MATLAB®

Graphics

MATLAB®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Contents

Polar Plots

3

Contour Plots

4

Specialized Charts

5

6

7

Titles and Labels

8

Axes Appearance

9

Coloring Graphs

10

Lighting

11

Transparency

12

Displaying Bit-Mapped Images

15

Printing and Saving

16

Graphics Properties

17

18

19

20

Group Objects

21

Controlling Graphics Output

22

Developing Classes of Chart Objects

23

Optimize Performance of Graphics Programs

24

Line Plots

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"	Scatter and Bubble Charts	"Data Distribution Plots"	"Discrete Data Plots"	"Geographic Plots"	"Polar Plots"	"Contour Plots"	"Vector Fields"	"Surface and Mesh Plots"	"Volume Visualization"	"Animation"	"Images"
plot	scatter	histogram	bar	geoplot	polarplot	contour	quiver	surf	streamline	animatedline	image
plot3	scatter3	histogram2	barh	geoscatter	polarhistogram	contourf	quiver3	surfc	streamslice	comet	imagesc
stairs	bubblechart	pie	bar3	geobubble	polarscatter	contour3	feather	surfl	streamparticles	comet3	
errorbar	bubblechart3	pie3	bar3h		polarbubblechart	contourslice		ribbon	streamribbon		
area	swarmchart	scatterhistogram	pareto		compass	fcontour		pcolor	streamtube		

"Line Plots"	Scatter and Bubble Charts	"Data Distribution Plots"	"Discrete Data Plots"	"Geographic Plots"	"Polar Plots"	"Contour Plots"	"Vector Fields"	"Surface and Mesh Plots"	"Volume Visualization"	"Animation"	"Images"
stackedplot	swarmchart3	swarmchart	stem		ezpolar			fsurf	coneplot		
loglog	spy	swarmchart3	stem3					fimplicit3	slice		
semilogx		wordcloud	stairs					mesh			
semilogy		bubblecloud						meshc			
fplot		heatmap						meshz			
fplot3		parallelplot						waterfall			
fimplicit		plotmatrix						fmesh			

See Also

Related Examples

- "Create 2-D Line Plot" on page 1-14
- MATLAB Plot Gallery

Create Common 2-D Plots

This example shows how to create a variety of 2-D plots in MATLAB®.

Line Plots

The `plot` function creates simple line plots of *x* and *y* values.

```
x = 0:0.05:5;
y = sin(x.^2);
figure
plot(x,y)
```
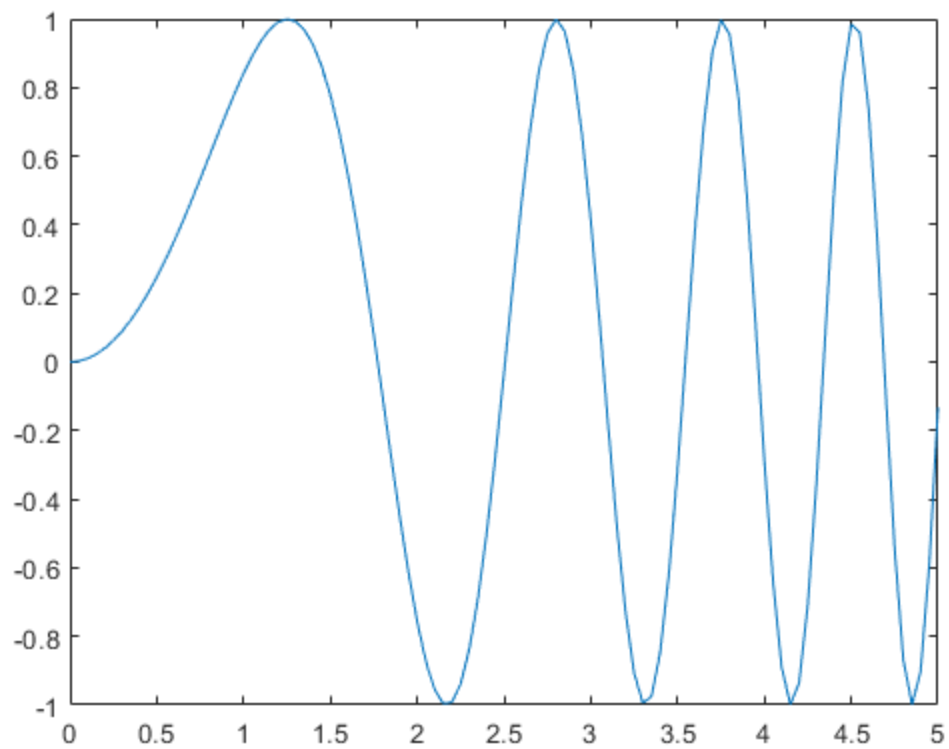

Line plots can display multiple sets of *x* and *y* data.

```
y1 = sin(x.^2);
y2 = cos(x.^2);
plot(x,y1,x,y2)
```
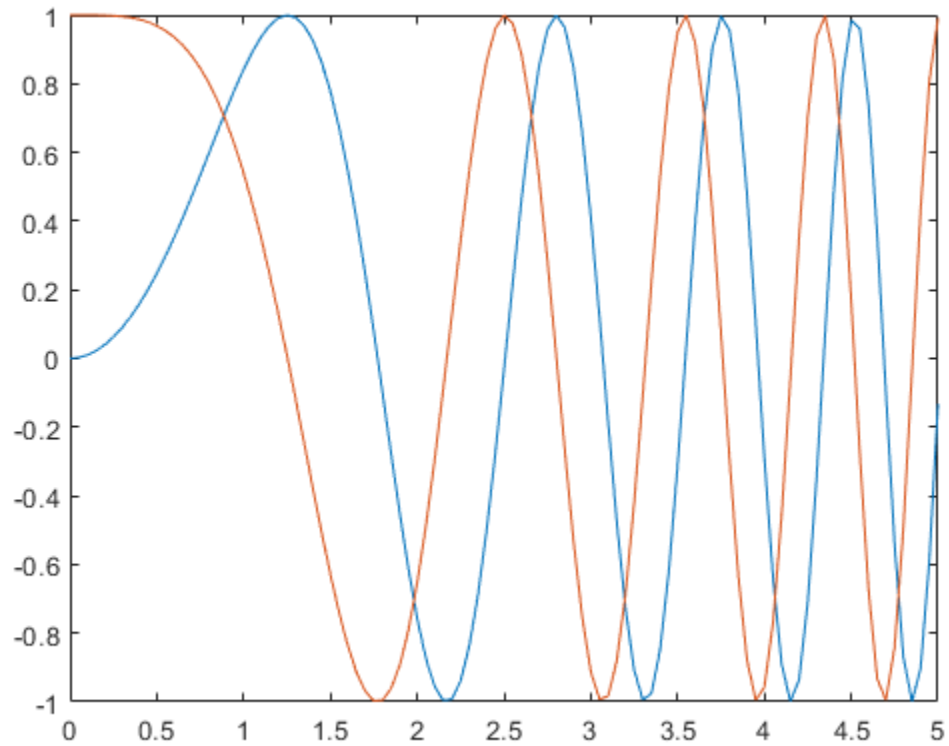
Bar Plots

