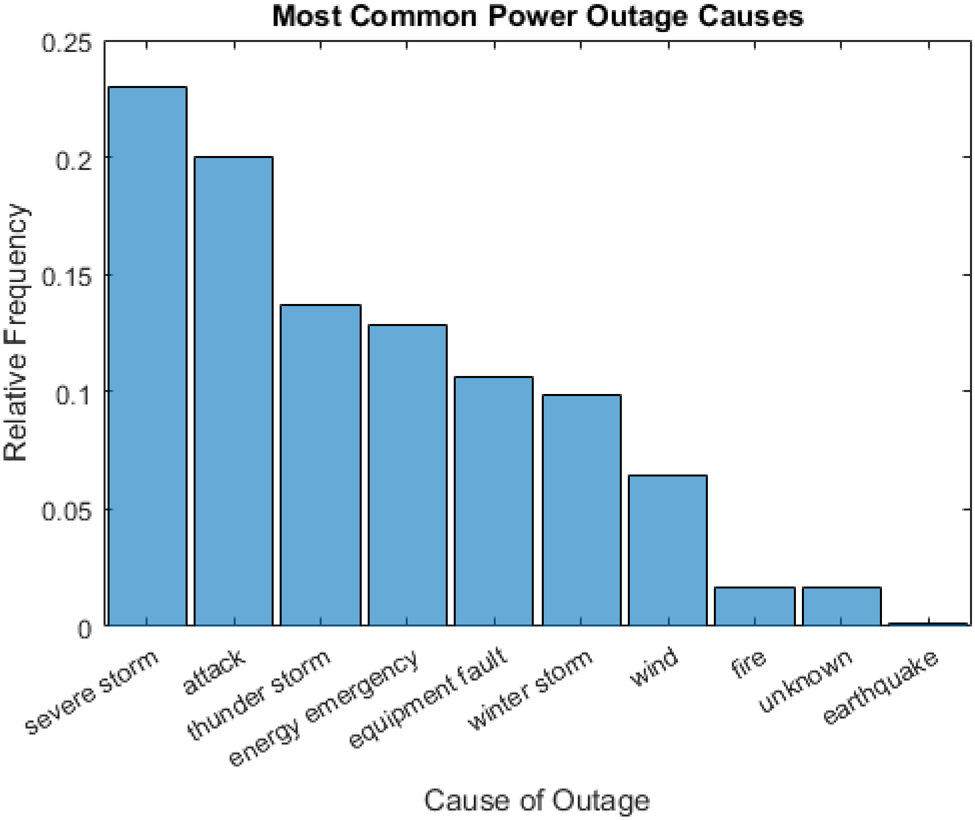
```
h.Normalization = 'probability';
ylabel('Relative Frequency')
```



Change Display Order

Use the `'DisplayOrder'` option to sort the bins from largest to smallest.

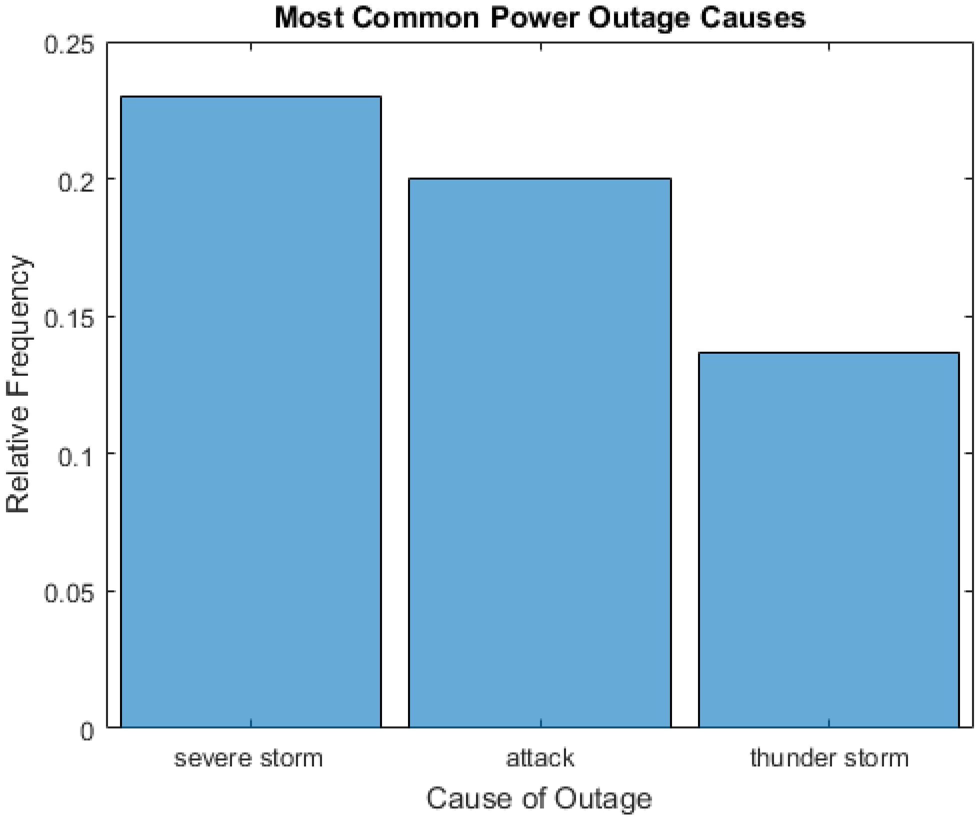
```
h.DisplayOrder = 'descend';
```



Truncate Number of Bars Displayed

Use the `'NumDisplayBins'` option to display only three bars in the plot. The displayed probabilities no longer add to 1 since the undisplayed data is still taken into account for normalization.

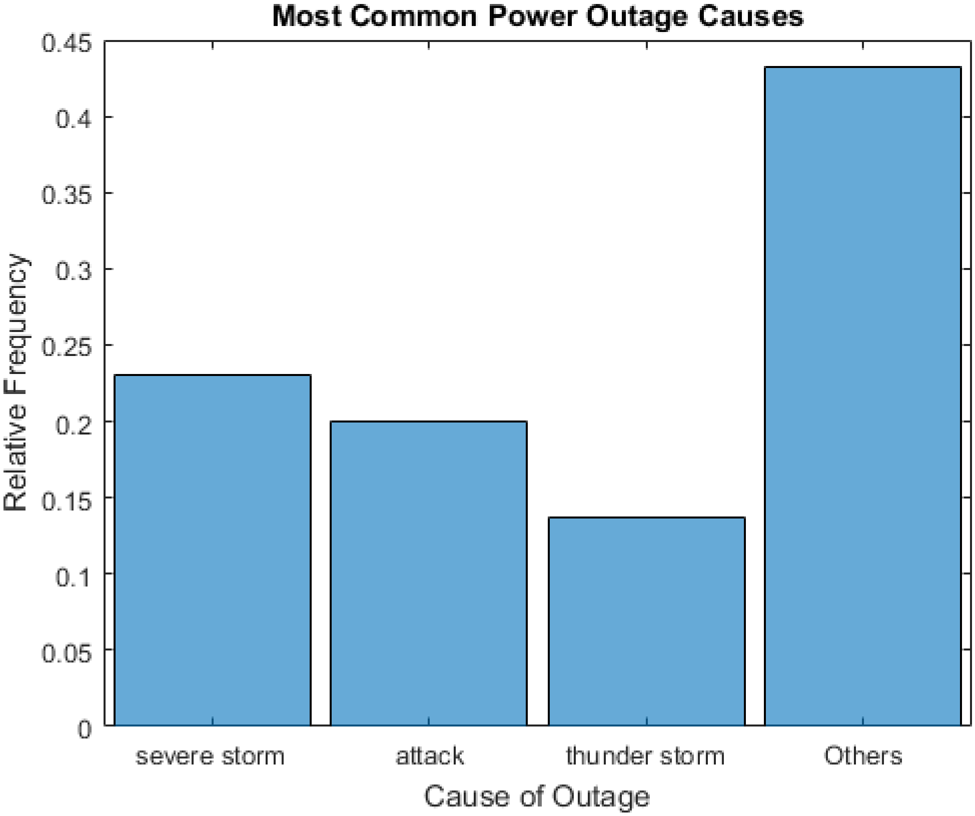
```
h.NumDisplayBins = 3;
```



Summarize Excluded Data

Use the `'ShowOthers'` option to summarize all of the excluded bars, so that the displayed probabilities again add to 1.

```
h.ShowOthers = 'on';
```

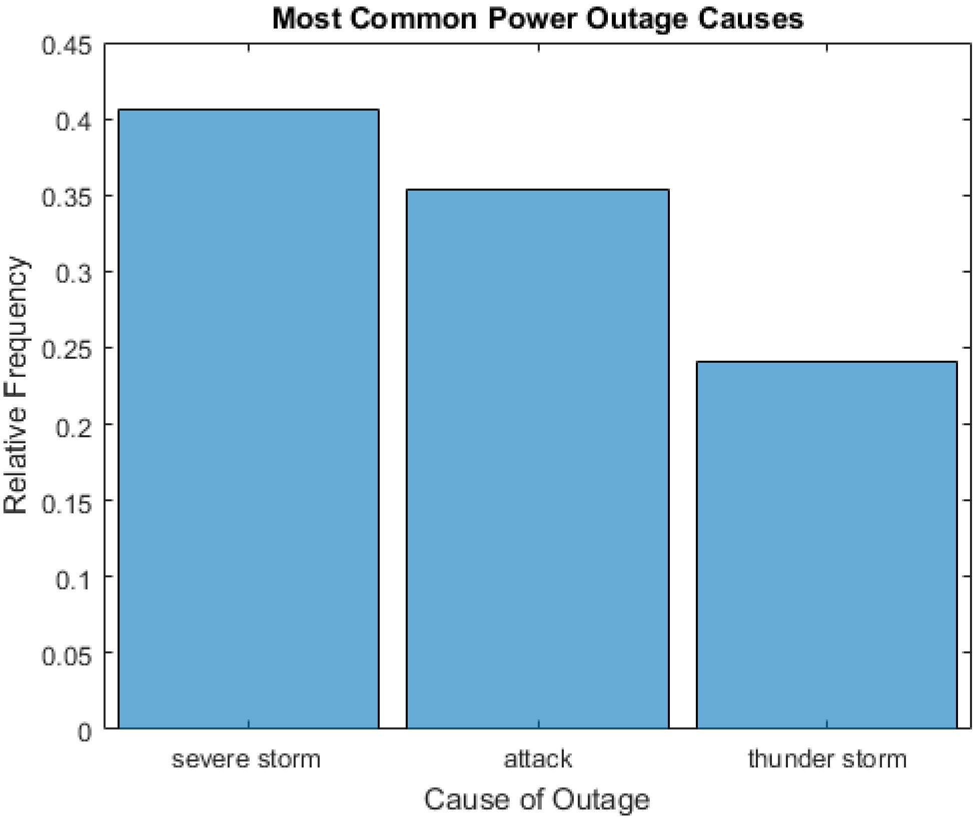


Limit Normalization to Display Data

Prior to R2017a, the `histogram` and `histcounts` functions used only *binned* data to calculate normalizations. This behavior meant that if some of the data ended up outside the bins, it was ignored for the purposes of normalization. However, in MATLAB® R2017a, the behavior changed to always normalize using the total number of elements in the input data. The new behavior is more intuitive, but if you prefer the old behavior, then you need to take a few special steps to limit the normalization only to the binned data.

Instead of normalizing over all of the input data, you can limit the probability normalization to the data that is displayed in the histogram. Simply update the `Data` property of the histogram object to remove the other categories. The `Categories` property reflects the categories displayed in the histogram. Use `setdiff` to compare the two property values and remove any category from `Data` that is not in `Categories`. Then remove all of the resulting `undefined` categorical elements from the data, leaving only elements in the displayed categories.

```
h.ShowOthers = 'off';
cats_to_remove = setdiff(categories(h.Data),h.Categories);
h.Data = removecats(h.Data,cats_to_remove);
h.Data = rmmissing(h.Data);
```



The normalization is now based only on the three remaining categories, so the three bars add to 1.

See Also

`categorical` | `histogram` | histogram Properties

Replace Discouraged Instances of hist and histc

In this section...

“Old Histogram Functions (`hist`, `histc`)” on page 6-58

“Recommended Histogram Functions” on page 6-58

“Differences Requiring Code Updates” on page 6-59

Old Histogram Functions (`hist`, `histc`)

Earlier versions of MATLAB use the `hist` and `histc` functions as the primary way to create histograms and calculate histogram bin counts. These functions, while good for some general purposes, have limited overall capabilities. The use of `hist` and `histc` in new code is discouraged for these reasons (among others):

- After using `hist` to create a histogram, modifying properties of the histogram is difficult and requires recomputing the entire histogram.
- The default behavior of `hist` is to use 10 bins, which is not suitable for many data sets.
- Plotting a normalized histogram requires manual computations.
- `hist` and `histc` do not have consistent behavior.

Recommended Histogram Functions

The `histogram`, `histcounts`, and `discretize` functions dramatically advance the capabilities of histogram creation and calculation in MATLAB, while still promoting consistency and ease of use. `histogram`, `histcounts`, and `discretize` are the recommended histogram creation and computation functions for new code.

Of particular note are the following changes, which stand as *improvements* over `hist` and `histc`:

- `histogram` can return a histogram object. You can use the object to modify properties of the histogram.
- Both `histogram` and `histcounts` have automatic binning and normalization capabilities, with several common options built-in.
- `histcounts` is the primary calculation function for `histogram`. The result is that the functions have consistent behavior.

- `discretize` provides additional options and flexibility for determining the bin placement of each element.

Differences Requiring Code Updates

Despite the aforementioned improvements, there are several important *differences* between the old and now recommended functions, which might require updating your code. The tables summarize the differences between the functions and provide suggestions for updating code.

Code Updates for `hist`

Difference	Old behavior with `hist`	New behavior with `histogram`

Input matrices	`hist` creates a histogram for each column of an input matrix and plots the histograms side-by-side in the same figure.
	`A = randn(100,2);`
	`hist(A)`
		`histogram` treats the input matrix as a single tall vector and creates a single histogram. To plot multiple histograms, create a different histogram object for each column of data. Use the `hold on` command to plot the histograms in the same figure.
		`A = randn(100,2);`
		`h1 = histogram(A(:,1),10)`
		`edges = h1.BinEdges;`
		`hold on`
		`h2 = histogram(A(:,2),edges)`
		The above code example uses the same bin edges for each histogram, but in some cases it is better to set the `BinWidth` of each histogram to be the same instead. Also, for display purposes, it might be helpful to set the `FaceAlpha` property of each histogram, as this affects the

Difference	Old behavior with `hist`	New behavior with `histogram`
		transparency of overlapping bars.
Bin specification	`hist` accepts the bin *centers* as a second input.	`histogram` accepts the bin *edges* as a second input. To convert bin centers into bin edges for use with `histogram`, see “Convert Bin Centers to Bin Edges” on page 6-64. **Note:** In cases where the bin centers used with `hist` are integers, such as `hist(A,-3:3)`, use the new built-in binning method of `histogram` for integers. `histogram(A,'BinLimits',[-3,3],'BinM`

Output arguments	`hist` returns the bin counts as an output argument, and optionally can return the bin centers as a second output argument.
	`A = randn(100,1);`
	`[N, Centers] = hist(A)`
		`histogram` returns a histogram object as an output argument. The object contains many properties of interest (bin counts, bin edges, and so on). You can modify aspects of the histogram by changing its property values. For more information, see histogram.
		`A = randn(100,1);`
		`h = histogram(A);`
		`N = h.Values`
		`Edges = h.BinEdges`

Difference	Old behavior with `hist`	New behavior with `histogram`
		Note: To calculate bin counts (without plotting a histogram), replace `[N, Centers] = hist(A)` with `[N,edges] = histcounts(A,nbins)`.

Default number of bins	`hist` uses 10 bins by default.	Both `histogram` and `histcounts` use an automatic binning algorithm by default. The number of bins is determined by the size and spread of the input data.
		`A = randn(100,1);`
		`histogram(A)`
		`histcounts(A)`

Bin limits	`hist` uses the minimum and maximum finite data values to determine the left and right edges of the first and last bar in the plot. `-Inf` and `Inf` are included in the first and last bin, respectively.	If `BinLimits` is not set, then `histogram` uses rational bin limits based on, but not exactly equal to, the minimum and maximum finite data values. `histogram` ignores `Inf` values unless one of the bin edges explicitly specifies `Inf` or `-Inf` as a bin edge.
		To reproduce the results of `hist(A)` for finite data (no `Inf` values), use 10 bins and explicitly set `BinLimits` to the minimum and maximum data values.
		`A = randi(5,100,1);`
		`histogram(A,10,'BinLimits',[min(A) m`

Code Updates for `histc`

Difference	Old behavior with `histc`	New behavior with `histcounts`

Input matrices	`histc` calculates the bin counts for each column of input data. For an input matrix of size `m`-by-`n`, `histc` returns a matrix of bin counts of size `length(edges)`-by-`n`.
	`A = randn(100,10);`
	`edges = -4:4;`
	`N = histc(A,edges)`
		`histcounts` treats the input matrix as a single tall vector and calculates the bin counts for the entire matrix.
		`A = randn(100,10);`
		`edges = -4:4;`
		`N = histcounts(A,edges)`
		Use a for-loop to calculate bin counts over each column.
		`A = randn(100,10);`
		`nbins = 10;`
		`N = zeros(nbins, size(A,2));`
		`for k = 1:size(A,2)`
		`   N(:,k) = histcounts(A(:,k),nbins)`
		`end`
		If performance is a problem due to a large number of columns in the matrix, then consider continuing to use `histc` for the column-wise bin counts.

Values included in last bin	`histc` includes an element `A(i)` in the last bin if `A(i) == edges(end)`. The output, `N`, is a vector with `length(edges)` elements containing the bin counts. Values falling outside the bins are not counted.	`histcounts` includes an element `A(i)` in the last bin if `edges(end-1) <= A(i) <= edges(end)`. In other words, `histcounts` combines the last two bins from `histc` into a single final bin. The output, `N`, is a vector with `length(edges)-1` elements containing the bin counts. If you specify the bin edges,

Difference	Old behavior with `histc`	New behavior with `histcounts`

		then values falling outside the bins are not counted. Otherwise, `histcounts` automatically determines the proper bin edges to use to include all of the data.
		`A = 1:4;`
		`edges = [1 2 2.5 3]`
		`N = histcounts(A)`
		`N = histcounts(A,edges)`
		The last bin from `histc` is primarily useful to count integers. To do this integer counting with `histcounts`, use the `'integers'` bin method:
		`N = histcounts(A,'BinMethod','intege`

Output arguments	`histc` returns the bin counts as an output argument, and optionally can return the bin indices as a second output argument.
	`A = randn(15,1);`
	`edges = -4:4;`
	`[N,Bin] = histc(A,edges)`
		• For bin count calculations like `N = histc(A,edges)` or `[N,bin] = histc(A,edges)`, use `histcounts`. The `histcounts` function returns the bin counts as an output argument, and optionally can return the bin edges as a second output, or the bin indices as a third output.
		`A = randn(15,1);`
		`[N,Edges,Bin] = histcounts(A)`
		• For bin placement calculations like `[~,Bin]`

Difference	Old behavior with `histc`	New behavior with `histcounts`

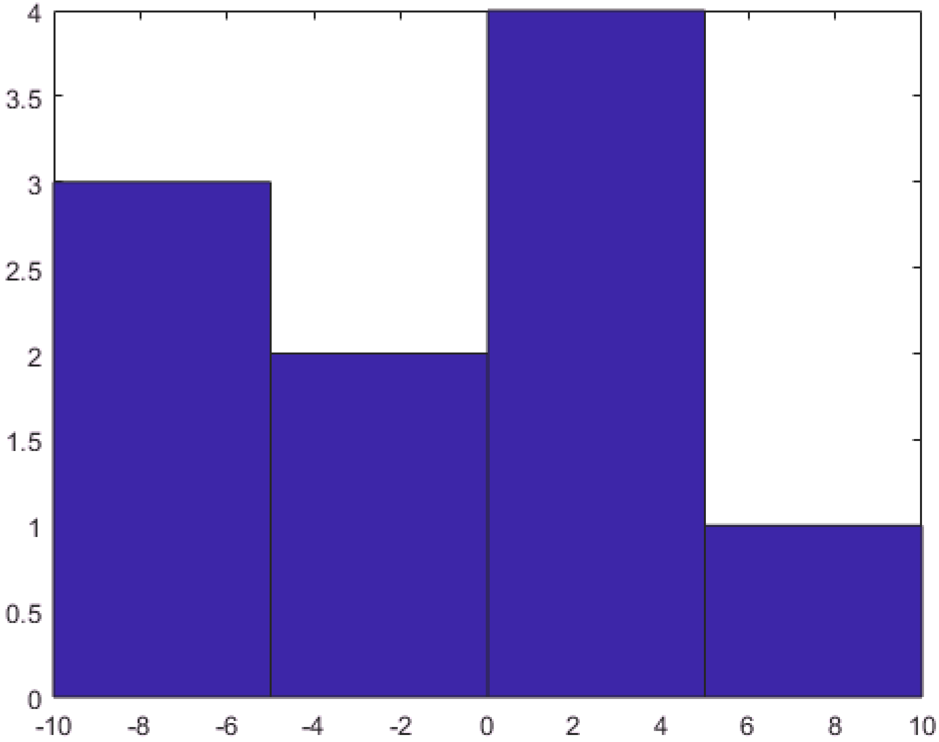
		= histc(A,edges), use `discretize`. The `discretize` function offers additional options for determining the bin placement of each element.
		`A = randn(15,1);`
		`edges = -4:4;`
		`Bin = discretize(A,edges)`

Convert Bin Centers to Bin Edges

The `hist` function accepts bin centers, whereas the `histogram` function accepts bin edges. To update code to use `histogram`, you might need to convert bin centers to bin edges to reproduce results achieved with `hist`.

For example, specify bin centers for use with `hist`. These bins have a uniform width.

```
A = [-9 -6 -5 -2 0 1 3 3 4 7];
centers = [-7.5 -2.5 2.5 7.5];
hist(A,centers)
```



To convert the bin centers into bin edges, calculate the midpoint between consecutive values in `centers`. This method reproduces the results of `hist` for both uniform and nonuniform bin widths.

```
d = diff(centers)/2;
edges = [centers(1)-d(1), centers(1:end-1)+d, centers(end)+d(end)];
```

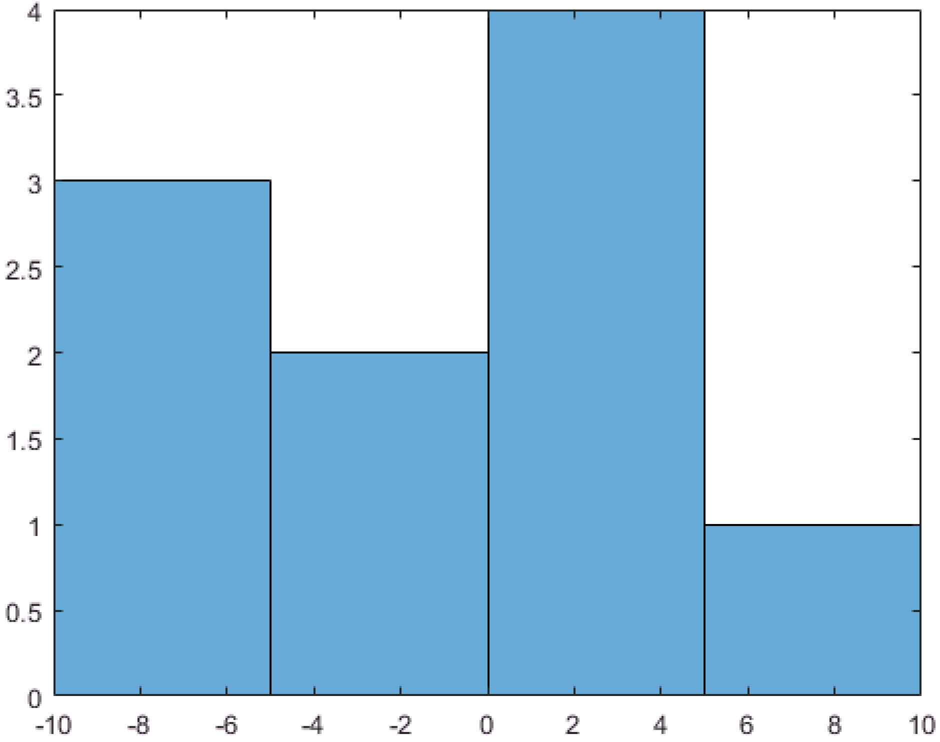
The `hist` function includes values falling on the right edge of each bin (the first bin includes both edges), whereas `histogram` includes values that fall on the left edge of each bin (and the last bin includes both edges). Shift the bin edges slightly to obtain the same bin counts as `hist`.

```
edges(2:end) = edges(2:end)+eps(edges(2:end))
```

```
edges =

   -10.0000    -5.0000     0.0000     5.0000    10.0000
```

Now, use `histogram` with the bin edges.

```
histogram(A,edges)
```

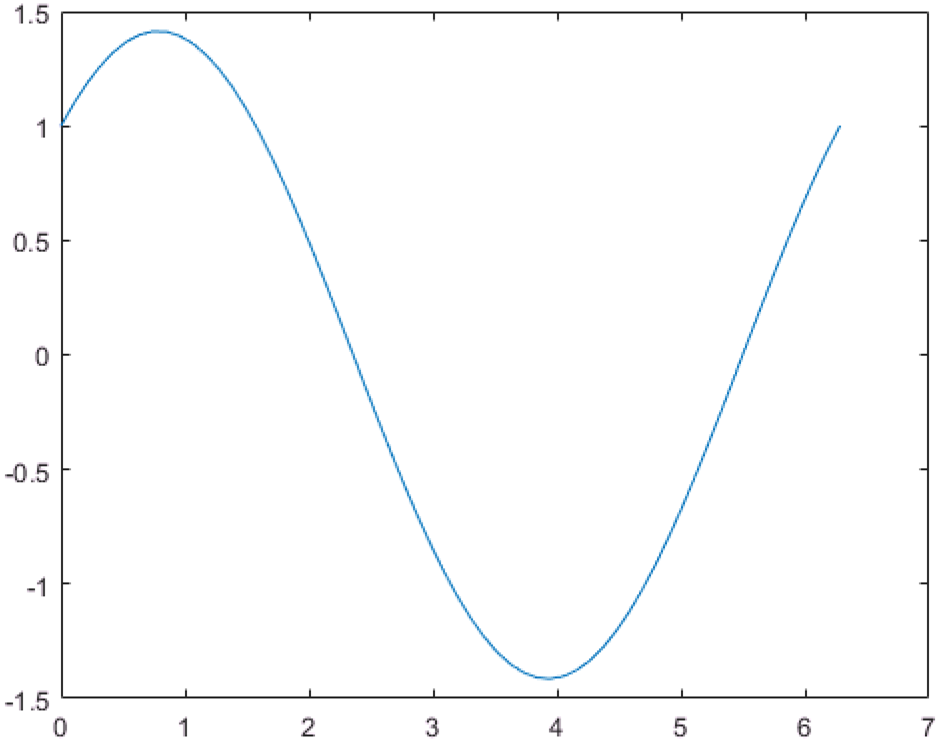


Combine Line and Stem Plots

This example shows how to combine a line plot and two stem plots. Then, it shows how to add a title, axis labels, and a legend.

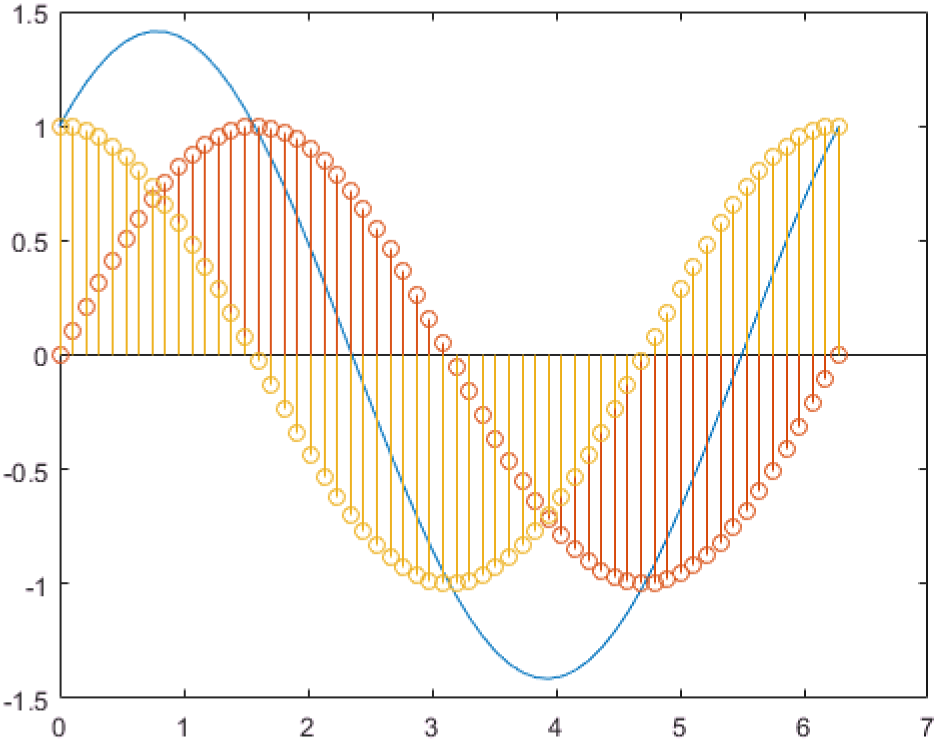
Create the data and plot a line.

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



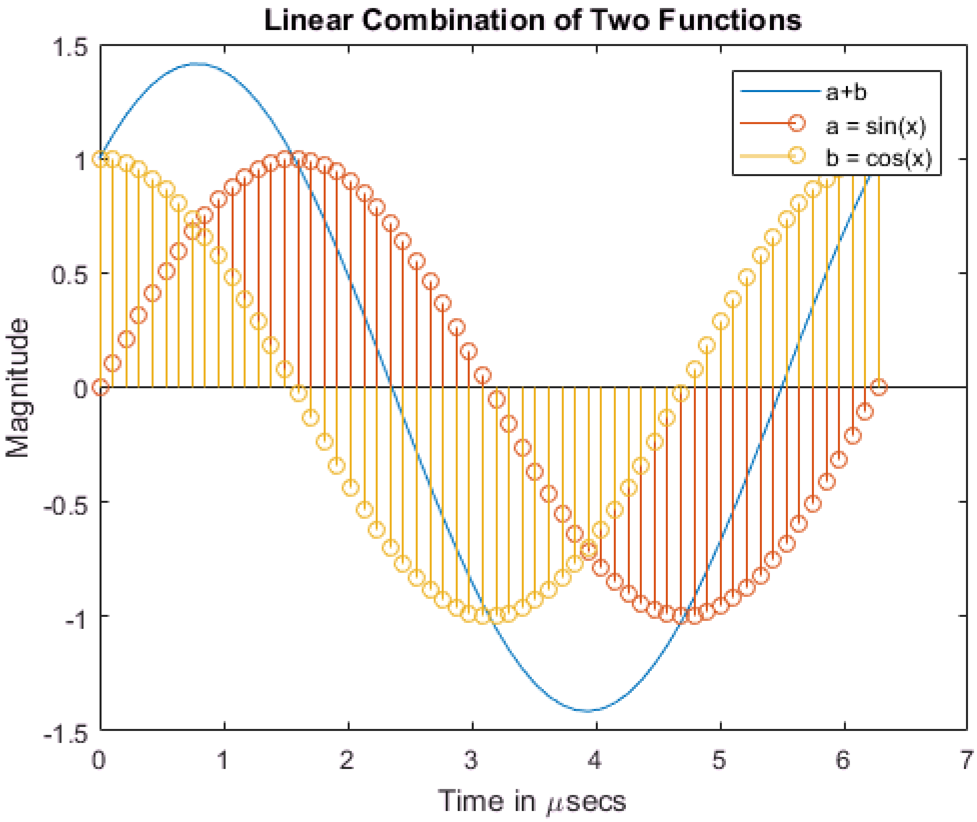
Add two stem plots to the axes. Prevent new plots from replacing existing plots using `hold on`.

```
hold on
stem(x,a)
stem(x,b)
hold off
```



Add a title, axis labels, and a legend. Specify the legend descriptions in the order that you create the plots.

```
title('Linear Combination of Two Functions')
xlabel('Time in \musecs')
ylabel('Magnitude')
legend('a+b','a = sin(x)','b = cos(x)')
```


Linear Combination of Two Functions
a+b
a = sin(x)
b = cos(x)
Magnitude
1.5
1
0.5
0
-0.5
-1
-1.5
0
1
2
3
4
5
6
7
Time in μsecs

Overlay Stairstep Plot and Line Plot

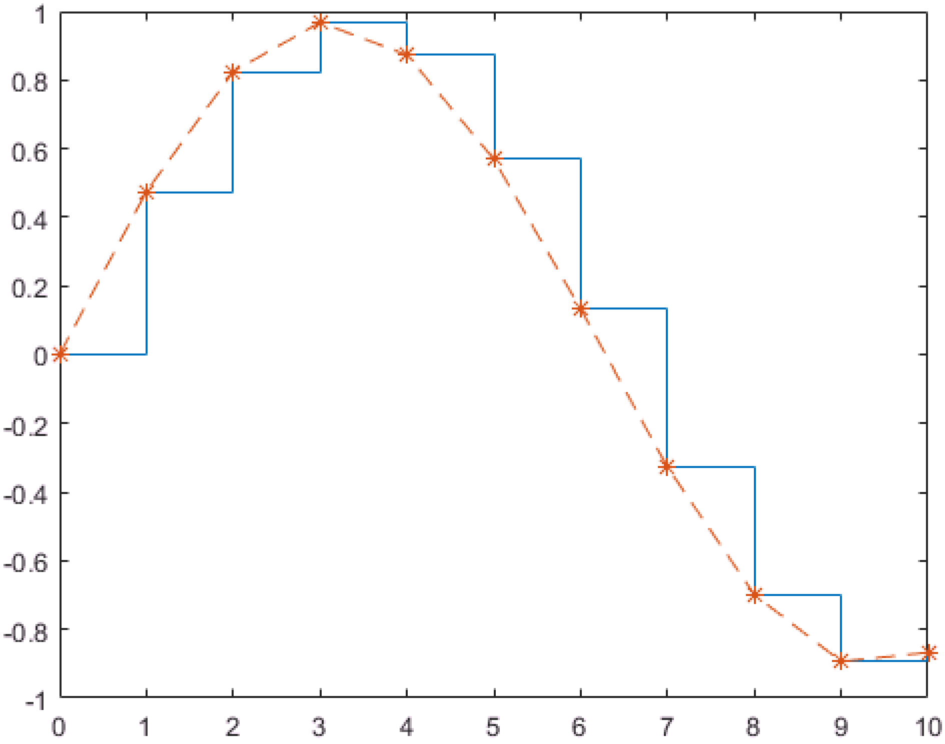
This example shows how to overlay a line plot on a stairstep plot.

Define the data to plot.

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

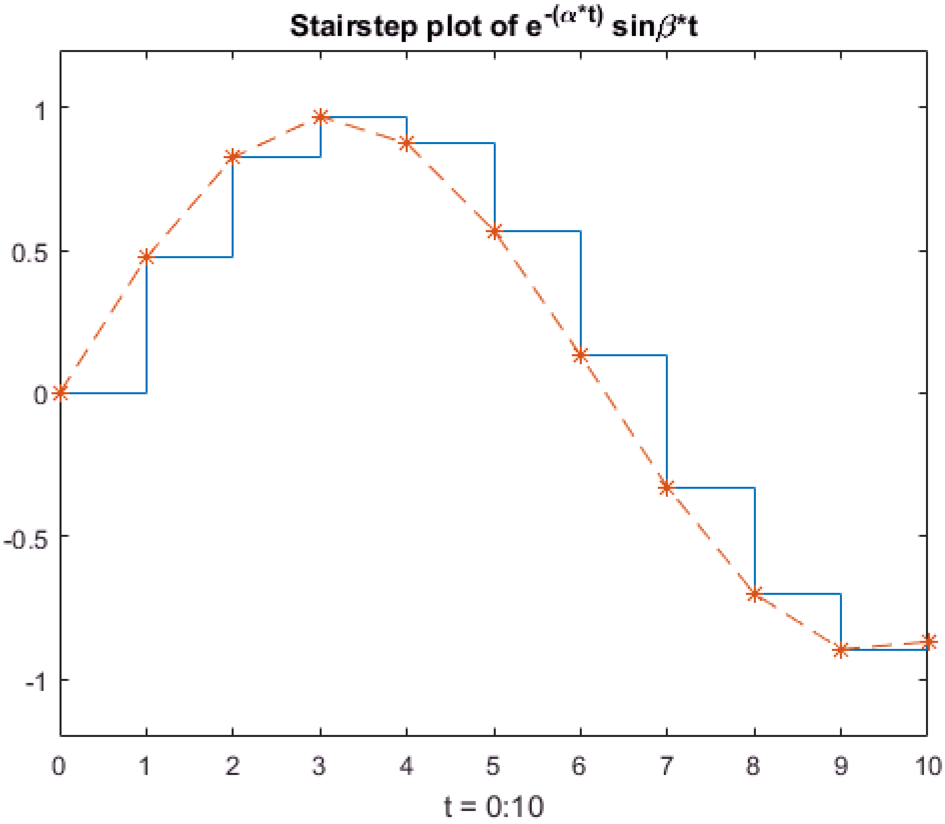
Display `f` as a stairstep plot. Use the `hold` function to retain the stairstep plot. Add a line plot of `f` using a dashed line with star markers.

```
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

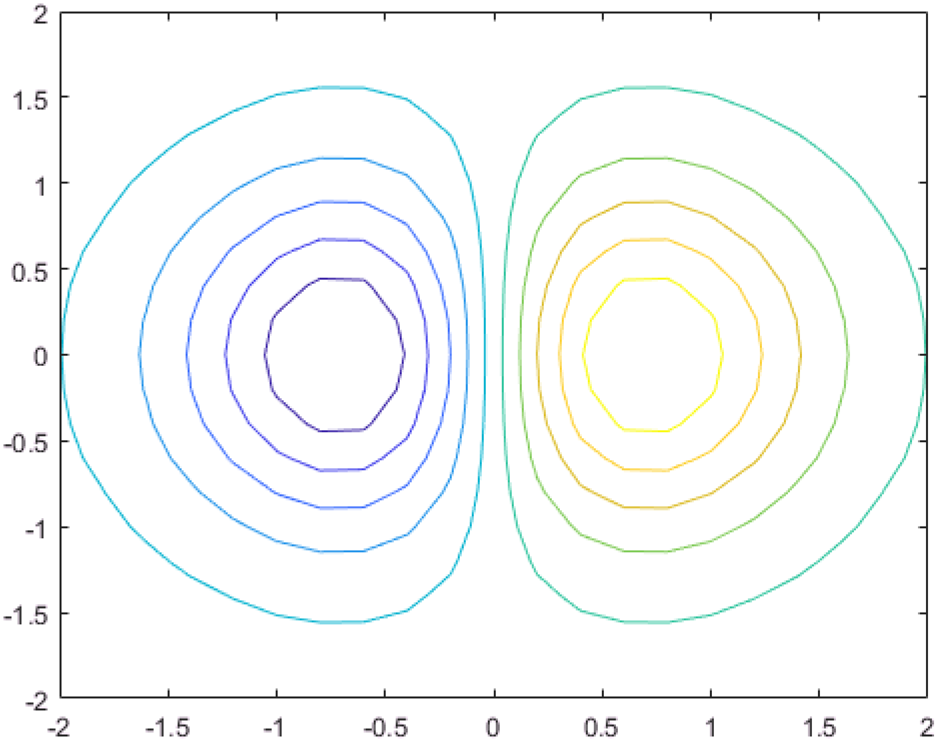
`axis` | `plot` | `stairs`

Combine Contour Plot and Quiver Plot

This example shows how to combine a contour plot and a quiver plot using the `hold` function.

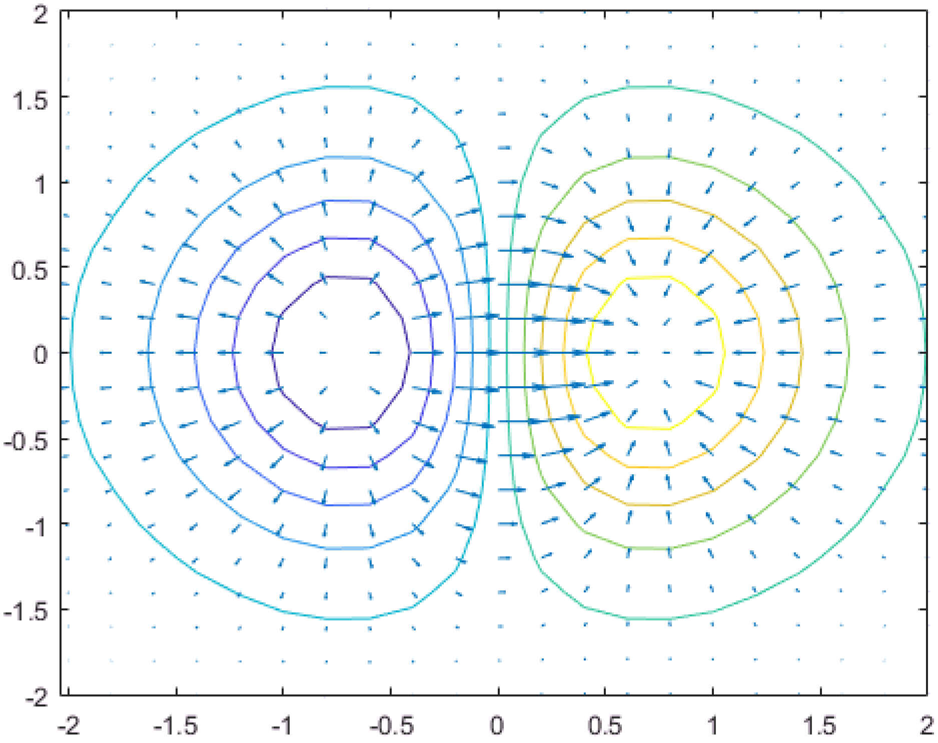
Plot 10 contours of $xe^{-x^2-y^2}$ over a grid from -2 to 2 in the *x* and *y* directions.

```
[X,Y] = meshgrid(-2:0.2:2);
Z = X .* exp(-X.^2 - Y.^2);
contour(X,Y,Z,10)
```



Calculate the 2-D gradient of Z using the gradient function. The gradient function returns U as the gradient in the *x*-direction and V as the gradient in the *y*-direction. Display arrows indicating the gradient values using the quiver function.

```
[U,V] = gradient(Z,0.2,0.2);
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, vz and a. Calculate z as the height as time varies from 0 to 1.