The `bar` function creates vertical bar charts. The `barh` function creates horizontal bar charts.

```
x = -2.9:0.2:2.9;
y = exp(-x.*x);
bar(x,y)
```
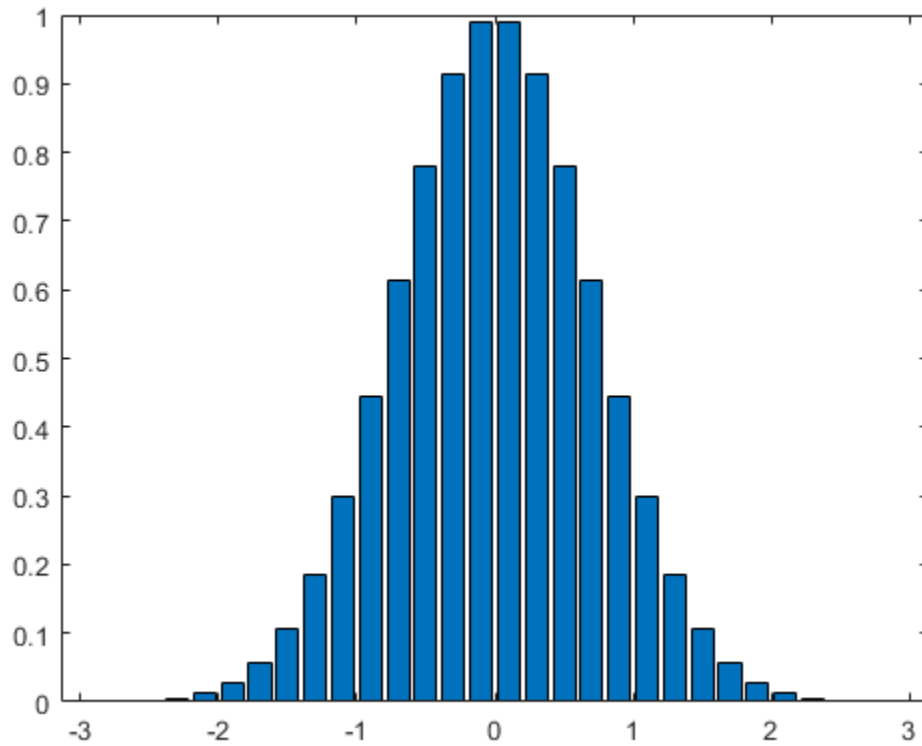
Stairstep Plots

The `stairs` function creates a stairstep plot. It can create a stairstep plot of Y values only or a stairstep plot of *x* and *y* values.

```
x = 0:0.25:10;
y = sin(x);
stairs(x,y)
```
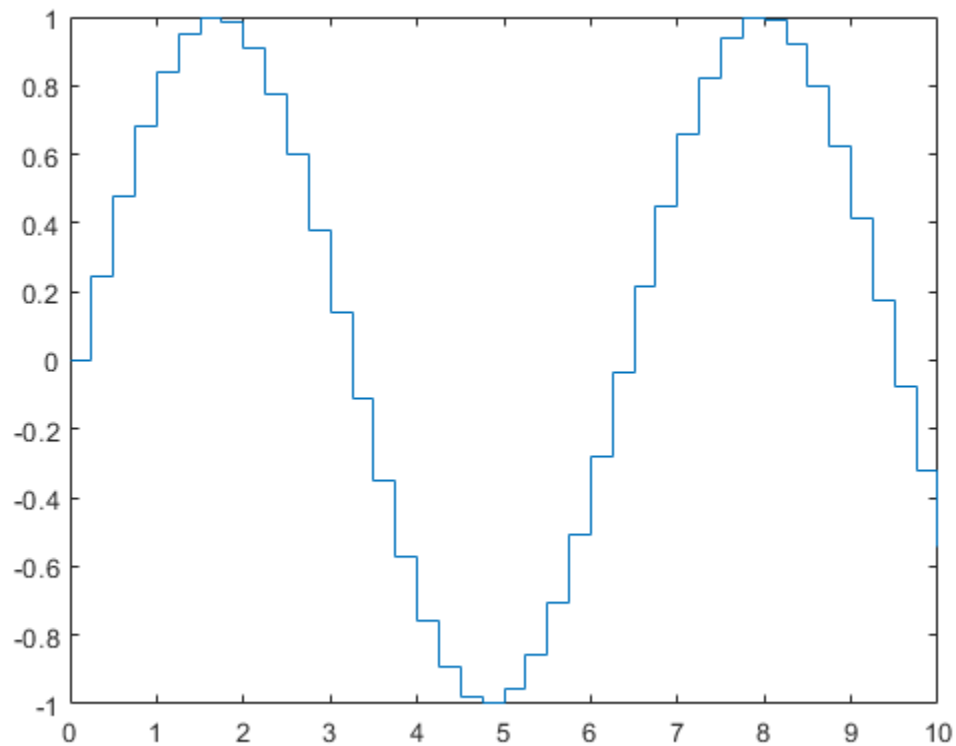
Errorbar Plots

The `errorbar` function draws a line plot of x and y values and superimposes a vertical error bar on each observation. To specify the size of the error bar, pass an additional input argument to the `errorbar` function.

```
x = -2:0.1:2;
y = erf(x);
eb = rand(size(x))/7;
errorbar(x,y,eb)
```
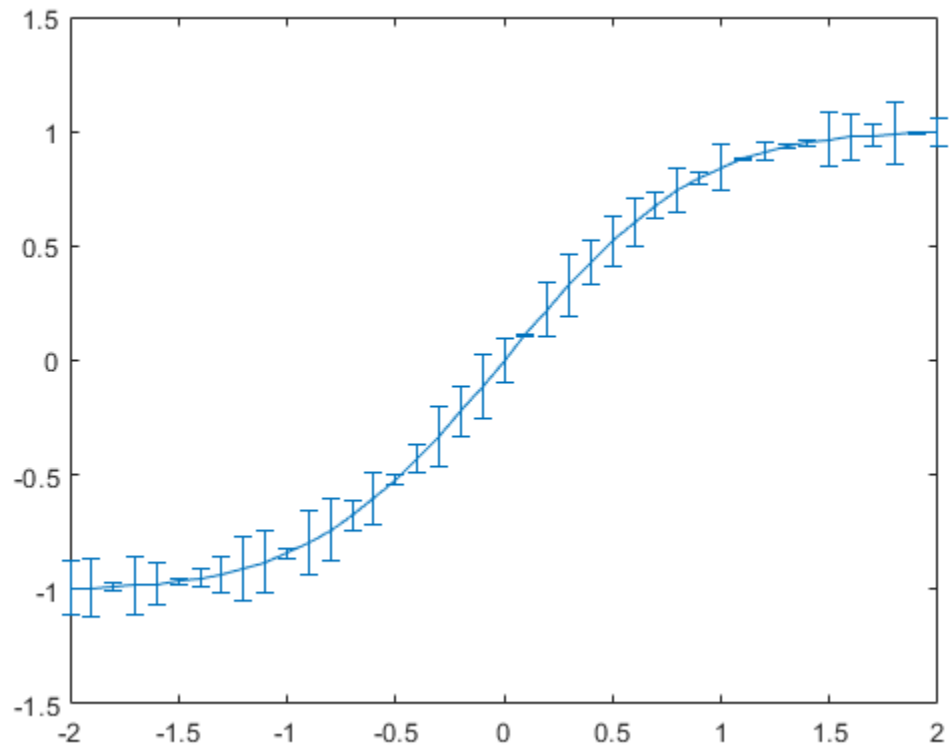
Polar Plots

The `polarplot` function draws a polar plot of the angle values in `theta` (in radians) versus the radius values in `rho`.

```
theta = 0:0.01:2*pi;                      % angle
rho = abs(sin(2*theta).*cos(2*theta));    % radius
polarplot(theta,rho)
```
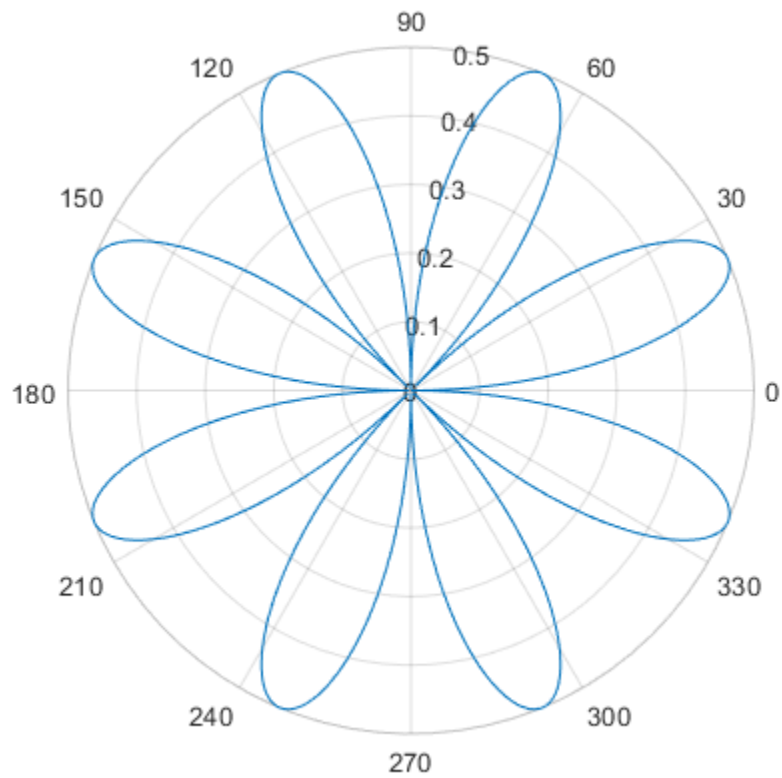
Stem Plots

The `stem` function draws a marker for each *x* and *y* value with a vertical line connected to a common baseline.

```
x = 0:0.1:4;
y = sin(x.^2).*exp(-x);
stem(x,y)
```
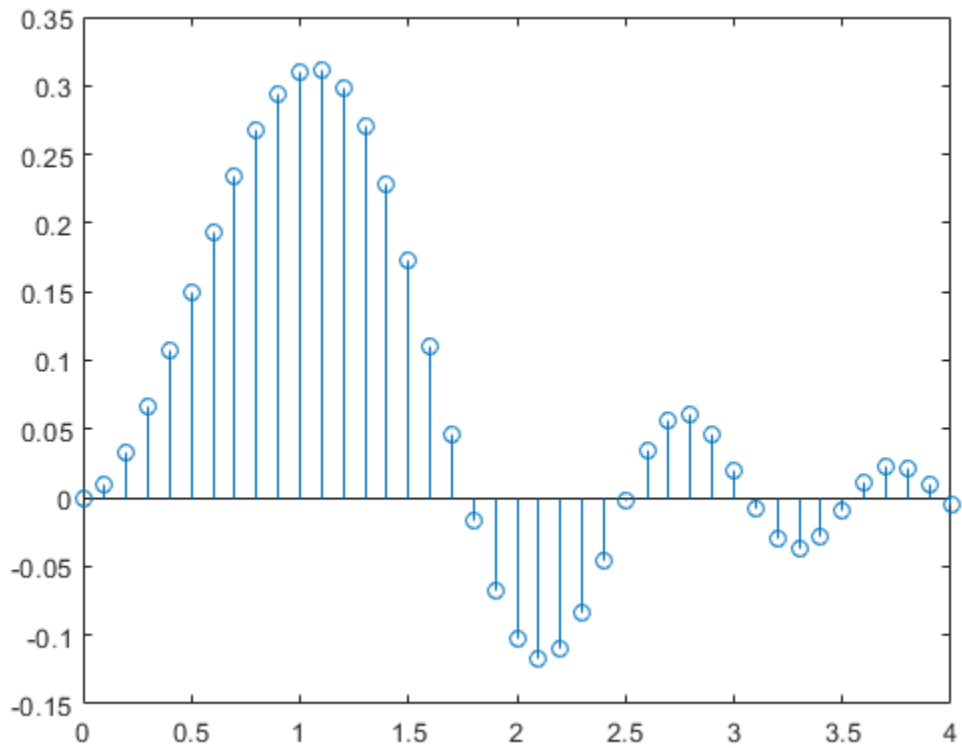
Scatter Plots