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

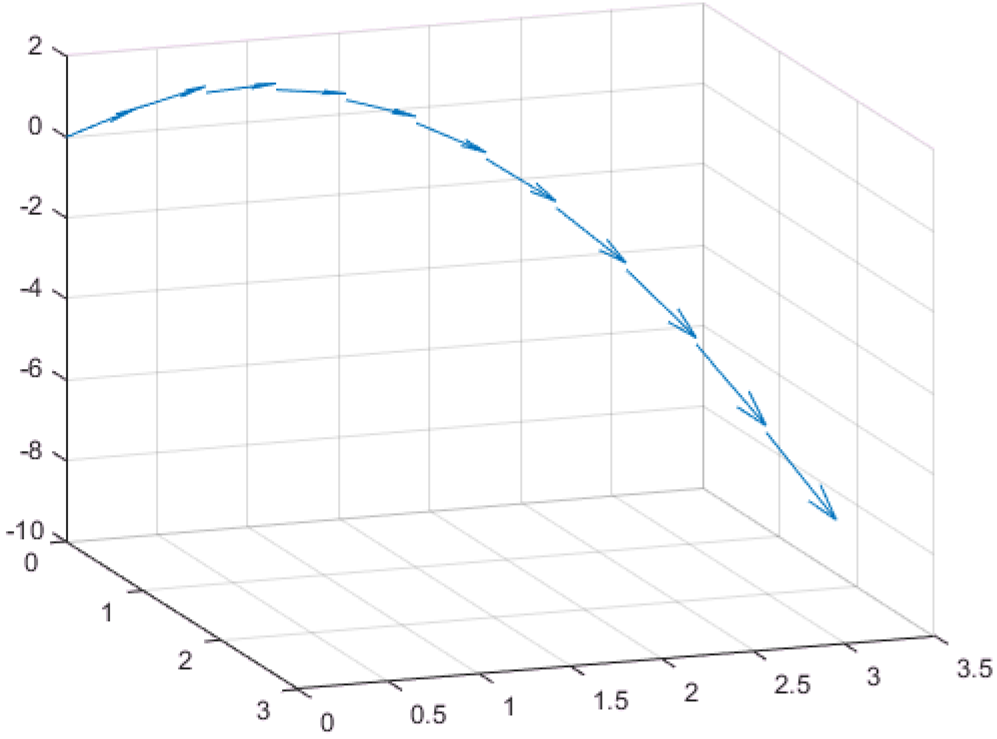
```
vz = 10; % velocity constant
a = -32; % acceleration constant
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
```

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])
```


2
0
-2
-4
-6
-8
-10
0
1
2
3
0
0.5
1
1.5
2
2.5
3
3.5

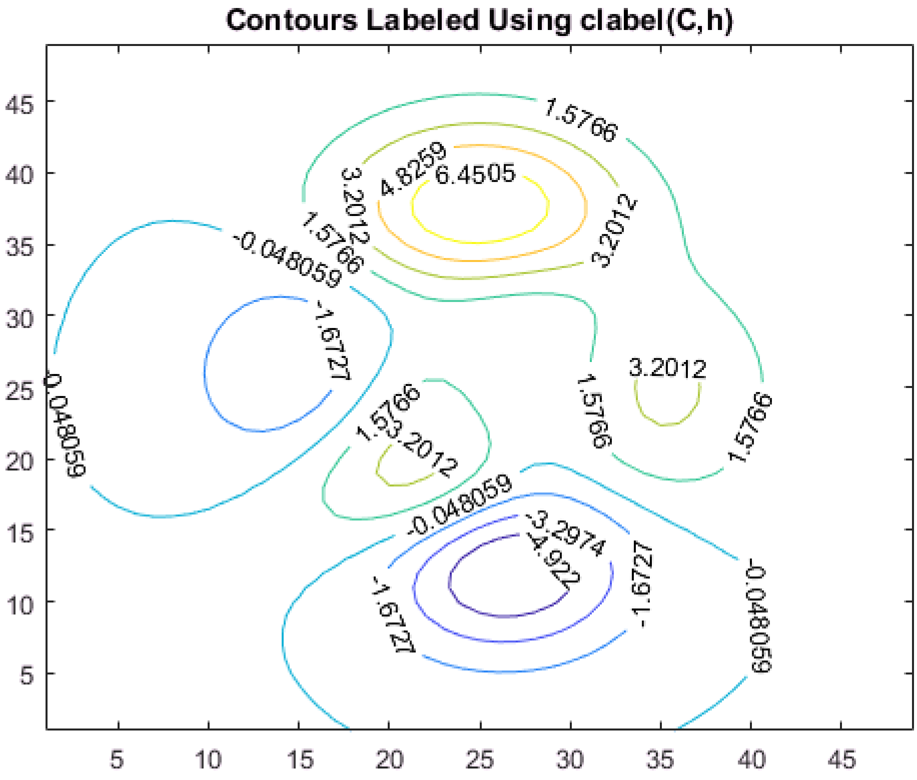
Label Contour Plot Levels

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`

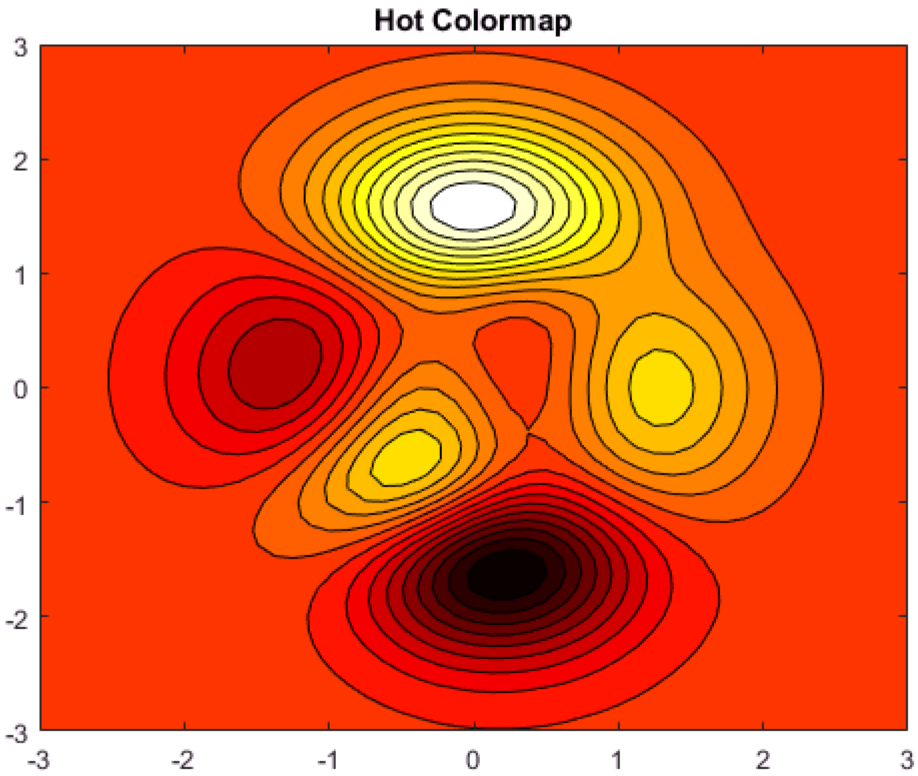
Change Fill Colors for Contour Plot

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.

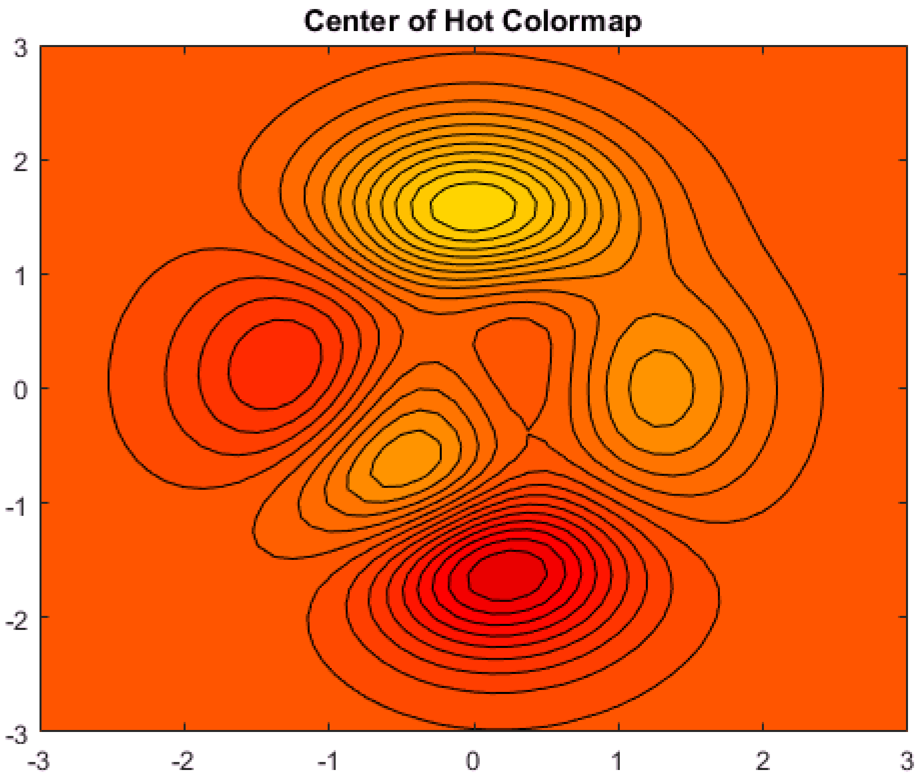
```
[X,Y,Z] = peaks;
figure
contourf(X,Y,Z,20)
colormap(hot)
title('Hot Colormap')
```



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.

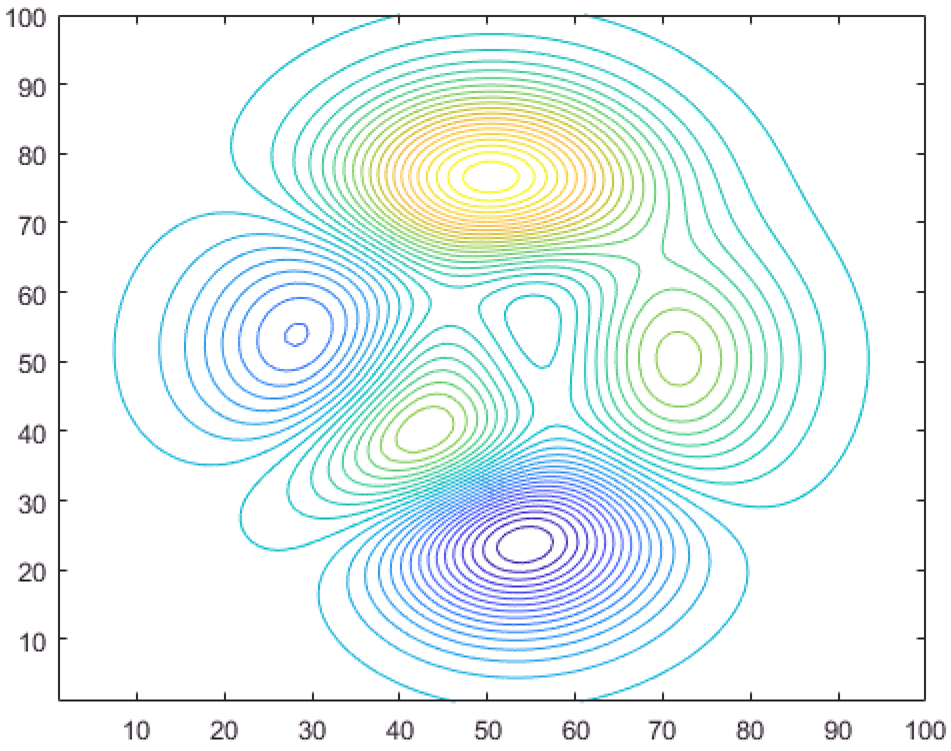
```
Z = peaks(100);
```

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)
```

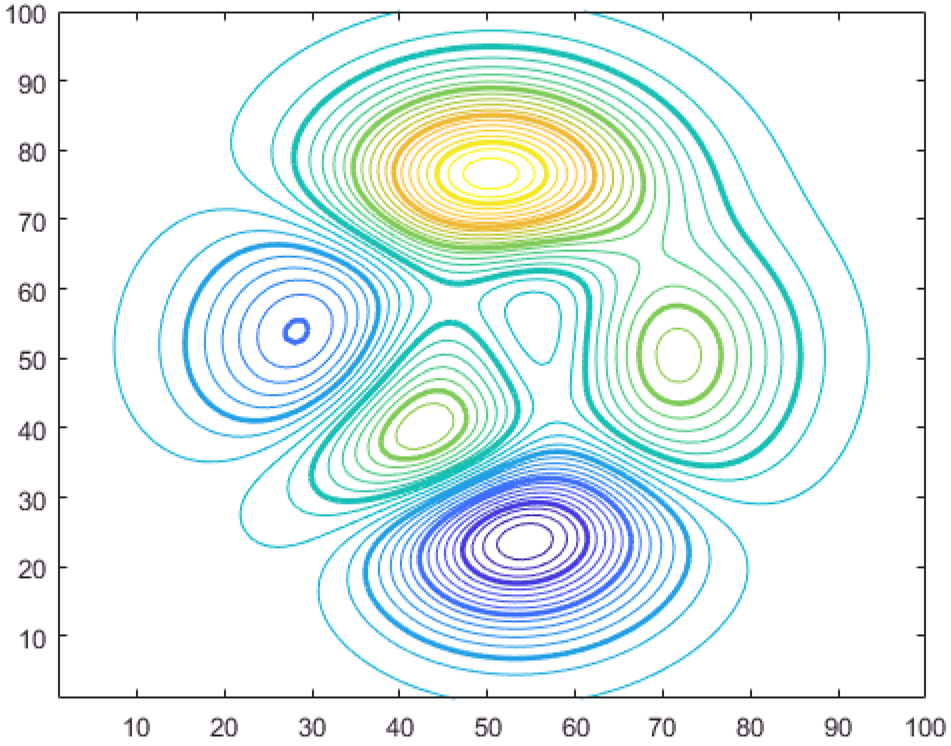


Define `zindex` as a vector of integer values between `zmin` and `zmax` indexed by 2.

```
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`

Animation Techniques

In this section...

"Updating the Screen" on page 6-84
"Optimizing Performance" on page 6-84

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

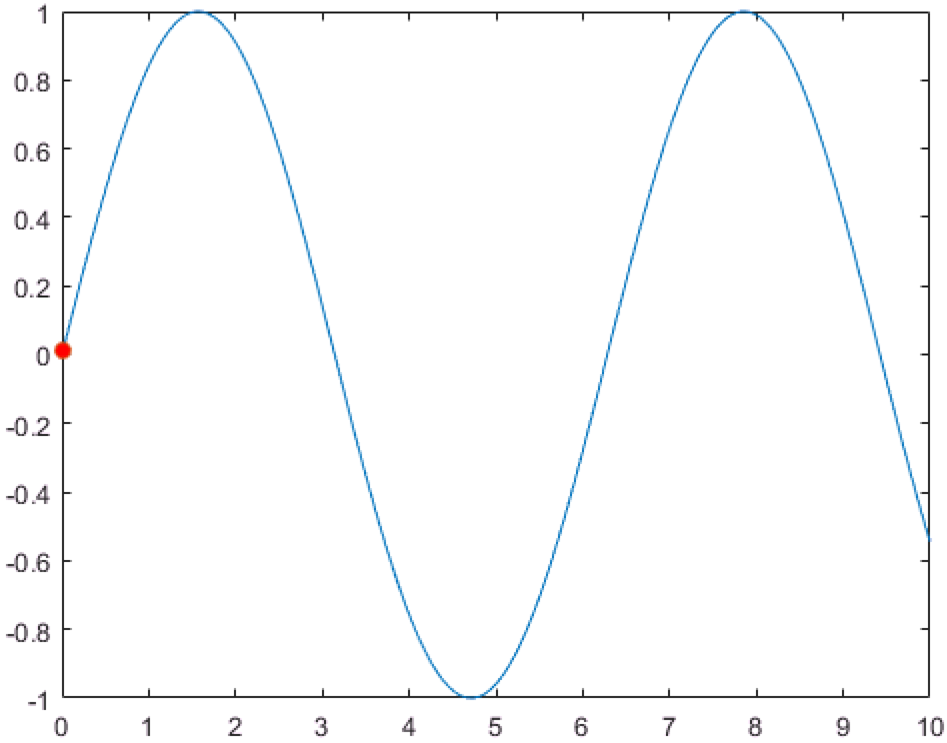
- “Trace Marker Along Line” on page 6-86
- “Move Group of Objects Along Line” on page 6-89
- “Line Animations” on page 6-97
- “Record Animation for Playback” on page 6-100

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.

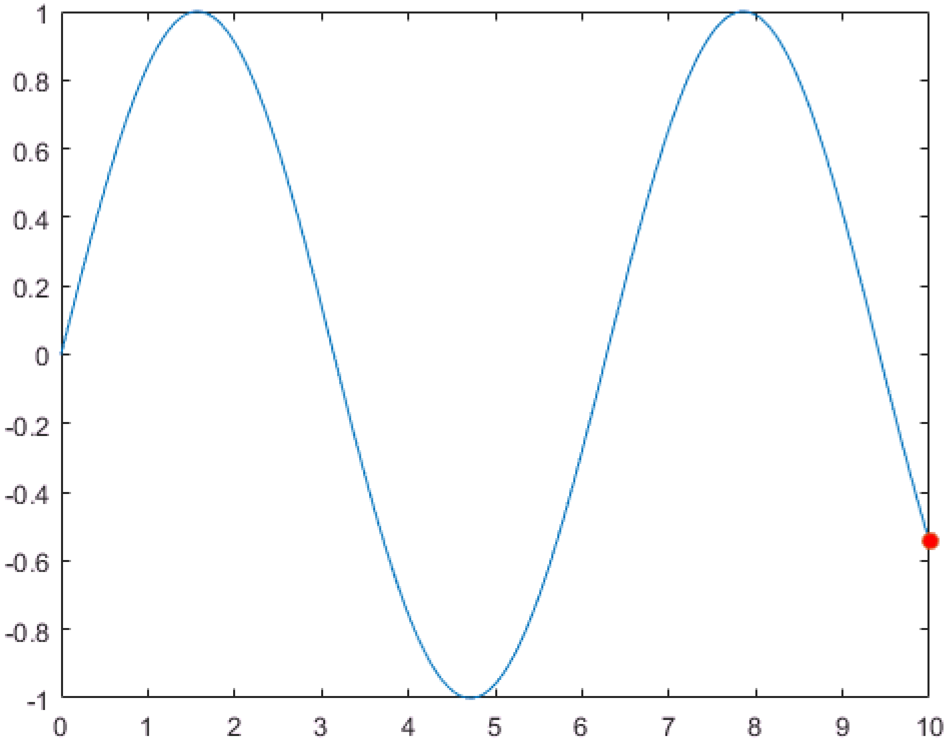
```
x = linspace(0,10,1000);
y = sin(x);
plot(x,y)
hold on
p = plot(x(1),y(1),'o','MarkerFaceColor','red');
hold off
axis manual
```



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

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead, such as `set(p,'XData',x(k))`.

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



The animation shows the marker moving along the line.

See Also

`drawnow` | `linspace` | `plot`

Related Examples

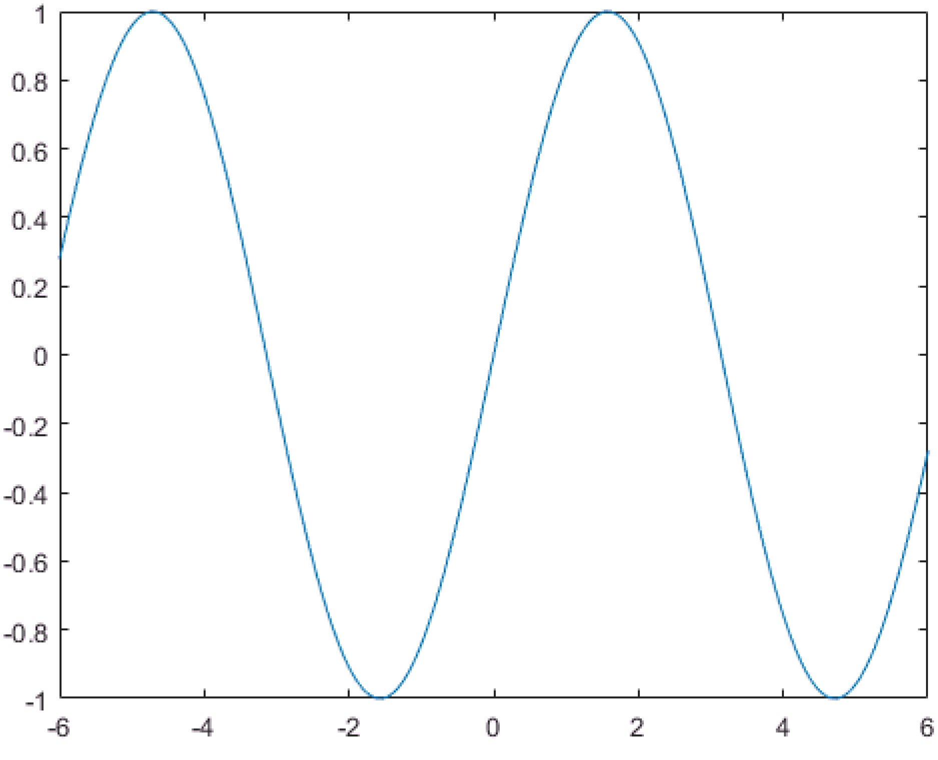
- “Move Group of Objects Along Line” on page 6-89
- “Animate Graphics Object” on page 6-93
- “Record Animation for Playback” on page 6-100
- “Line Animations” on page 6-97

Move Group of Objects Along Line

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

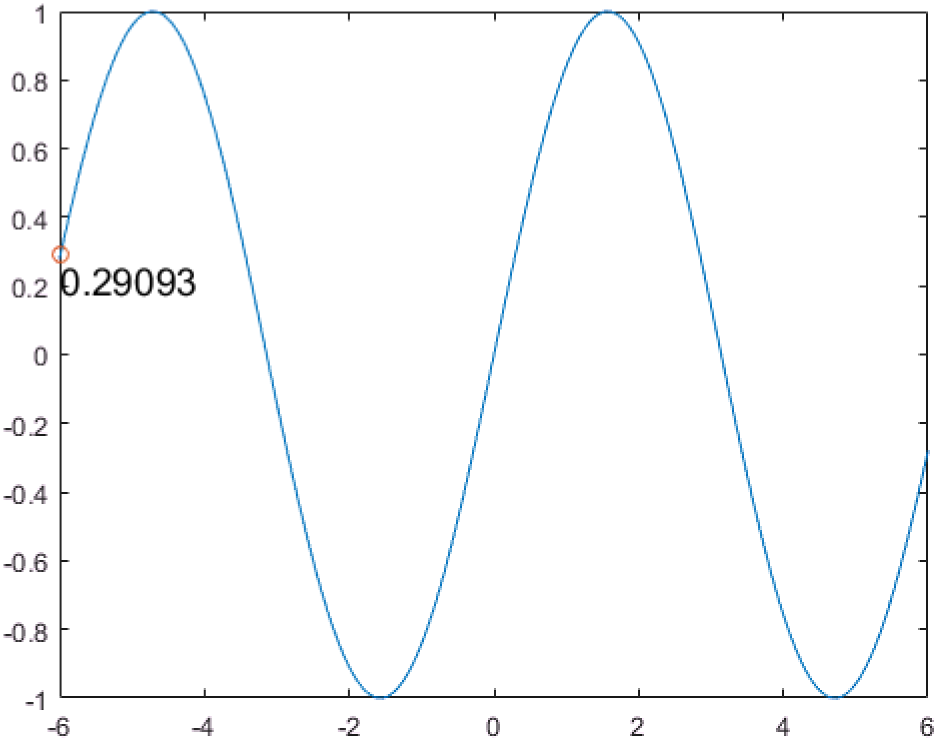
Plot a sine wave and set the axis limits mode to manual to avoid recalculating the limits during the animation loop.

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



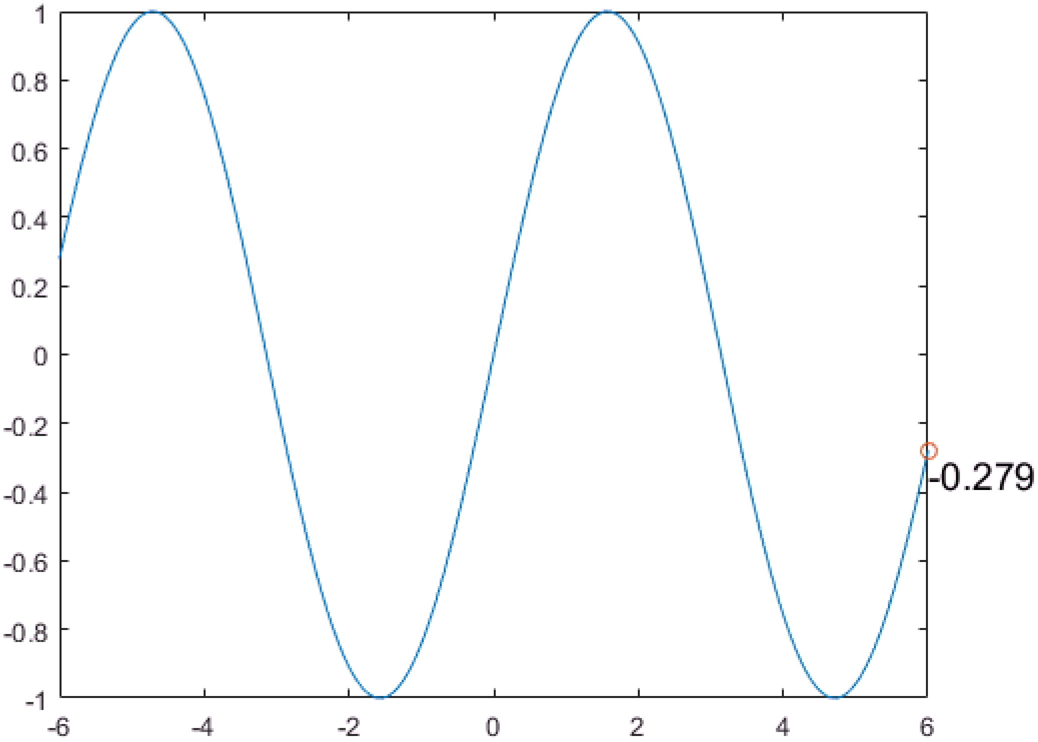
Create a transform object and set its parent to the current axes. Plot a marker and a text annotation at the beginning of the line. Use the `num2str` function to convert the *y*-value at that point to text. Group the two objects by setting their parents to the transform object.

```
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 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 = makehgtform('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 limitrate` instead of drawnow for a faster animation. However, `drawnow limitrate` might not draw every update on the screen.

See Also

`axis` | `drawnow` | `hgtransform` | `makehgtform` | `plot` | `text`

Related Examples

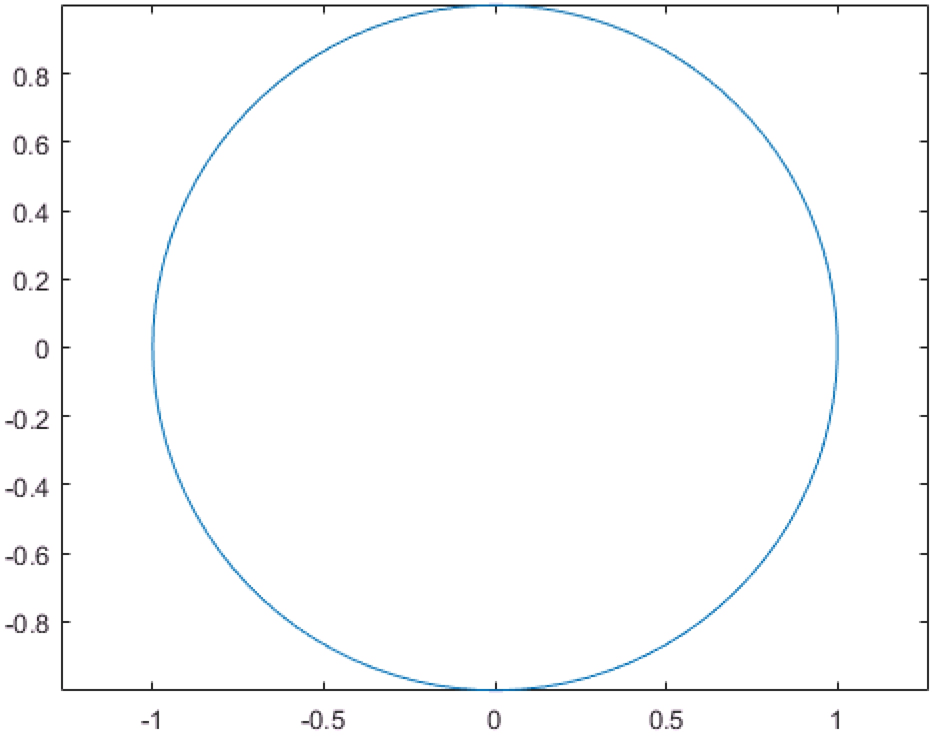
- “Animate Graphics Object” on page 6-93
- “Record Animation for Playback” on page 6-100
- “Line Animations” on page 6-97

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.

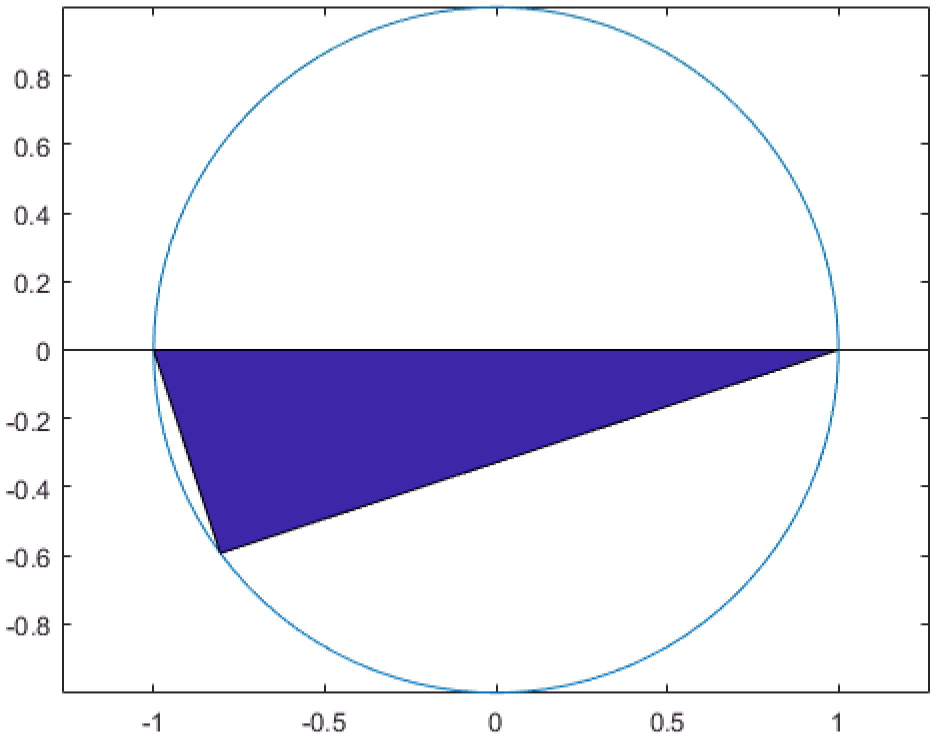
Plot the circle and set the axis limits so that the data units are the same in both directions.

```
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 limitrate` command to display the updates after each iteration. `drawnow limitrate` 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 limitrate % 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 6-86
- "Line Animations" on page 6-97
- "Record Animation for Playback" on page 6-100

More About

- "Animation Techniques" on page 6-84

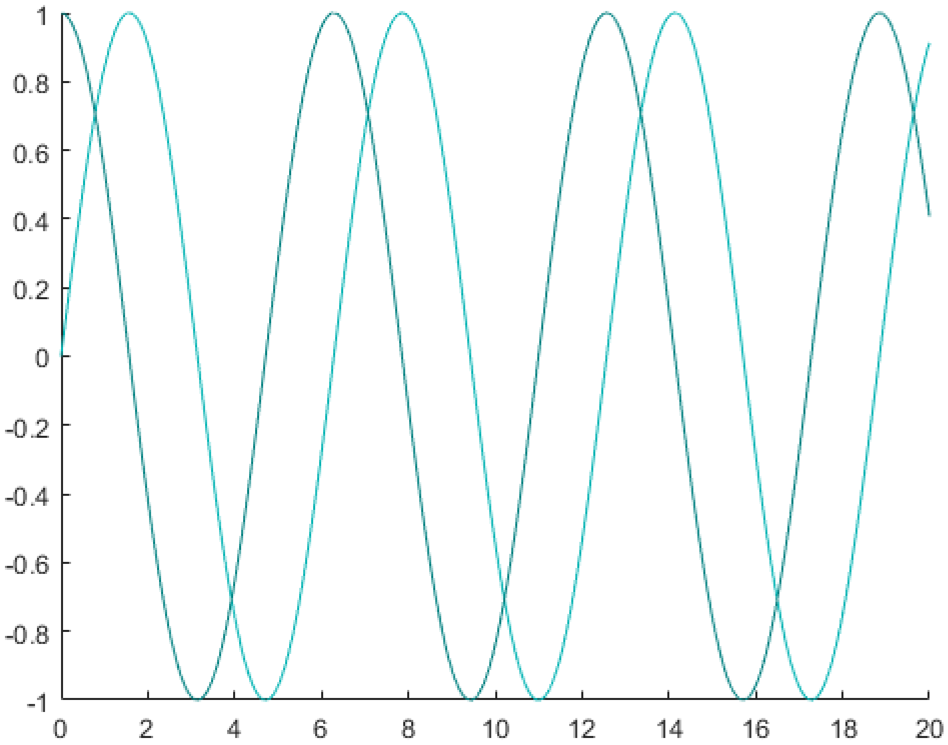
Line Animations

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 limitrate` 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 limitrate
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 6-86
- "Move Group of Objects Along Line" on page 6-89
- "Record Animation for Playback" on page 6-100

More About

- "Animation Techniques" on page 6-84

Record Animation for Playback

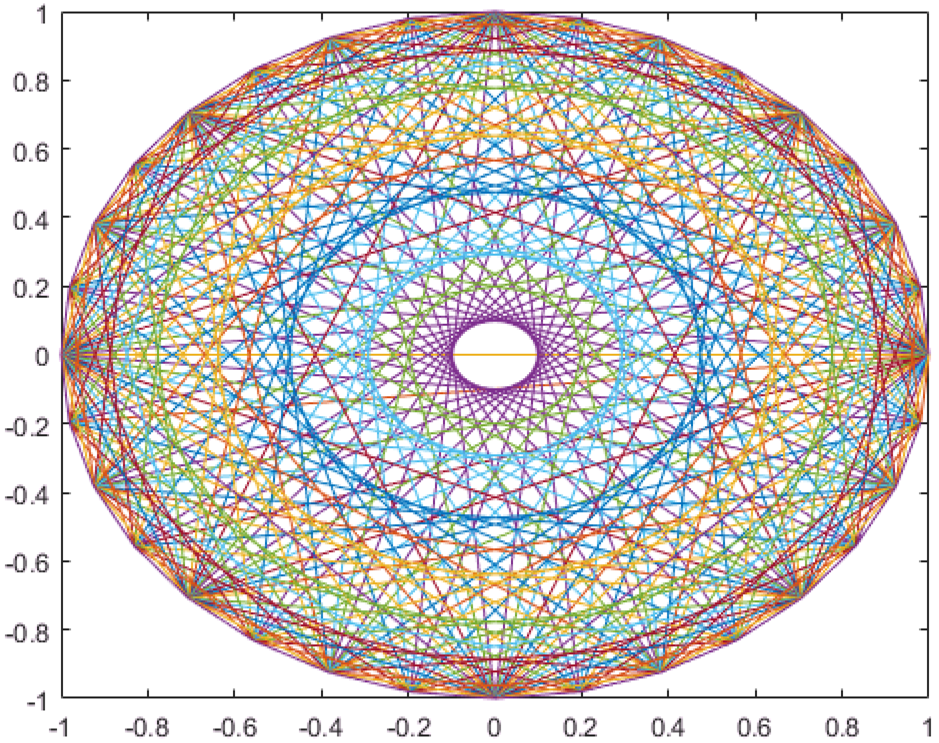
These examples show how to record animations as movies that you can replay.

In this section...
"Record and Play Back Movie" on page 6-100
"Capture Entire Figure for Movie" on page 6-101

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.

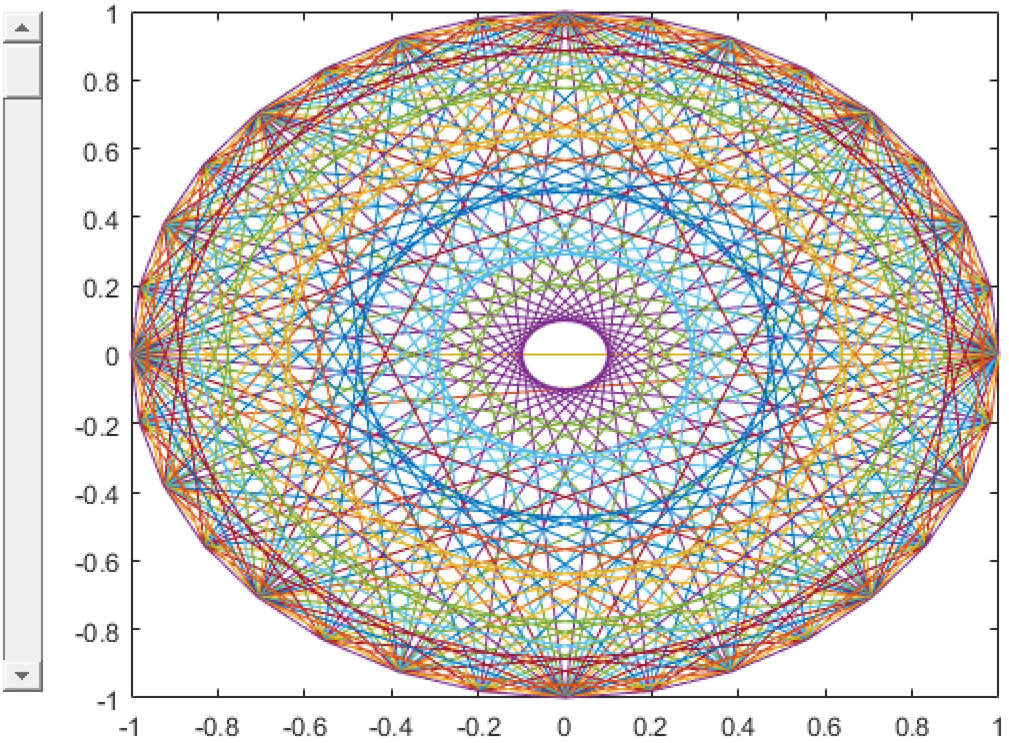
```
figure
movie(M,5)
```

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 fives 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 6-93
- "Line Animations" on page 6-97

More About

- "Animation Techniques" on page 6-84

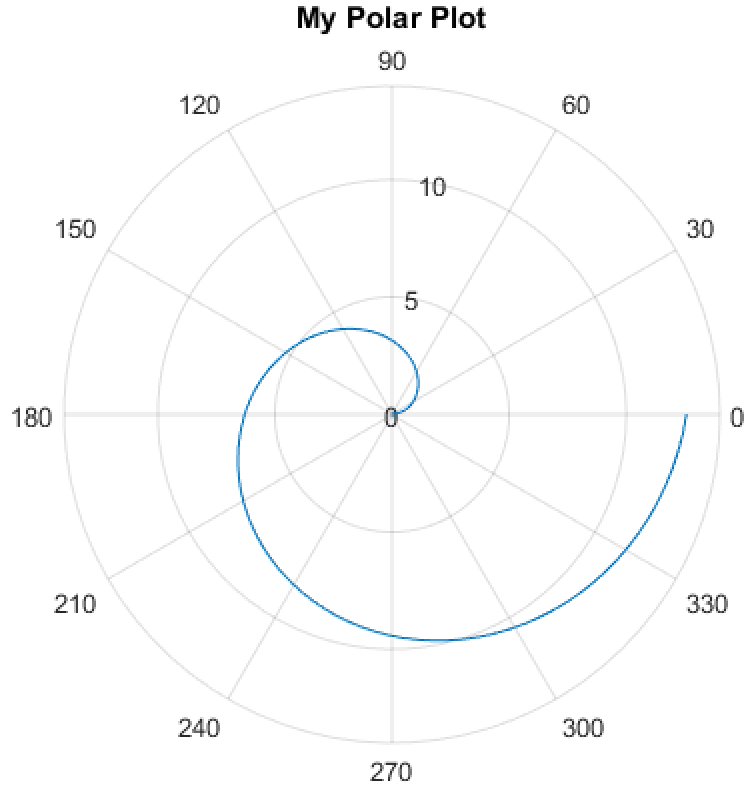
Customize Polar Axes Grid Lines and Appearance

You can modify certain aspects of polar axes in order to make the chart more readable. For example, you can change the grid line locations and associated labels. You also can change the grid line colors and label font size.

Create Polar Plot

Plot a line in polar coordinates and add a title.

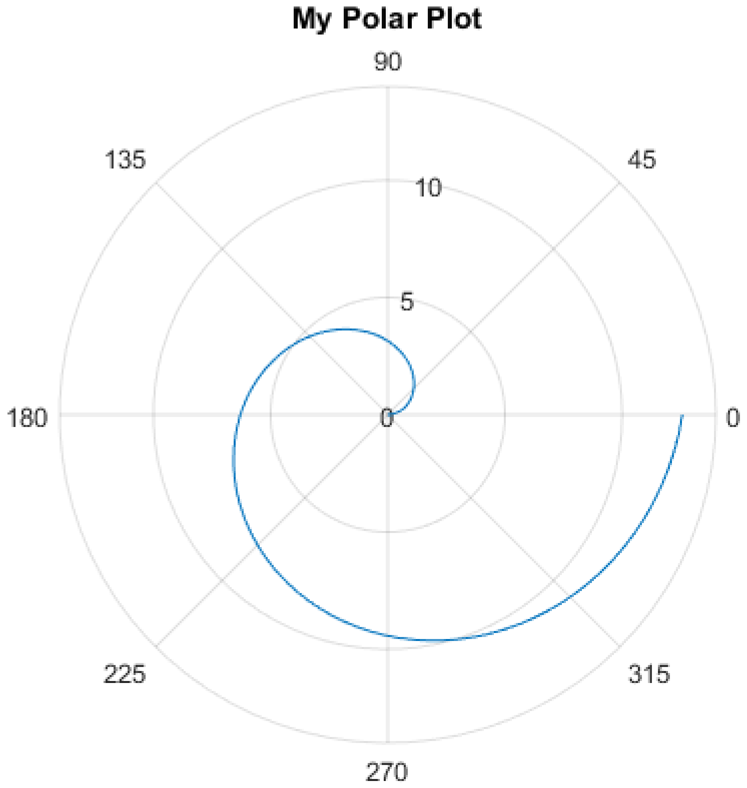
```
theta = linspace(0,2*pi);
rho = 2*theta;
figure
polarplot(theta,rho)
title('My Polar Plot')
```



Change *theta*-Axis Grid Line Locations and Units

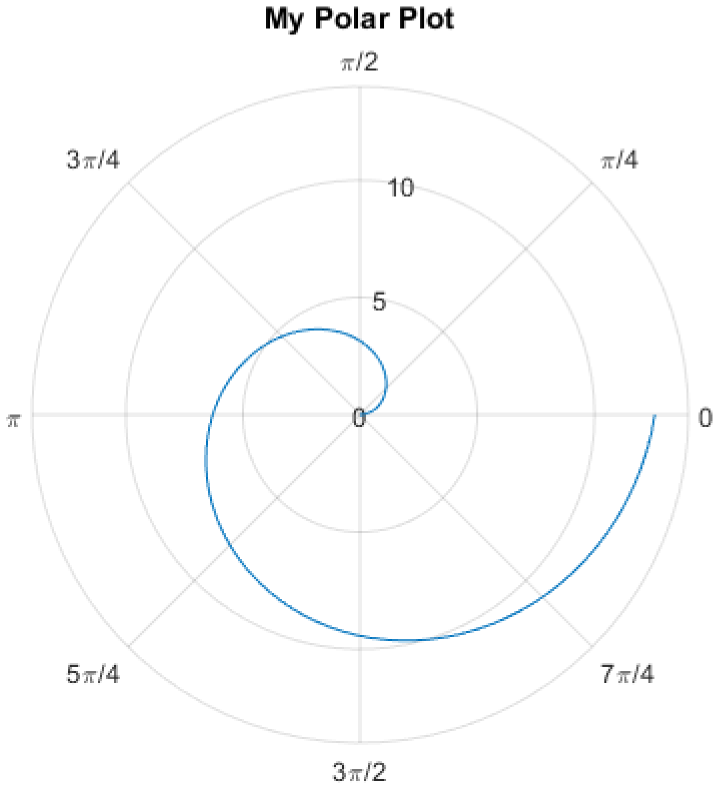
Display grid lines along the *theta*-axis every 45 degrees. Specify the grid line locations as a vector of increasing values.