The `scatter` function draws a scatter plot of *x* and *y* values.

```matlab
load patients Height Weight Systolic    % load data
scatter(Height,Weight)                  % scatter plot of Weight vs. Height
xlabel('Height')
ylabel('Weight')
```
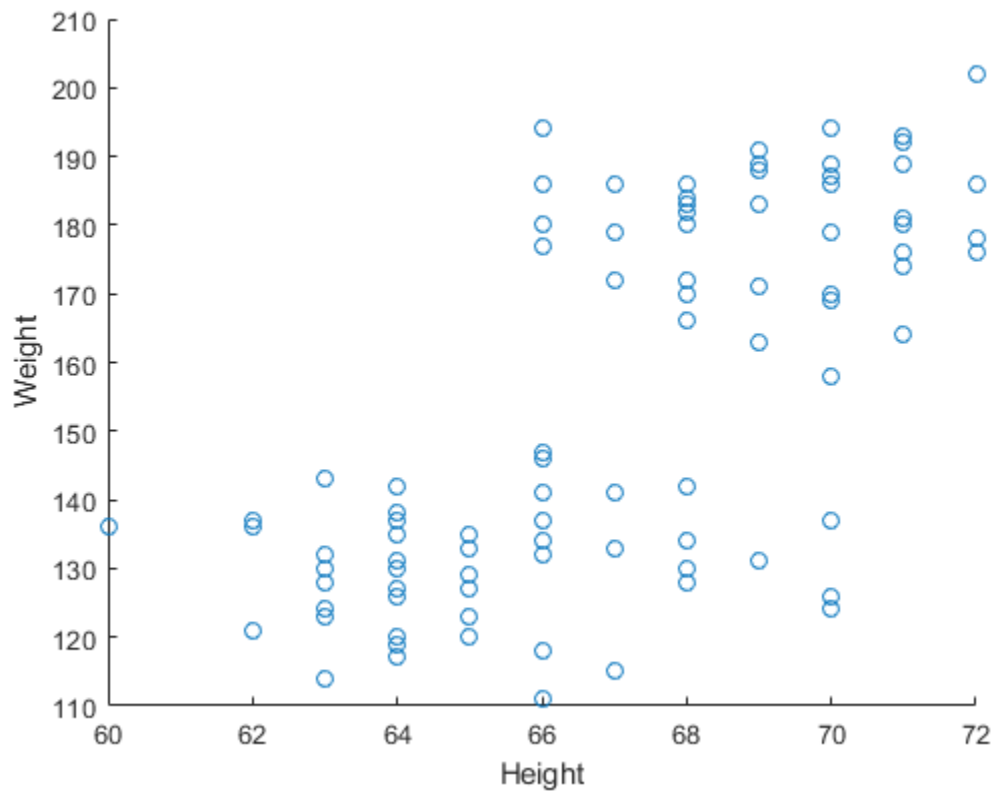
Use optional arguments to the `scatter` function to specify the marker size and color. Use the `colorbar` function to show the color scale on the current axes.

```
scatter(Height,Weight,20,Systolic)    % color is systolic blood pressure
xlabel('Height')
ylabel('Weight')
colorbar
```
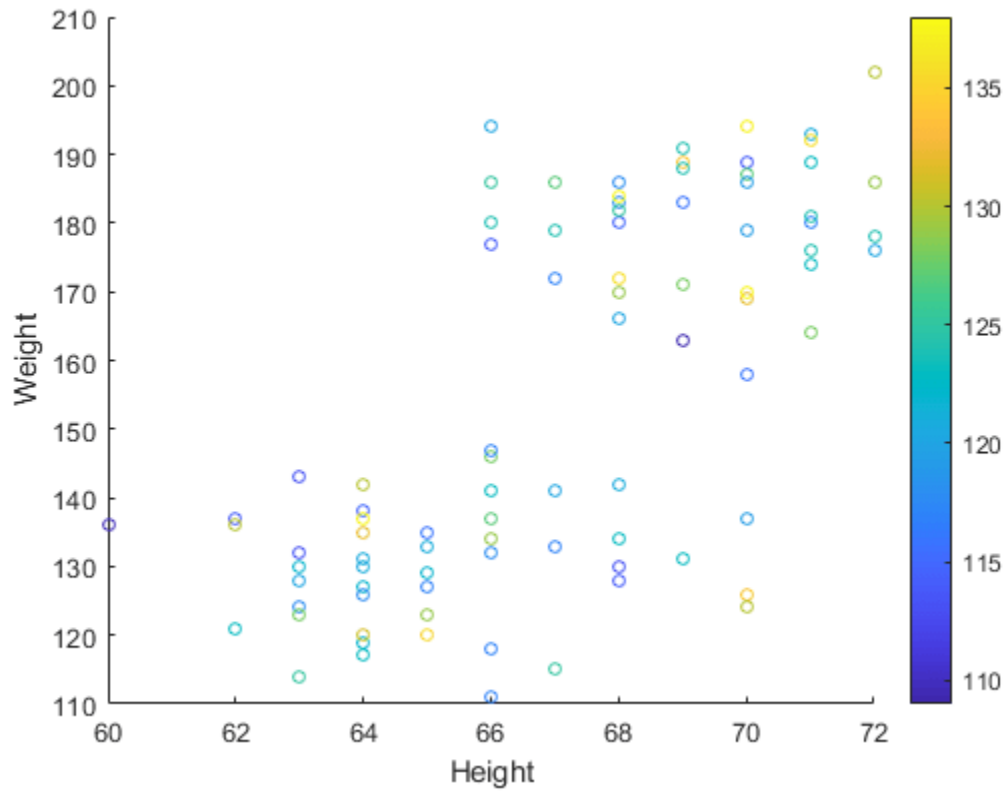
See Also

Related Examples

- "Create 2-D Line Plot" on page 1-14

Create 2-D Line Plot

Create a simple line plot and label the axes. Customize the appearance of plotted lines by changing the line color, the line style, and adding markers.

Create Line Plot

Create a two-dimensional line plot using the `plot` function. For example, plot the value of the sine function from 0 to 2π.