```
thetaticks(0:45:315)
```



Label the *theta*-axis values in radians instead of degrees by changing the `ThetaAxisUnits` property of the polar axes object. Use `gca` to assign the polar axes object to the variable `pax` in order to access its properties.

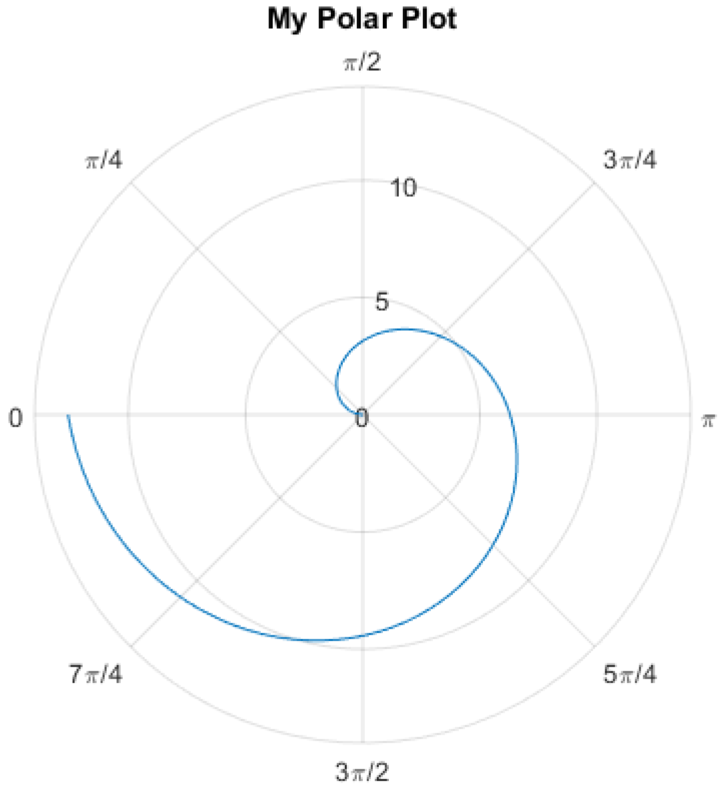
```
pax = gca;
pax.ThetaAxisUnits = 'radians';
```



Rotate *theta*-Axis and Reverse Direction

Change the angles to increase in a clockwise direction. Also, rotate the *theta*-axis so that the zero reference angle is on the left side.

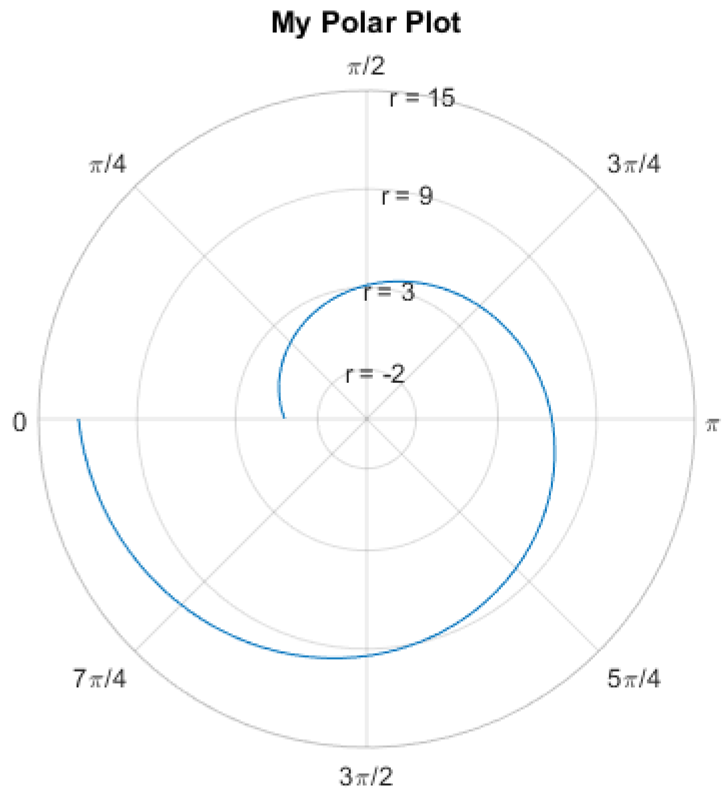
```
pax = gca;
pax.ThetaDir = 'clockwise';
pax.ThetaZeroLocation = 'left';
```



Change *r*-Axis Limits, Grid Line Locations, and Labels

Change the limits of the *r*-axis so that the values range from -5 to 15. Display grid lines at the values -2, 3, 9, and 15. Then, change the labels that appear next to each grid line. Specify the labels as a cell array of character vectors.

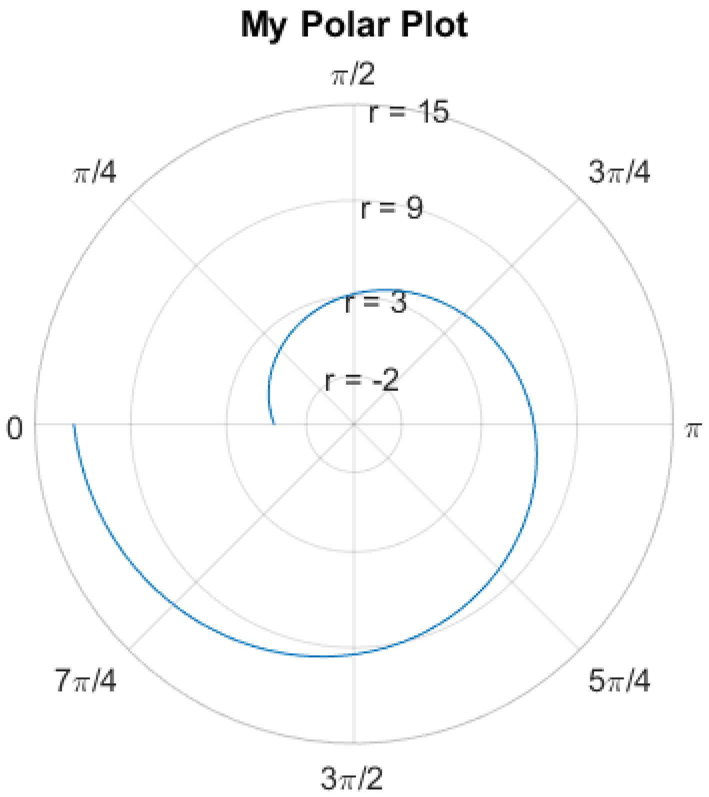
```
rlim([-5 15])
rticks([-2 3 9 15])
rticklabels({'r = -2','r = 3','r = 9','r = 15'})
```



Change Polar Axes Font Size

Change the font size for the polar axes labels.

```
pax = gca;
pax.FontSize = 12;
```

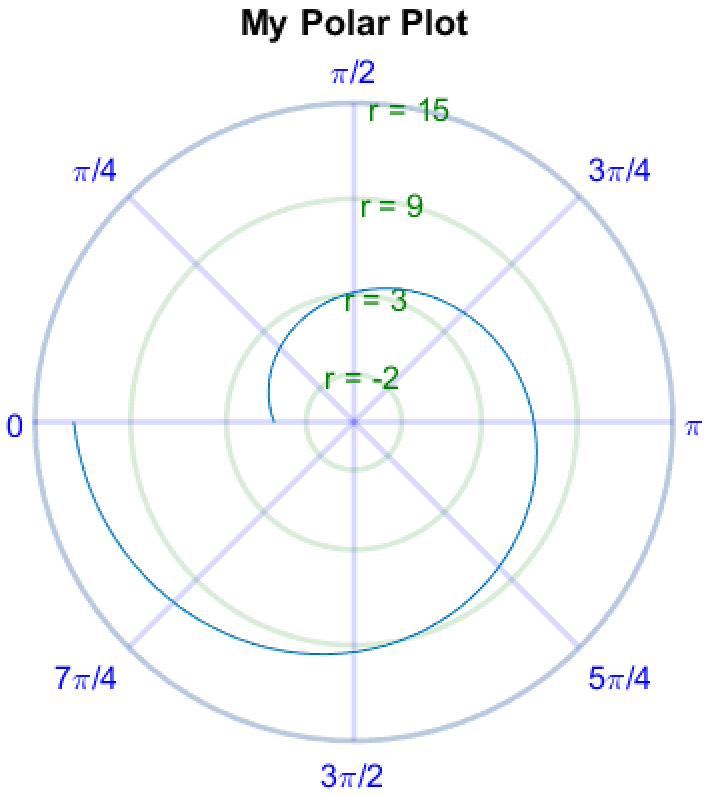


Change Polar Axes Colors and Line Width

Use different colors for the *theta*-axis and *r*-axis grid lines and associated labels by setting the `ThetaColor` and `RColor` properties. Change the width of the grid lines by setting the `LineWidth` property.

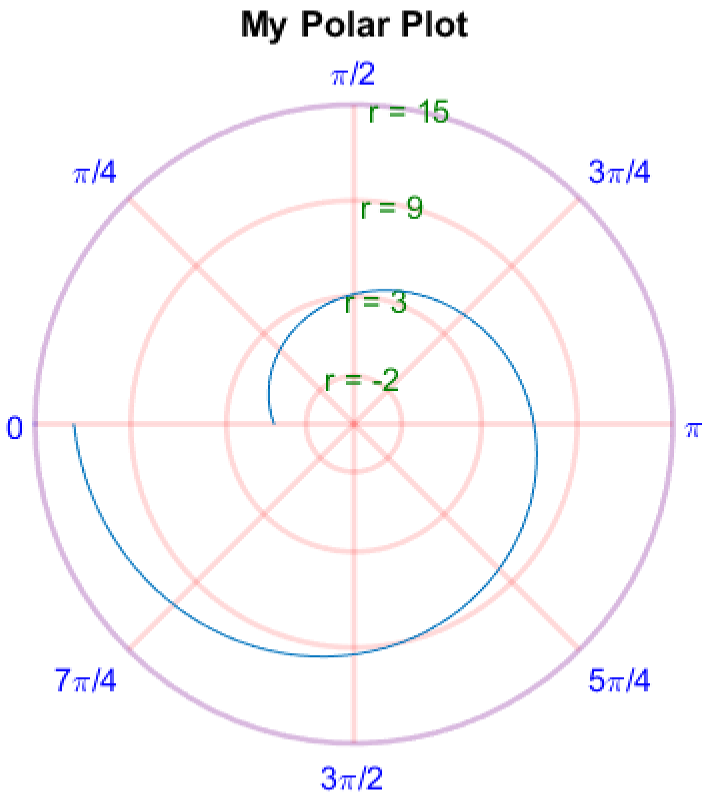
Specify the colors using either a character vector of a color name, such as `'blue'`, or an RGB triplet. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1], for example, [0.4 0.6 0.7].

```
pax = gca;
pax.ThetaColor = 'blue';
pax.RColor = [0 .5 0];
pax.LineWidth = 2;
```



Change the color of the all the grid lines without affecting the labels by setting the `GridColor` property.

```
pax.GridColor = 'red';
```



When you specify the `GridColor` property, the `ThetaColor` and `RColor` properties no longer affect the grid lines. If you want the `ThetaColor` and `RColor` properties to affect the grid lines, then set the `GridColorMode` property back to `'auto'`.

See Also

Polar Axes Properties | `polarplot` | `rticklabels` | `rticks` | `thetaticklabels` | `thetaticks`

Related Examples

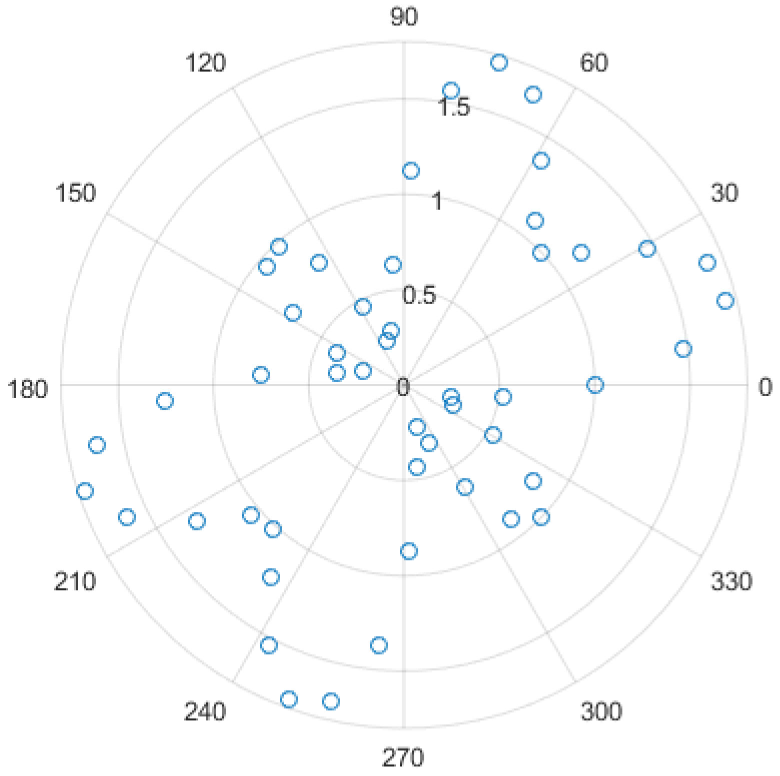
- "Compass Labels on Polar Axes" on page 6-113

Compass Labels on Polar Axes

This example shows how to plot data in polar coordinates. It also shows how to specify the angles at which to draw grid lines and how to specify the labels.

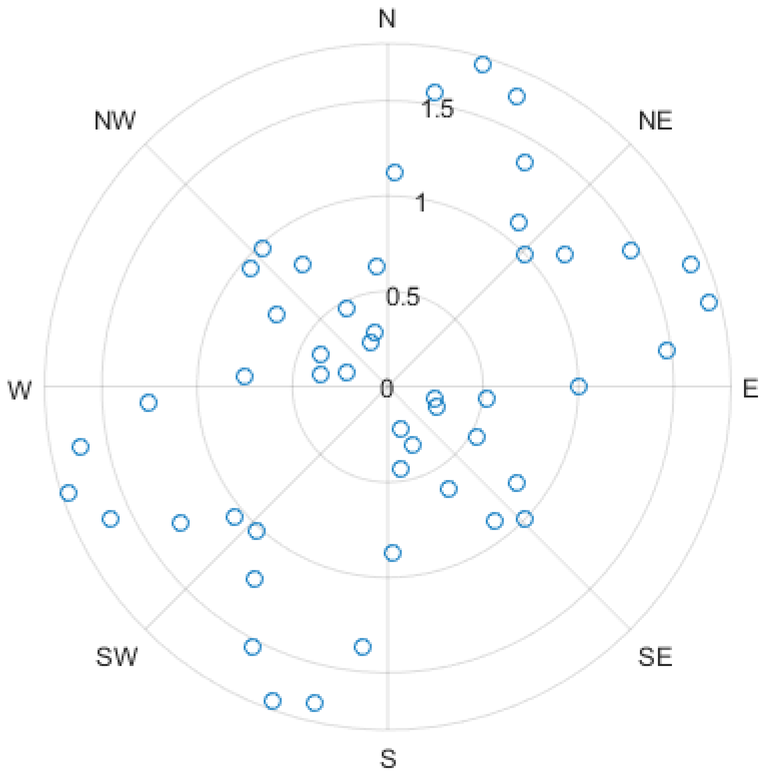
Plot data in polar coordinates and display a circle marker at each data point.

```
theta = linspace(0,2*pi,50);
rho = 1 + sin(4*theta).*cos(2*theta);
polarplot(theta,rho,'o')
```



Use `gca` to access the polar axes object. Specify the angles at which to draw grid lines by setting the `ThetaTick` property. Then, specify the label for each grid line by setting the `ThetaTickLabel` property.

```
pax = gca;
angles = 0:45:360;
pax.ThetaTick = angles;
labels = {'E','NE','N','NW','W','SW','S','SE'};
pax.ThetaTickLabel = labels;
```



See Also

Polar Axes Properties | `polarplot` | `rlim`

Related Examples

- “Customize Polar Axes Grid Lines and Appearance” on page 6-104

Create Line Plot with Markers

Adding markers to a line plot can be a useful way to distinguish multiple lines or to highlight particular data points. Add markers in one of these ways:

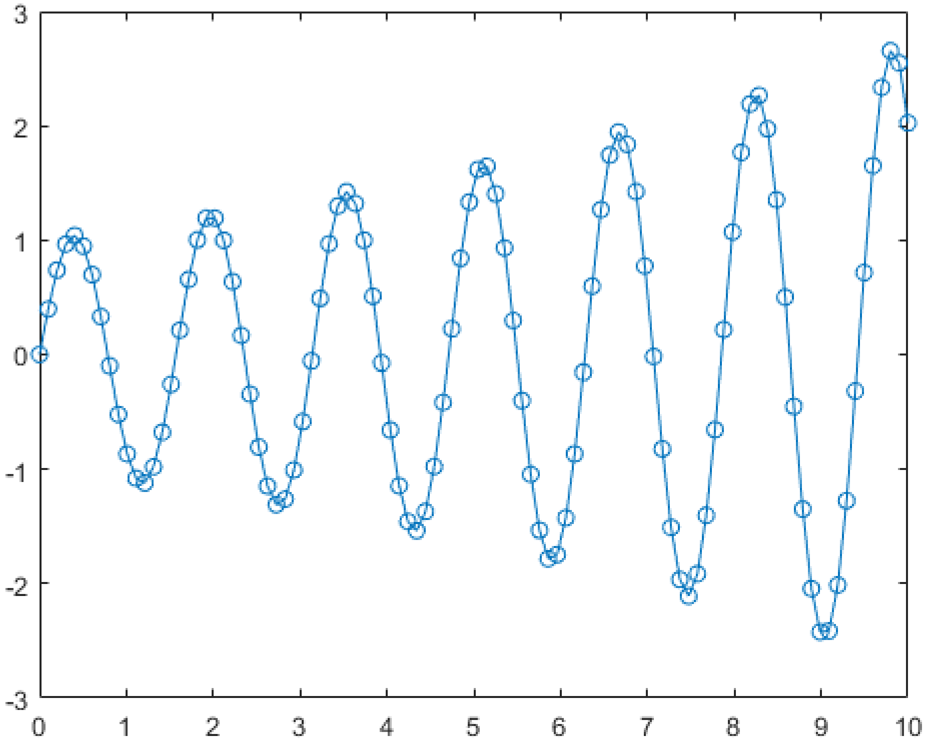
- Include a marker symbol in the line-specification input argument, such as `plot(x,y,'-s')`.
- Specify the `Marker` property as a name-value pair, such as `plot(x,y,'Marker','s')`.

For a list of marker options, see “Supported Marker Symbols” on page 6-123.

Add Markers to Line Plot

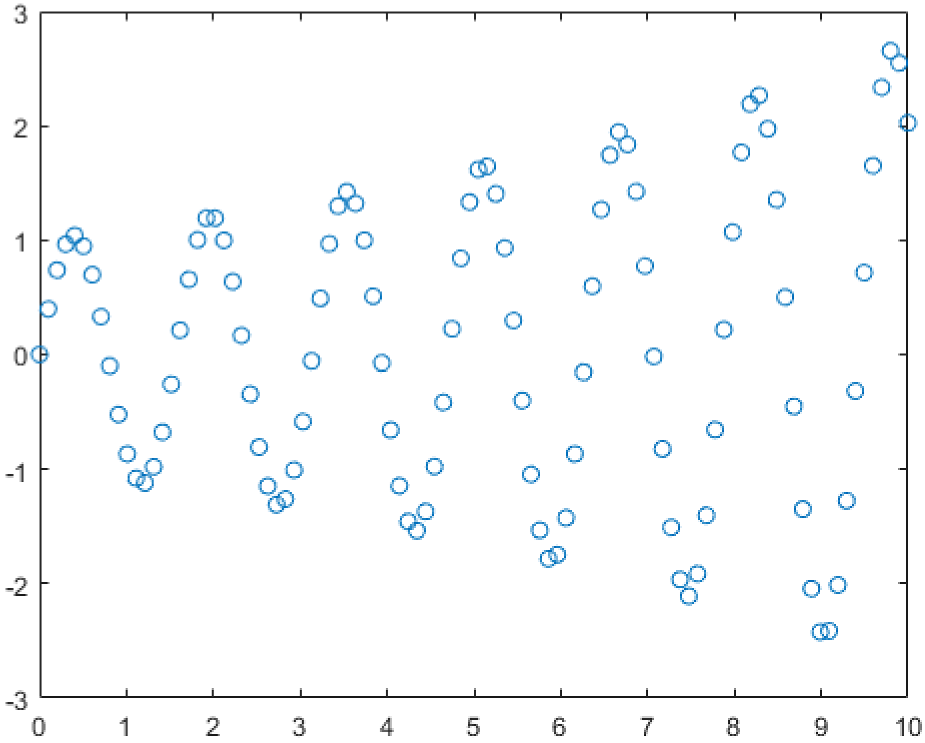
Create a line plot. Display a marker at each data point by including the line-specification input argument when calling the `plot` function. For example, use `'-o'` for a solid line with circle markers.

```
x = linspace(0,10,100);
y = exp(x/10).*sin(4*x);
plot(x,y,'-o')
```



If you specify a marker symbol and do not specify a line style, then `plot` displays only the markers with no line connecting them.

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



Alternatively, you can add markers to a line by setting the `Marker` property as a name-value pair. For example, `plot(x,y,'Marker','o')` plots a line with circle markers.

Specify Marker Size and Color

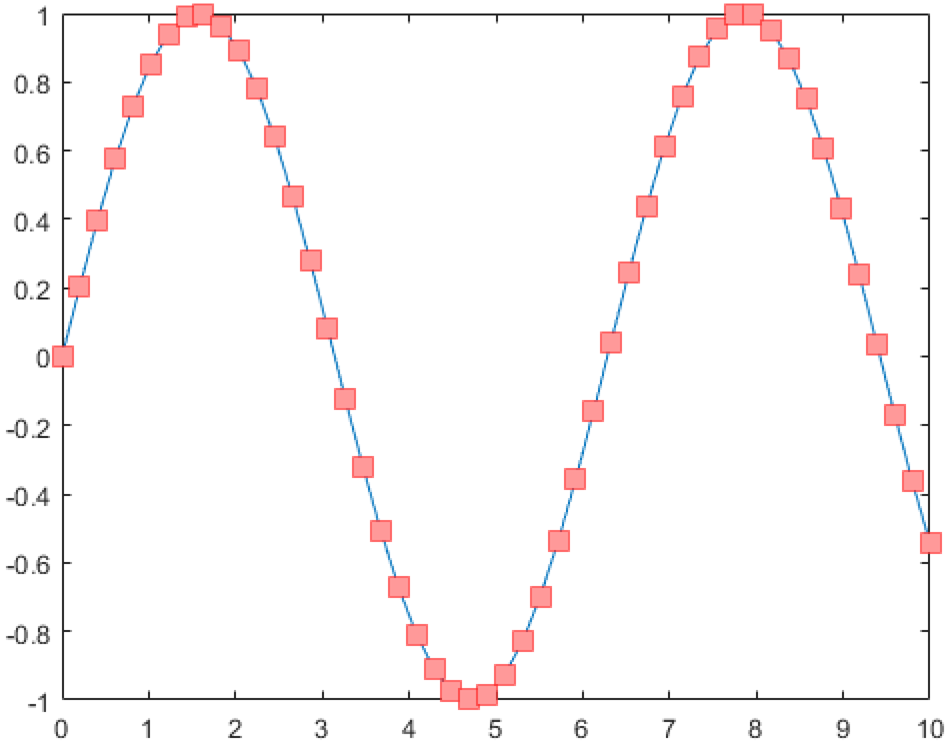
Create a line plot with markers. Customize the markers by setting these properties using name-value pair arguments with the `plot` function:

- `MarkerSize` - Marker size, which is specified as a positive value.
- `MarkerEdgeColor` - Marker outline color, which is specified as a color name or an RGB triplet.

- `MarkerFaceColor` - Marker interior color, which is specified as a color name or an RGB triplet.

Specify the colors using either a character vector of a color name, such as `'red'`, or an RGB triplet, such as `[0.4 0.6 0.7]`. An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1].

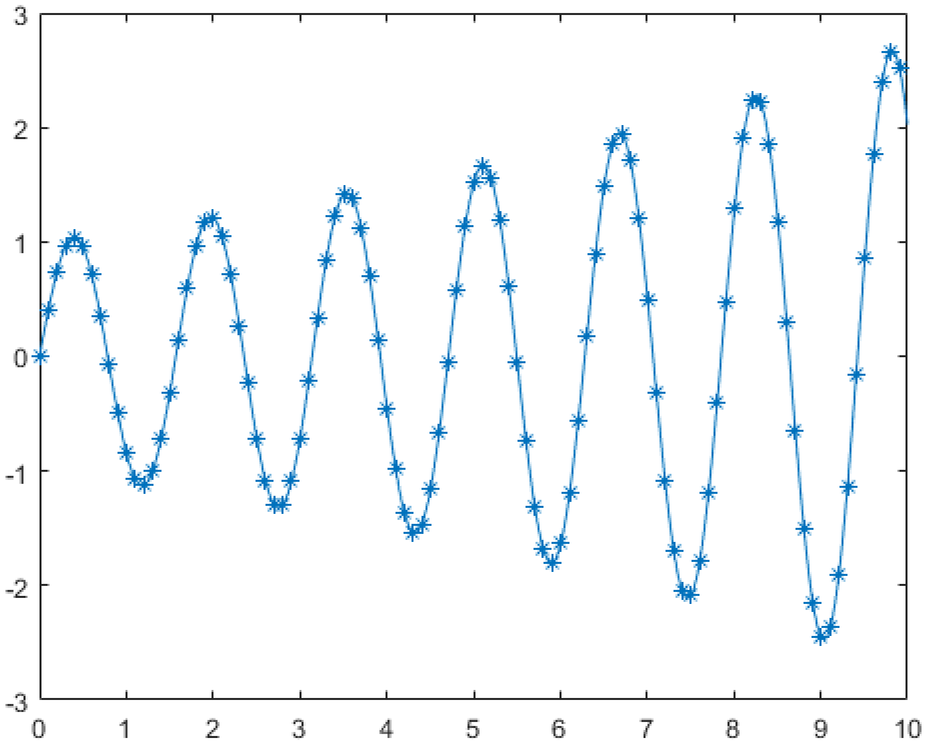
```
x = linspace(0,10,50);
y = sin(x);
plot(x,y,'-s','MarkerSize',10,...
    'MarkerEdgeColor','red',...
    'MarkerFaceColor',[1 .6 .6])
```



Control Placement of Markers Along Line

Create a line plot with 1,000 data points, add asterisks markers, and control the marker positions using the `MarkerIndices` property. Set the property to the indices of the data points where you want to display markers. Display a marker every tenth data point, starting with the first data point.

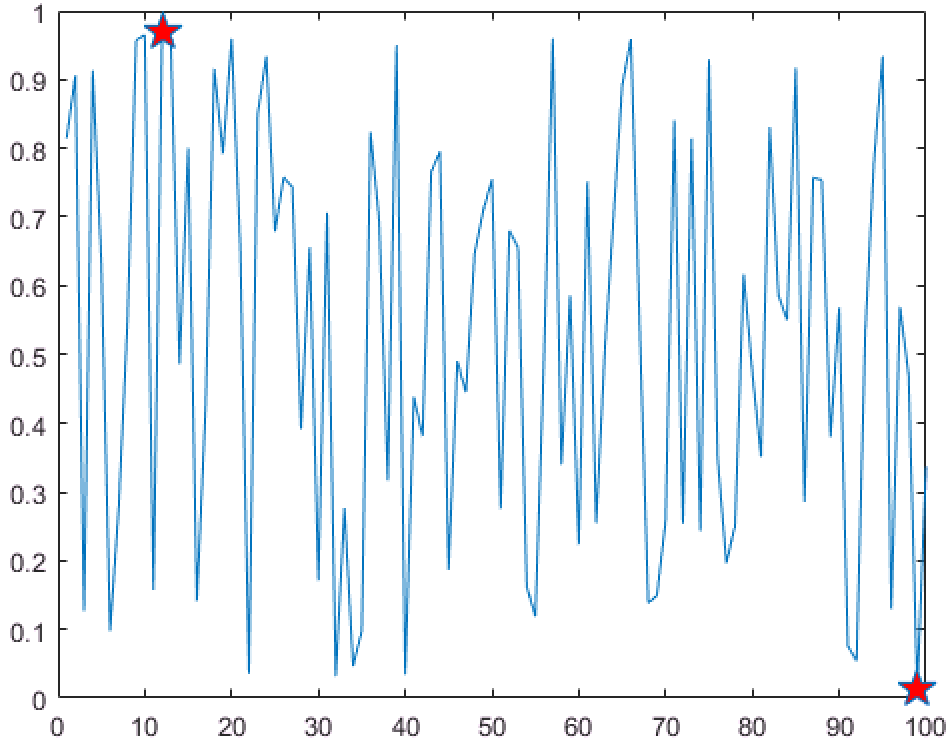
```
x = linspace(0,10,1000);
y = exp(x/10).*sin(4*x);
plot(x,y,'-*','MarkerIndices',1:10:length(y))
```



Display Markers at Maximum and Minimum Data Points

Create a vector of random data and find the index of the minimum and maximum values. Then, create a line plot of the data. Display red markers at the minumum and maximum data values by setting the `MarkerIndices` property to a vector of the index values.

```
x = 1:100;
y = rand(100,1);
idxmin = find(y == max(y));
idxmax = find(y == min(y));
plot(x,y,'-p','MarkerIndices',[idxmin idxmax],...
    'MarkerFaceColor','red',...
    'MarkerSize',15)
```

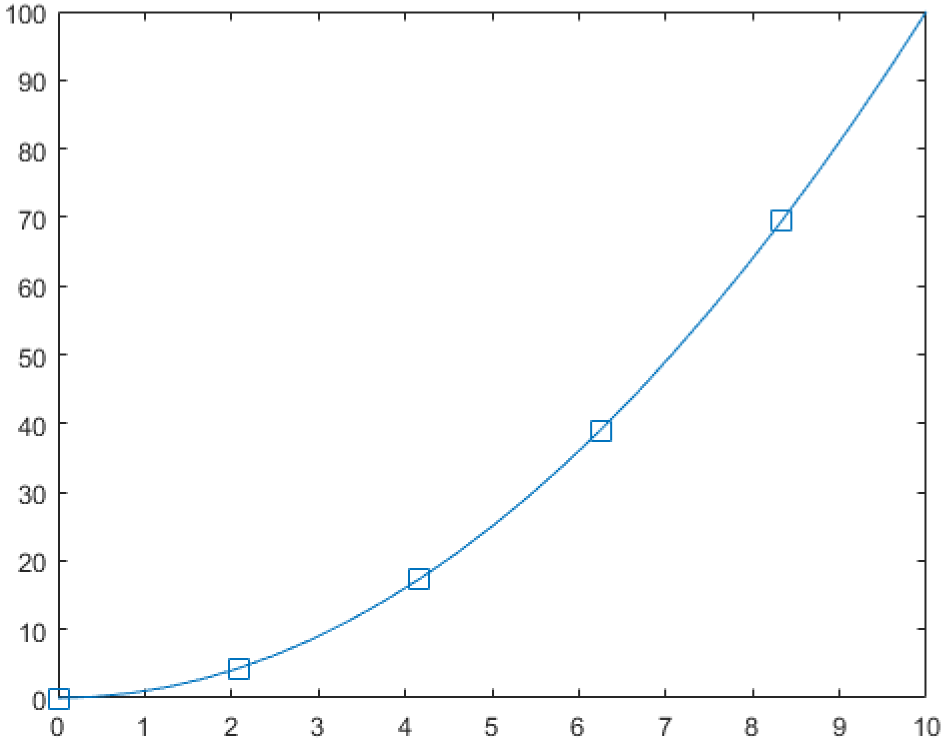


Revert to Default Marker Locations

Modify the marker locations, then revert back to the default locations.

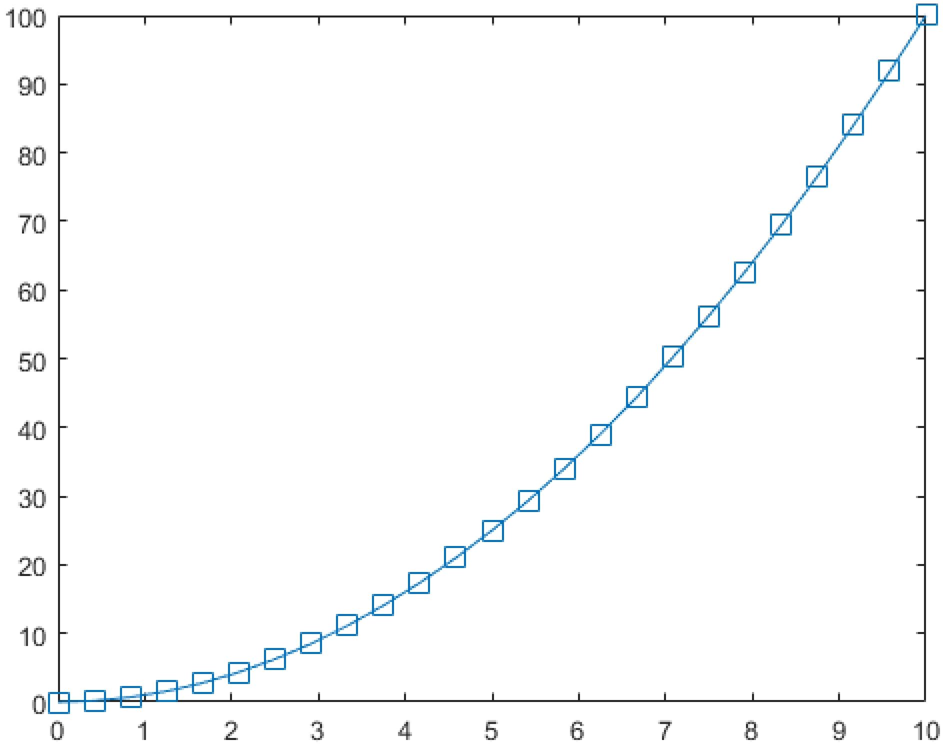
Create a line plot and display large, square markers every five data points. Assign the chart line object to the variable p so that you can access its properties after it is created.

```
x = linspace(0,10,25);
y = x.^2;
p = plot(x,y,'-s');
p.MarkerSize = 10;
p.MarkerIndices = 1:5:length(y);
```



Reset the `MarkerIndices` property to the default value, which is a vector of all index values from 1 to the number of data points.

```
p.MarkerIndices = 1:length(y);
```



Supported Marker Symbols

Value	Description
`'o'`	Circle
`'+'`	Plus sign
`'*'`	Asterisk
`'.'`	Point
`'x'`	Cross
`'square'` or `'s'`	Square

Value	Description
`'diamond'` or `'d'`	Diamond
`'^'`	Upward-pointing triangle
`'v'`	Downward-pointing triangle
`'>'`	Right-pointing triangle
`'<'`	Left-pointing triangle
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

The line-specification input argument does not support marker options that are more than one character. Use the one character alternative or set the `Marker` property instead.

See Also

Functions

`loglog` | `plot` | `plot3` | `scatter`

Properties

Chart Line Properties

Displaying Bit-Mapped Images

- “Working with Images in MATLAB Graphics” on page 7-2
- “Image Types” on page 7-5
- “8-Bit and 16-Bit Images” on page 7-10
- “Read, Write, and Query Image Files” on page 7-18
- “Displaying Graphics Images” on page 7-22
- “The Image Object and Its Properties” on page 7-27
- “Printing Images” on page 7-34
- “Convert Image Graphic or Data Type” on page 7-35

Working with Images in MATLAB Graphics

In this section...

“What Is Image Data?” on page 7-2
“Supported Image Formats” on page 7-3
“Functions for Reading, Writing, and Displaying Images” on page 7-4

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 7-10 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 an 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
`axis`	Plot axis scaling and appearance.	Display
`image`	Display image (create image object).	Display
`imagesc`	Scale data and display as image.	Display
`imread`	Read image from graphics file.	File I/O
`imwrite`	Write image to graphics file.	File I/O
`imfinfo`	Get image information from graphics file.	Utility
`ind2rgb`	Convert indexed image to RGB image.	Utility

Image Types

In this section...

"Indexed Images" on page 7-5
"Intensity Images" on page 7-7
"RGB (Truecolor) Images" on page 7-8

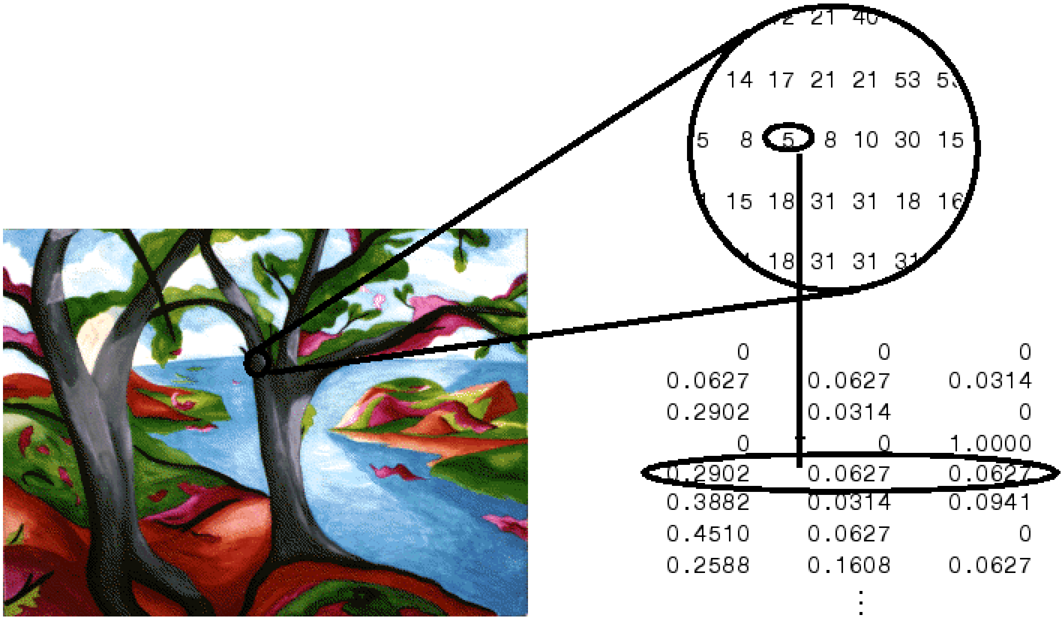
Indexed Images

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.



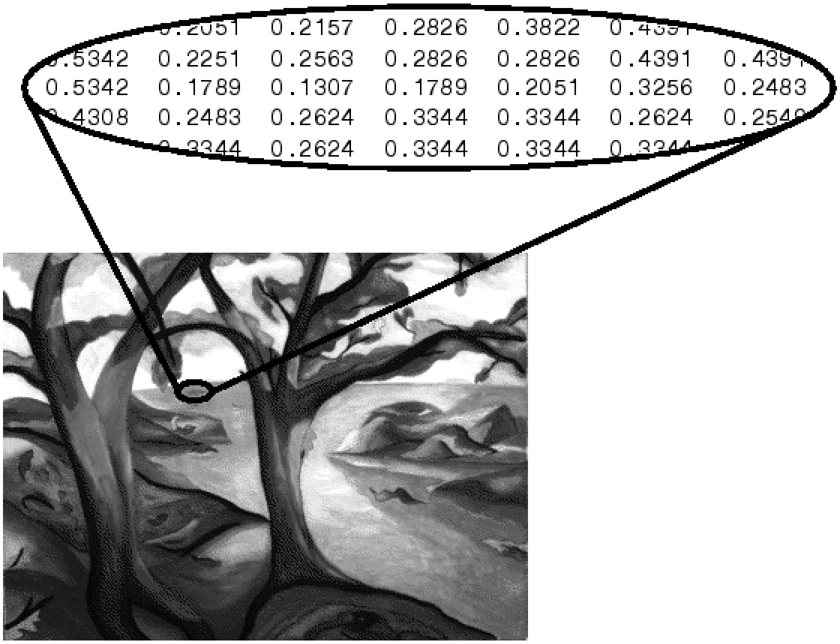
The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or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

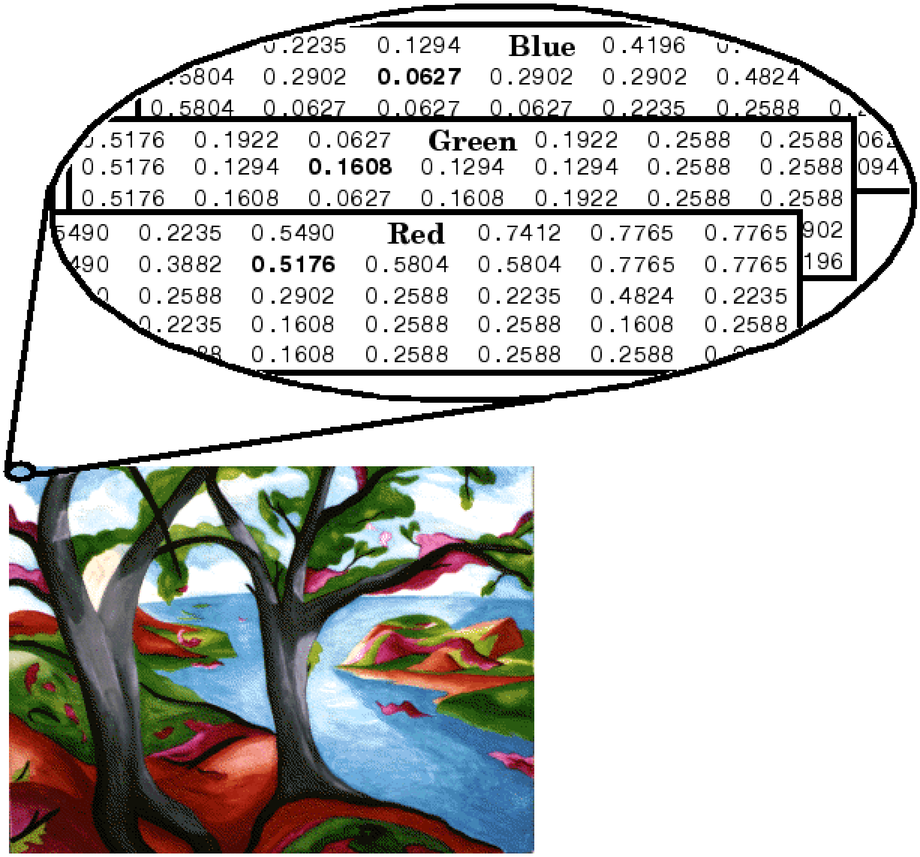
An RGB image, sometimes referred to as a *truecolor* image, is stored as an *m*-by-*n*-by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities 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.



```
  .2235   0.1294    Blue   0.4196   0.
 .5804   0.2902   0.0627   0.2902   0.2902   0.4824
 0.5804   0.0627   0.0627   0.0627   0.2235   0.2588   0.
 .5176   0.1922   0.0627   Green   0.1922   0.2588   0.2588 062
 0.5176   0.1294   0.1608   0.1294   0.1294   0.2588   0.2588 094
 0.5176   0.1608   0.0627   0.1608   0.1922   0.2588   0.2588
5490   0.2235   0.5490    Red    0.7412   0.7765   0.7765 902
 490   0.3882   0.5176   0.5804   0.5804   0.7765   0.7765 196
   0   0.2588   0.2902   0.2588   0.2235   0.4824   0.2235
   .2235   0.1608   0.2588   0.2588   0.1608   0.2588
     88   0.1608   0.2588   0.2588   0.2588   0.
```

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 7-10

“Intensity Images” on page 7-11

“RGB Images” on page 7-11

“Mathematical Operations Support for uint8 and uint16” on page 7-12

“Other 8-Bit and 16-Bit Array Support” on page 7-12

“Converting an 8-Bit RGB Image to Grayscale” on page 7-13

“Summary of Image Types and Numeric Classes” on page 7-17

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 `uint`8 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:

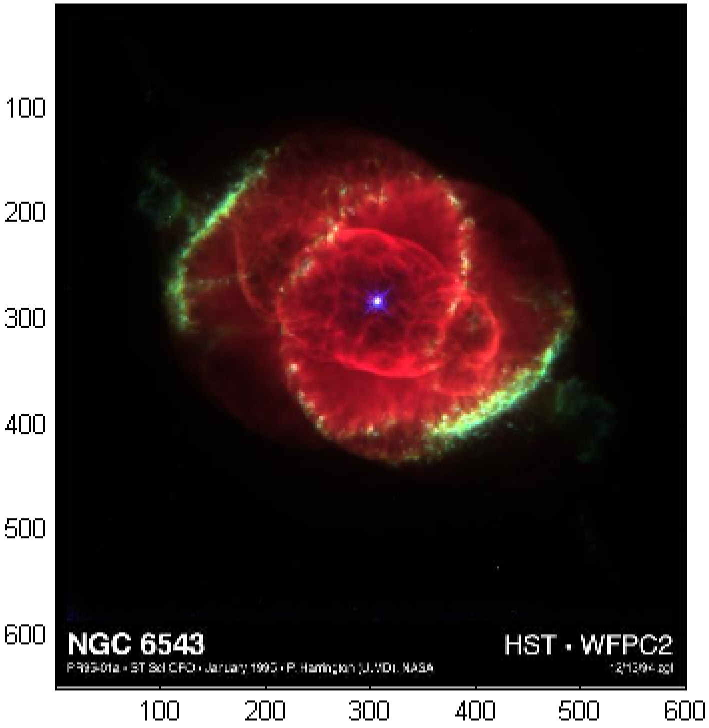
- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading `uint8` and `uint16` arrays in MAT-files using `save` and `load`. (Remember that if you are loading or saving a graphics file format image, you must use 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. Orkowski (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
```

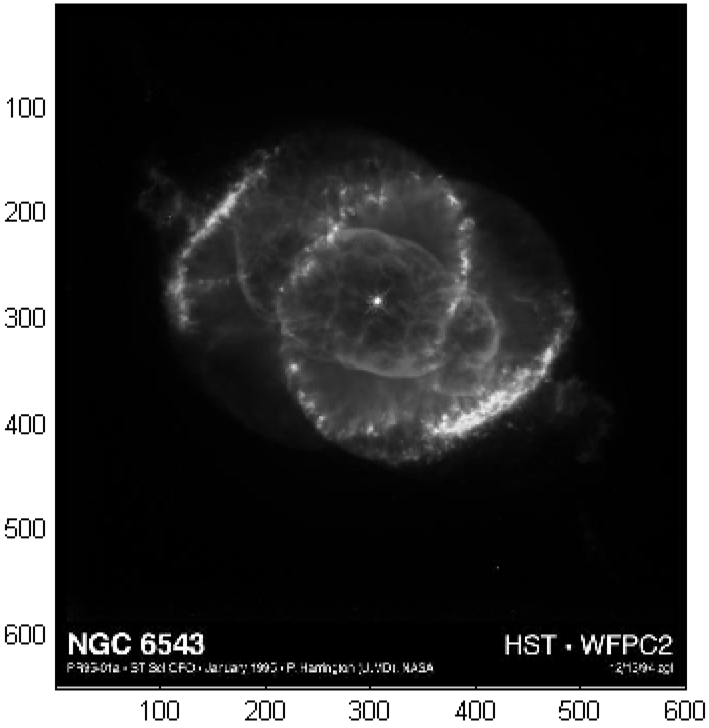
to a maximum of 255:

```
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	Image is an m-by-n array of integers in the range [1, p]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n array of integers in the range [0, p –1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1].
Intensity	Image is an m-by-n array of floating-point values that are linearly scaled to produce colormap indices. The typical range of values is [0, 1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.	Image is an m-by-n array of integers that are linearly scaled to produce colormap indices. The typical range of values is [0, 255] or [0, 65535]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.
RGB (Truecolor)	Image is an m-by-n-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

Read, Write, and Query Image Files

In this section...

"Working with Image Formats" on page 7-18
"Reading a Graphics Image" on page 7-19
"Writing a Graphics Image" on page 7-19
"Subsetting a Graphics Image (Cropping)" on page 7-20
"Obtaining Information About Graphics Files" on page 7-21

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.	`load` `save`
Load or save graphics file format image, e.g., BMP, TIFF.	`imread` `imwrite`
Display any image loaded into the MATLAB workspace.	`image`

Procedure	Functions to Use
	`imagesc`
Utilities	`imfinfo` `ind2rgb`

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...

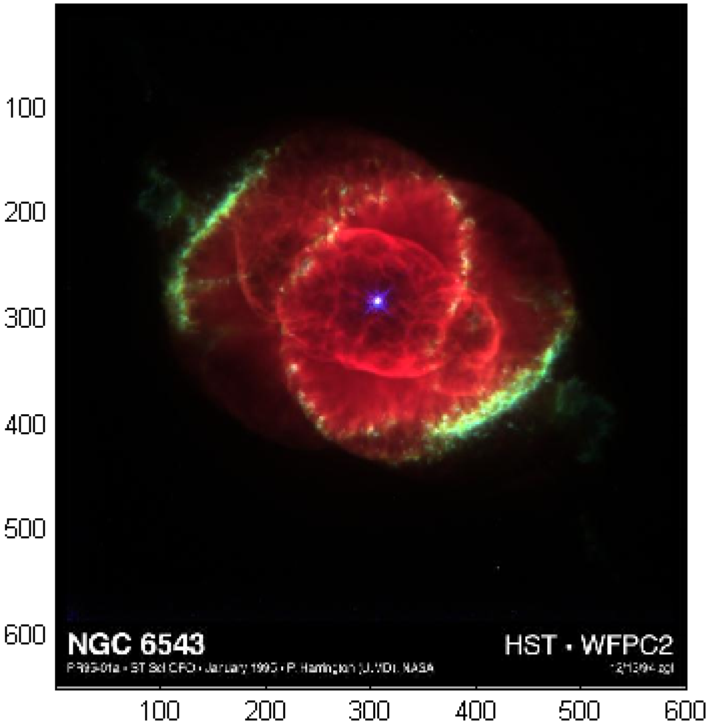
"Image Types and Display Methods" on page 7-22

"Controlling Aspect Ratio and Display Size" on page 7-24

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

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

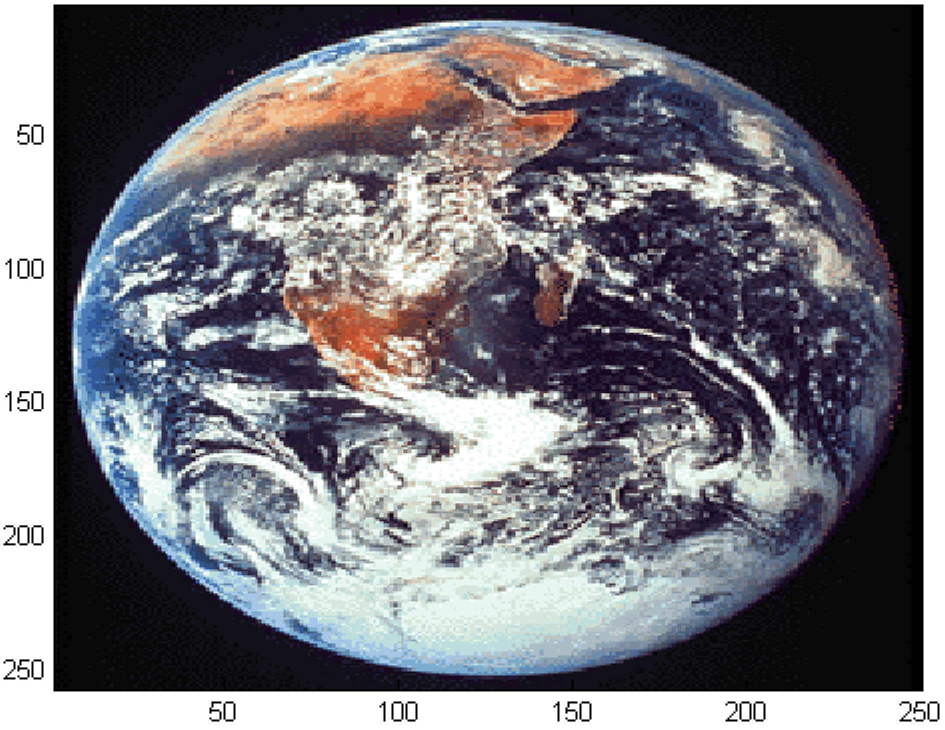
RGB (truecolor)	`image(RGB)`	No

Controlling Aspect Ratio and Display Size

The `image` function displays the image in a default-sized figure and axes. 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.

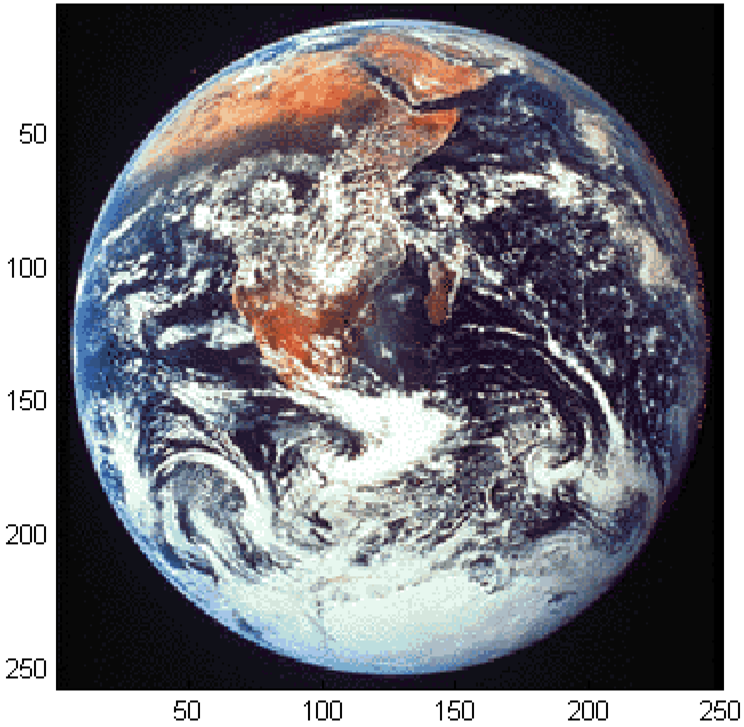
For example, these commands display the `earth` image using the default figure and axes positions:

```
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.

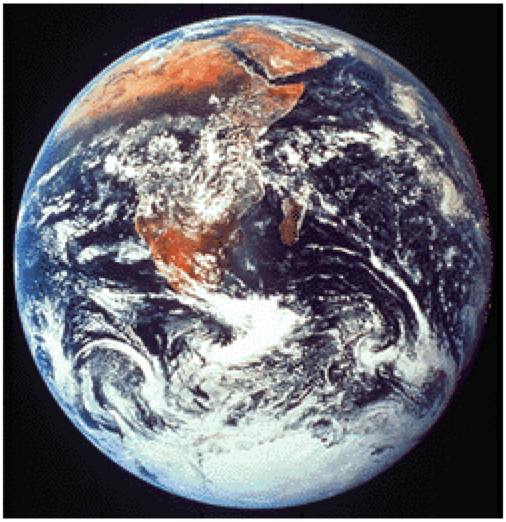
```
axis image
```



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, use imshow. For example, this command displays the earth image so that one data element corresponds to one screen pixel:

```
imshow(X,map)
```



The Image Object and Its Properties

In this section...

"Image CData" on page 7-27

"Image CDataMapping" on page 7-27

"XData and YData" on page 7-28

"Add Text to Image Data" on page 7-30

"Additional Techniques for Fast Image Updating" on page 7-32

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 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:

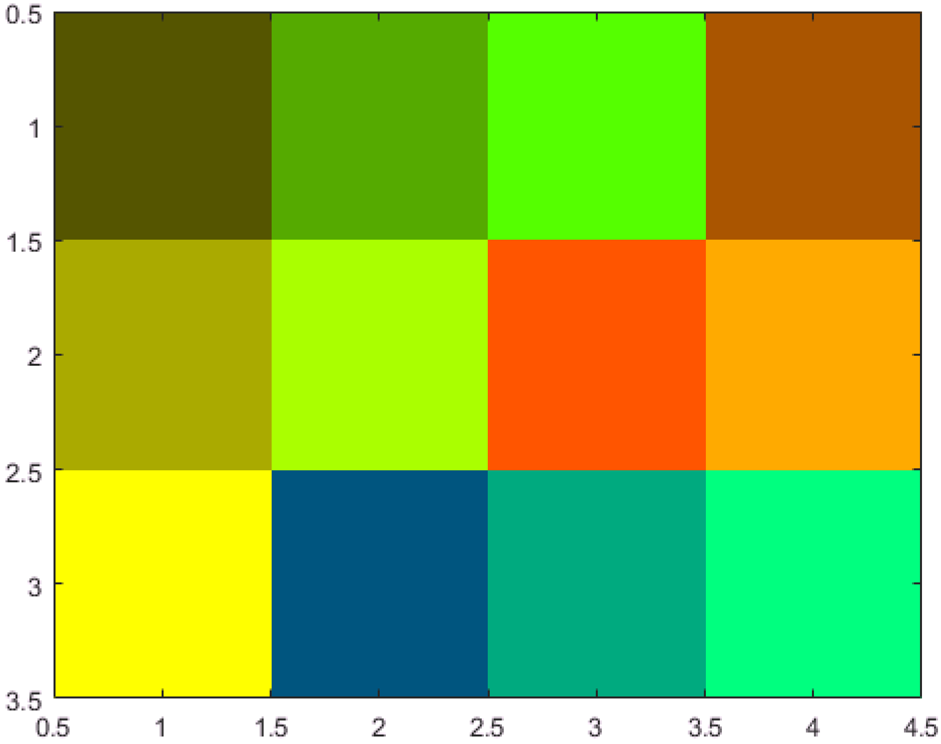
- The left column of the image has an *x*-coordinate of 1.
- The right column of the image has an *x*-coordinate of *n*.
- The top row of the image has a *y*-coordinate of 1.
- The bottom row of the image has a *y*-coordinate of *m*.

Coordinate System for Images

Use Default Coordinate System

Display an image using the default coordinate system. Use colors from the `colorcube` map.

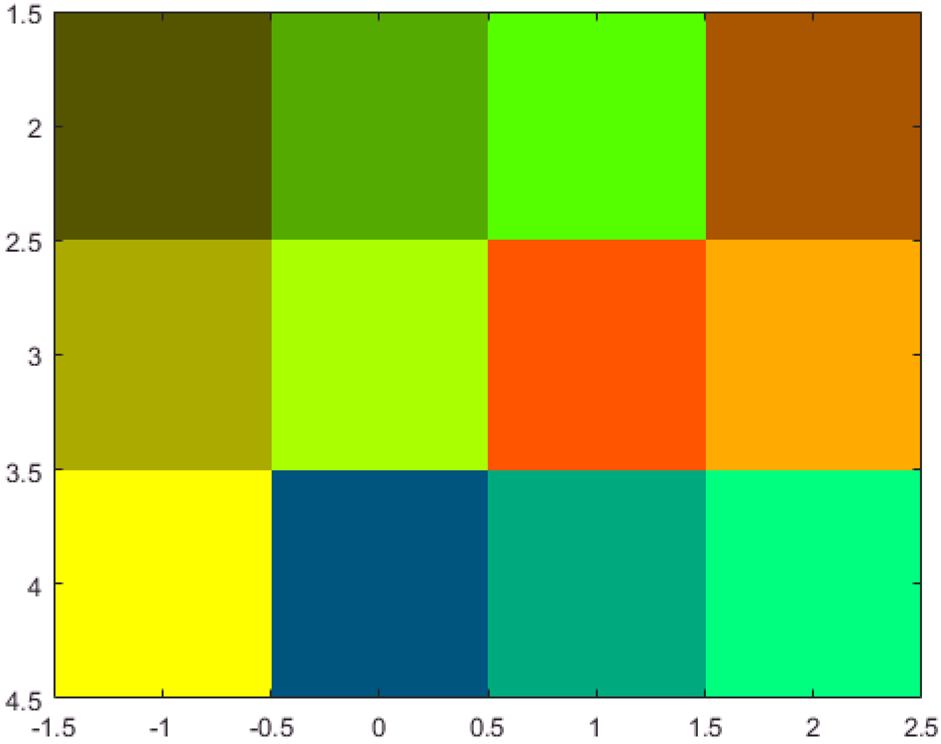
```
C = [1 2 3 4; 5 6 7 8; 9 10 11 12];
im = image(C);
colormap(colorcube)
```



Specify Coordinate System

Display an image and specify the coordinate system. Use colors from the `colorcube` 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(colorcube)
```



Add Text to Image Data

This example shows how to use array indexing to rasterize text 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.

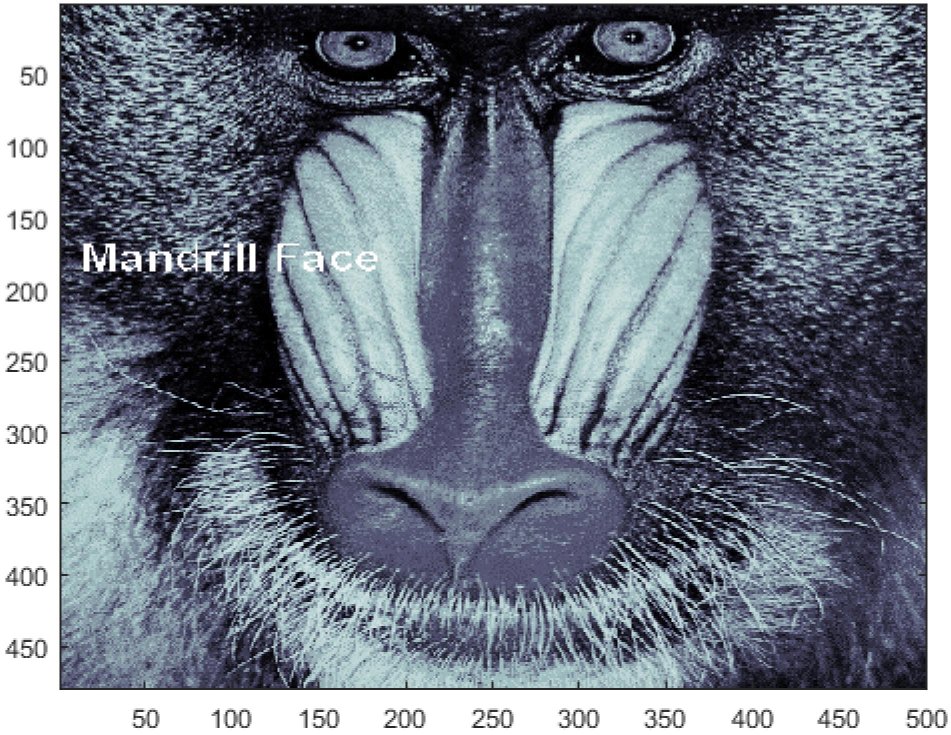
```
fig = figure;
t = text(.05,.1,'Mandrill Face','FontSize',20,'FontWeight','bold');
F = getframe(gca,[10 10 200 200]);
close(fig)
```

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.
- 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',...
  'ylimmode','manual',...
  'zlimmode','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')
```

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 7-5 and "Indexed Images" on page 7-10 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

- "Print Figure from File Menu" on page 8-2
- "Copy Figure to Clipboard from Edit Menu" on page 8-7
- "Customize Figure Interactively Before Saving" on page 8-11
- "Save Figure to Open in Another Application" on page 8-19
- "Save Figure Preserving Background Color" on page 8-24
- "Save Figure at Specific Size and Resolution" on page 8-27
- "Save Figure to Reopen in MATLAB Later" on page 8-33
- "Save Axes Without Saving UIControls" on page 8-36
- "Save Figure with Minimal White Space" on page 8-40

Print Figure from File Menu

In this section...

"Simple Printout" on page 8-2
"Preserve Background Color and Tick Values" on page 8-2
"Figure Size and Placement" on page 8-3
"Line Width and Font Size" on page 8-5

Simple Printout

To print a figure, use **File** > **Print**. For example, create a bar chart to print.

```
x = [3 5 2 6 1 8 2 3];
bar(x)
```

Click **File** > **Print**, select a printer, and click **OK**. The printer must be set up on your system. If you do not see a printer that is set up already, then restart MATLAB.

To print the figure programmatically, use the `print` function.

Preserve Background Color and Tick Values

Some details of the printed figure can look different from the figure on the display. By default, printed figures use a white figure background color. Also, if the printed figure size is different from the original figure size, then the axis limits and tick values can differ.

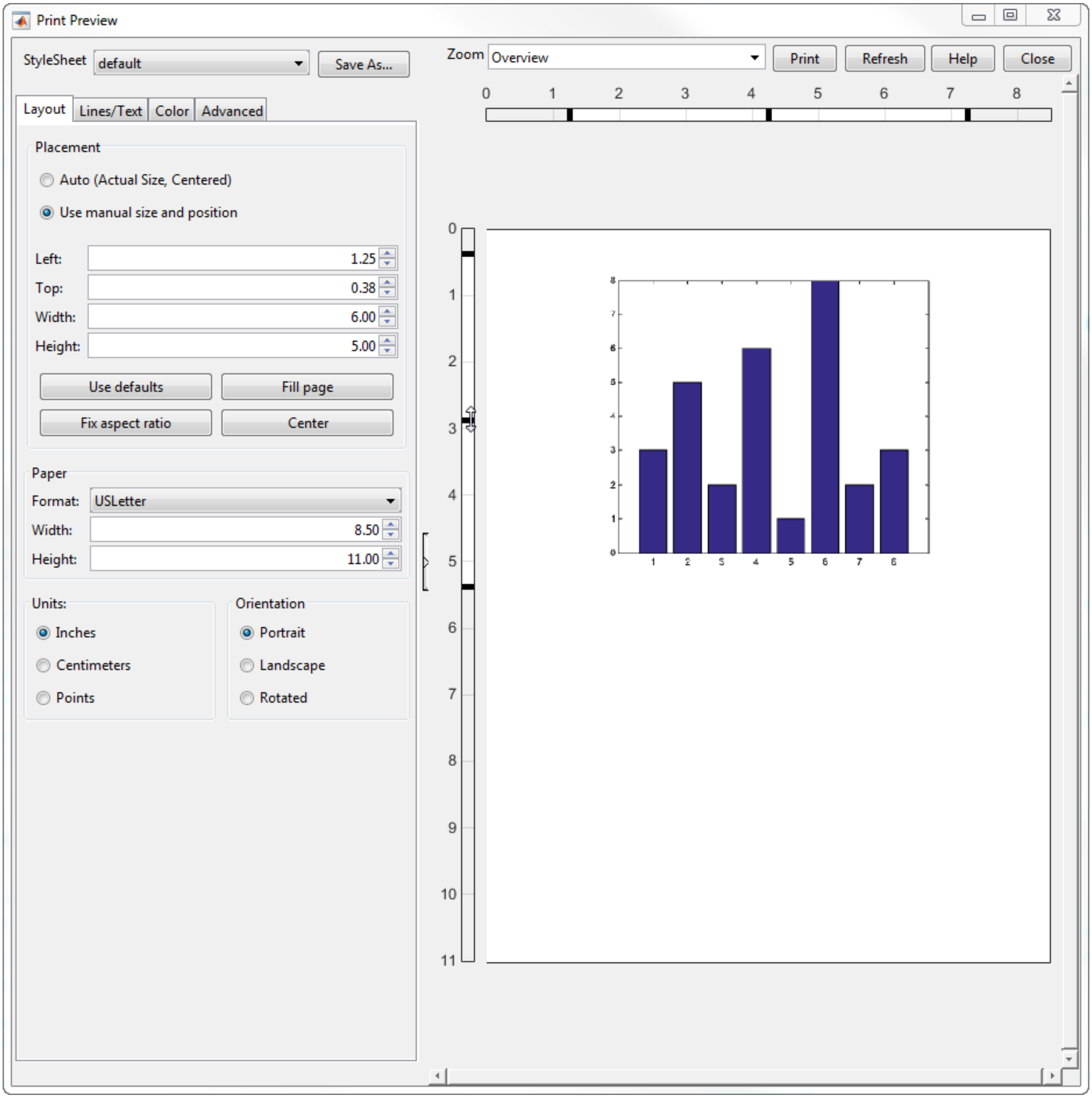
- Preserve the figure background color by clicking **File** > **Print Preview** > **Color** tab. Select **Same as figure** for the background color. Select **Color** for the color scale.
- Preserve the axis limits and tick value locations by clicking **File** > **Print Preview** > **Advanced** tab. Then, for the **Axis limits and ticks** option, select **Keep screen limits and ticks**.

To retain the color scheme programmatically, set the InvertHardcopy property of the figure to `'off'`. To keep the same axis limits and tick marks, set the `XTickMode`, `YTickMode`, and `ZTickMode` properties for the axes to `'manual'`.

Figure Size and Placement

To print a figure with specific dimensions, click **File** > **Print Preview** > **Layout** tab. Then, for the **Placement** option, select **Use manual size and position**. Specify the dimensions you want in the text boxes. Alternatively, use the sliders to the left and top of the figure preview to adjust the size and placement.

MATLAB changes the figure size in the print preview, but does not change the size of the actual figure.



StyleSheet: default

Zoom: Overview

Placement

○ Auto (Actual Size, Centered)

● Use manual size and position

Left: 1.25

Top: 0.38

Width: 6.00

Height: 5.00

Paper

Format: USLetter

Width: 8.50

Height: 11.00

Units:

● Inches

○ Centimeters

○ Points

Orientation

● Portrait

○ Landscape

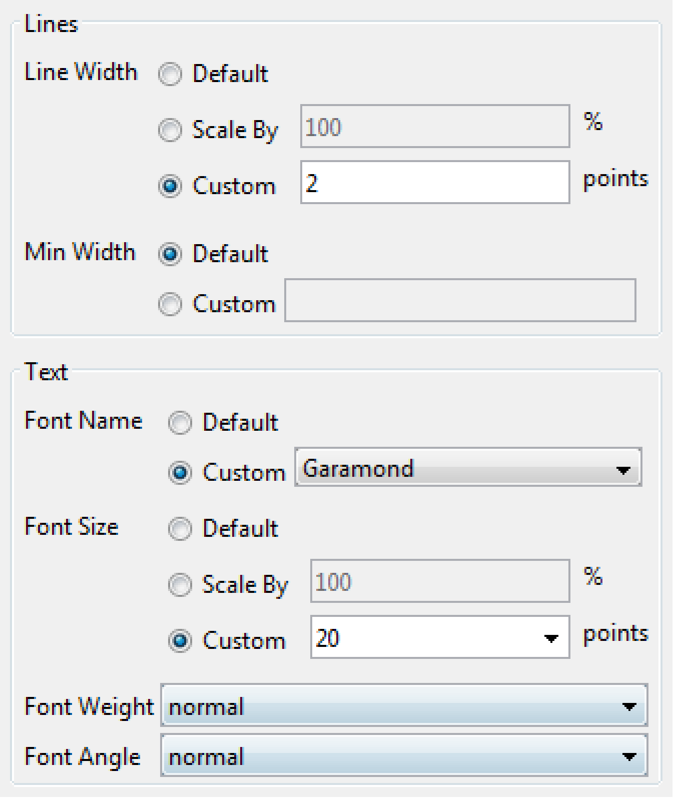
○ Rotated

To specify the printed figure size and placement programmatically, use the PaperPosition property for the figure.

Line Width and Font Size

To change the line width, font size, and font name for the printed output, click **File** > **Print Preview** > **Lines/Text** tab. Specify a custom line width in the appropriate text box, for example, 2 points. Select a font name from the dropdown list of fonts and specify a custom font size. For example, use 20 point Garamond font.

MATLAB changes the line width and font in the print preview, but does not change the appearance of the actual figure.



Lines

Line Width ○ Default
○ Scale By 100 %
● Custom 2 points

Min Width ● Default
○ Custom

Text

Font Name ○ Default
● Custom Garamond

Font Size ○ Default
○ Scale By 100 %
● Custom 20 points

Font Weight normal

Font Angle normal

To change the line width and font size programmatically, set properties of the graphics objects. For a list, see “Graphics Object Properties”.

See Also

`print` | `saveas`

Related Examples

- “Copy Figure to Clipboard from Edit Menu” on page 8-7
- “Customize Graph Using Plot Tools” on page 1-6

Copy Figure to Clipboard from Edit Menu

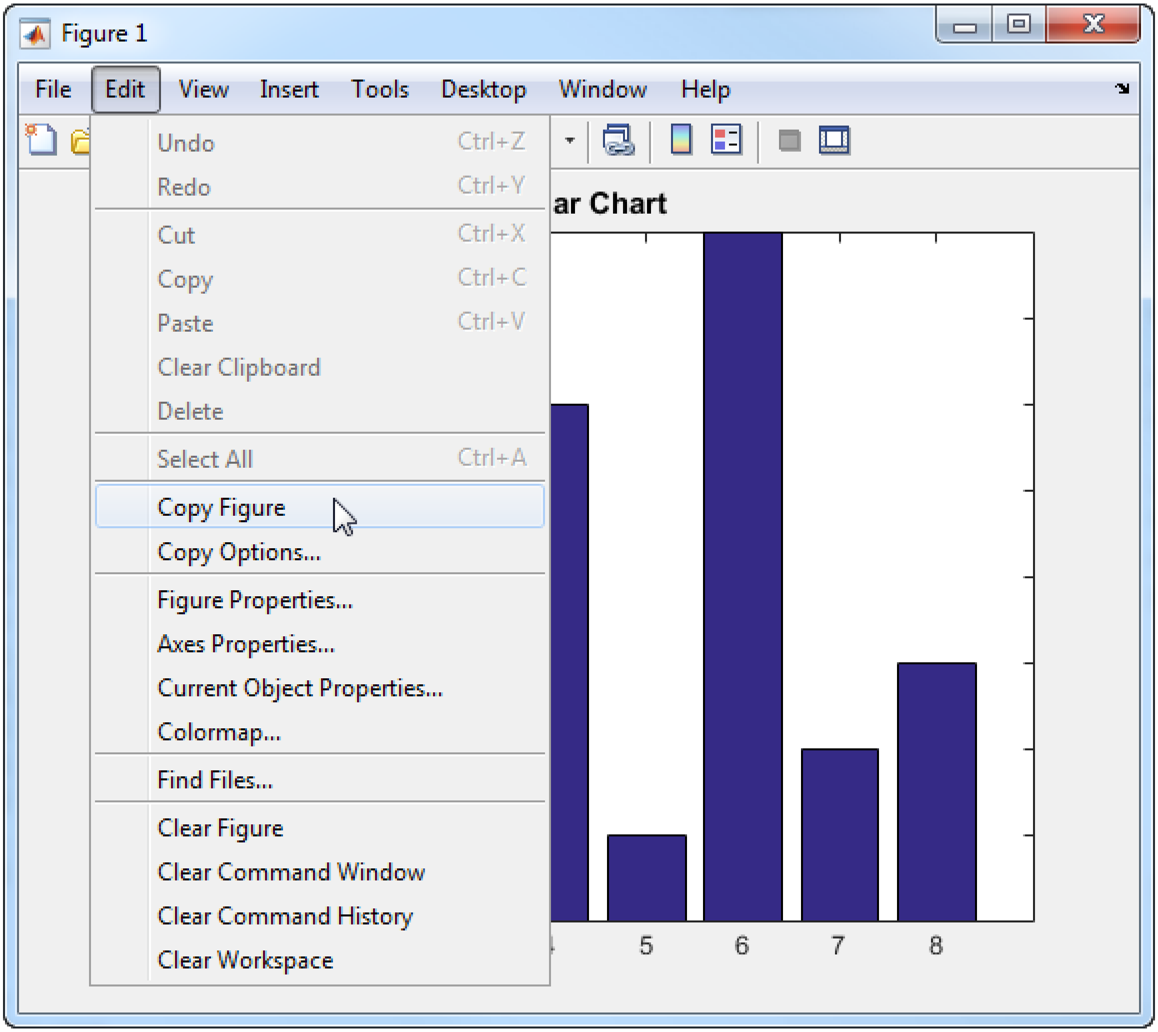
This example shows how to copy a figure to the clipboard and how to set copy options. When a figure is on the clipboard, you can paste it into other applications, such as a document or presentation.

In this section...
"Copy Figure to Clipboard" on page 8-7
"Specify Format, Background Color, and Size Options" on page 8-9

Copy Figure to Clipboard

Create a bar chart with a title. Copy the figure to your system clipboard by clicking **Edit** > **Copy Figure**.

```
x = [3 5 2 6 1 8 2 3];
bar(x)
title('Bar Chart')
```



Paste the copied figure into other applications, typically by right-clicking. By default, MATLAB converts the background color of the copied figure to white.

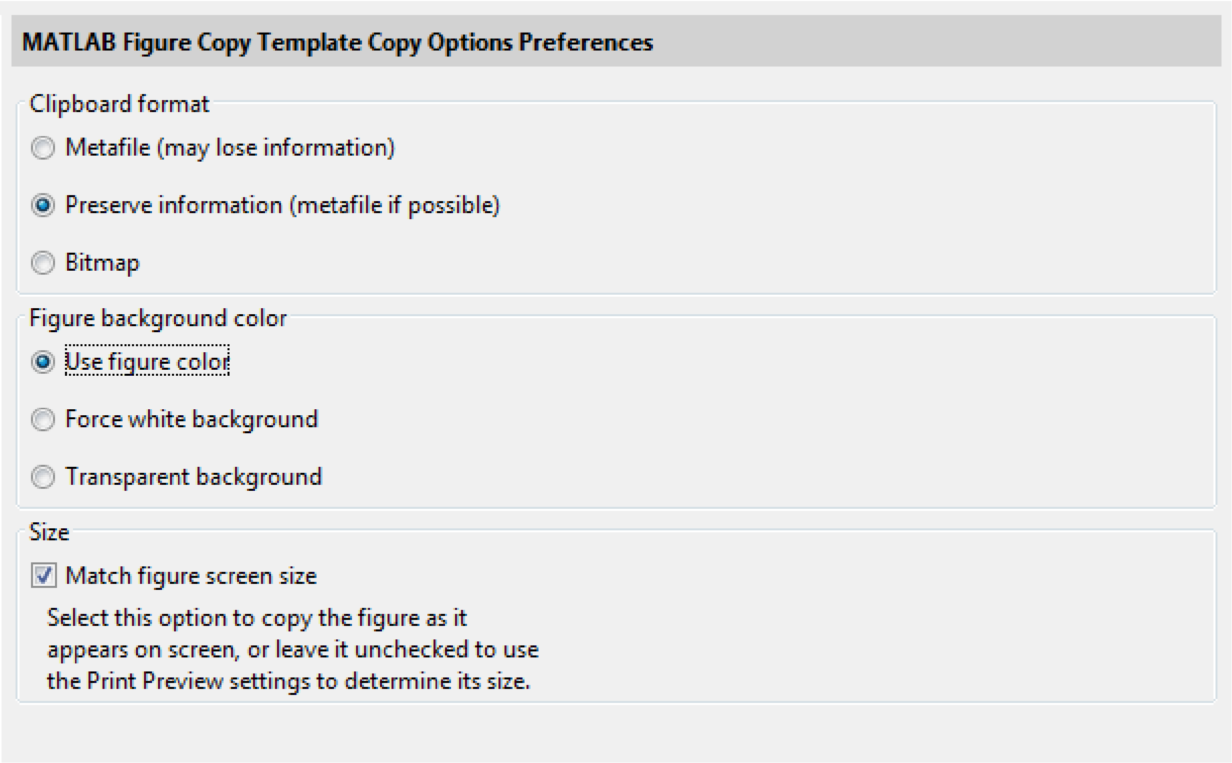
Note: The **Copy Figure** option is not available on Linux® systems. Use the programmatic alternative.

To copy the figure programmatically, use the `'-clipboard'` option with `print`. Specify the format as either `'-dbitmap'`, `'-dpdf'`, or `'-dmeta'`. The metafile format, `'-dmeta'`, is supported on Windows systems only.

Specify Format, Background Color, and Size Options

You can adjust certain settings for figures that are copied to the clipboard. Access these options by selecting**Edit** > **Copy Options** from the figure menu. The settings apply to all future figures copied to the clipboard. They do not affect the way the figure looks on the screen.

Note: This window is available on Windows systems only. On Mac and Linux systems, use the programmatic alternatives.



MATLAB Figure Copy Template Copy Options Preferences

Clipboard format

○ Metafile (may lose information)

● Preserve information (metafile if possible)

○ Bitmap

Figure background color

● Use figure color

○ Force white background

○ Transparent background

Size

☑ Match figure screen size

Select this option to copy the figure as it appears on screen, or leave it unchecked to use the Print Preview settings to determine its size.

Set the clipboard format to one of these options:

- Metafile — Copy the figure in an EMF color vector format.
- Preserve information — Select the format based on the figure's renderer. If the renderer is Painters, then the format is a metafile. If the renderer is OpenGL®, then the format is a bitmap image.
- Bitmap — Copy the figure in a bitmap format.

Set the figure background color to one of these options:

- Use figure color — Keep the background color the same as it appears on the screen. To use the programmatic alternative, set the `InvertHardcopy` property for the figure to `'off'` before copying.
- Force white background — Copy the figure with a white background. To use the programmatic alternative, set the `InvertHardcopy` property for the figure to `'on'` before copying.
- Transparent background — Copy the figure with a transparent background. To use the programmatic alternative, set the `Color` property for the figure to `'none'` and the `InvertHardcopy` property to `'off'` before copying. Metafile and PDF formats support transparency. Bitmap formats do not support transparency.

Copy the figure with the same size as it appears on the screen by selecting **Match figure screen size**. Clear this option to use the width and height specified in the Export Setup dialog box.

See Also

`print` | `saveas`

Related Examples

- "Save Figure to Open in Another Application" on page 8-19
- "Customize Graph Using Plot Tools" on page 1-6

Customize Figure Interactively Before Saving

This example shows how to use the Export Setup window to customize a figure before saving it. It shows how to change the figure size, background color, font size, and line width. It also shows how to save the settings as an export style that you can apply to other figures before saving them.

In this section...