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

Label the axes and add a title.

```
xlabel('x')
ylabel('sin(x)')
title('Plot of the Sine Function')
```
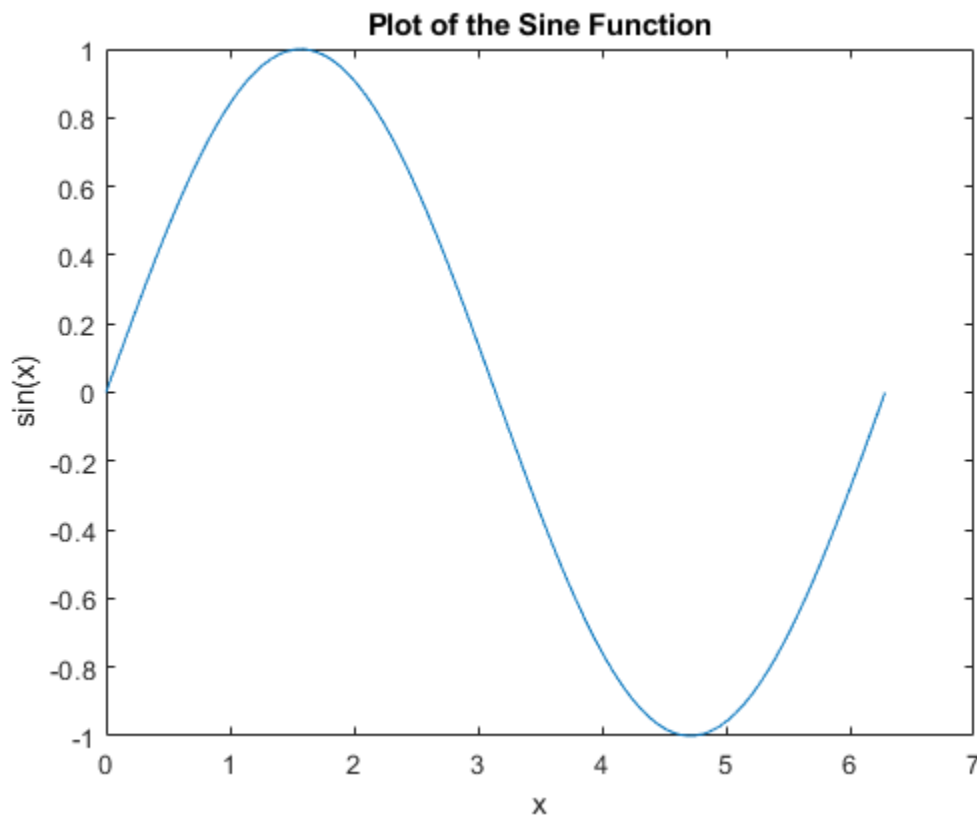
Plot Multiple Lines

By default, MATLAB clears the figure before each plotting command. Use the `figure` command to open a new figure window. You can plot multiple lines using the `hold on` command. Until you use `hold off` or close the window, all plots appear in the current figure window.

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

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

hold on
y2 = cos(x);
plot(x,y2)
hold off
```
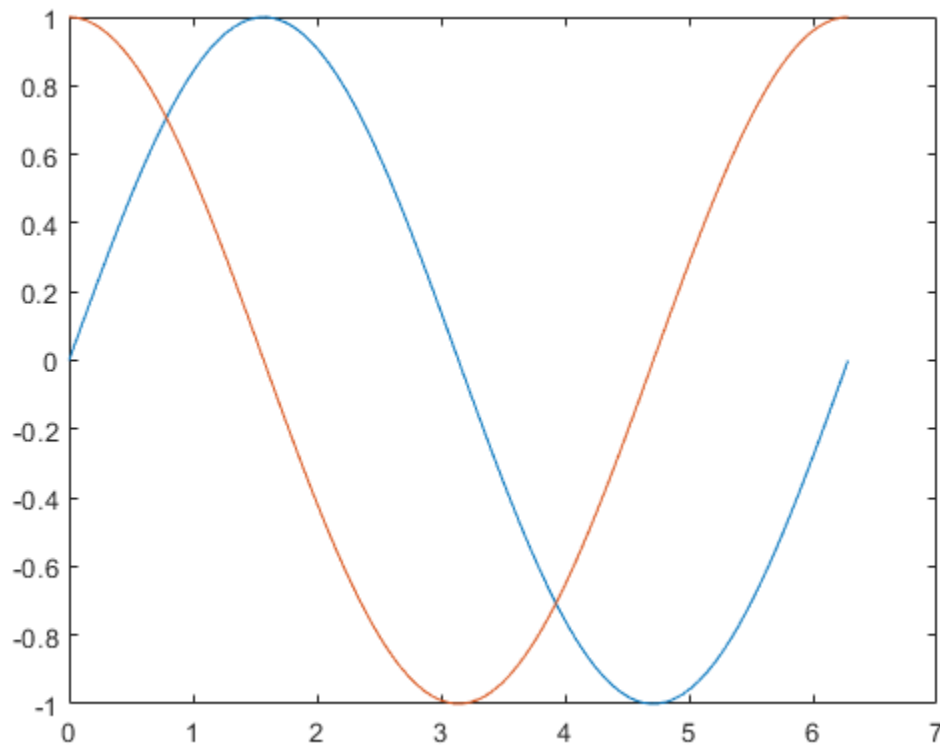

Change Line Appearance

You can change the line color, line style, or add markers by including an optional line specification when calling the `plot` function. For example:

- `':'` plots a dotted line.
- `'g:'` plots a green, dotted line.
- `'g:*'` plots a green, dotted line with star markers.
- `'*'` plots star markers with no line.

The symbols can appear in any order. You do not need to specify all three characteristics (line color, style, and marker). For more information about the different style options, see the `plot` function page.

For example, plot a dotted line. Add a second plot that uses a dashed, red line with circle markers.

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

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

hold on
y2 = cos(x);
plot(x,y2,'--ro')
hold off
```
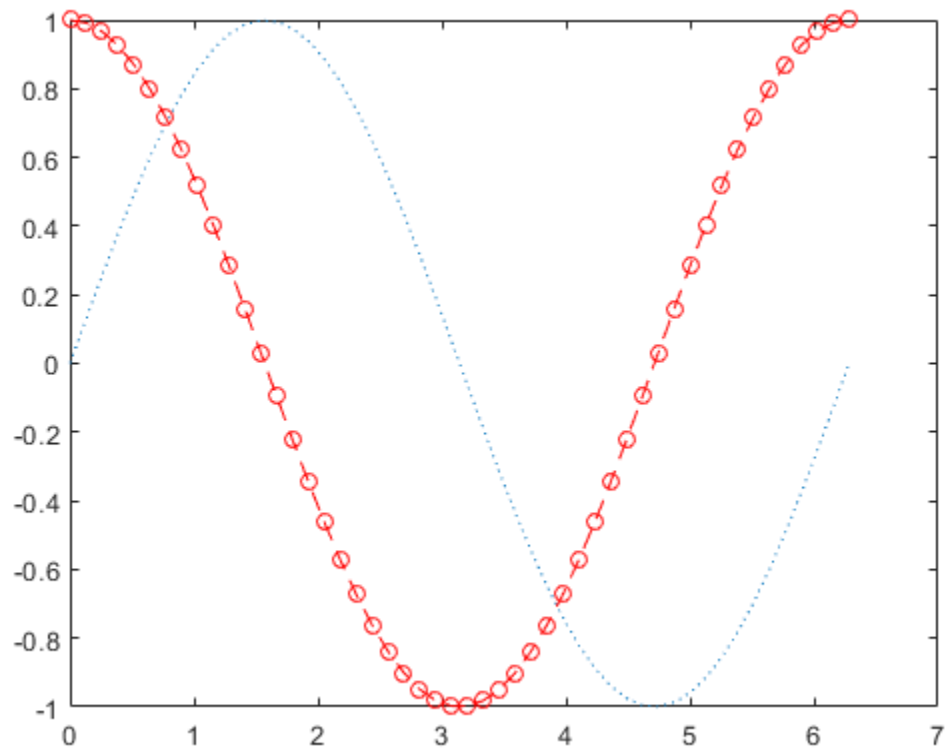