"Set Figure Size" on page 8-11

"Set Figure Background Color" on page 8-13

"Set Figure Font Size and Line Width" on page 8-14

"Save Figure to File" on page 8-16

"Save Figure Settings for Future Use" on page 8-17

"Apply Settings to Another Figure" on page 8-17

"Restore Figure to Original Settings" on page 8-18

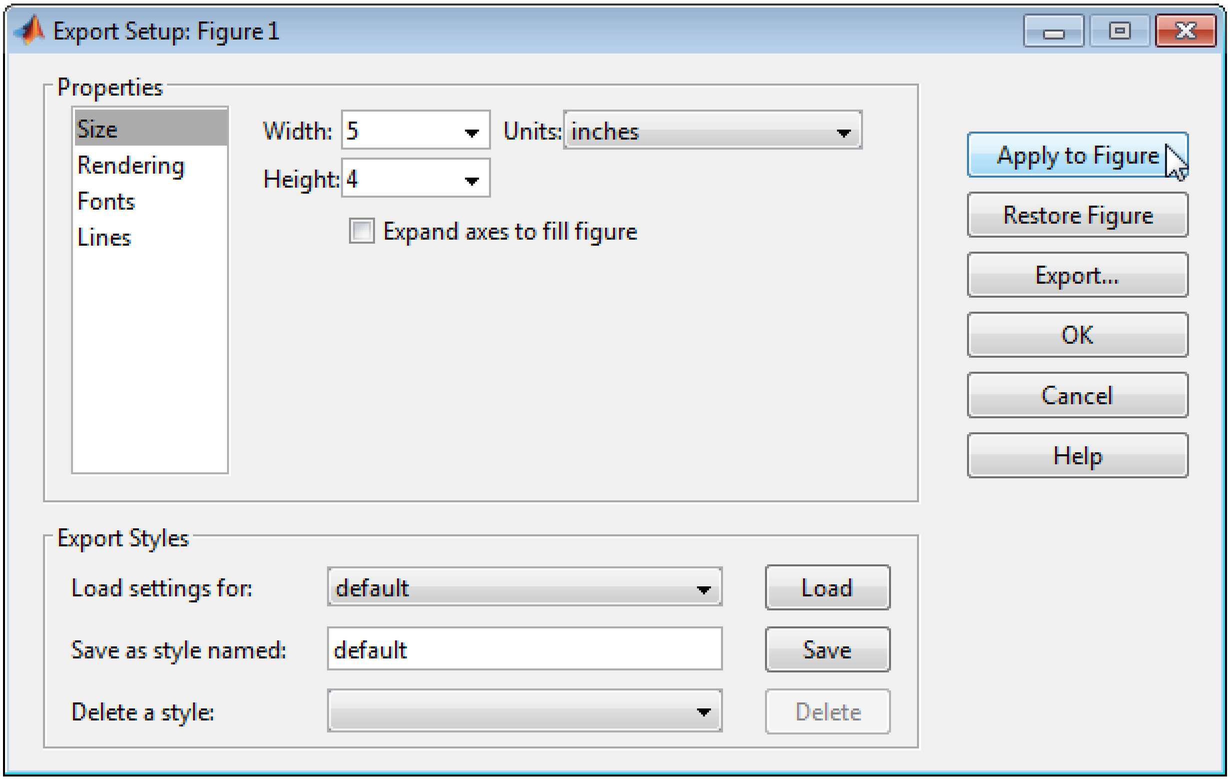
Set Figure Size

Create a line plot.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
```

Set the figure size by clicking **File** > **Export Setup**. Specify the desired dimensions in the **Width** and **Height** fields, for example 5-by-4 inches. The dimensions include the entire figure window except for the frame, title bar, menu bar, and any tool bars. If the specified width and height are too large, then the figure might not reach the specified size.

To make the axes fill the figure, select **Expand axes to fill figure**. This option only affects axes with an `ActivePositionProperty` property set to `'outerposition'`. By default, it does not affect subplots since subplots have an `ActivePositionProperty` set to `'position'`.



Width: 5 Units: inches

Height: 4

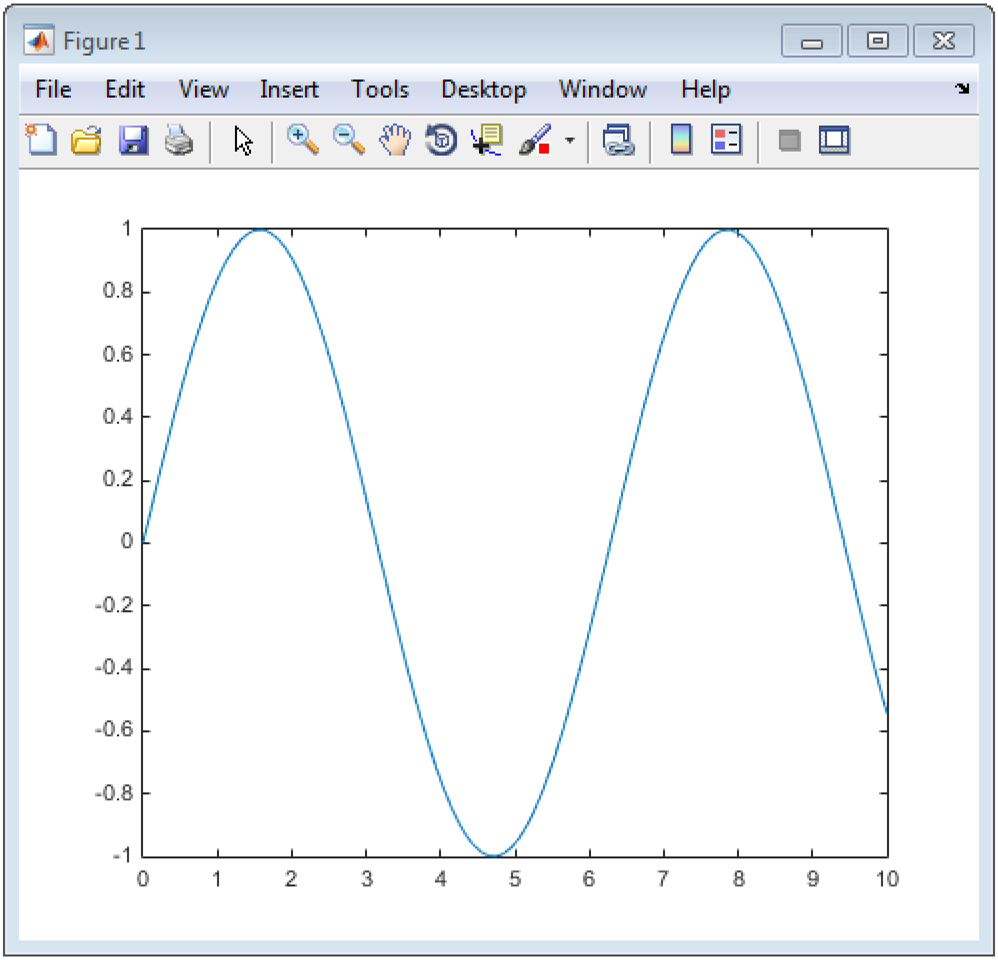
□ Expand axes to fill figure

Load settings for: default

Save as style named: default

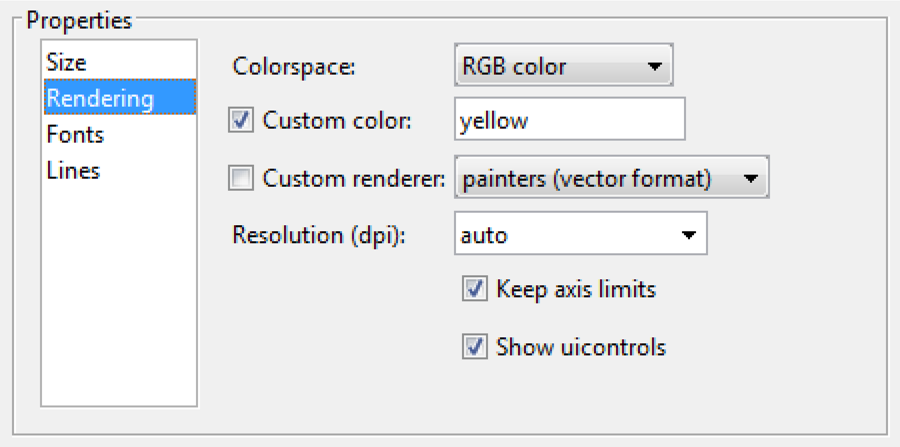
Delete a style:

Click **Apply to Figure**. Applying the settings changes the appearance of the figure on the screen. All settings from the Export Setup dialog are applied to the figure. Thus, more than just the figure size can change. For example, by default, MATLAB converts the background color of the saved figure to white.



Set Figure Background Color

Set the figure background color by clicking the **Rendering** property in the Export Setup window. In the Custom color field, specify either a color name from the table or an RGB triplet. For example, set the background color to yellow.



Colorspace: RGB color

☑ Custom color: yellow

□ Custom renderer: painters (vector format)

Resolution (dpi): auto

☑ Keep axis limits

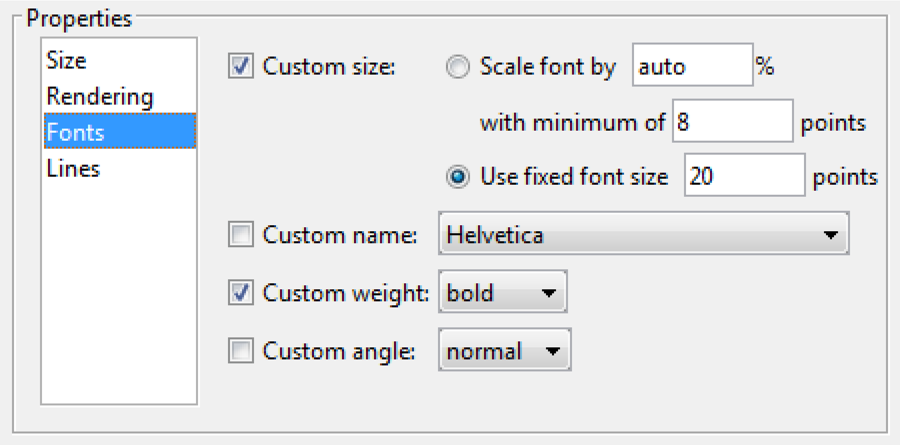
☑ Show uicontrols

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists some common RGB triplets that have corresponding color names. To specify the default gray background color, set the Custom color field to `default`.

Long Name	Short Name	Corresponding RGB Triplet
`white`	`w`	`[1 1 1]`
`yellow`	`y`	`[1 1 0]`
`magenta`	`m`	`[1 0 1]`
`red`	`r`	`[1 0 0]`
`cyan`	`c`	`[0 1 1]`
`green`	`g`	`[0 1 0]`
`blue`	`b`	`[0 0 1]`
`black`	`k`	`[0 0 0]`

Set Figure Font Size and Line Width

Change the font by clicking the **Fonts** property. Specify a fixed font size and select a font name, font weight, and font angle. For example, use 20 point bold font. The tick mark locations might change to accommodate the new font size.



Properties

Size
Rendering
Fonts
Lines

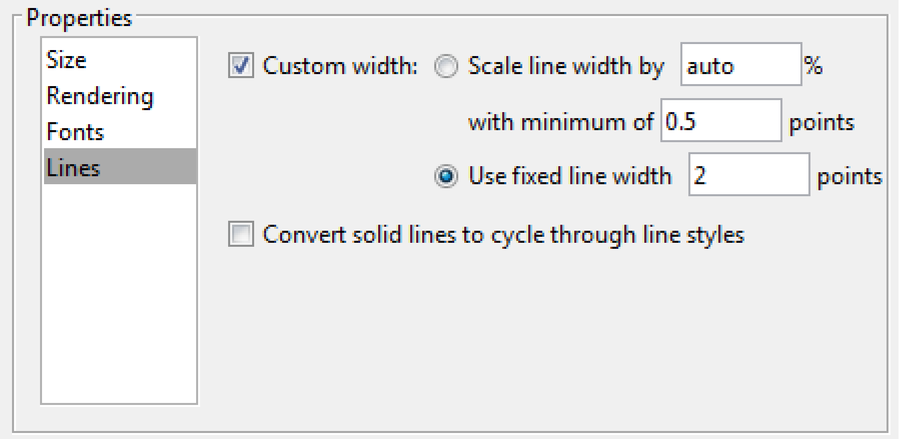
☑ Custom size: ○ Scale font by auto %
with minimum of 8 points
● Use fixed font size 20 points

□ Custom name: Helvetica

☑ Custom weight: bold

□ Custom angle: normal

Change the line width by clicking the **Lines** property. Specify a fixed line width, for example, 2 points.



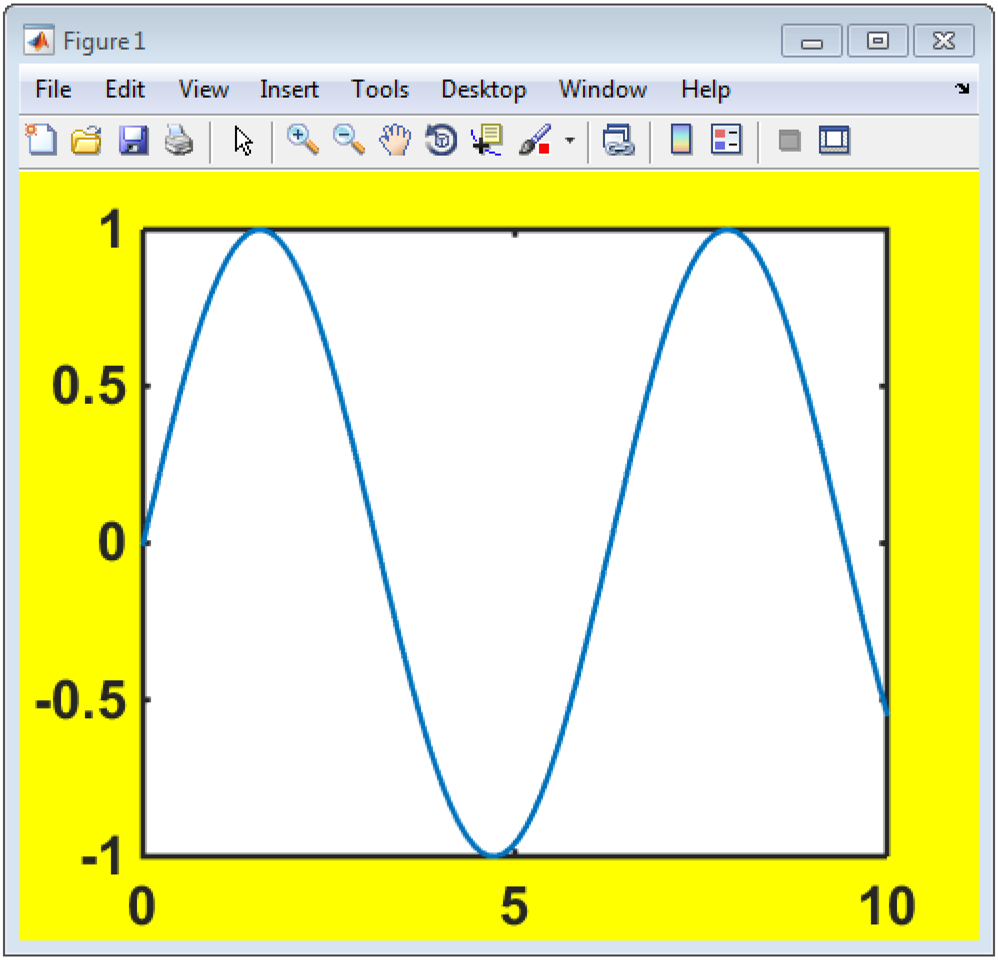
Properties

Size
Rendering
Fonts
Lines

☑ Custom width: ○ Scale line width by auto %
with minimum of 0.5 points
● Use fixed line width 2 points

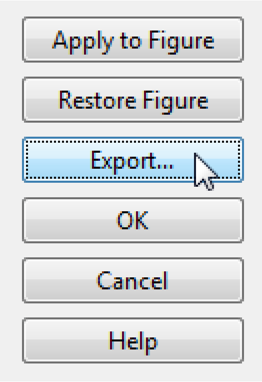
□ Convert solid lines to cycle through line styles

Click **Apply to Figure** on the right side of the Export Setup dialog.



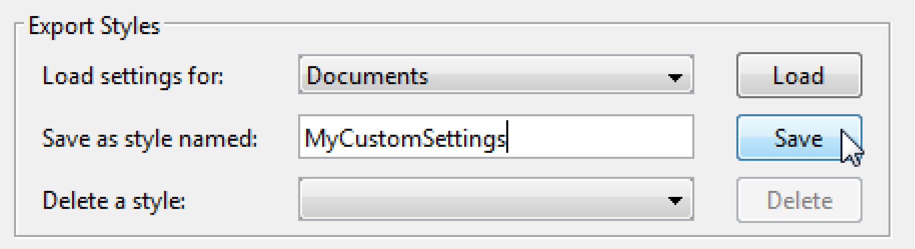
Save Figure to File

Save the figure to a file by first clicking **Export**, and then specifying a file name, location, and desired format. For more information about file formats, see `saveas`.



Save Figure Settings for Future Use

Save your settings to use for future figures by creating an export style. In the Export Styles section, type a style name, for example `MyCustomSettings`. Then, click Save.


Export Styles

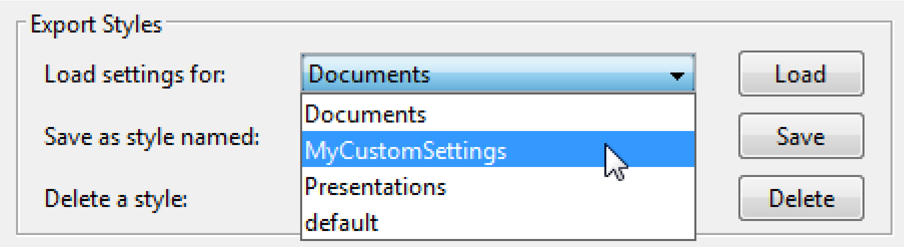
Load settings for: Documents

Save as style named: MyCustomSettings

Delete a style:

Apply Settings to Another Figure

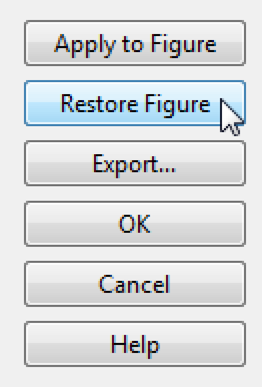
Apply your settings to another figure by opening the Export Setup box from its figure menu. In the Export Styles section, select the style name and click **Load**. Next, click **Apply to Figure** on the right side of the Export Setup dialog. MATLAB applies the saved style settings to the figure.



Load settings for: Documents

Restore Figure to Original Settings

Restore the figure on the screen to the original settings by clicking **Restore Figure**.



See Also

`print` | `saveas`

Related Examples

- "Save Figure to Open in Another Application" on page 8-19
- "Save Figure at Specific Size and Resolution" on page 8-27
- "Customize Graph Using Plot Tools" on page 1-6

Save Figure to Open in Another Application

In this section...

“Choose File Format” on page 8-19

“Save Figure for Document or Presentation” on page 8-19

“Save Figure for Editing in Another Application” on page 8-20

“Customize Figure Before Saving” on page 8-20

“Include Figure in Microsoft Application or LaTeX Document” on page 8-21

“File Format Options” on page 8-21

Choose File Format

Before saving the figure, consider the file format you want to use. You can save the figure as either a bitmap image or a vector graphics file.

- Bitmap images contain a pixel-based representation of the figure. This type of format is widely used by web browsers and other applications that display graphics. However, bitmap images do not scale well. You cannot modify individual graphics objects, such as lines and text, in other graphics applications. Supported bitmap image formats include PNG and JPEG.
- Vector graphics files store commands that redraw the figure. This type of format scales well, but can result in a large file. Also, vector graphics files might not produce the correct 3-D arrangement of objects in certain cases. Some applications support extensive editing of vector graphics formats, but some applications support only resizing the graphic. In general, try to make all the necessary changes while your figure is still in MATLAB. Supported vector graphics formats include PDF, EPS, and SVG.

For a full list of supported file formats, see “File Format Options” on page 8-21.

Save Figure for Document or Presentation

To save a figure, use either the `saveas` or `print` function. For example, save a bar chart as a PNG file. Use `gcf` to save the current figure.

```
bar([1 10 7 8 2 2 9 3 6])
saveas(gcf,'BarChartFile.png')
```

If you want additional control over the saved output, such as setting the resolution or controlling the renderer, use the `print` function to save the figure instead.

Note: Details of saved and printed figures can differ from the figure on the display. To get output that is more consistent with the display, see “Save Figure Preserving Background Color” on page 8-24 and “Save Figure at Specific Size and Resolution” on page 8-27.

Save Figure for Editing in Another Application

If you want to edit a figure in another application, save it as a vector graphics file, such as PDF or EPS. Use either the `saveas` or `print` function to save the figure.

For example, save a bar chart as an EPS file with color using the `'epsc'` file format.

```
bar([1 10 7 8 2 2 9 3 6])
saveas(gcf,'BarChartFile','epsc')
```

`saveas` saves the bar chart as `BarChartFile.eps`. For a black and white EPS file, use the `'eps'` format instead. For a full list of supported file formats, see “File Format Options” on page 8-21.

Customize Figure Before Saving

Customizing your figure in MATLAB before you save it can eliminate the need to use another application for editing.

To customize the figure programmatically, set properties of the graphics objects. Typically, graphics functions return output arguments that you can use to access and modify graphics objects. For example, assign the chart line objects returned from the `plot` function to a variable and set their `LineWidth` property.

```
p = plot(rand(5));
set(p,'LineWidth',3)
```

If you do not return the graphics objects as output arguments, you can use `findobj` to find objects with certain properties. For example, find all objects in the current figure with a `Type` property set to `'line'`. Then, set their `LineWidth` property.

```
plot(rand(5))
```

```
p = findobj(gcf,'Type','line')
set(p,'LineWidth',3);
```

For a list of all graphics objects and their properties, see “Graphics Object Properties”.

To customize the figure interactively, use either the Plot Tools or the Export Setup dialog. For more information on the Plot Tools, see “Customize Graph Using Plot Tools” on page 1-6. For more information on the Export Setup dialog, see “Customize Figure Interactively Before Saving” on page 8-11.

Include Figure in Microsoft Application or LaTeX Document

To import a figure into a Microsoft application, such as Word or PowerPoint®, click **Insert** > **Picture** > **From File** in the application. Then, navigate to your saved file.

To add a figure to a LaTeX document, first save the figure using an EPS format. For example, `saveas(gcf,'BarChart','epsc')`. Then, use the `\includegraphics` element in the LaTeX document to include the file. For example:

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{BarChart.eps}}
\caption{Bar Chart from MATLAB}
\end{figure}

\end{document}
```

File Format Options

This table lists the supported bitmap image formats.

Option	Bitmap Image Format	Corresponding File Extension
`'jpeg'`	JPEG 24-bit	`.jpg`
`'png'`	PNG 24-bit	`.png`
`'tiff'`	TIFF 24-bit (compressed)	`.tif`
`'tiffn'`	TIFF 24-bit (not compressed)	`.tif`

Option	Bitmap Image Format	Corresponding File Extension
`'meta'`	Enhanced metafile (Windows only)	`.emf`
`'bmpmono'`	BMP Monochrome	`.bmp`
`'bmp'`	BMP 24-bit	`.bmp`
`'bmp16m'`	BMP 24-bit	`.bmp`
`'bmp256'`	BMP 8-bit (256 color, uses a fixed colormap)	`.bmp`
`'hdf'`	HDF 24-bit	`.hdf`
`'pbm'`	PBM (plain format) 1-bit	`.pbm`
`'pbmraw'`	PBM (raw format) 1-bit	`.pbm`
`'pcxmono'`	PCX 1-bit	`.pcx`
`'pcx24b'`	PCX 24-bit color (three 8-bit planes)	`.pcx`
`'pcx256'`	PCX 8-bit newer color (256 color)	`.pcx`
`'pcx16'`	PCX older color (EGA/VGA 16-color)	`.pcx`
`'pgm'`	PGM (plain format)	`.pgm`
`'pgmraw'`	PGM (raw format)	`.pgm`
`'ppm'`	PPM (plain format)	`.ppm`
`'ppmraw'`	PPM (raw format)	`.ppm`

This table lists the supported vector graphics formats.

Option	Vector Graphics Format	Corresponding File Extension
`'pdf'`	Full page Portable Document Format (PDF) color	`.pdf`
`'eps'`	Encapsulated PostScript® (EPS) Level 3 black and white	`.eps`

Option	Vector Graphics Format	Corresponding File Extension
`'epsc'`	Encapsulated PostScript (EPS) Level 3 color	`.eps`
`'eps2'`	Encapsulated PostScript (EPS) Level 2 black and white	`.eps`
`'epsc2'`	Encapsulated PostScript (EPS) Level 2 color	`.eps`
`'meta'`	Enhanced Metafile (Windows only)	`.emf`
`'svg'`	SVG (Scalable Vector Graphics)	`.svg`
`'ps'`	Full-page PostScript (PS) Level 3 black and white	`.ps`
`'psc'`	Full-page PostScript (PS) Level 3 color	`.ps`
`'ps2'`	Full-page PostScript (PS) Level 2 black and white	`.ps`
`'psc2'`	Full-page PostScript (PS) Level 2 color	`.ps`

See Also

`print` | `saveas`

Related Examples

- “Save Figure at Specific Size and Resolution” on page 8-27
- “Save Figure to Reopen in MATLAB Later” on page 8-33

Save Figure Preserving Background Color

In this section...

"Retain Current Background Color" on page 8-24
"Change Background Color" on page 8-25

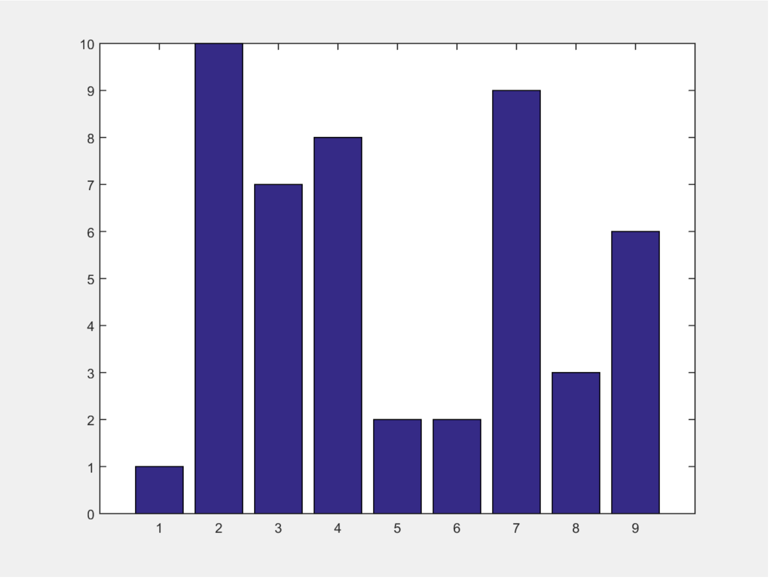
Retain Current Background Color

By default, saved figures have a white background. Ensure that the colors of the saved figure match the colors on the display by setting the InvertHardcopy property of the figure to `'off'`. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

For example, create a bar chart and save it as a PNG file. Retain the figure background color in the saved output.

```
bar([1 10 7 8 2 2 9 3 6])
fig = gcf;
fig.InvertHardcopy = 'off';
saveas(gcf,'GrayBackground.png')
```

`saveas` saves the file, `GrayBackground.png`, in your current folder. The saved figure has the same gray background color as the onscreen figure.

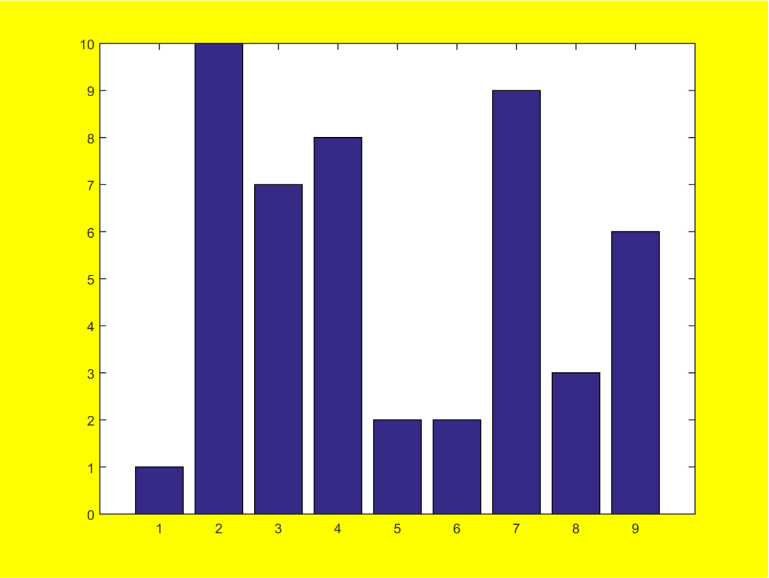


Change Background Color

To change the figure color, set the Color property for the figure. For example, change the color to yellow before saving the figure.

```
bar([1 10 7 8 2 2 9 3 6])
fig = gcf;
fig.Color = 'yellow';
fig.InvertHardcopy = 'off';
saveas(gcf,'YellowBackground.png')
```

`saveas` saves the file, `YellowBackground.png`, in your current folder. The saved figure has the same yellow background color as the onscreen figure.



See Also

`print` | `saveas`

Related Examples

- "Save Figure at Specific Size and Resolution" on page 8-27
- "Save Axes Without Saving UIControls" on page 8-36

Save Figure at Specific Size and Resolution

In this section...

“Use Screen Size and Resolution” on page 8-27

“Expand Figure to Fill Page” on page 8-28

“Use Specific Dimensions” on page 8-29

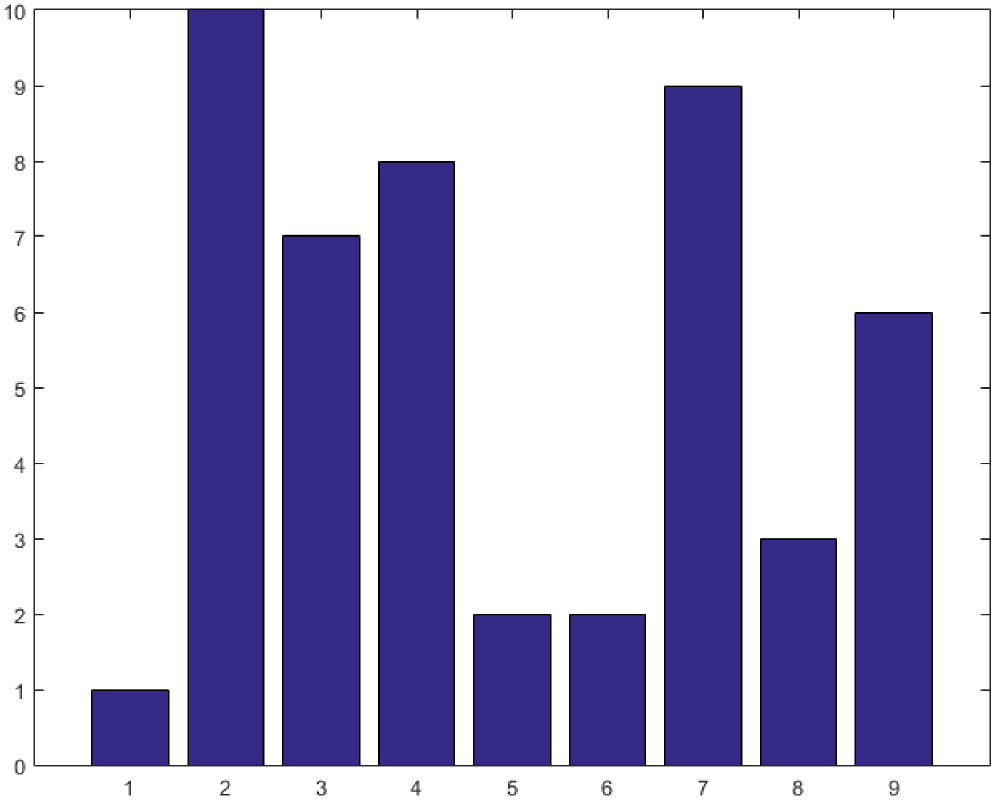
“Preserve Axis Limits and Tick Values” on page 8-31

Use Screen Size and Resolution

To print or save figures that are the same size as the figure on the screen, ensure that the PaperPositionMode property of the figure is set to `'auto'` before printing. To generate output that matches the on-screen size in pixels, include the `'-r0'` resolution option when using the `print` function.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, for example, `set(fig,'PaperPositionMode','auto')`.

```
bar([1 10 7 8 2 2 9 3 6])
fig = gcf;
fig.PaperPositionMode = 'auto';
print('ScreenSizeFigure','-dpng','-r0')
```



Expand Figure to Fill Page

To print or save a figure that fills the page, use `print` with either the `'-fillpage'` or `'-bestfit'` option. Both options are valid only when printing a figure to a printer or saving it to a paged format (PDF and full page PostScript).

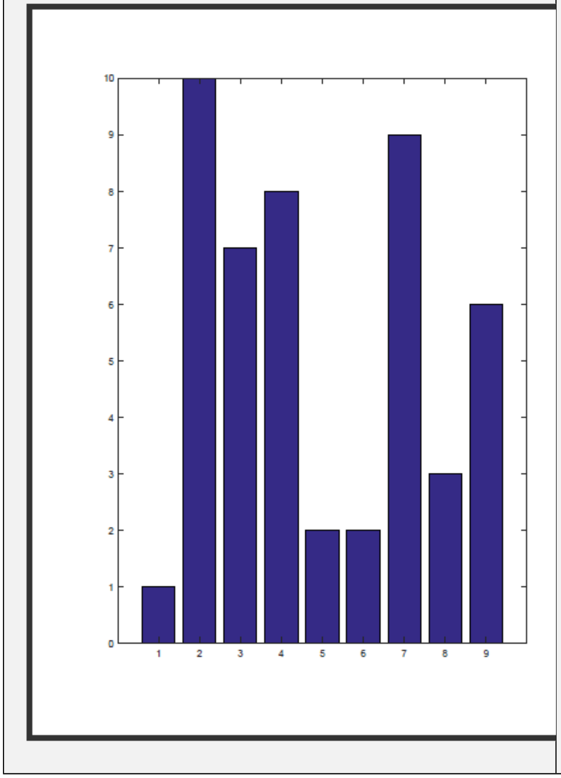
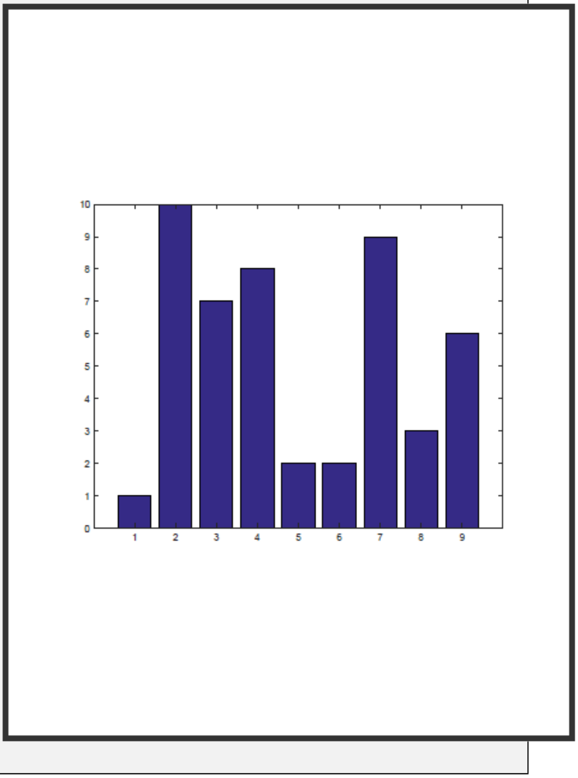
- The `'-fillpage'` option maximizes the size of the figure to fill the page and leaves a .25 inch page margin. The tick marks, layout, and aspect ratio of the figure might change.

- The `'-bestfit'` option maximizes the size of the figure to fill the page, but preserves the aspect ratio of the figure. The figure might not fill the entire page. This option leaves a minimum page margin of .25 inches.

This table shows an example of each option and the resulting output size.

Fill Page Option	Best Fit Option

`bar([1 10 7 8 2 2 9 3 6])`
`print('-fillpage','FillPageFigure','-d`
	`bar([1 10 7 8 2 2 9 3 6])`
	`print('-bestfit','BestFitFigure','-dpdf')`

Use Specific Dimensions

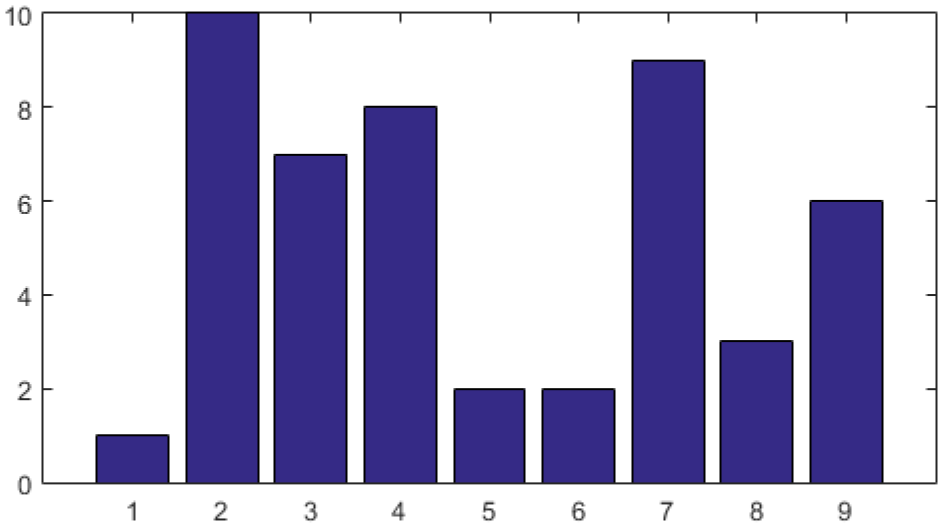
To save or print a figure with specific dimensions, set the PaperPosition property of the figure to the desired dimensions. The `PaperPosition` property affects the size of saved and printed figures, but does not affect the size of the figure on the display. Set the property to a four-element vector of the form `[left bottom width height]`.

- `left` and `bottom` values — Control the distance from the lower left corner of the page to the lower left corner of the figure. These values are ignored when saving a figure to a nonpage format, such as a PNG or EPS format.
- `width` and `height` values — Control the figure dimensions. The dimensions include the entire figure window except for the frame, title bar, menu bar, and any tool bars. If the width and height values are too large, then the figure might not reach the specified size. If the figure does not reach the specific size, then any UI components on the figure, such as uicontrols or a uitable, might not save or print as expected.

For example, save the figure with 6-by-3 inch dimensions using screen resolution.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, for example, `set(fig,'PaperUnits','inches')`.

```
bar([1 10 7 8 2 2 9 3 6])
fig = gcf;
fig.PaperUnits = 'inches';
fig.PaperPosition = [0 0 6 3];
print('5by3DimensionsFigure','-dpng','-r0')
```



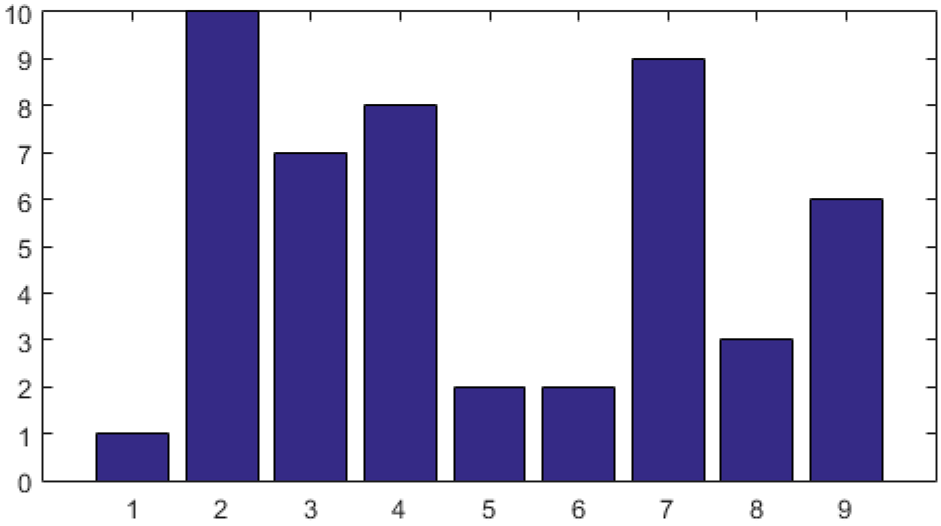
Preserve Axis Limits and Tick Values

If the size of the saved or printed figure is different from the size on screen, the axis limits and tick values can change to accommodate the new size. To keep the axis limits and tick values from changing, set the tick value mode and limit mode properties for the axes to `'manual'`.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead, for example, `set(ax,'XTickMode','manual')`.

```
bar([1 10 7 8 2 2 9 3 6])
ax = gca;
ax.XTickMode = 'manual';
ax.YTickMode = 'manual';
ax.ZTickMode = 'manual';
ax.XLimMode = 'manual';
ax.YLimMode = 'manual';
ax.ZLimMode = 'manual';
```

```
fig = gcf;
fig.PaperUnits = 'inches';
fig.PaperPosition = [0 0 6 3];
print('SameAxisLimits','-dpng','-r0')
```



See Also

`print` | `saveas`

Related Examples

- "Save Figure Preserving Background Color" on page 8-24
- "Save Figure to Open in Another Application" on page 8-19

Save Figure to Reopen in MATLAB Later

This example shows how to save a figure so that you can reopen it in MATLAB later. You can either save the figure to a FIG-file or you can generate and save the code.

In this section...

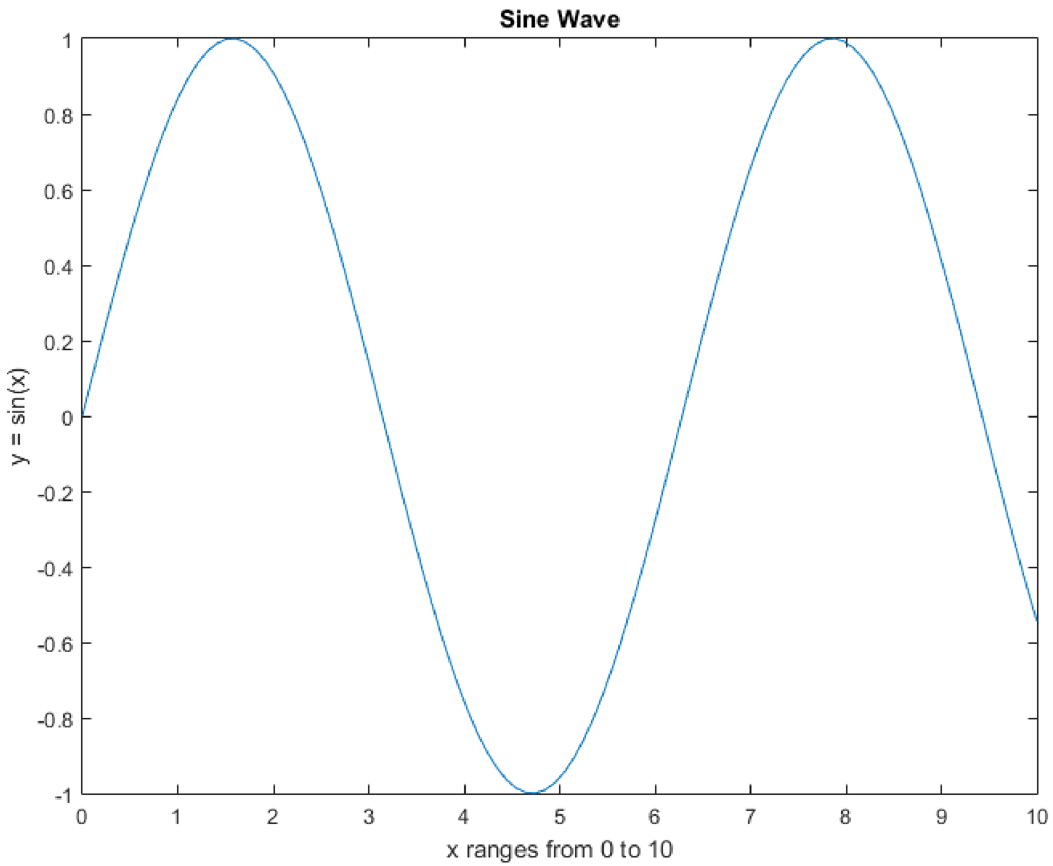
"Save Figure to FIG-File" on page 8-33

"Generate Code to Recreate Figure" on page 8-35

Save Figure to FIG-File

Create a plot to save. Add a title and axis labels.

```
x = linspace(0,10);
y = sin(x);
plot(x,y)
title('Sine Wave')
xlabel('x ranges from 0 to 10')
ylabel('y = sin(x)')
```



Save the figure to a FIG-file using the `savefig` function. The FIG-file stores the information required to recreate the figure.

```
savefig('SineWave.fig')
```

Close the figure, then reopen the saved figure using the `openfig` function.

```
close(gcf)
openfig('SineWave.fig')
```

`openfig` creates a new figure, a new axes, and a new line 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, any current default values apply to the new figure. You can interact with the figure. For example, you can pan, zoom, and rotate the axes.

Note: FIG-files open in MATLAB only. If you want to save the figure in a format that can be opened in another application, see “Save Figure to Open in Another Application” on page 8-19.

Generate Code to Recreate Figure

Alternatively, generate the MATLAB code for the plot and then use the code to reproduce the graph. Generating the code captures modifications that you make using the plot tools.

Click **File** > **Generate Code...**. The generated code displays in the MATLAB Editor. Save the code by clicking **File** > **Save As**.

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.

See Also

`openfig` | `saveas` | `savefig`

Related Examples

- “Save Figure to Open in Another Application” on page 8-19
- “Customize Graph Using Plot Tools” on page 1-6

Save Axes Without Saving UIControls

To save only the axes from a figure that has uicontrols, you can use `print` with the `'-noui'` option. Alternatively, you can copy the axes to a new figure and save the new figure.

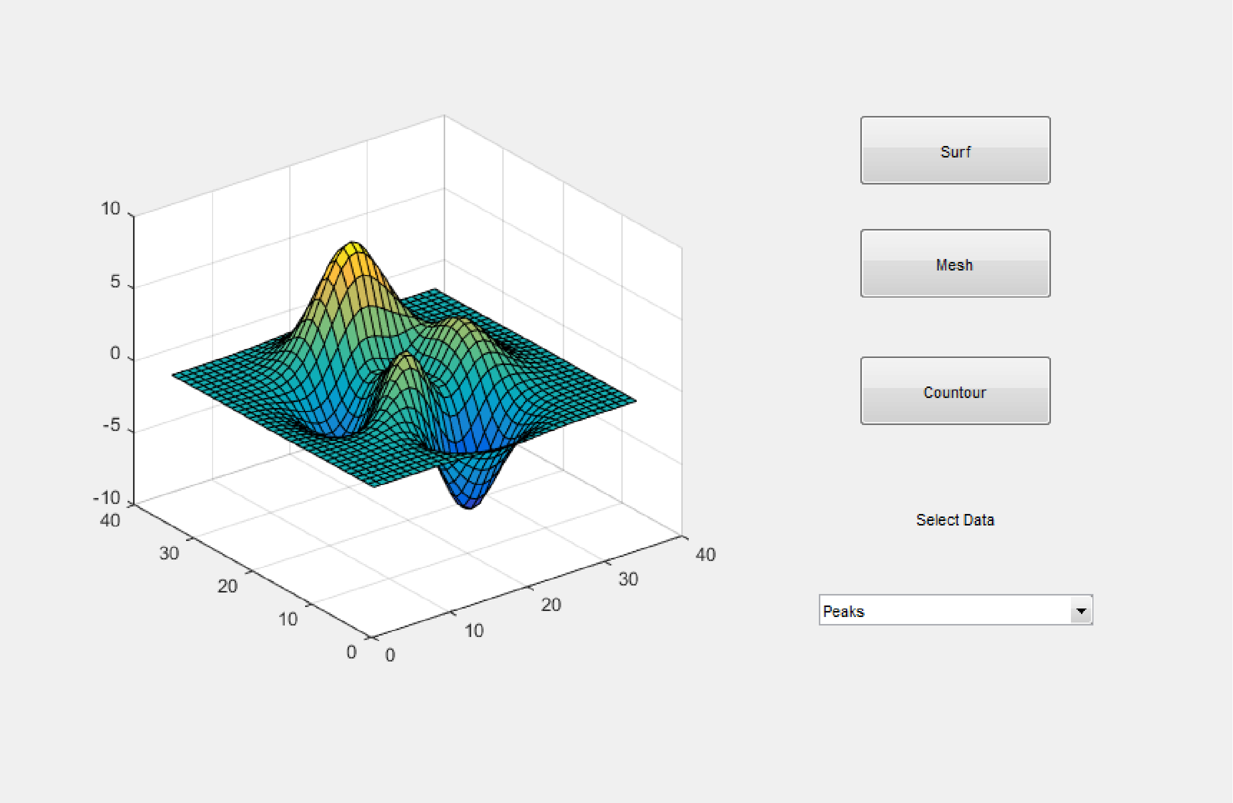
In this section...

"Create Figure with UIControls" on page 8-36
"Save Axes Without Saving UIControls" on page 8-37
"Copy Axes to New Figure and Save" on page 8-38

Create Figure with UIControls

To create an example of a figure with uicontrols, set your current folder to one to which you have write access. Then, copy this example code.

```
copyfile(fullfile(docroot,'techdoc','creating_guis','examples','simple_gui2*.*'));
simple_gui2
```

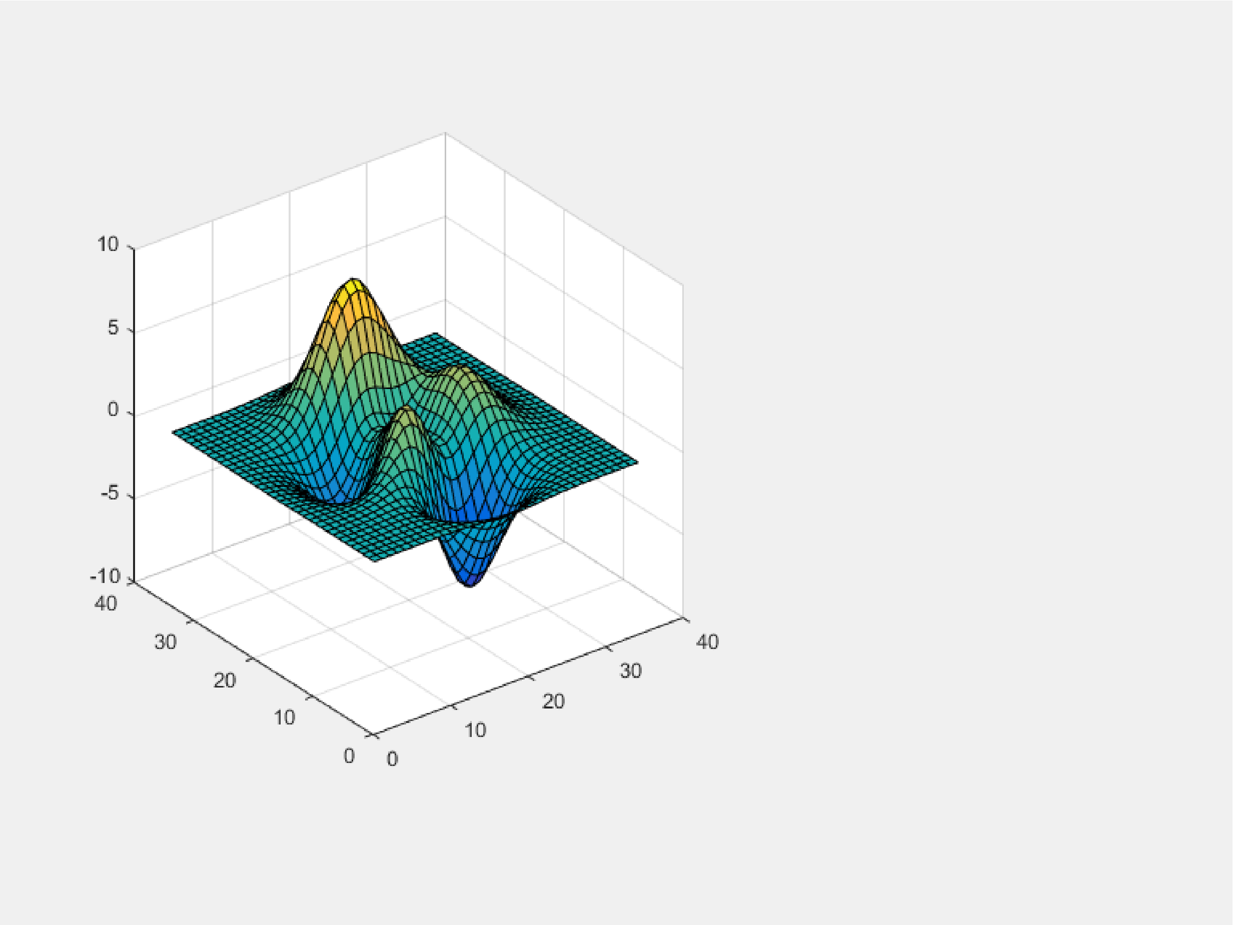


Save Axes Without Saving UIControls

To save the figure and exclude the uicontrols from the saved output, use `print` with the `'-noui'` option. `print` leaves blank space in place of the uicontrols. If you do not specify the `'-noui'` option, then `print` includes the uicontrols in the saved output.

To maintain the current figure background color in the saved figure, set the `InvertHardcopy` property of the figure to `'off'`. Otherwise, the saved figure has a white background. Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the `set` function instead.

```
fig = gcf;
fig.InvertHardcopy = 'off';
print('PlotWithoutUIControls','-dpng','-noui')
```



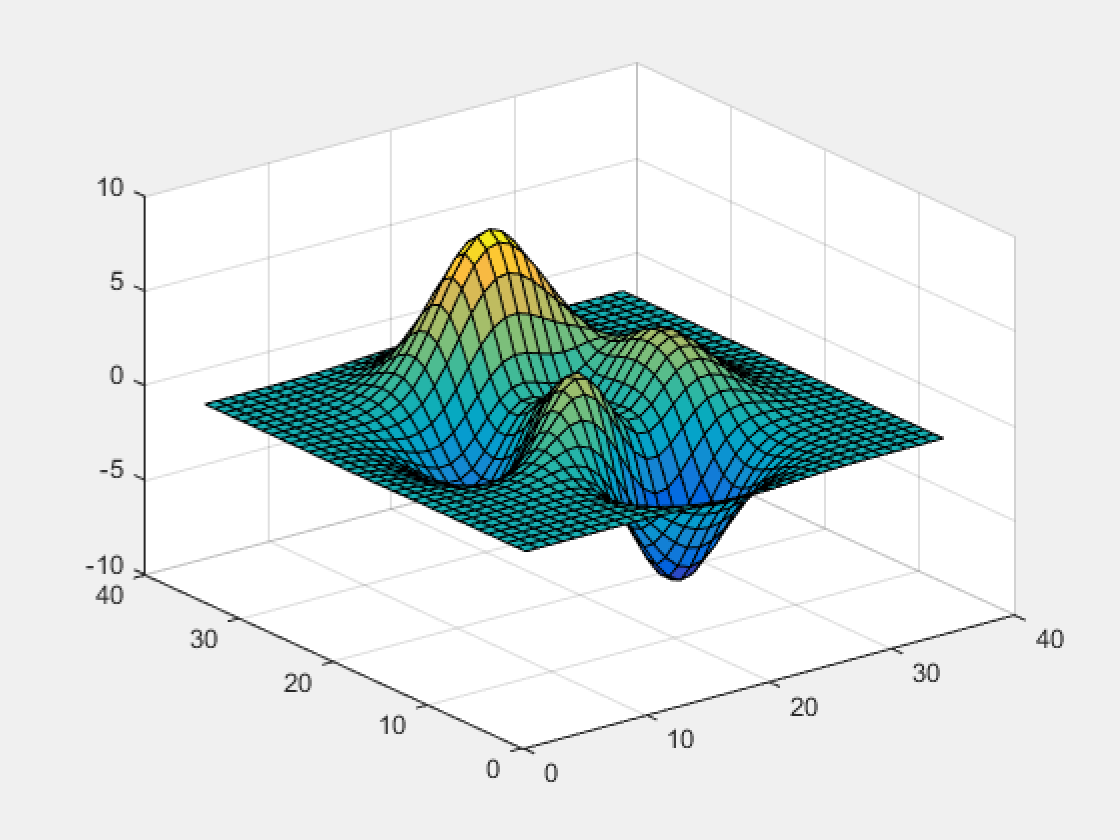
Copy Axes to New Figure and Save

To eliminate the blank space, copy the axes to a new figure and resize the axes to fill the figure. For example, click the axes to make it the current axes. Use `copyobj` to copy it to a new figure. Then, set the `Position` property of the new axes to fill the figure.

```
ax_old = gca;
f_new = figure;
ax_new = copyobj(ax_old,f_new)
set(ax_new,'Position','default')
```

Save the new figure using either `saveas` or `print`.

```
print(f_new,'AxesOnly','-dpng')
```



See Also

`print` | `saveas`

Related Examples

- “Save Figure to Open in Another Application” on page 8-19
- “Save Figure at Specific Size and Resolution” on page 8-27

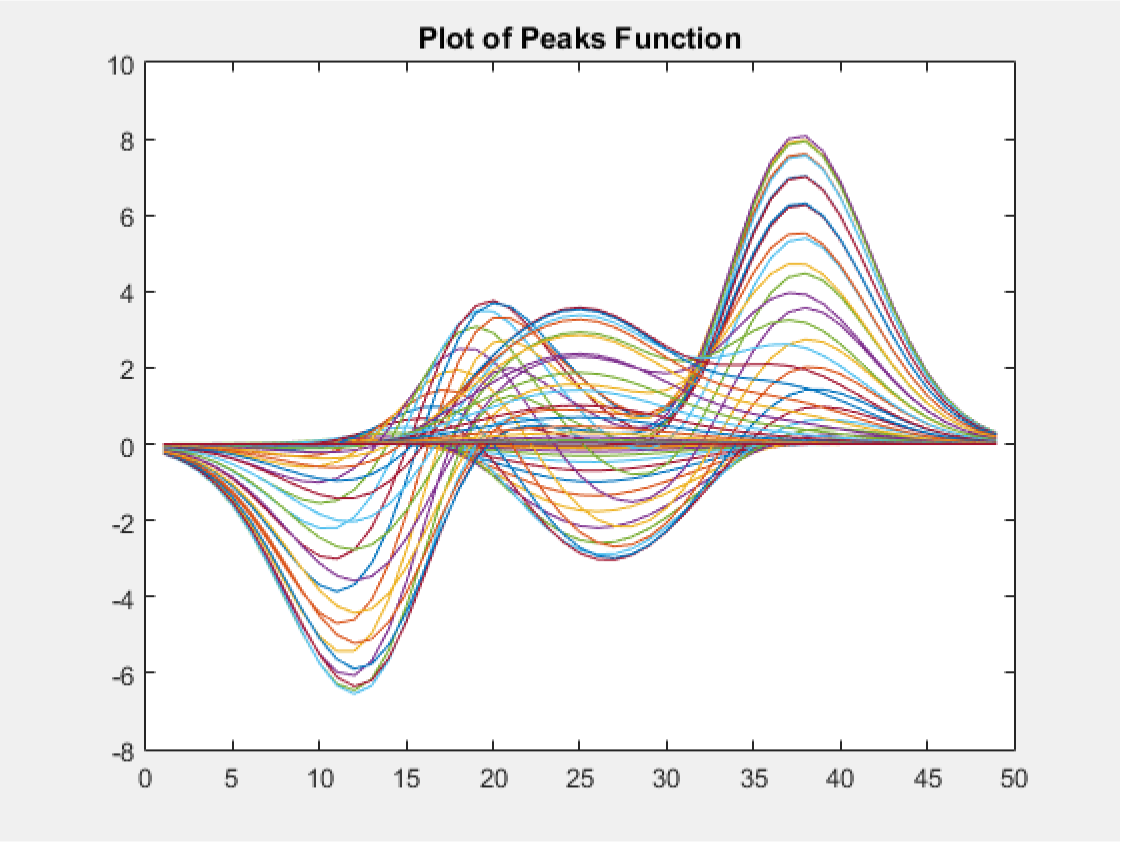
Save Figure with Minimal White Space

This example shows how to save a figure so that the saved figure has a tight margin of white space around the axes.

Create Plot to Save

Create a plot to save and add a title.

```
plot(peaks)
title('Plot of Peaks Function')
```

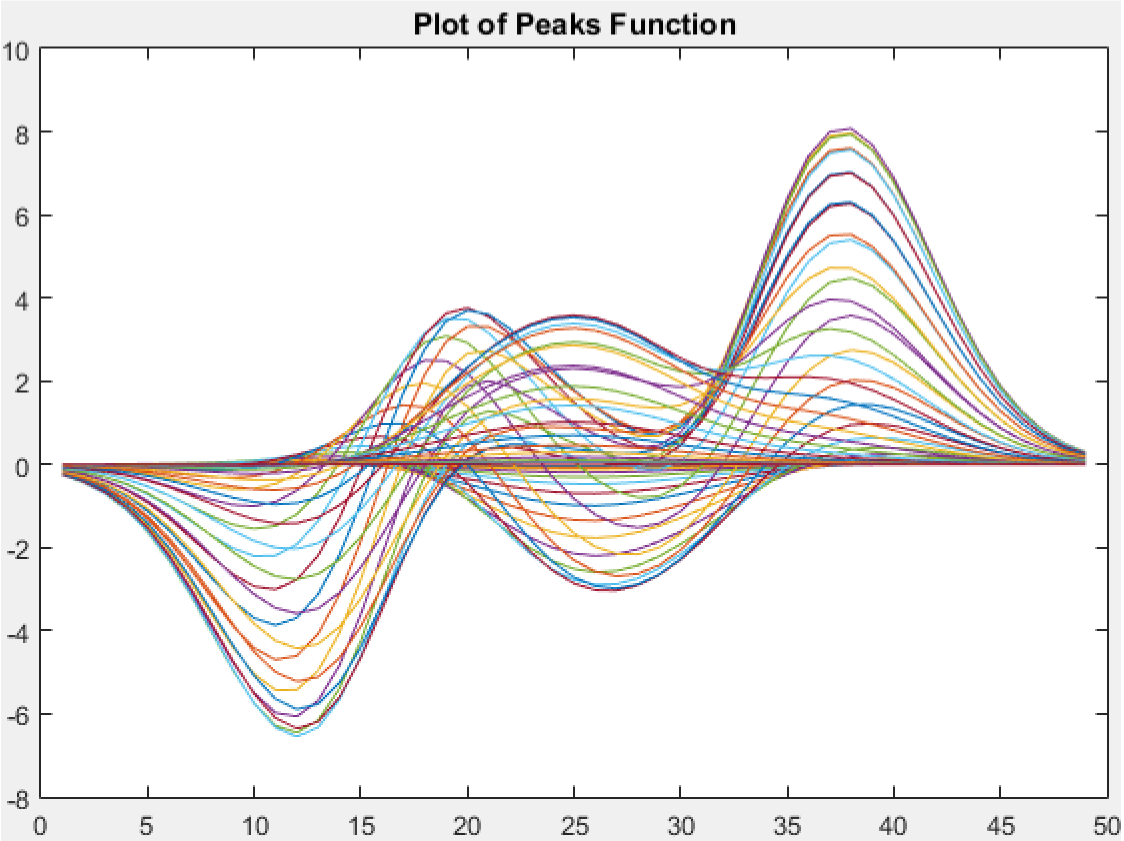


Expand Axes to Fill Figure

Expand the axes size so that it fills the maximum available space in the figure. Get the dimensions of the maximum available space from the `OuterPosition` property of the axes. Account for the space needed for the tick values and text labels using the margin values stored in the `TightInset` property.

Note: Starting in R2014b, you can use dot notation to query and set properties. If you are using an earlier release, use the `get` and `set` functions instead.

```
ax = gca;
outerpos = ax.OuterPosition;
ti = ax.TightInset;
left = outerpos(1) + ti(1);
bottom = outerpos(2) + ti(2);
ax_width = outerpos(3) - ti(1) - ti(3);
ax_height = outerpos(4) - ti(2) - ti(4);
ax.Position = [left bottom ax_width ax_height];
```



Note: If you have multiple subplot axes in your figure, then expand the size of each subplot. The space allocated for subplots does not typically extend to the figure edges.

Alternatively, you can interactively expand the size of the axes to fill the figure using the **File** > **Export Setup** dialog. In the **Properties** section, select **Size**, and then select **Expand axes to fill figure**. This option only affects axes with an `ActivePositionProperty` property set to `'outerposition'`. By default, it does not affect subplots since subplots have an `ActivePositionProperty` set to `'position'`.

Specify Figure Size and Page Size

Set the page size equal to the figure size to ensure that there is no extra whitespace. This step is necessary only if you are saving to a PDF or PostScript file format. If you are saving to an image file format, this step is not necessary. Image formats automatically use a page size that fits tightly around the saved figure.

```
fig = gcf;
fig.PaperPositionMode = 'auto'
fig_pos = fig.PaperPosition;
fig.PaperSize = [fig_pos(3) fig_pos(4)];
```

Save Figure to File Format

Save the figure to a file.

```
print(fig,'MySavedFile','-dpdf')
```

See Also

`print` | `saveas`

Related Examples

- "Save Figure to Open in Another Application" on page 8-19
- "Save Figure at Specific Size and Resolution" on page 8-27
- "Customize Figure Interactively Before Saving" on page 8-11

9

Axes Active Position

Control Axes Layout

In this section...
"Axes Position-Related Properties" on page 9-2
"Position and Margin Boundaries" on page 9-2
"Controlling Automatic Resize Behavior" on page 9-4
"Stretch-to-Fill Behavior" on page 9-5

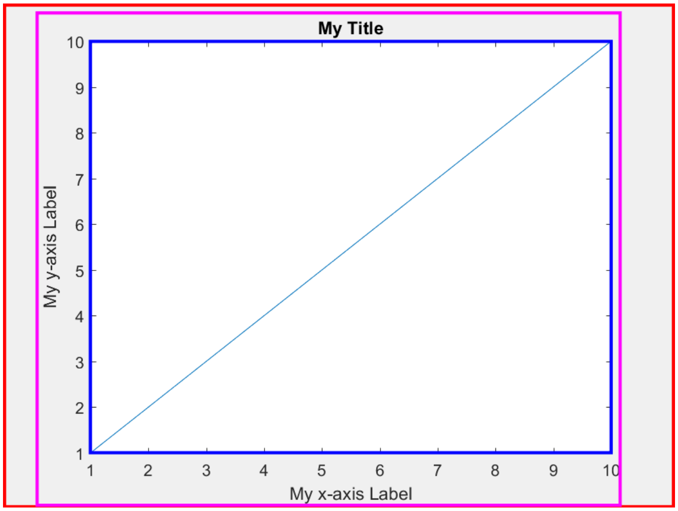
Axes Position-Related Properties

The `Axes` object has several properties that control the axes size and the layout of titles and axis labels within a figure.

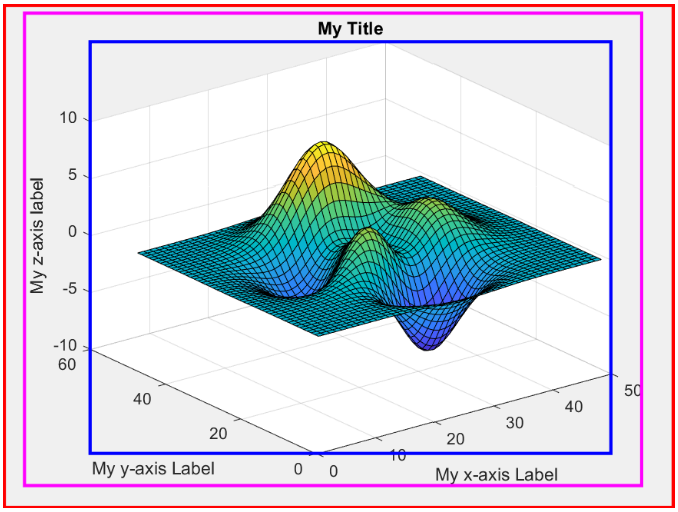
- OuterPosition — Outer boundary of the axes, including the title, labels, and a margin. Specify this property as a vector of the form `[left bottom width height]`. The `left` and `bottom` values indicate the distance from the lower left corner of the figure to the lower left corner of the outer boundary. The `width` and `height` values indicate the outer boundary dimensions.
- Position — Boundary of the inner axes where plots appear, excluding the title, labels, and a margin. Specify this property as a vector of the form `[left bottom width height]`.
- TightInset — Margins added to the width and height of the `Position` property values, specified as a vector of the form `[left bottom right top]`. This property is read-only. When you add axis labels and a title, MATLAB updates the values to accommodate the text. The size of the boundary defined by the `Position` and `TightInset` properties includes all graph text.
- ActivePositionProperty — Position property preserved when the `Axes` object changes size, specified as either `'outerposition'` (the default) or `'position'`.
- Units — Position units. The units must be set to `'normalized'` (the default) to enable automatic axes resizing. When the position units are a unit of length, such as inches or centimeters, then the `Axes` object is a fixed size.

Position and Margin Boundaries

This figure shows a 2-D view of the axes areas defined by the `OuterPosition` values (red), the `Position` values (blue), and the `Position` expanded by the `TightInset` values (magenta).



This figure shows a 3-D view of the axes areas defined by the `OuterPosition` values (red), the `Position` values (blue), and the `Position` expanded by the `TightInset` values (magenta).



Controlling Automatic Resize Behavior

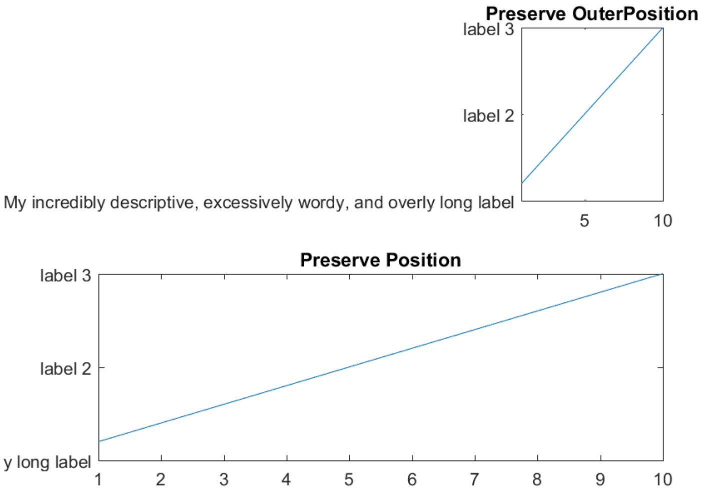
Some scenerios can trigger the `Axes` object to automatically resize. For example, interactively resizing the figure or adding a title or axis labels activates automatic resizing. Sometimes, the new axes size cannot satisfy both the `Position` and `OuterPosition` values, so the `ActivePositionProperty` indicates which values to preserve. Specify the `ActivePositionProperty` as one of these values:

- `'outerposition'` — Preserve the `OuterPosition` value. Use this option when you do not want the axes or any of the surrounding text to extend beyond a certain outer boundary. MATLAB adjusts the size of the inner area of the axes (where plots appear) to try to fit the contents within the outer boundary.
- `'position'` — Preserve the `Position` value. Use this option when you want the inner area of the axes to remain a certain size within the figure. This option sometimes causes text to run off the figure.

Usually, leaving the `ActivePositionProperty` value set to `'outerposition'` is preferable. However, an overly long axes title or labels can shrink the inner area of your axes to a size that is hard to read. In such a case, keeping the inner axes to a specific size can be preferable, even if the surrounding text runs off the figure.

For example, create a figure with two subplots. Set the `ActivePositionProperty` value to `'outerposition'` for the upper subplot and to `'position'` for the lower subplot. Notice that in the upper subplot, the inner area of the axes shrinks to accommodate the text, but the text does not run outside the figure. In the lower subplot, the size of the inner area of the axes is preserved, but some of the text is cut off.

```
figure;
ax1 = subplot(2,1,1);
ax1.ActivePositionProperty = 'outerposition';
plot(ax1,1:10)
title(ax1,'Preserve OuterPosition')
yticklabels(ax1,{'My incredibly descriptive, excessively wordy, and overly long label'
    'label 2','label 3'})

ax2 = subplot(2,1,2);
ax2.ActivePositionProperty = 'position';
plot(ax2,1:10)
title(ax2,'Preserve Position')
yticklabels(ax2,{'My incredibly descriptive, excessively wordy, and overly long label'
    'label 2','label 3'})
```



Preserve OuterPosition

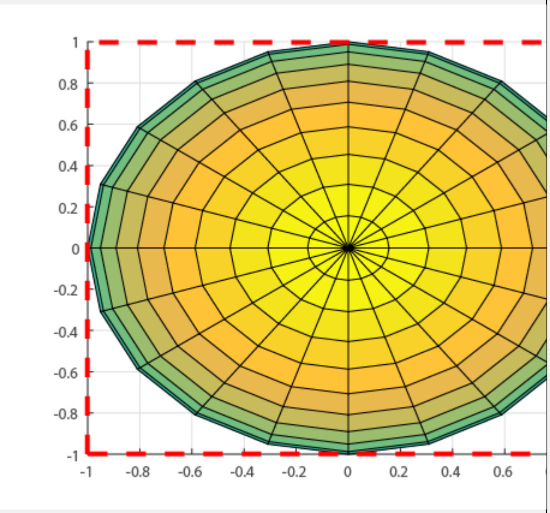
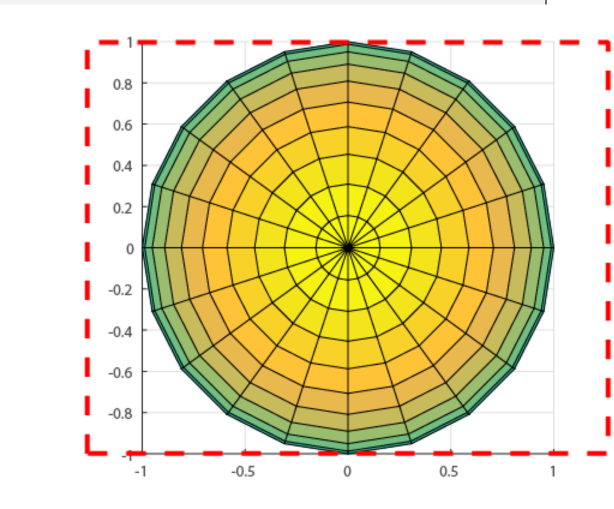
Preserve Position

Stretch-to-Fill Behavior

By default, MATLAB stretches the axes to fill the available space. This “stretch-to-fill” behavior can cause some distortion. The axes might not exactly match the data aspect ratio, plot box aspect ratio, and camera-view angle values stored in the `DataAspectRatio`, `PlotBoxAspectRatio`, and `CameraViewAngle` properties. The “stretch-to-fill” behavior is enabled when the `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` properties of the `Axes` object are set to `'auto'`.

If you specify the data aspect ratio, plot box aspect ratio, or camera-view angle, then the “stretch-to-fill” behavior is disabled. When the “stretch-to-fill” behavior is disabled, MATLAB makes the axes as large as possible within the available space and strictly adheres to the property values so that there is no distortion.

For example, this figure shows the same plot with and without the “stretch-to-fill” behavior enabled. The dotted line shows the available space as defined by the `Position` property. In both versions, the data aspect ratio, plot box aspect ratio, and camera-view angle values are the same. However, in the left plot, the stretching introduces some distortion.

Stretch-to-fill enabled (some distortion)	Stretch-to-fill disabled (no distortion)
	

See Also

Functions

`axes` | `daspect` | `pbaspect` | `subplot` | `title`

Properties

Axes Properties

Related Examples

- "Save Figure with Minimal White Space" on page 8-40

Controlling Graphics Output

- "Control Graph Display" on page 10-2
- "Prepare Figures and Axes for Graphs" on page 10-5
- "Use newplot to Control Plotting" on page 10-9
- "Responding to Hold State" on page 10-12
- "Prevent Access to Figures and Axes" on page 10-14

Control Graph Display

In this section...
"What You Can Control" on page 10-2
"Targeting Specific Figures and Axes" on page 10-2

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 10-5
"How the NextPlot Properties Control Behavior" on page 10-5
"Control Behavior of User-Written Plotting Functions" on page 10-7

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 `clf`.	Removes all axes child objects whose handles are not hidden before adding new graphics objects. Does not reset axes properties. Equivalent to `cla`.
`replace`	Removes all axes objects and resets figure properties to their defaults before adding new objects. Equivalent to `clf reset`.	Removes all child objects and resets axes properties to their defaults before adding new objects. Equivalent to `cla reset`. (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`).
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.
add	Makes the figure the current figure.
new	Creates a new figure and makes it the current figure.
replacechildren	Deletes the figure's children (axes objects and their descendants) and makes this figure the current figure.
replace	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 NextPlot Property Value	newplot Function
No axes in current figure	Creates an axes and makes it the current axes
`add`	Makes the axes the current axes and returns its handle.
`replacechildren`	Deletes the axes' children and makes this axes the current axes.
`replace`	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:

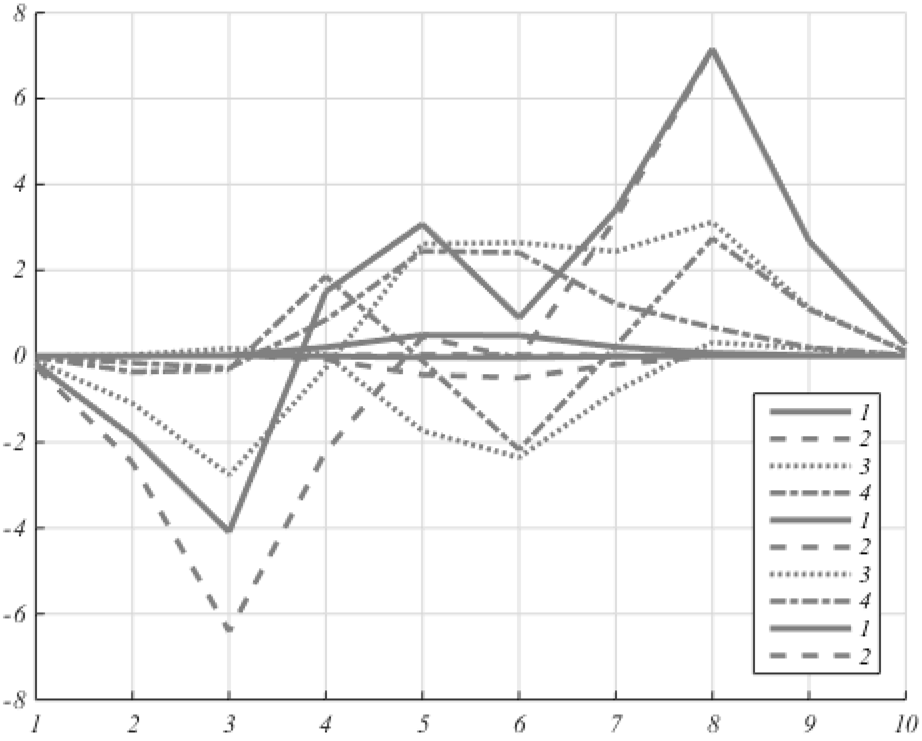
- Customizes the axes and figure appearance for a particular publication requirement.
- Uses revolving line styles and a single color for multiline graphs.
- Adds a legend with specified display names.

Note: Starting in R2014b, you can set properties using dot notation. If you are using an earlier release, use the `set` function instead, such as `set(cax,'FontName','Times')`.

```
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.

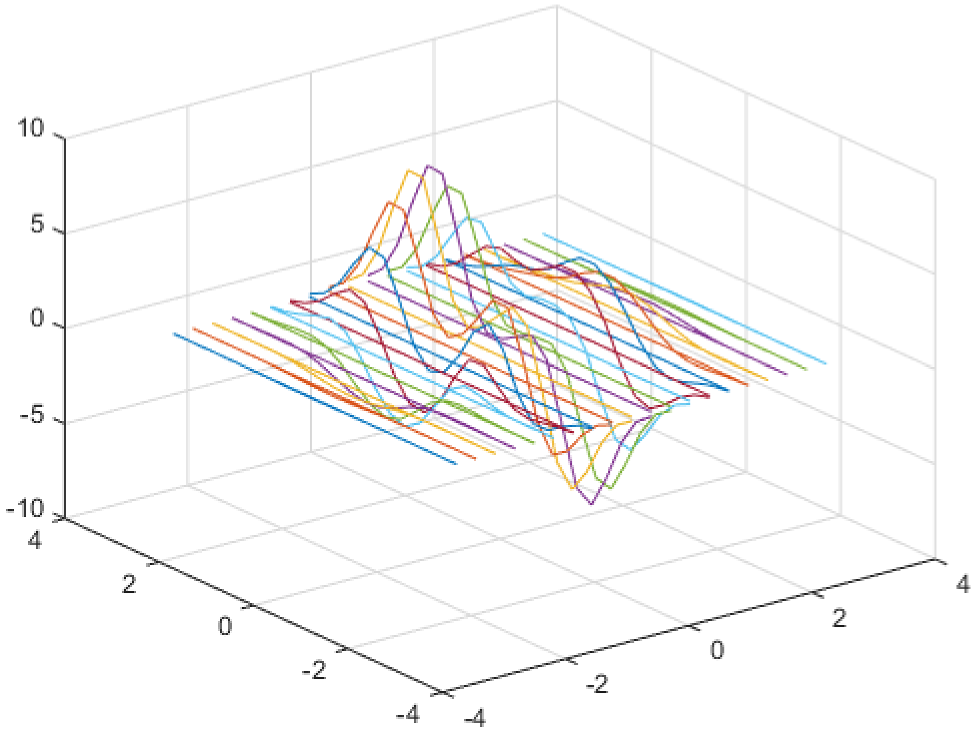
Plotting functions usually change various axes parameters to accommodate different data. The `myPlot3D` function:

- 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)
```


10
5
0
-5
-10
4
2
0
-2
-4
-4
-2
0
2
4

Prevent Access to Figures and Axes

In this section...

“Why Prevent Access” on page 10-14

“How to Prevent Access” on page 10-14

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 11-2
- "Default Values for Automatically Calculated Properties" on page 11-6
- "How MATLAB Finds Default Values" on page 11-8
- "Factory-Defined Property Values" on page 11-9
- "Define Default Line Styles" on page 11-10
- "Multilevel Default Values" on page 11-12

Default Property Values

In this section...

"Predefined Values for Properties" on page 11-2

"Specify Default Values" on page 11-2

"Where in Hierarchy to Define Default" on page 11-3

"List Default Values" on page 11-3

"Set Properties to the Current Default" on page 11-4

"Remove Default Values" on page 11-4

"Set Properties to Factory-Defined Values" on page 11-4

"List Factory-Defined Property Values" on page 11-4

"Reserved Words" on page 11-5

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 character vector with these three parts:

```
'default' ObjectType PropertyName
```

- The word `default`
- The object type (for example, `Line`)

- The property name (for example, `LineWidth`)

A character vector that specified the default line `LineWidth` would be:

```
'defaultLineLineWidth'
```

Use this character vector 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 character vector `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,'factory`*`ObjectType`*`')` — List all factory-defined property values for a specific object
- `get(groot,'factory`*`ObjectTypePropertyName`*`')` — 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 11-6
"Default Values for Automatically Calculated Properties" on page 11-6

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`).

Default Values for Automatically Calculated Properties

Defining a default value for an automatically calculated property requires two steps:

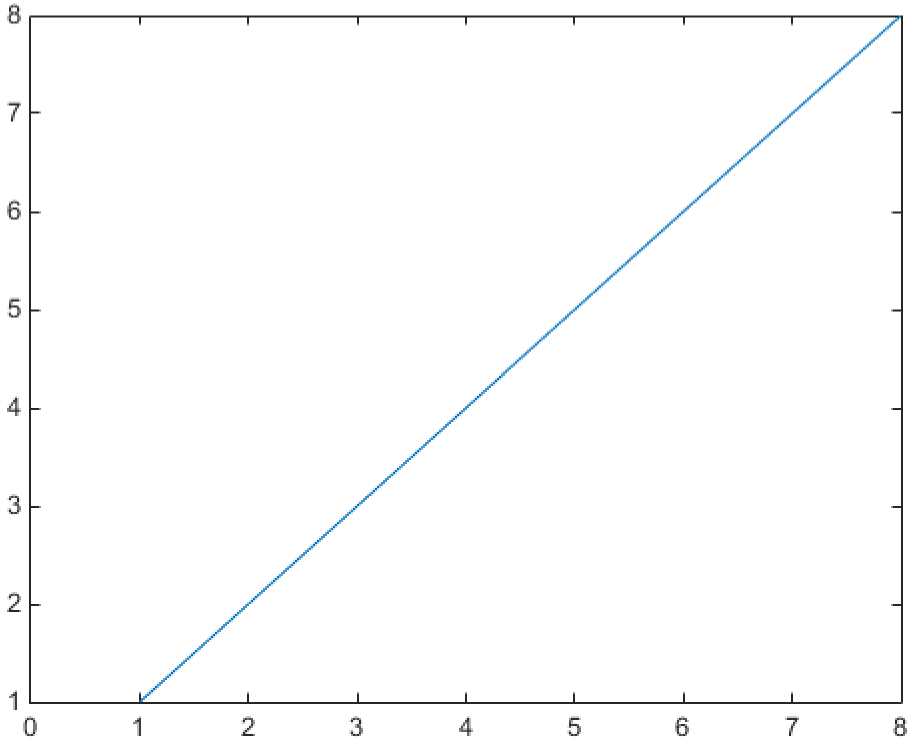
- Define the property default value
- Define the default value of the mode property as `manual`

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]`:


8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
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.

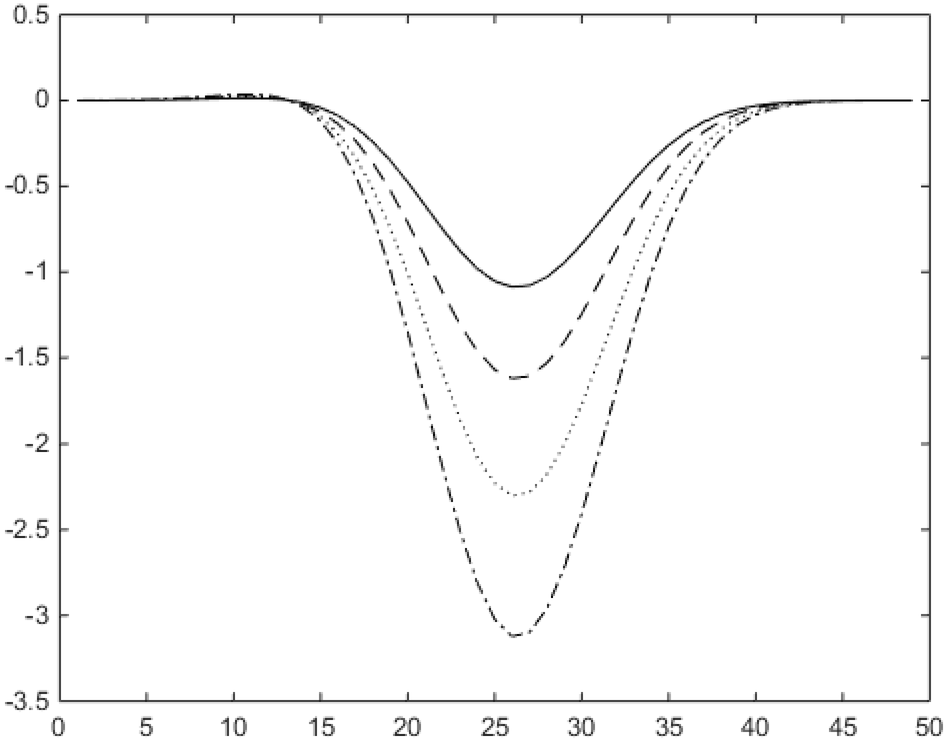
The `plot` function cycles through the colors defined by the axes ColorOrder property when displaying multiline plots. If you define more than one value for the axes LineStyleOrder property, `plot` increments the line style after each cycle through the colors.

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`.

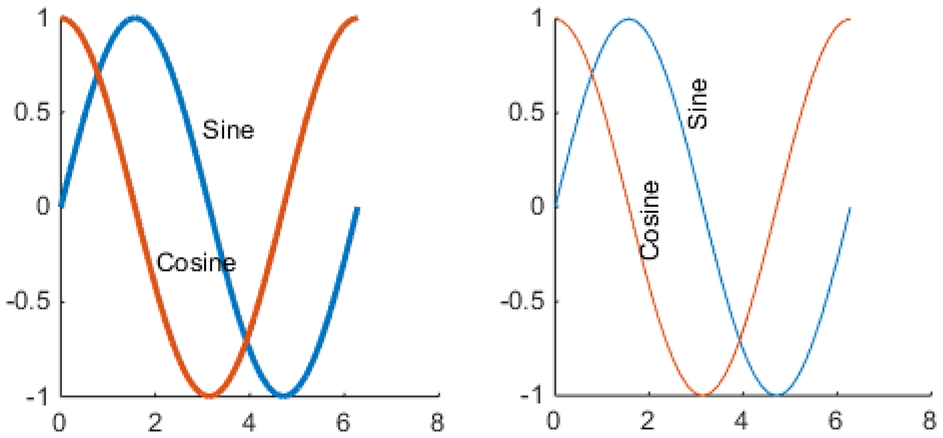
```
Z = peaks;
x = 1:length(Z);
y = Z(4:7,:);
plot(x,y)
```


0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
-3.5
0
5
10
15
20
25
30
35
40
45
50

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.

12

Graphics Object Callbacks

- "Callbacks — Programmed Response to User Action" on page 12-2
- "Callback Definition" on page 12-4
- "Button Down Callback Function" on page 12-7
- "Define a Context Menu" on page 12-9
- "Define an Object Creation Callback" on page 12-11
- "Define an Object Deletion Callback" on page 12-13
- "Capturing Mouse Clicks" on page 12-14
- "Pass Mouse Click to Group Parent" on page 12-18
- "Pass Mouse Click to Obscured Object" on page 12-21

Callbacks — Programmed Response to User Action

In this section...
"What Are Callbacks?" on page 12-2
"Window Callbacks" on page 12-2

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 12-6.

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 12-4

“Callback Function Syntax” on page 12-4

“Related Information” on page 12-5

“Define a Callback as a Default” on page 12-6

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 character vector that evaluates to a valid MATLAB expression. MATLAB evaluates the character vector in the base workspace.

Defining a callback as a character vector 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 12-4.

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,'ButtonDownFcn',@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,~)
   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,'ButtonDownFcn',{@lineCallback,'--','*'})
```

You can use an anonymous function to pass additional arguments. For example:

```
plot(x,y,'ButtonDownFcn',...
    @(src,eventdata)lineCallback(src,eventdata,'--','*'))
```

Related Information

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.

Button Down Callback Function

In this section...

"When to Use a Button Down Callback" on page 12-7

"How to Define a Button Down Callback" on page 12-7

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 12-9

"How to Define a Context Menu" on page 12-9

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 defining callback functions, see “Callback Definition” on page 12-4

Define an Object Deletion Callback

You can create an object deletion callback that executes code when you delete the object.

For example, create an object deletion callback for a figure so that when you delete the figure a dialog appears asking if you want to delete all the figures. Copy the following code to a new function file and save it as `figDelete.m` either in the current folder or in a folder on the MATLAB search path.

```
function figDelete(~,~)
yn = questdlg('Delete all figures?',...
    'Figure Menu',...
    'Yes','No','No');
switch yn
    case 'Yes'
        allfigs = findobj(get(groot,'Children'),'Type','figure' );
        set(allfigs,'DeleteFcn',[]);
        delete(allfigs)
    case 'No'
        return
end
end
```

Then create two figures and assign the `figDelete` function to the DeleteFcn properties. Delete one of the figures and choose an option on the dialog that appears.