Plot only the data points by omitting the line style option from the line specification.

```
x = linspace(0,2*pi,25);
y = sin(x);
plot(x,y,'o')
```
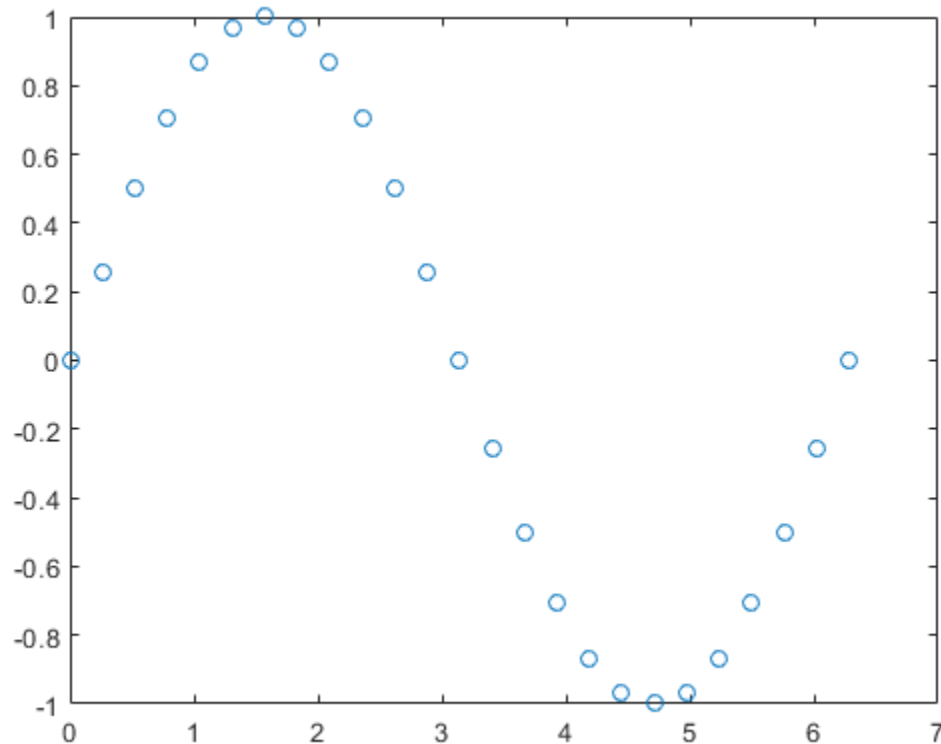
Change Line Object Properties

You also can customize the appearance of the plot by changing properties of the `Line` object used to create the plot.

Create a line plot. Assign the `Line` object created to the variable `ln`. The display shows commonly used properties, such as `Color`, `LineStyle`, and `LineWidth`.

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

ln =
  Line with properties:

              Color: [0 0.4470 0.7410]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x25 double]
              YData: [1x25 double]
              ZData: [1x0 double]

  Show all properties
```

To access individual properties, use dot notation. For example, change the line width to 2 points and set the line color to an RGB triplet color value, in this case `[0 0.5 0.5]`. Add blue, circle markers.