```
figure('DeleteFcn',@figDelete)
figure('DeleteFcn',@figDelete)
```

Capturing Mouse Clicks

In this section...

"Properties That Control Response to Mouse Clicks" on page 12-14

"Combinations of PickablePart/HitTest Values" on page 12-14

"Passing Mouse Click Up the Hierarchy" on page 12-15

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.

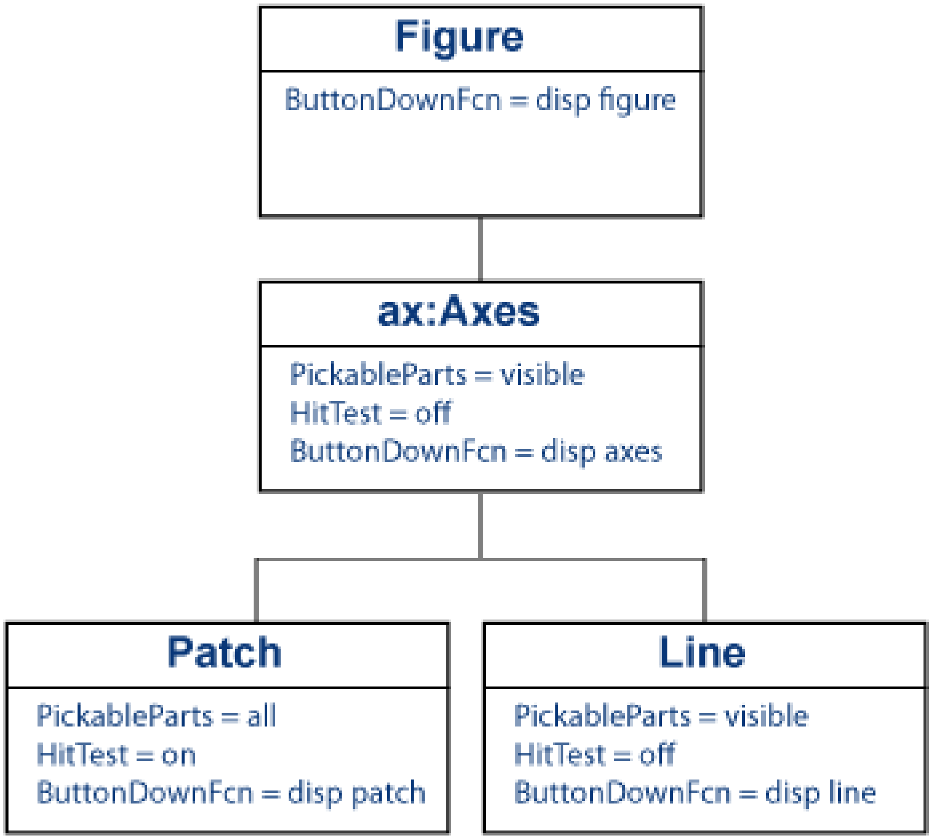
This table summarizes the response to a mouse click based on property values.

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 12-18

"Object Hierarchy and Key Properties" on page 12-18

"MATLAB Code" on page 12-19

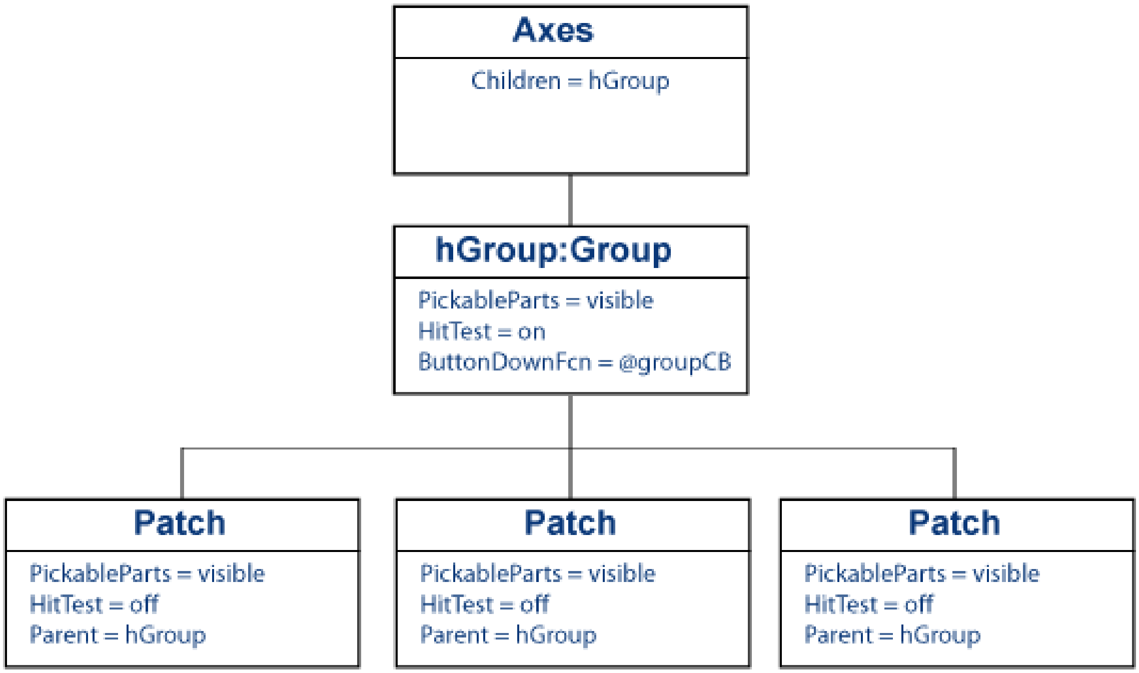
Objective and Design

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 12-4

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
```

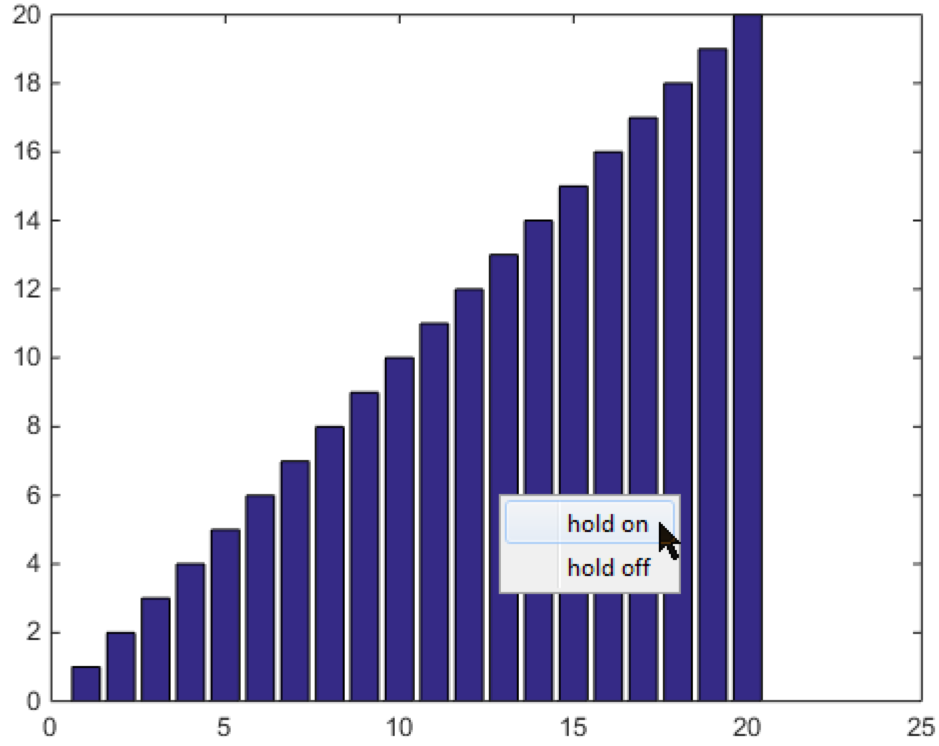
Create a bar graph and set the PickableParts property of the Bar objects:

```
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:


20
18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
hold on
hold off

13

Graphics Objects

- "Graphics Objects" on page 13-2
- "Features Controlled by Graphics Objects" on page 13-7

Graphics Objects

In this section...

"MATLAB Graphics Objects" on page 13-2
"Graphs Are Composed of Specific Objects" on page 13-2
"Organization of Graphics Objects" on page 13-2

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.

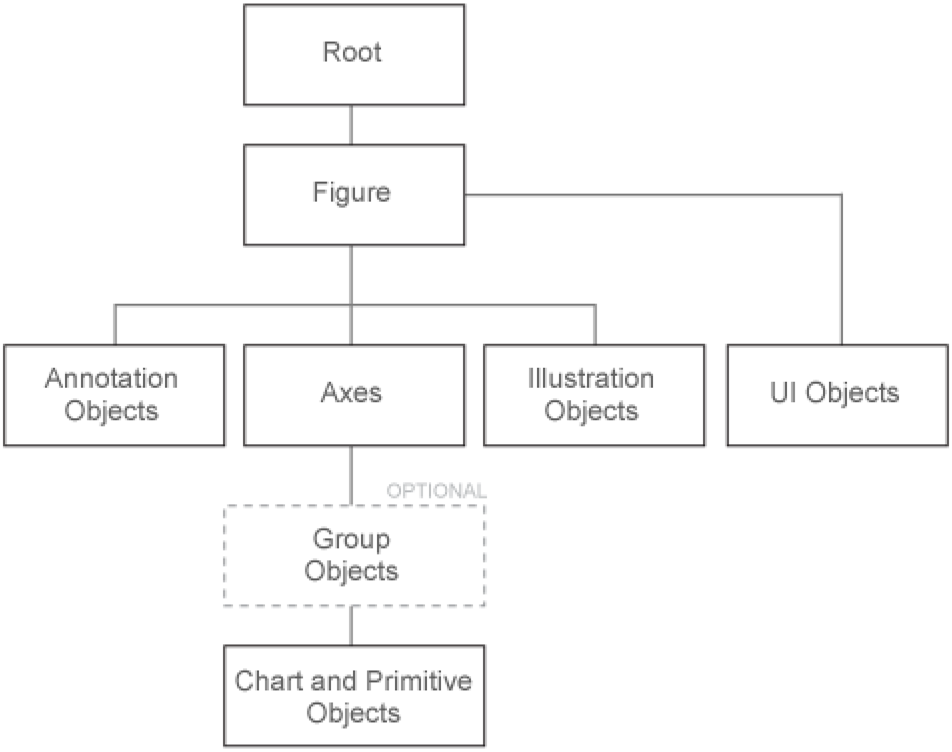
Each object has a unique identifier called a *handle*. Using this handle, you can manipulate the characteristics of an existing graphics object by setting object *properties*. You can also specify values for properties when you create a graphics object. Typically, you create graphics objects using plotting 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.

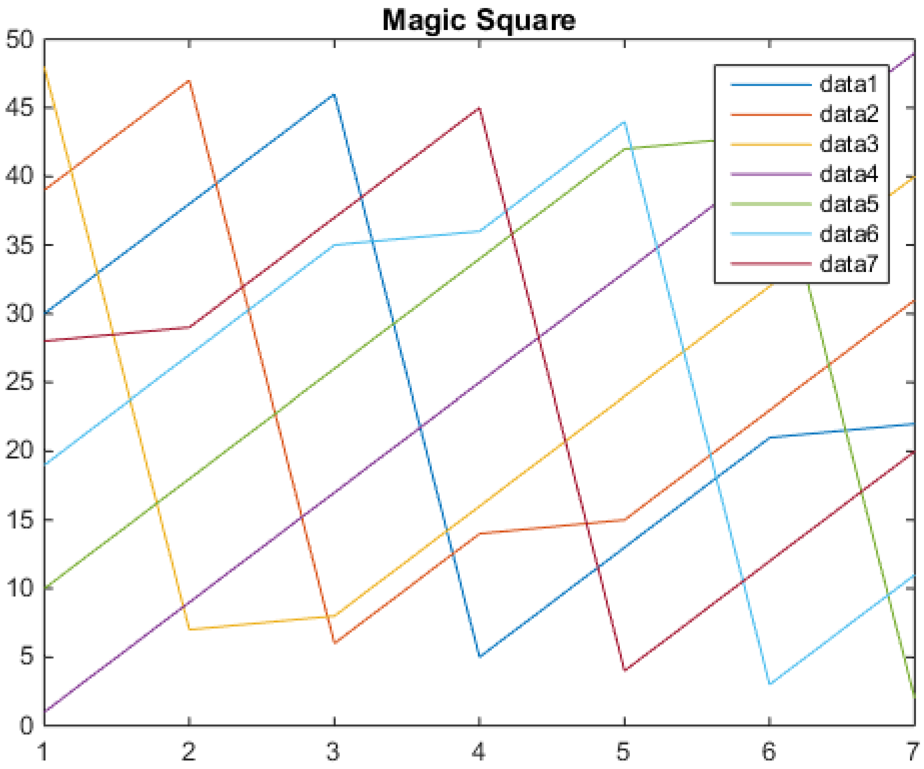


The hierarchical nature of graphics objects reflects the containment of objects by other objects. Each object plays a specific role in the graphics display.

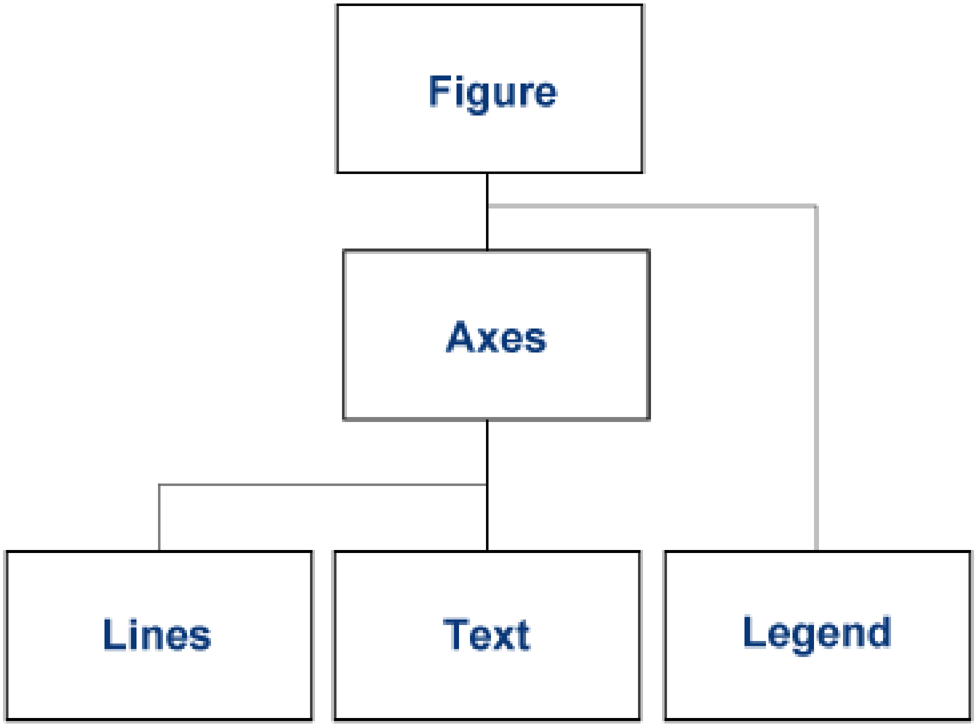
For example, suppose you create a line graph with the `plot` function. An axes object defines a frame of reference for the lines that represent data. A figure is the window to display the graph. The figure contains the axes and the axes contains the lines, text, legends, and other objects used to represent 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 13-7

"Figures" on page 13-7

"Axes" on page 13-8

"Objects That Represent Data" on page 13-9

"Group Objects" on page 13-10

"Annotation Objects" on page 13-11

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:

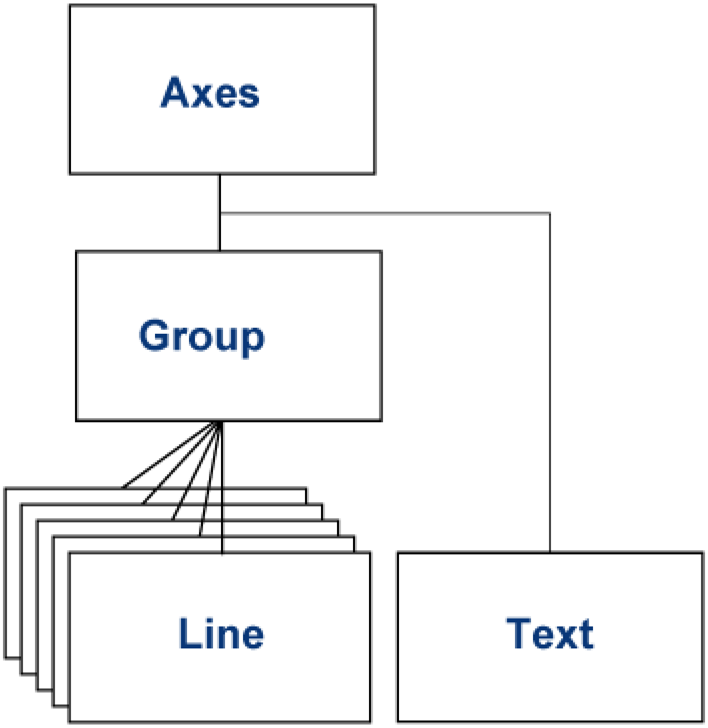
- High-level functions — Create complete graphs that replace existing graphs with new ones. High-level functions include `plot`, `bar`, `scatter`, and so on. For a summary of high-level functions, see "Types of MATLAB Plots" on page 1-2.
- Low-level functions — Add graphics objects with minimal changes 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 14-2
- "Create Object Groups" on page 14-3
- "Transforms Supported by hgtransform" on page 14-5
- "Rotate About an Arbitrary Axis" on page 14-10
- "Nest Transforms for Complex Movements" on page 14-14

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 14-14. 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 14-4

“Visible and Selected Properties of Group Children” on page 14-4

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 14-10 and “Nest Transforms for Complex Movements” on page 14-14.

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 14-5

“Rotation” on page 14-5

“Translation” on page 14-6

“Scaling” on page 14-6

“The Default Transform” on page 14-7

“Disallowed Transforms: Perspective” on page 14-7

“Disallowed Transforms: Shear” on page 14-7

“Absolute vs. Relative Transforms” on page 14-8

“Combining Transforms into One Matrix” on page 14-8

“Undoing Transform Operations” on page 14-9

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 `makehgtform` function simplifies the construction of matrices to perform rotation, translation, and scaling. For information on creating transform matrices using `makehgtform`, see “Nest Transforms for Complex Movements” on page 14-14.

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 14-9.

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 14-8.

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`.

For more information on the identity matrix, see the `eye` function

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 14-10
"Rotate Surface" on page 14-10

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.

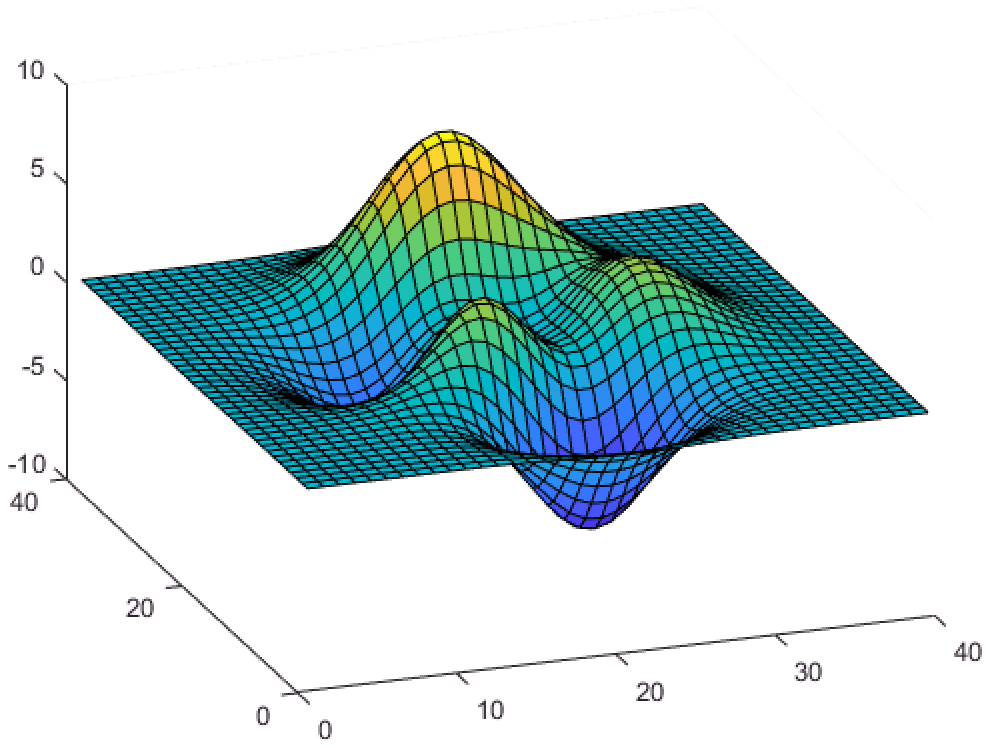
Rotate Surface

This example shows how to rotate a surface about the *y*-axis.

Create Surface and Transform

Parent the surface to the transform object.

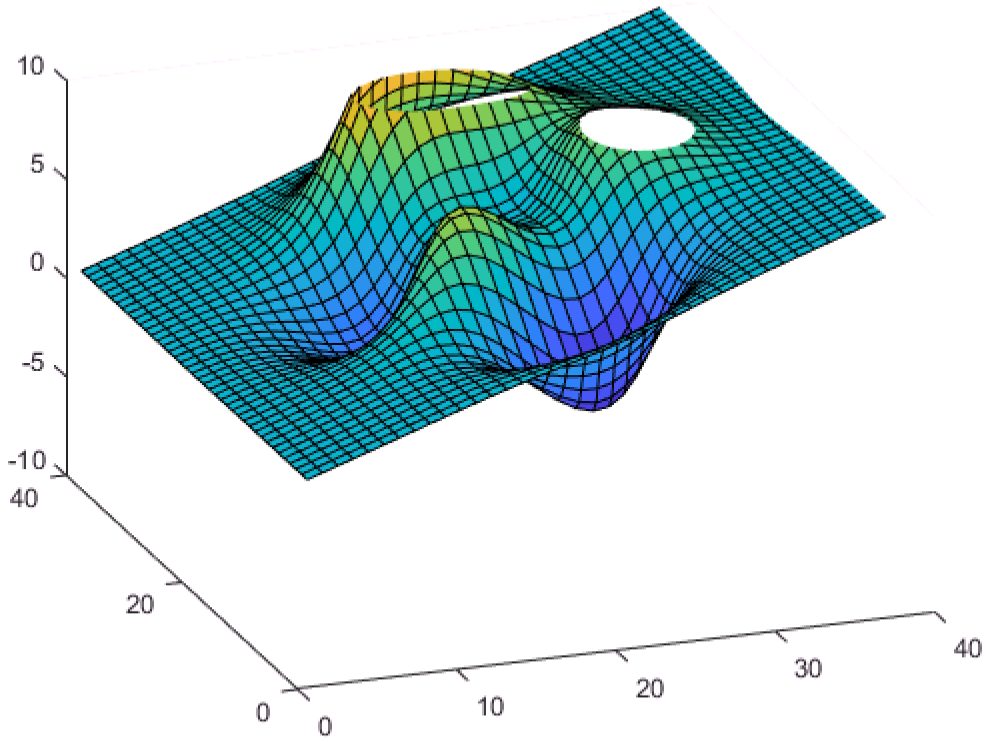
```
t = hgtransform;
surf(peaks(40),'Parent',t)
view(-20,30)
axis manual
```



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;
```



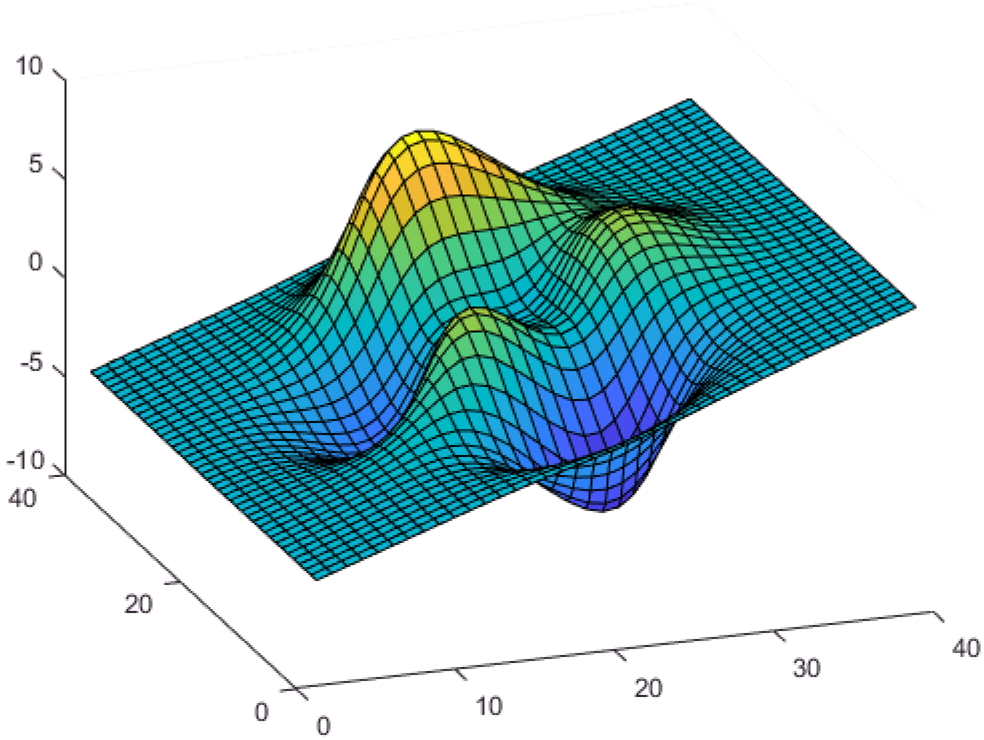
The surface rotated -15 degrees about the *y*-axis that passes through the origin.

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.

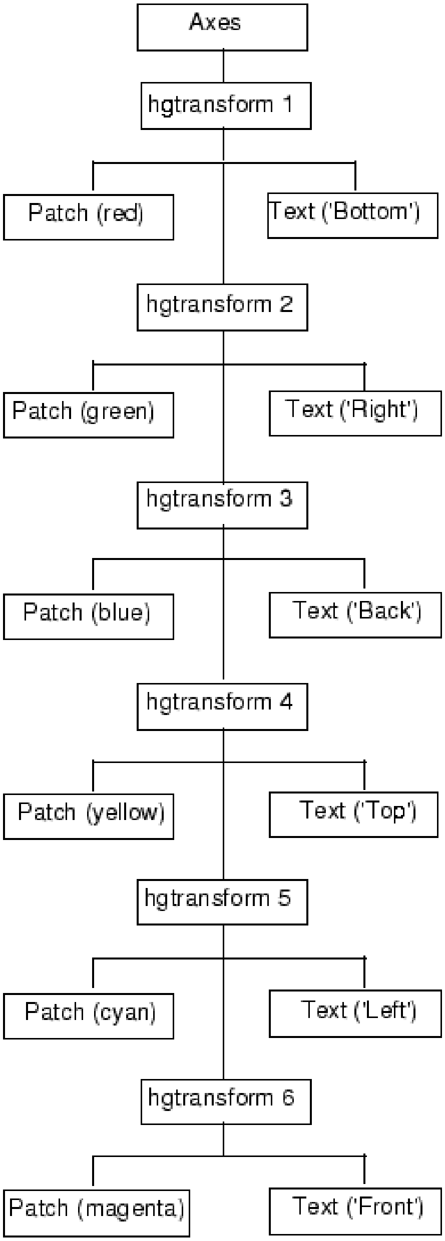
```
Tx1 = makehgtform('translate',[-20 0 0]);
Tx2 = makehgtform('translate',[20 0 0]);
t.Matrix = Tx2*Ry*Tx1;
```


10
5
0
-5
-10
40
20
0
0
10
20
30
40

Nest Transforms for Complex Movements

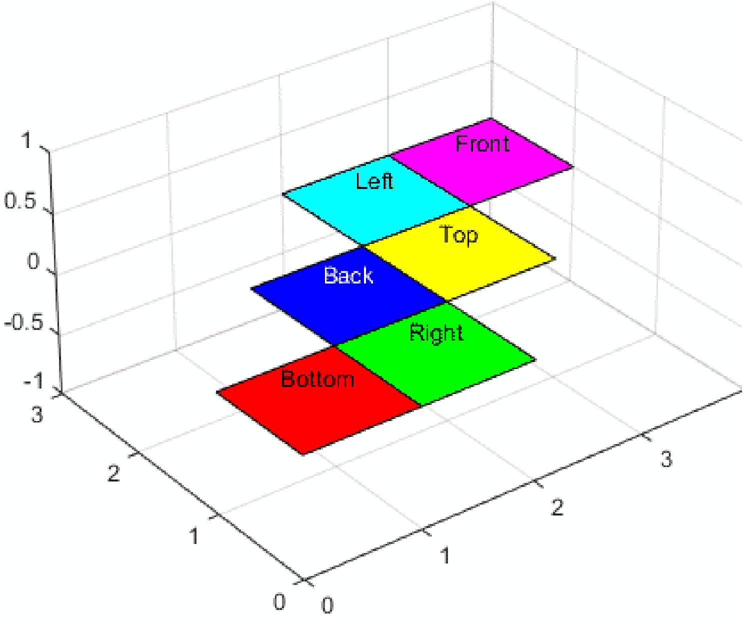
This example creates a nested hierarchy of transform objects, which are then transformed in sequence to create a cube from six squares. The example illustrates how you can parent transform objects to other transform objects to create a hierarchy, and how transforming members of a hierarchy affects subordinate 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 file named `transform_foldbox.m` 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 transform objects
   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
```

15

Control Legend Content

Create Interactive Legends Using Callbacks

In this section...

"Toggle Chart Visibility" on page 15-2
"Toggle Chart Line Width" on page 15-3

You can create interactive legends so that when you click an item in the legend, the associated chart updates in some way. For example, you can toggle the visibility of the chart or change its line width. Set the ItemHitFcn property of the legend to a callback function that controls how the charts change.

Toggle Chart Visibility

This example shows how to toggle the visibility of a chart when you click the chart icon or label in a legend. It creates a callback function that changes the `Visible` property of the chart to either `'on'` or `'off'`.

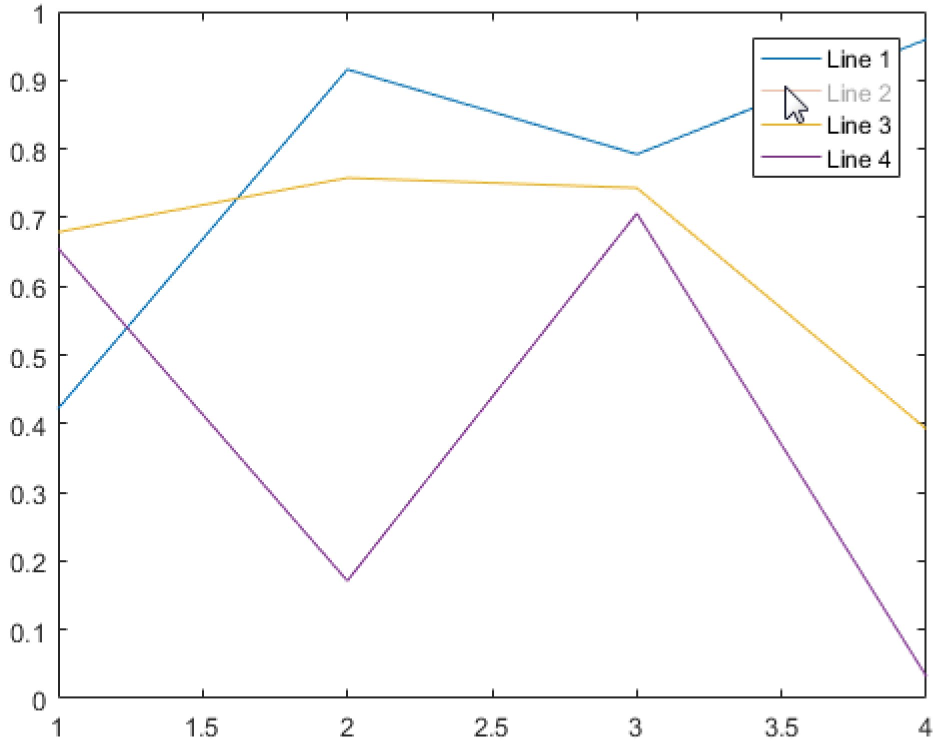
Copy the following code to a new function file and save it as `hitcallback_ex1.m` either in the current folder or in a folder on the MATLAB search path. The two input arguments, `src` and `evnt`, are the legend object and an event data structure. MATLAB automatically passes these inputs to the callback function when you click an item in the legend. Use the `Peer` field of the event data structure to access properties of the chart object associated with the clicked legend item.

```
function hitcallback_ex1(src,evnt)

if strcmp(evnt.Peer.Visible,'on')
    evnt.Peer.Visible = 'off';
else
    evnt.Peer.Visible = 'on';
end

end
```

Then, plot four lines, create a legend, and assign the legend object to a variable. Set the `ItemHitFcn` property of the legend object to the callback function. Click items in the legend to show or hide the associated chart. The legend label changes to gray when you hide a chart.

```
plot(rand(4));
```

```
l = legend('Line 1','Line 2','Line 3','Line 4');
l.ItemHitFcn = @hitcallback_ex1;
```



Toggle Chart Line Width

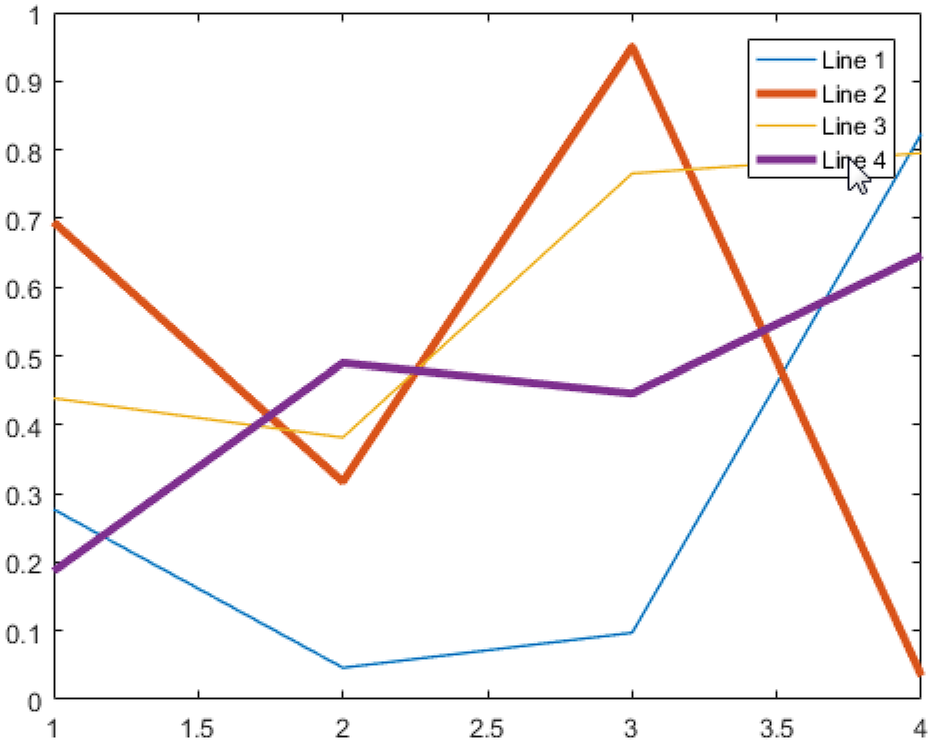
This example shows how to toggle the line width of a chart when you click the chart icon or label in a legend. It creates a callback function that changes the `LineWidth` property of the chart. Copy the following code to a new function file and save it as `hitcallback_ex2.m` either in the current folder or in a folder on the MATLAB search path.

```
function hitcallback_ex2(src,evnt)
```

```
if evnt.Peer.LineWidth == 3
    evnt.Peer.LineWidth = 0.5;
else
    evnt.Peer.LineWidth = 3;
end

end
```

Then, plot four lines, create a legend, and assign the legend object to a variable. Set the `ItemHitFcn` property of the legend object to the callback function. Click items in the legend to toggle the line width of the associated line. The legend icon updates to match the line width.

```
plot(rand(4));
l = legend('Line 1','Line 2','Line 3','Line 4');
l.ItemHitFcn = @hitcallback_ex2;
```



See Also

Legend Properties | `legend`

Related Examples

- "Callback Definition" on page 12-4

Working with Graphics Objects

- "Graphics Object Handles" on page 16-2
- "Preallocate Arrays of Graphics Objects" on page 16-4
- "Test for Valid Handle" on page 16-5
- "Handles in Logical Expressions" on page 16-6
- "Graphics Arrays" on page 16-9

Graphics Object Handles

In this section...

"What You Can Do with Handles" on page 16-2
"What You Cannot Do with Handles" on page 16-3

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 character vectors or use handles in character vector operations.

More About

- “Graphics Arrays” on page 16-9
- “Dominant Argument in Overloaded Graphics Functions”

Preallocate Arrays of Graphics Objects

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 16-6
“If Result Is Empty” on page 16-6
“If Handles Are Equal” on page 16-7

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 to the character vector `'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 =
```

1

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
```

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;
hl = plot(1:10);
harray = [hf,ha,hl]
```

```
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;
hl = plot(rand(5));
harray = [hf,ha,hl];
Error using horzcat
Dimensions of matrices being concatenated are not consistent.
```

To form an array, you must transpose hl:

```
harray = [hf,ha,hl']
```

```
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;
hl = plot(rand(5));
harray(1) = hf;
harray(2) = ha;
for k = 1:length(hl)
   harray(k+2) = hl(k);
end
```

Object Identification

- "Special Object Identifiers" on page 17-2
- "Find Objects" on page 17-5
- "Copy Objects" on page 17-11
- "Delete Graphics Objects" on page 17-14

Special Object Identifiers

In this section...

“Getting Handles to Special Objects” on page 17-2

“The Current Figure, Axes, and Object” on page 17-2

“Callback Object and Callback Figure” on page 17-4

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 objects include:

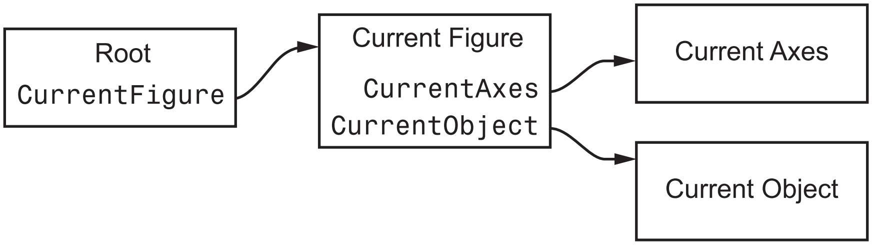
- Current figure — Handle of the figure that is the current target for graphics commands.
- Current axes— Handle of the axes in the current figure that is the target for graphics commands.
- Current object — Handle of the object that is selected
- Callback object — Handle of the object whose callback 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 object to set an axes property:

```
set(gca,'Color','black')
```

You can get the handles of all the graphic objects in the current axes (except hidden handles):

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

and then determine the types of the objects:

```
get(h,'Type')
```

```
ans =
      'text'
      'patch'
      'surface'
      'line'
```

Although `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 know the 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 10-14.

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 Root 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 12-4

Find Objects

In this section...

"Find Objects with Specific Property Values" on page 17-5
"Find Text by String Property" on page 17-5
"Use Regular Expressions with findobj" on page 17-7
"Limit Scope of Search" on page 17-9

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 character vector. You can then search for the specific property/value pair. For example, suppose that 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'
```

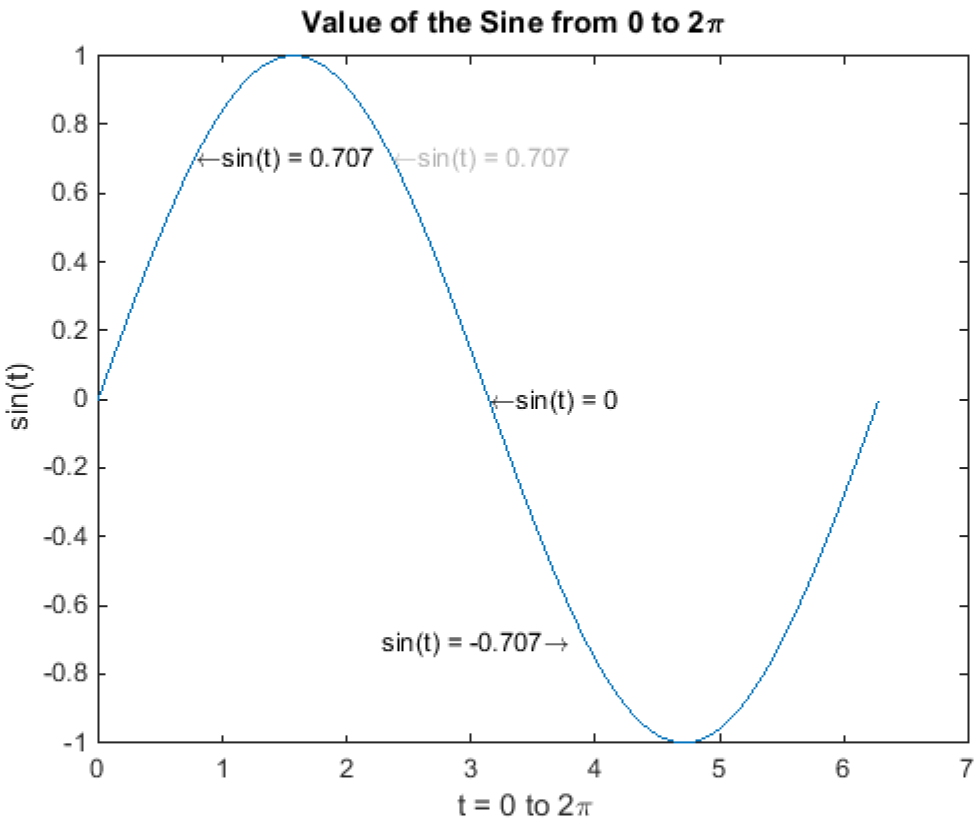
If you do not specify a starting object, `findobj` searches from the root object, finding all occurrences of the property name/property value combination that you specify.

To find objects with hidden handles, use `findall`.

Find Text by String Property

This example shows how to find text objects using the `String` property.

The following graph contains text objects labeling particular values of the function.



Suppose that you want to move the text 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','\leftarrowsin(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 that the text object of interest is in the current axes, so you can type:

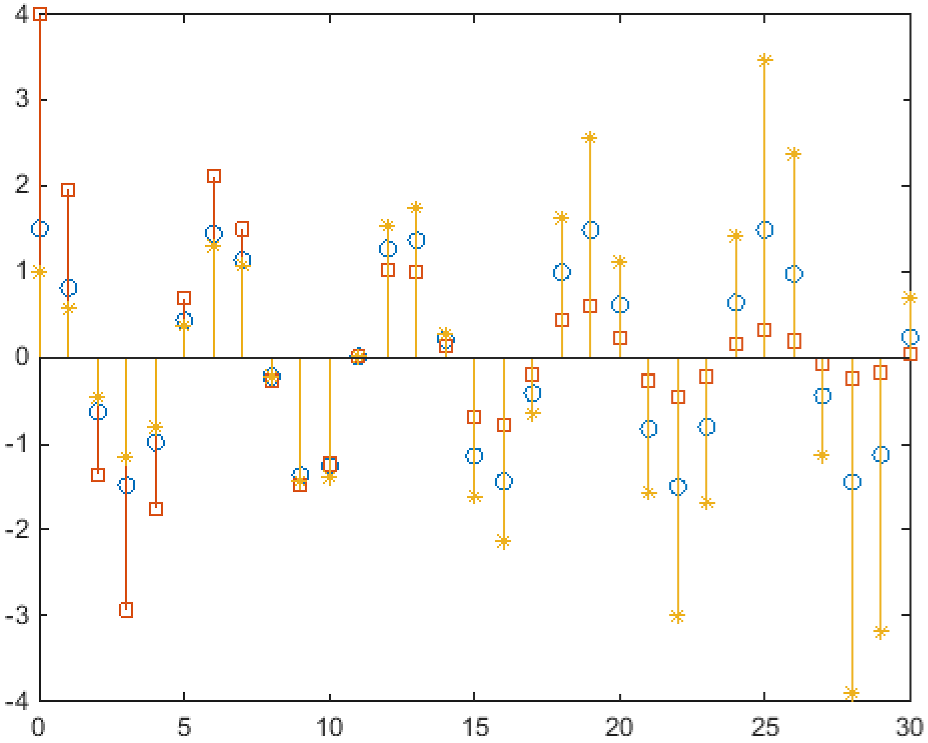
```
hText = findobj(gca,'String','\leftarrowsin(t) = .707');
```

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 that 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 specific patterns. For example, suppose that 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

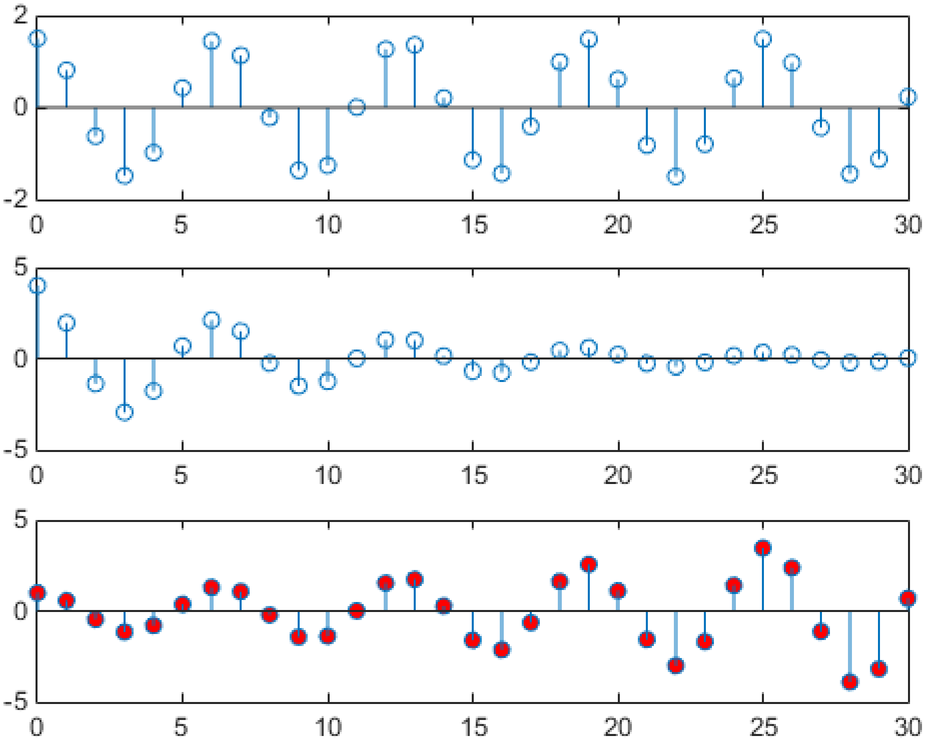
Specify the starting point in the object tree to limit the scope of the search. The starting point can be the handle of a figure, axes, or a group of object handles.

For example, suppose that 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 17-11
"Copy Single Object to Multiple Destinations." on page 17-11
"Copying Multiple Objects" on page 17-12

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 various objects to a new parent, or one object to several 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 that 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, 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 Graphics Objects" on page 17-14
"Handles to Deleted Objects" on page 17-15

How to Delete Graphics Objects

Remove graphics objects with the `delete` function. Pass the object handle as an argument to `delete`. 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`. For example, if `h1`, `h2`, and `h3` are handles to graphics objects that you want to delete, concatenate the handles into a single array.

```
h = [h1,h2,h3];
delete(h)
```

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;
```

Close the figure:

```
close(f)
```

MATLAB deletes each object.

```
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 that contains the object handle. However, the variable becomes an invalid handle because the object it referred to no longer exists.

You can delete graphics objects explicitly using the `delete` function or by closing the figure that contains the graphics objects. For example, create a bar graph.

```
f = figure;
y = rand(1,5);
b = bar(y);
```

Close the figure containing the bar graph.

```
close(f)
```

The handle variables still exist after closing the figure, but the graphics objects no longer exist.

```
whos
```

```
  Name      Size            Bytes  Class

  f         1x1               104  matlab.ui.Figure
  b         1x1               104  matlab.graphics.chart.primitive.Bar
  y         1x5                40  double
```

Use `isgraphics` to determine the validity of a graphics object handle.

```
isgraphics(b)
```

```
ans =

     0
```

You cannot access properties with the invalid handle variable.

```
h.FaceColor
```

```
Invalid or deleted object.
```

To remove the variable, use the `clear` function.

```
clear h
```

See Also

`isvalid`

Related Examples

- “Find Objects” on page 17-5

Optimize Performance of Graphics Programs

- “Finding Code Bottlenecks” on page 18-2
- “What Affects Code Execution Speed” on page 18-4
- “Judicious Object Creation” on page 18-6
- “Avoid Repeated Searches for Objects” on page 18-8
- “Screen Updates” on page 18-10
- “Getting and Setting Properties” on page 18-12
- “Avoid Updating Static Data” on page 18-15
- “Transforming Objects Efficiently” on page 18-17
- “Use Low-Level Functions for Speed” on page 18-18
- “System Requirements for Graphics” on page 18-19
- “Resolving Low-Level Graphics Issues” on page 18-22

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.

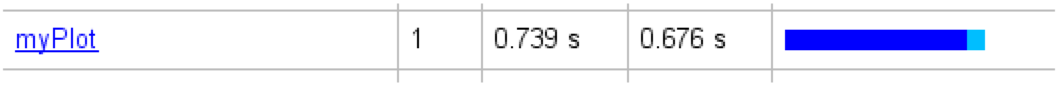
Once you have optimized your code, use the following techniques to reduce the overhead of object creation and updating the display.

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:



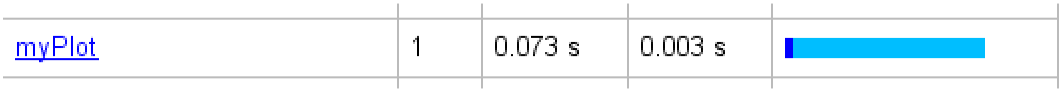
myPlot	1	0.739 s	0.676 s	

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(:),y(:),'LineStyle','none','Marker','o','Color','b');
end
```

```
profile on
myPlot
profile viewer
```

Object creation time is a major factor in this case:



myPlot	1	0.073 s	0.003 s	

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 18-4
"How to Improve Performance" on page 18-4

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 execution time = T creating objects + T updating + (T 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 18-6.
- Updating an object that has some percentage of static data? See "Avoid Updating Static Data" on page 18-15.

- Searching for object handles. See “Avoid Repeated Searches for Objects” on page 18-8.
- Rotating, translating, or scaling objects? See “Transforming Objects Efficiently” on page 18-17.
- Querying and setting properties in the same loop? See “Getting and Setting Properties” on page 18-12.

Judicious Object Creation

In this section...

"Object Overhead" on page 18-6

"Do Not Create Unnecessary Objects" on page 18-6

"Use NaNs to Simulate Multiple Lines" on page 18-7

"Modify Data Instead of Creating New Objects" on page 18-7

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 10000-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.

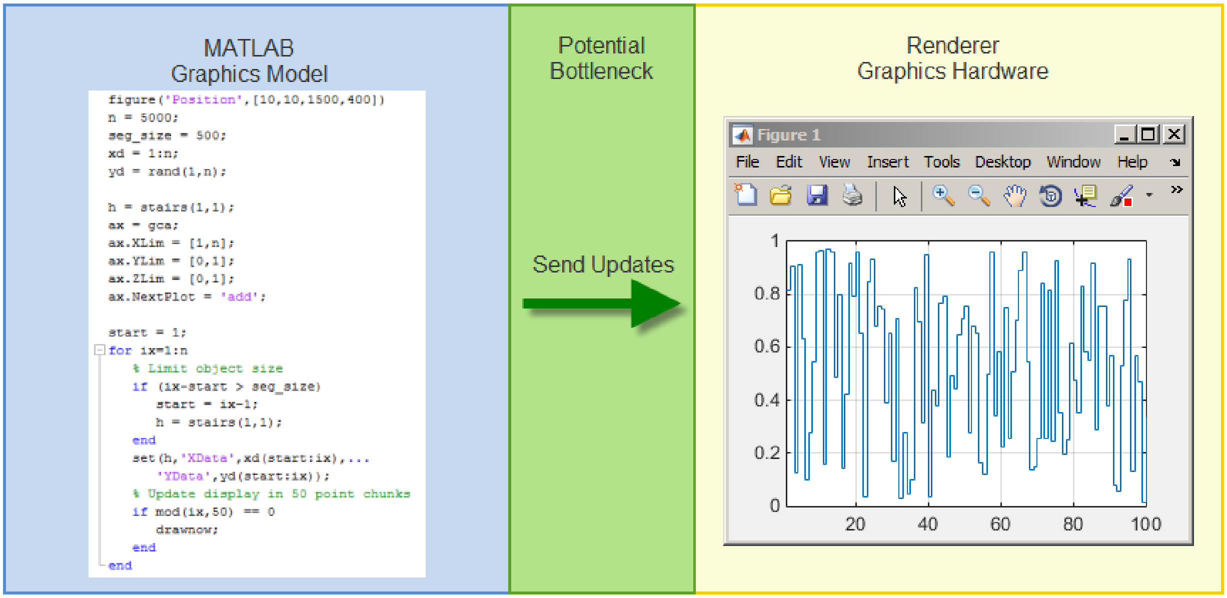
For more information, see "Find Objects" on page 17-5

Screen Updates

In this section...
"MATLAB Graphics System" on page 18-10
"Managing Updates" on page 18-11

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.



```
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];
ax.YLim = [0,1];
ax.ZLim = [0,1];
ax.NextPlot = 'add';

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
```

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 18-12).
- Completes execution and returns control to the MATLAB prompt or debugger.

Getting and Setting Properties

In this section...

"Automatically Calculated Properties" on page 18-12
"Inefficient Cycles of Sets and Gets" on page 18-13
"Changing Text `Extent` to Rotate Labels" on page 18-14

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	`CameraPosition`, `CameraTarget`, `CameraUpVector`, `CameraViewAngle`
	`Position`, `OuterPosition`, `TightInset`
	`XLim`, `YLim`, `ZLim`
	`XTick`, `YTick`, `ZTick`, `XMinorTick`, `YMinorTick`, `ZMinorTick`
	`XTickLabel`, `YTickLabel`, `ZTickLabel`, `TickDir`
	`SortMethod`
Text	`Extent`

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.

<table>
<tr>
<th>Code with Poor Performance</th>
<th>Code with Better Performance</th>
</tr>
<tr>
<td>

```
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
```

</td>
<td>

```
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
```

</td>
</tr>
<tr>
<td>

This code performs poorly because:

- The `Extent` property depends on other values, such as screen resolution, figure size, and axis limits, so querying this property can cause a full update.

</td>
<td>

The performance is better because this code:

- Queries all property values in one loop and stores these values in an array.
- Sets all property values in a separate loop.
- Calls `drawnow` after the second loop finishes.

</td>
</tr>
</table>

Code with Poor Performance	Code with Better Performance
• Each set of the `Position` property makes a full update necessary when the next get of the `Extent` 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.

<table>
<tr><th>Code with Poor Performance</th><th>Code with Better Performance</th></tr>
<tr><td>In this example, a marker moves along the surface by creating both objects with each pass through the loop.
<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></td><td>Create the surface, then update the XData, YData, and ZData of the marker in the loop. Only the marker data changes in each iteration.
<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></td></tr>
</table>

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.

<table>
<tr><th>Code with Poor Performance</th><th>Code with Better Performance</th></tr>
<tr><td><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></td><td><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></td></tr>
</table>

<table>
<tr><th>Code with Poor Performance</th><th>Code with Better Performance</th></tr>
<tr><td>

```
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
```

</td><td>

```
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
```

The performance of this code is better because the limiting factor is the amount of data sent during updates.

</td></tr>
</table>

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` function 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` function sends the transform matrix for the same rotation to the renderer thread.

<table>
<tr>
<th>Code with Poor Performance</th>
<th>Code with Better Performance</th>
</tr>
<tr>
<td>

When object data is large, the update bottleneck becomes a limiting factor.

```
% Using rotate
figure
[x,y,z] = sphere(270);

s = surf(x,y,z,z,'EdgeColor','none');
axis vis3d
for ang = 1:360
   rotate(s,[1,1,1],1)
   drawnow
end
```

</td>
<td>

Using `hgtransform` applies the transform on the renderer side of the bottleneck.

```
% Using hgtransform
figure
ax = axes;
[x,y,z] = sphere(270);

% Transform object contains the surface
grp = hgtransform('Parent',ax);
s = surf(ax,x,y,z,z,'Parent',grp,...
   'EdgeColor','none');

view(3)
grid on
axis vis3d

% Apply the transform
tic
for ang = linspace(0,2*pi,360)
   tm = makehgtform('axisrotate',[1,1,1],ang);
   grp.Matrix = tm;
   drawnow
end
toc
```

</td>
</tr>
</table>

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`

System Requirements for Graphics

In this section...

"Minimum System Requirements" on page 18-19
"Recommended System Requirements" on page 18-19
"Upgrade Your Graphics Drivers" on page 18-19
"Graphics Features with OpenGL Requirements" on page 18-20

Minimum System Requirements

All systems support most of the common MATLAB graphics features.

Recommended System Requirements

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 has OpenGL 2.1 or later. If you have an earlier version of OpenGL, most graphics features still work, but some advanced graphics features are unavailable. For more information, see "Graphics Features with OpenGL Requirements" on page 18-20. For the best performance, OpenGL 3.3 or later is recommended.
- The latest versions of graphics drivers available from your computer manufacturer or graphics hardware vendor.

For more information on determining your graphics hardware, see `opengl`.

Starting in R2015b, MATLAB is a DPI-aware application that takes advantage of your full system resolution. MATLAB graphics look sharp and properly scaled on all systems, including Macintosh systems connected to Apple Retina displays and high-DPI Windows systems.

Upgrade Your Graphics Drivers

Graphics hardware vendors frequently provide updated graphics drivers that improve hardware performance. To help ensure that your graphics hardware works with MATLAB, upgrade your graphics drivers to the latest versions available.

- On Windows systems, check your computer manufacturer website for driver updates, such as Lenovo®, HP®, or Dell®. 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.

Graphics Features with OpenGL Requirements

Most graphics features work on all systems. However, support for some graphics features depends on:

- Whether you are using hardware, basic hardware, or software OpenGL. By default, MATLAB uses hardware OpenGL if your graphics hardware supports it. Basic hardware and software OpenGL are alternate options that you can use to work around low-level graphics issues. In some cases, MATLAB automatically switches to software OpenGL. For more information, see `opengl`.
- The version of the OpenGL implementation, for example, OpenGL 2.1.

This table lists the graphics features with OpenGL requirements. For more information on the features, see `opengl`.

Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows (Uses OpenGL 1.1)	Software OpenGL on Linux (Uses OpenGL 2.1)
Graphics Smoothing	Supported for OpenGL 2.1 or higher	Supported for OpenGL 2.1 or higher	Not supported	Not supported
Depth Peel Transparency	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Supported
Align Vertex Centers	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Not supported

Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows (Uses OpenGL 1.1)	Software OpenGL on Linux (Uses OpenGL 2.1)
Hardware-accelerated markers	Supported for OpenGL 4.0 or higher	Disabled	Not supported	Not supported

See Also

Functions

`opengl`

More About

- “Resolving Low-Level Graphics Issues” on page 18-22

Resolving Low-Level Graphics Issues

MATLAB can encounter low-level issues when creating graphics on your system. For example, bar edges might be missing from bar charts, stems might be missing from stem plots, or your graphics hardware might run out of memory. You can encounter these issues while creating 2-D or 3-D charts, using a Simulink® model that contains scopes, or using UIs from a MathWorks® toolbox. These issues are often due to older graphics hardware or outdated graphics drivers. To resolve them, try the options described here.

In this section...

“Upgrade Your Graphics Hardware Drivers” on page 18-22

“Choose OpenGL Implementation for Your System” on page 18-23

“Fix Out-of-Memory Issues” on page 18-24

“Contact Technical Support” on page 18-24

Upgrade Your Graphics Hardware Drivers

Graphics hardware vendors frequently provide updated graphics drivers that improve hardware performance. To help ensure that your graphics hardware works with MATLAB, upgrade your graphics drivers to the latest versions available.

- On Windows systems, check for driver updates on the website of your manufacturer, such as Lenovo, HP, or Dell. If no updates are provided, then check the website of your graphics hardware vendor, such as AMD, NVIDIA, or Intel.
- On Linux systems, use proprietary vendor drivers instead of open-source replacements.
- On Macintosh systems, the graphics drivers are part of the operating system. Use the latest updates provided.

Use graphics hardware that supports a hardware-accelerated implementation of OpenGL 2.1 or later. Most graphics hardware released since 2006 has OpenGL 2.1 or later. If you have an earlier version of OpenGL, most graphics features still work, but some advanced graphics features are unavailable. For the best performance, OpenGL 3.3 or later is recommended. For more information on determining your graphics hardware, see `opengl`.

Choose OpenGL Implementation for Your System

MATLAB renders graphics using either a hardware-accelerated, basic hardware-accelerated, or software implementation of OpenGL. By default, MATLAB tries to use a hardware-accelerated implementation if your graphics hardware supports it. You can work around many graphics issues by switching to either a software implementation or a basic hardware-accelerated implementation. These alternate implementations do not support some advanced graphics features.

In some cases, MATLAB automatically switches to a software OpenGL implementation:

- If you do not have graphics hardware or if your graphics hardware does not support hardware OpenGL.
- If a previous MATLAB session crashed due to a graphics issue.
- If you are using a graphics driver with known issues, an older NVIDIA graphics driver, or graphics virtualization. Update your graphics drivers to the latest versions available.
- If you are using remote desktop on certain Windows systems, some graphics drivers do not support hardware OpenGL. If you try to use hardware OpenGL, MATLAB returns a warning message and uses software OpenGL instead. It is possible that updating your graphics drivers to the latest versions will enable support for hardware OpenGL.

To determine which implementation MATLAB is using, type `opengl info` at the command prompt and check the `Software` and `HardwareSupportLevel` fields. For more information, see `opengl`.

Specify OpenGL Implementation for Current Session

To specify the OpenGL implementation for the current session of MATLAB, use one of these techniques.

- Software OpenGL — Start MATLAB from the command prompt on your system using the command `matlab -softwareopengl`. This command works only Windows and Linux systems. Macintosh systems do not support software OpenGL.
- Basic hardware-accelerated OpenGL — Type `opengl hardwarebasic` at the MATLAB command prompt.
- Hardware-accelerated OpenGL — Type `opengl hardware` at the MATLAB command prompt.

Specify OpenGL Implementation for Future Sessions

To set your preferences so that MATLAB always starts with the specified implementation of OpenGL, use one of these techniques.

- Software OpenGL — Type `opengl('save','software')` at the MATLAB command prompt. Then, restart MATLAB.
- Basic hardware-accelerated OpenGL — Type `opengl('save','hardwarebasic')` at the MATLAB command prompt. Then, restart MATLAB.
- Hardware-accelerated OpenGL — Type `opengl('save','hardware')` at the MATLAB command prompt. Then, restart MATLAB.
- Undo preference setting — Execute `opengl('save','none')` at the MATLAB command line. Then, restart MATLAB.

Fix Out-of-Memory Issues

Graphics hardware with limited graphics memory can cause poor performance or lead to out-of-memory issues. Improve performance and work around memory issues with these changes:

- Use smaller figure windows.
- Turn off anti-aliasing by setting the GraphicsSmoothing property of the figure to `'off'`.
- Do not use transparency.
- Use software OpenGL.

Contact Technical Support

If you cannot resolve the issues using the options described here, then you might have encountered a bug in MATLAB. Contact MathWorks technical support and provide the following information:

- Output of executing `opengl info`.
- Whether your code runs without error when using software OpenGL.
- Whether your code runs without error on a different computer. Provide the output of `opengl info` for all computers you have tested your code on.

- Some error messages contain a link to a file with details about the graphics error you encountered. If a link to this file is provided, include this file with your service request.

Create a Service Request at http://www.mathworks.com/support/contact_us.

See Also

`opengl`

More About

- "System Requirements for Graphics" on page 18-19

19

set and get

Access Property Values

In this section...

“Object Properties and Dot Notation” on page 19-2
“Graphics Object Variables Are Handles” on page 19-4
“Listing Object Properties” on page 19-6
“Modify Properties with set and get” on page 19-6
“Multi Object/Property Operations” on page 19-7

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 new syntax to access object properties starting in R2014b. 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
  h         1x1               112  matlab.graphics.chart.primitive.Line
  h2        1x1               112  matlab.graphics.chart.primitive.Line
```

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
  h         1x1               112  matlab.graphics.chart.primitive.Line
  h2        1x1               112  matlab.graphics.chart.primitive.Line
```

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 vectors. You can use single quotes or a variable that is a character vector. 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

- "Control Ratio of Axis Lengths and Data Unit Lengths" on page 20-2
- "Create Chart with Multiple *x*-Axes and *y*-Axes" on page 20-10
- "Display Text Outside Axes" on page 20-14
- "Line Styles Used for Plotting — LineStyleOrder" on page 20-17

Control Ratio of Axis Lengths and Data Unit Lengths

In this section...

“Plot Box Aspect Ratio” on page 20-2

“Data Aspect Ratio” on page 20-5

“Revert Back to Default Ratios” on page 20-8

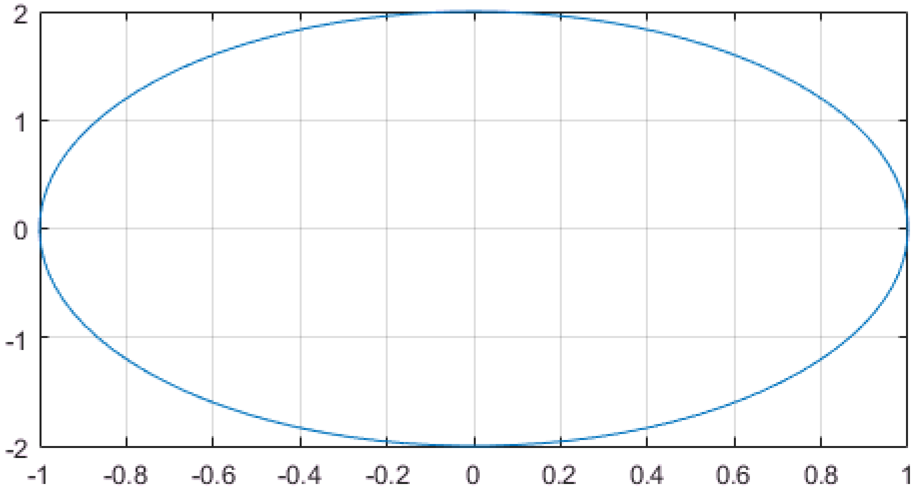
You can control the relative lengths of the *x*-axis, *y*-axis, and *z*-axis (plot box aspect ratio). You also can control the relative lengths of one data unit along each axis (data aspect ratio).

Plot Box Aspect Ratio

The plot box aspect ratio is the relative lengths of the *x*-axis, *y*-axis, and *z*-axis. By default, the plot box aspect ratio is based on the size of the figure. You can change the aspect ratio using the `pbaspect` function. Set the ratio as a three-element vector of positive values that represent the relative axis lengths.

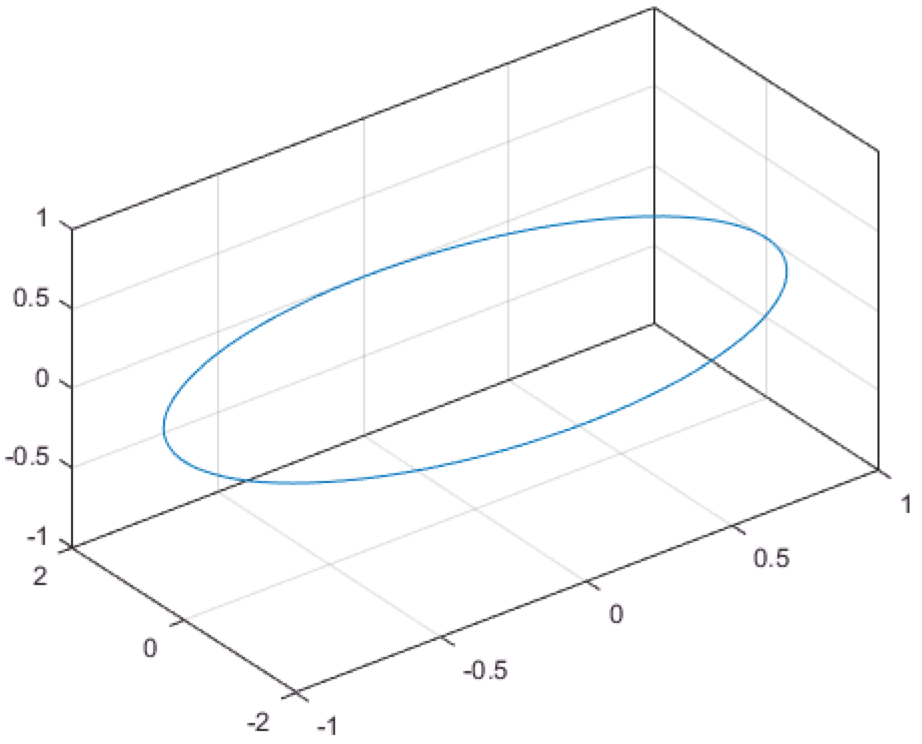
For example, plot an elongated circle. Then set the plot box aspect ratio so that the x-axis is twice the length of the *y*-axis and *z*-axis (not shown).

```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
pbaspect([2 1 1])
```



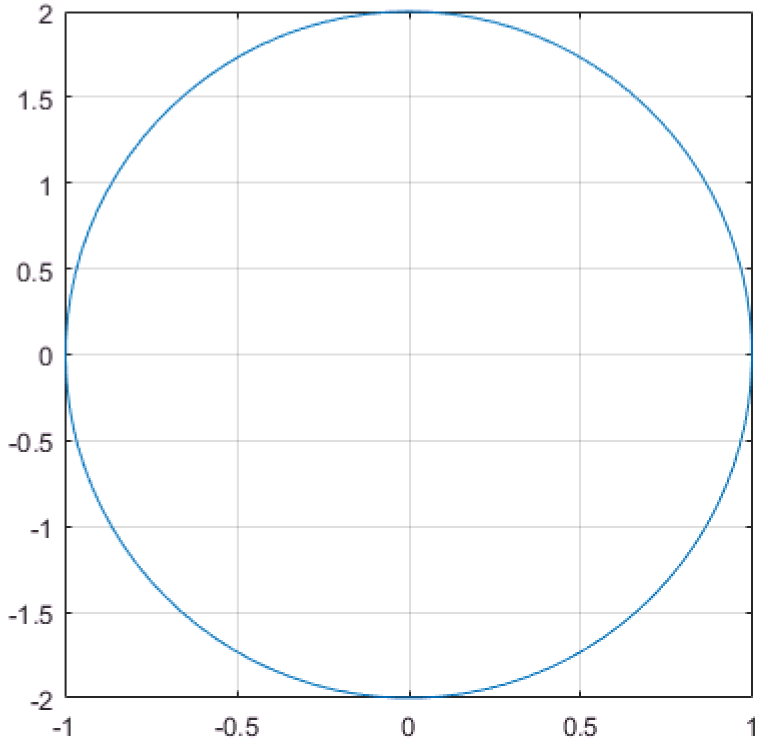
Show the axes in a 3-D view to see the *z*-axis.

```
view(3)
```



For square axes, use `[1 1 1]`. This value is similar to using the `axis square` command.

```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
pbaspect([1 1 1])
```



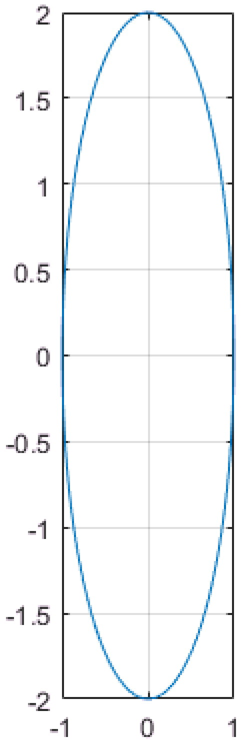
Data Aspect Ratio

The data aspect ratio is the relative length of the data units along the *x*-axis, *y*-axis, and *z*-axis. You can change the aspect ratio using the `daspect` function. Set the ratio as a three-element vector of positive values that represent the relative lengths of data units along each axis.

For example, set the ratio so that the length from 0 to 1 along the *x*-axis is equal to the length from 0 to 0.5 along the *y*-axis and 0 to 2 along the *z*-axis (not shown).

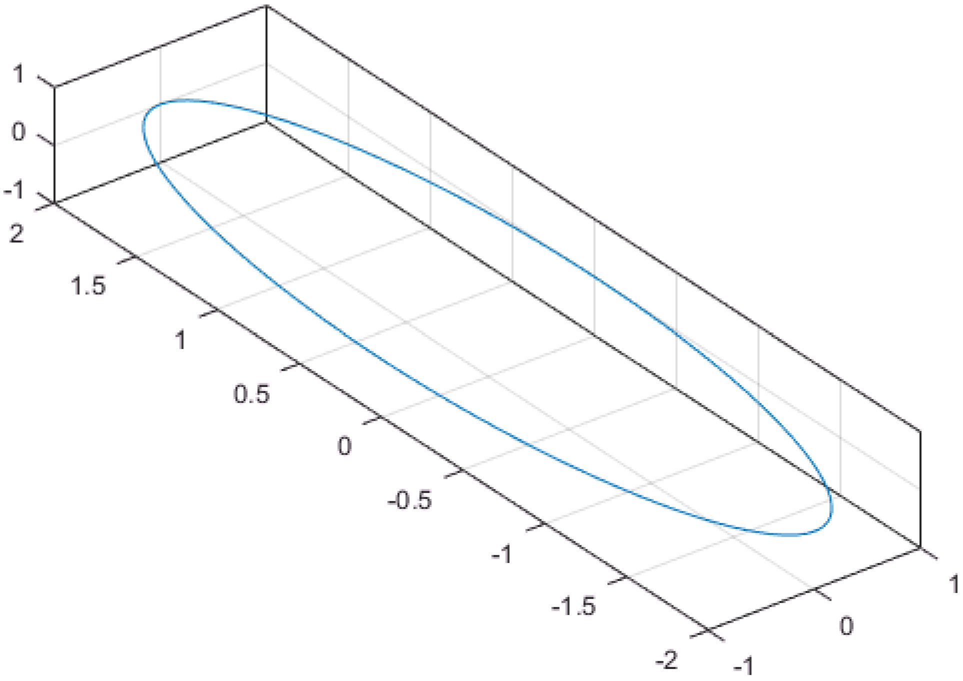
```
t = linspace(0,2*pi);
```

```
plot(sin(t),2*cos(t))
grid on
daspect([1 0.5 2])
```



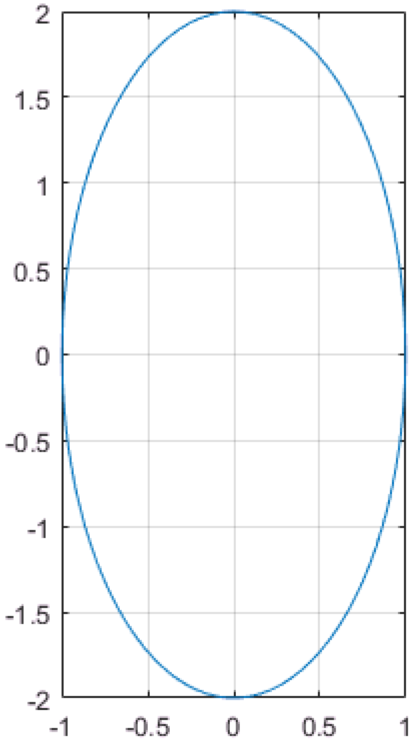
Show the axes in a 3-D view to see the *z*-axis.

```
view(3)
```



For equal data units in all directions, use `[1 1 1]`. This value is similar to using the `axis equal` command. One data unit in the *x* direction is the same length as one data unit in the *y* and *z* directions.

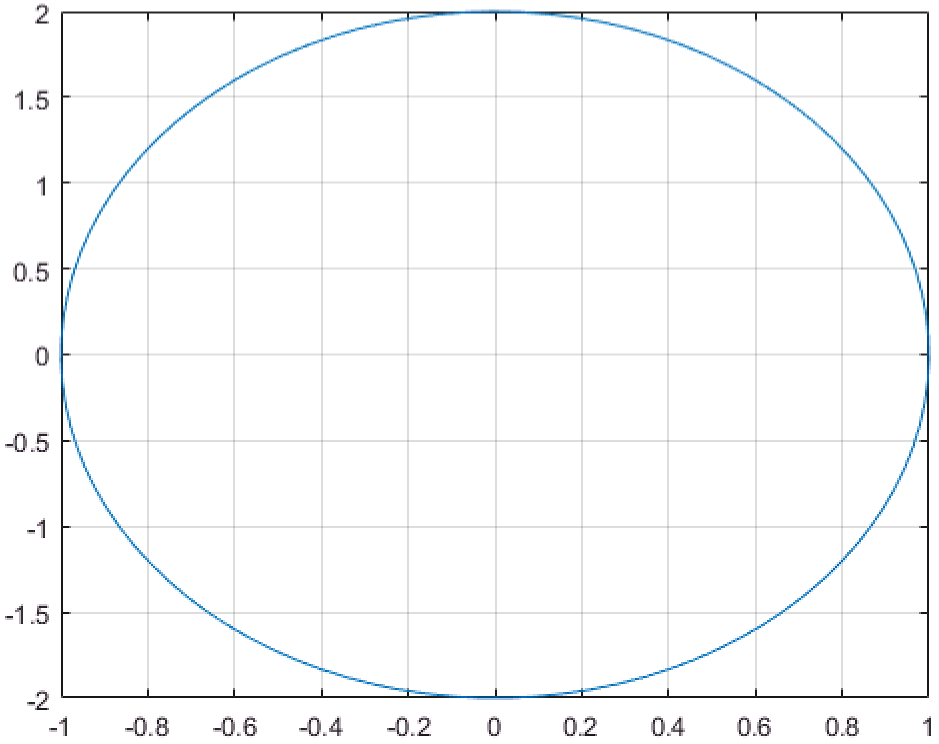
```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
daspect([1 1 1])
```



Revert Back to Default Ratios

Change the data aspect ratio. Then revert back to the default plot box and data aspect ratios using the `axis normal` command.

```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
daspect([1 1 1])
axis normal
```



See Also

Functions

`axis` | `daspect` | `pbaspect`

Related Examples

- "Specify Axis Limits" on page 2-19
- "Control Axes Layout" on page 9-2

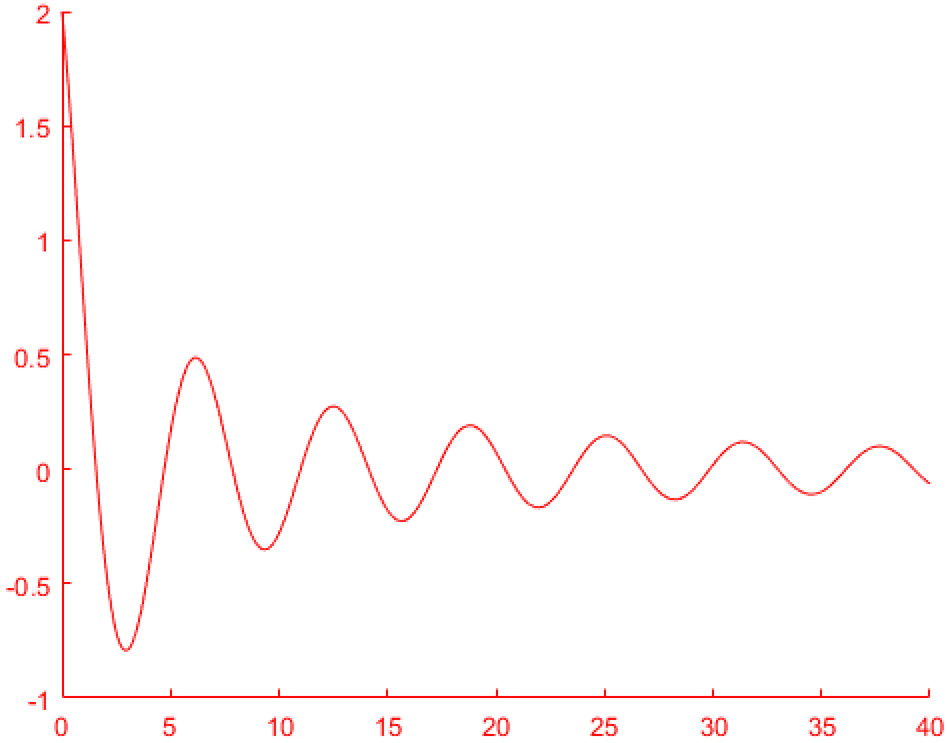
Create Chart with Multiple *x*-Axes and *y*-Axes

This example shows how to create a chart using the bottom and left sides of the axes for the first plot and the top and right sides for the second plot.

Plot a red line using the `line` function. Set the color for the *x*-axis and *y*-axis lines to red.

Note: Starting in R2014b, you can use dot notation to set properties. If you are using an earlier release, use the set function instead, such as `set(ax1,'XColor','r')`.

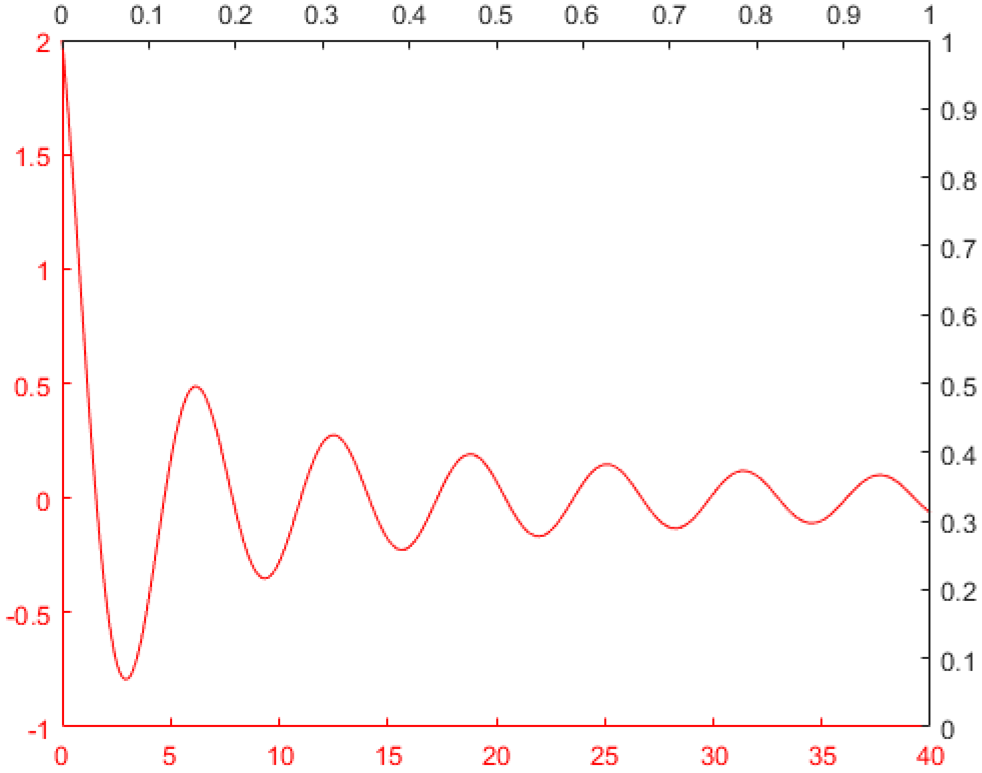
```
figure
x1 = 0:0.1:40;
y1 = 4.*cos(x1)./(x1+2);
line(x1,y1,'Color','r')
ax1 = gca; % current axes
ax1.XColor = 'r';
ax1.YColor = 'r';
```



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. Display the *x*-axis at the top of the axes and the *y*-axis on the right side. Set the axes `Color` to `'none'` so that the first axes is visible underneath the second axes.

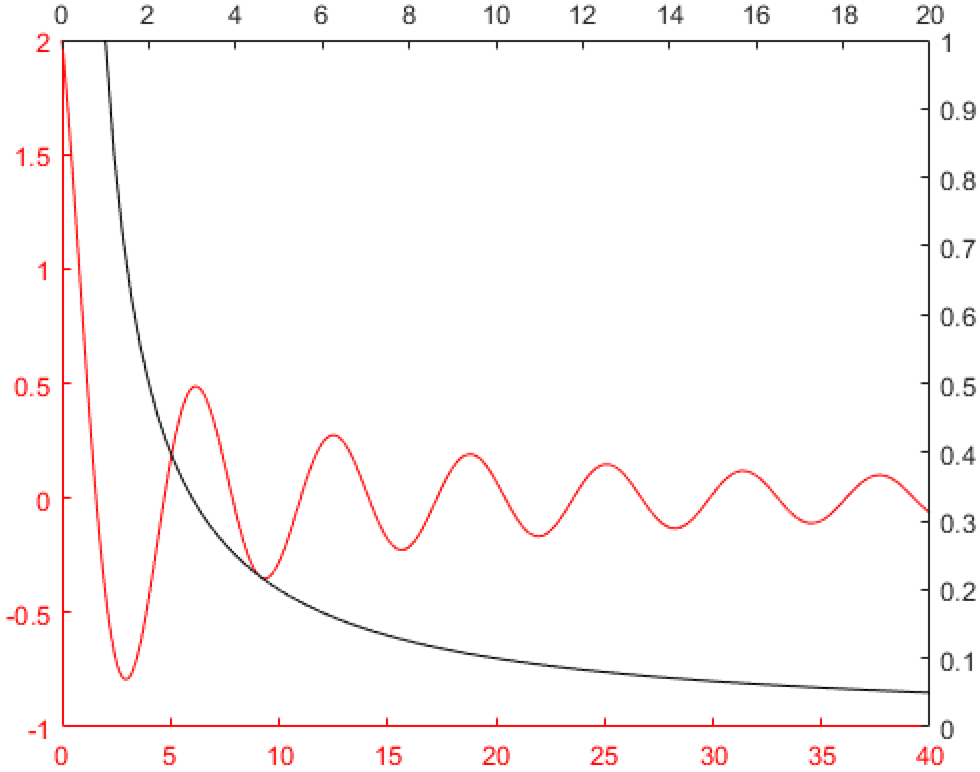
Note: Starting in R2014b, you can use dot notation to query properties. If you are using an earlier release, use the get function instead, such as `ax1_pos = get(ax1,'Position')`.

```
ax1_pos = ax1.Position; % position of first axes
ax2 = axes('Position',ax1_pos,...
    'XAxisLocation','top',...
    'YAxisLocation','right',...
    'Color','none');
```



Plot a line in the second axes. Set the line color to black so that it matches the color of the corresponding *x*-axis and *y*-axis.

```
x2 = 1:0.2:20;
y2 = x2.^2./x2.^3;
line(x2,y2,'Parent',ax2,'Color','k')
```



The chart 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

Functions

`axes` | `gca` | `line`

Related Examples

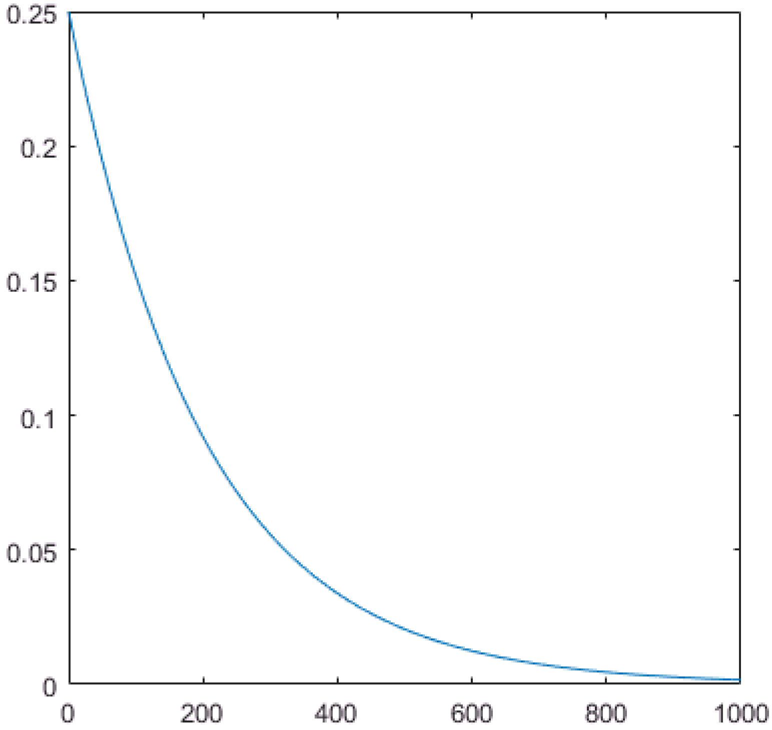
- "Create Chart with Two *y*-Axes" on page 2-55

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 specifying `ax2` as an input argument 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)
```



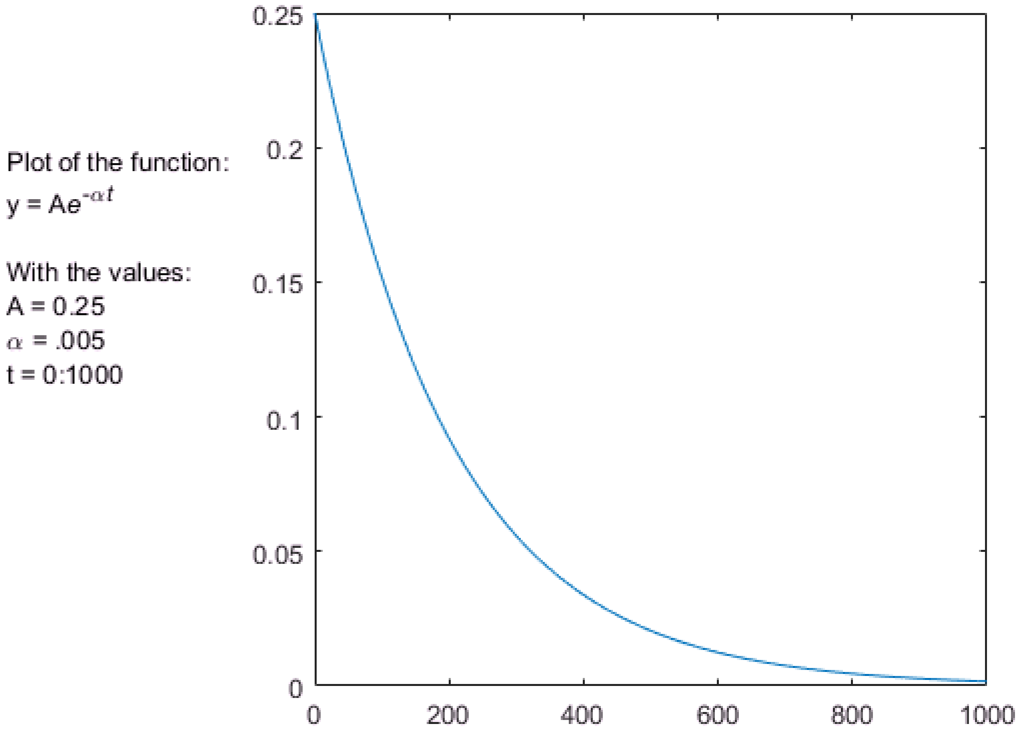
Define the text. Use a cell array to create multiline text.

```
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)
```



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.

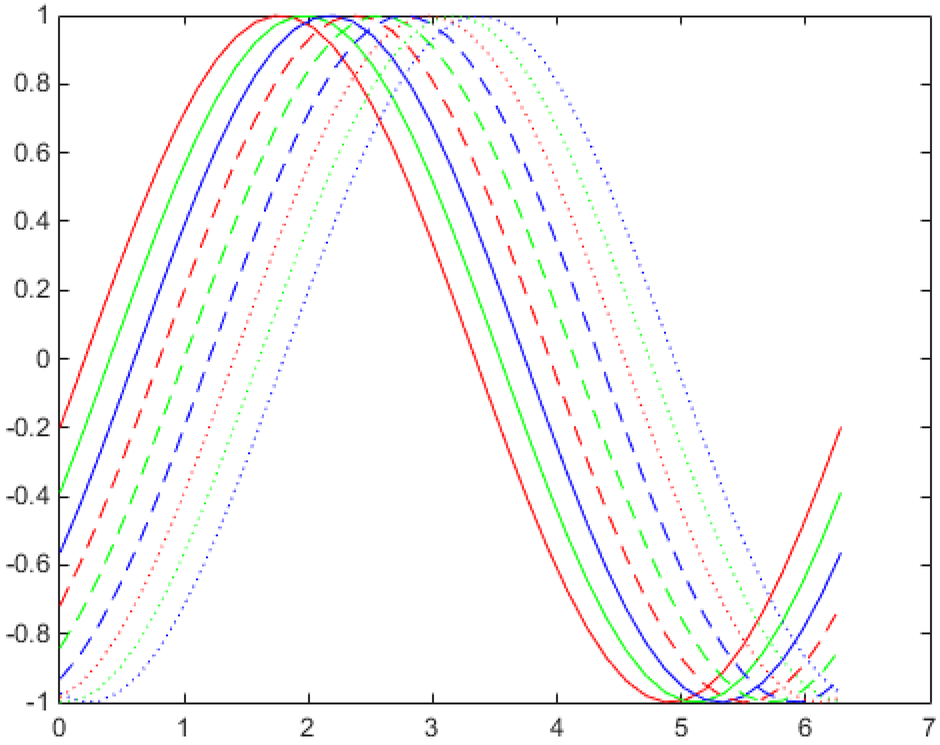
Axes increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

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” on page 11-2 for more information.