```
ln.LineWidth = 2;
ln.Color = [0 0.5 0.5];
ln.Marker = 'o';
ln.MarkerEdgeColor = 'b';
```
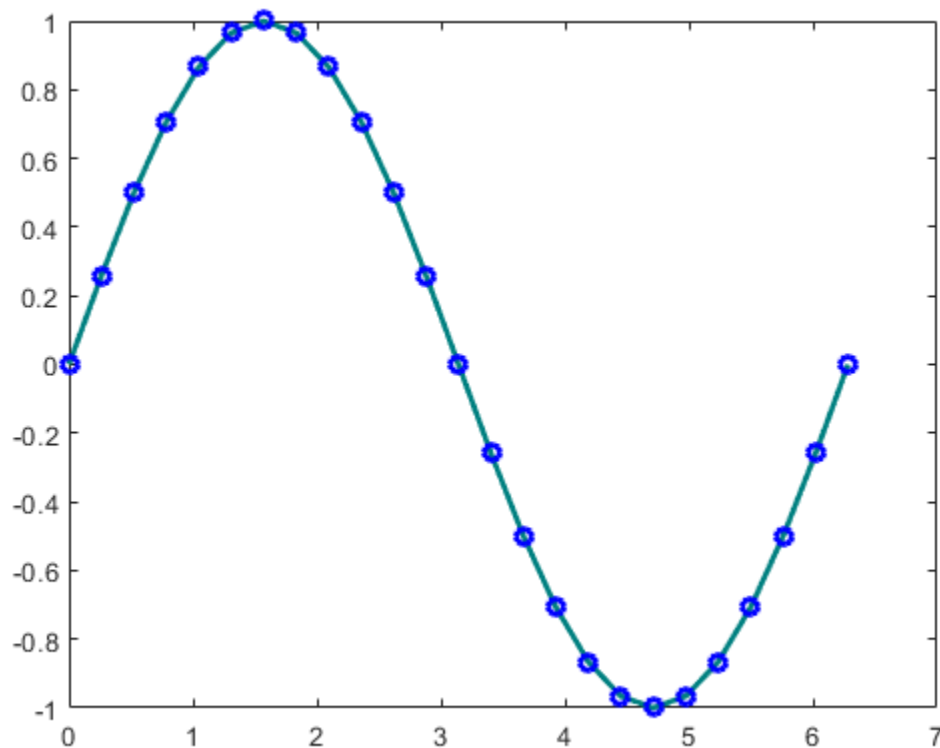

See Also
Line Properties | `loglog` | `plot` | `scatter`

Related Examples
- "Add Title and Axis Labels to Chart" on page 8-2
- "Specify Axis Limits" on page 9-2
- "Specify Axis Tick Values and Labels" on page 9-9
- Create Plot
- MATLAB Plot Gallery

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 1-25.

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')
```
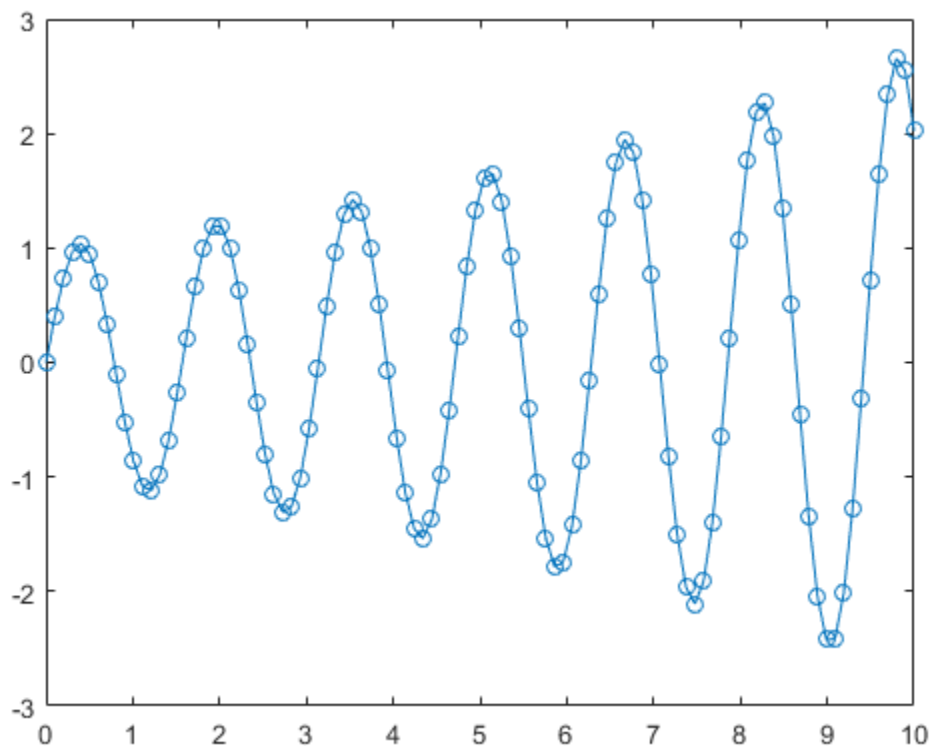

If you specify a marker symbol and do not specify a line style, then `plot` displays only the markers with no line connecting them.

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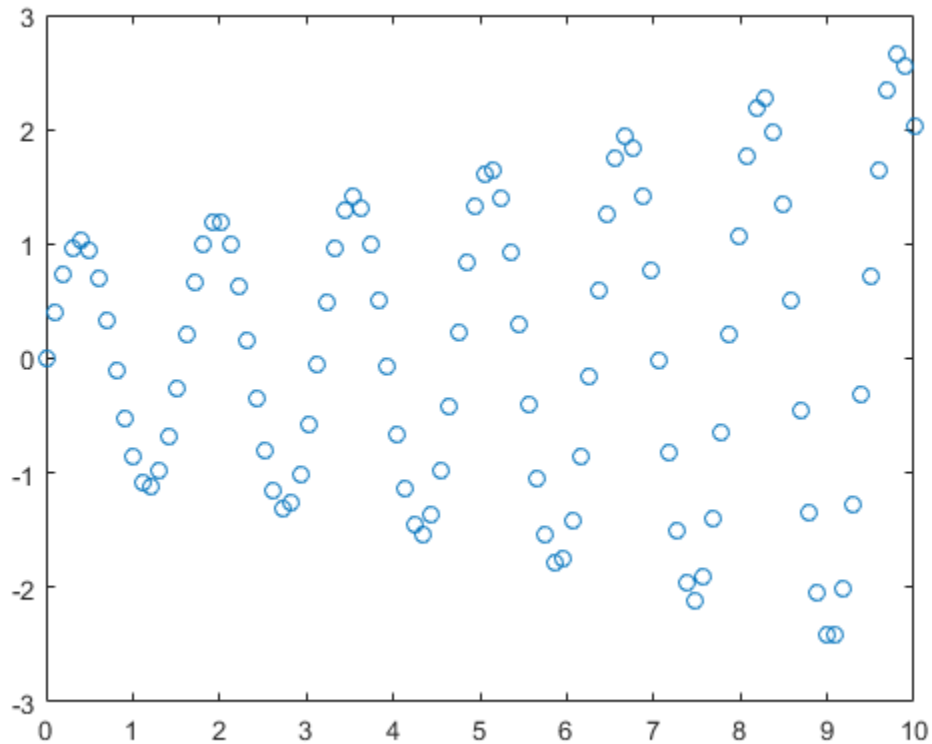
Alternatively, you can add markers to a line by setting the `Marker` property as a name-value pair. For example, `plot(x,y,'Marker','o')` plots a line with circle markers.

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])
```
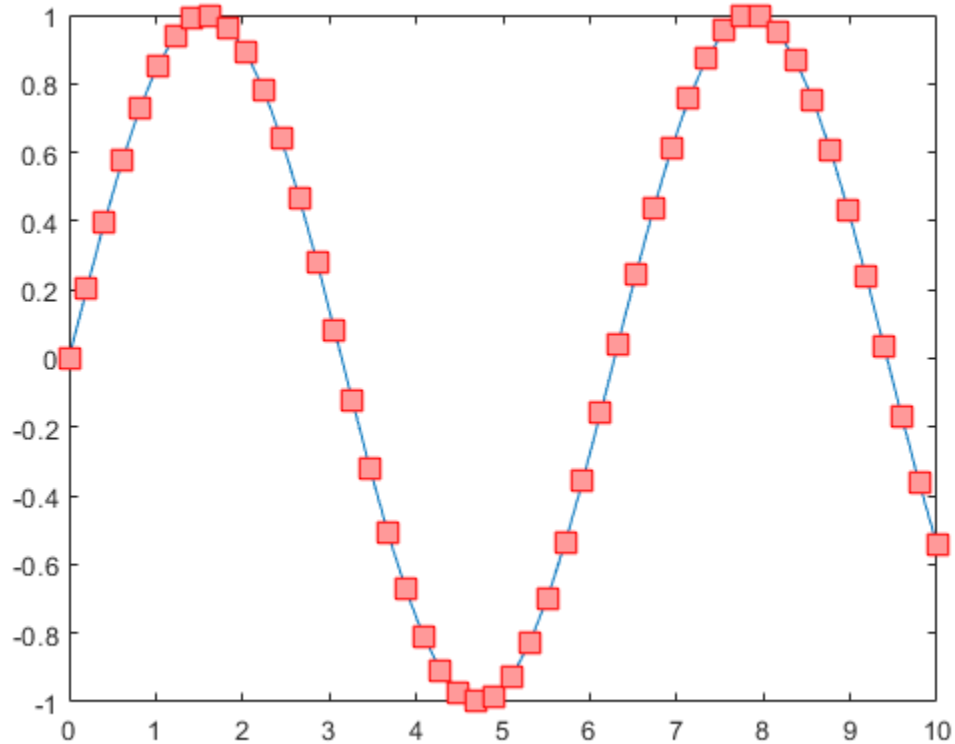
Control Placement of Markers Along Line

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))
```
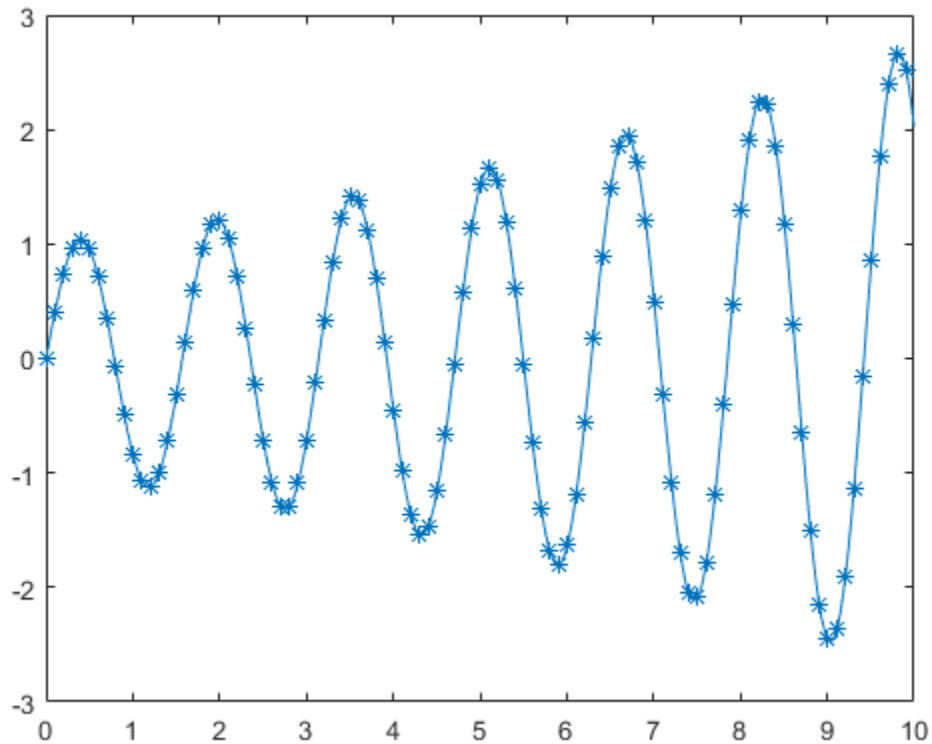
Display Markers at Maximum and Minimum Data Points

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 minimum 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)
```
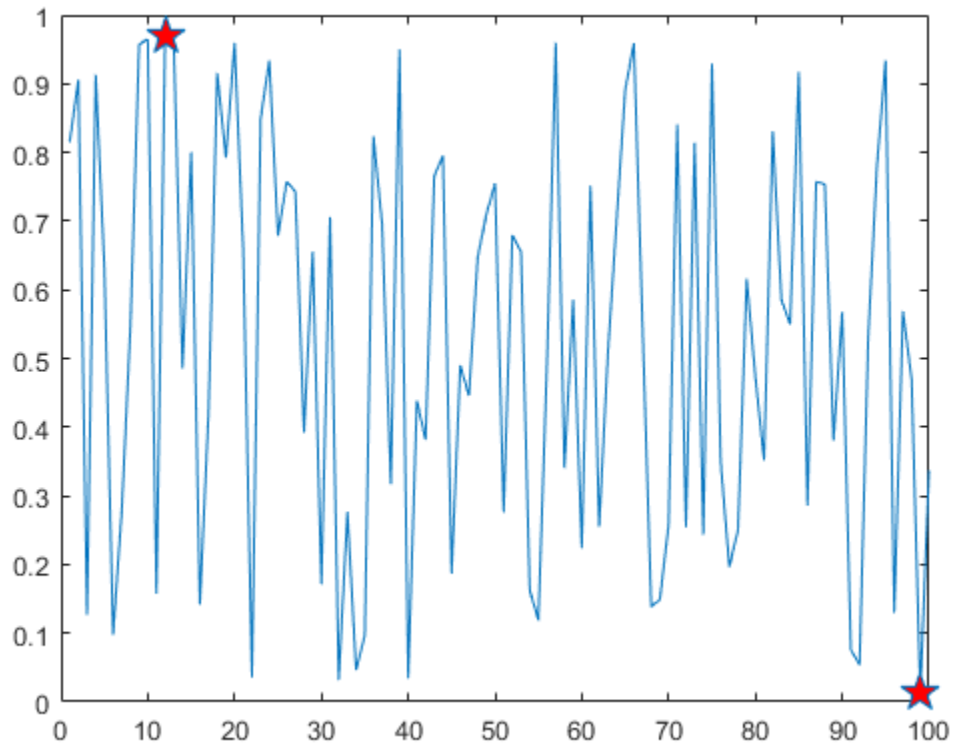
Revert to Default Marker Locations

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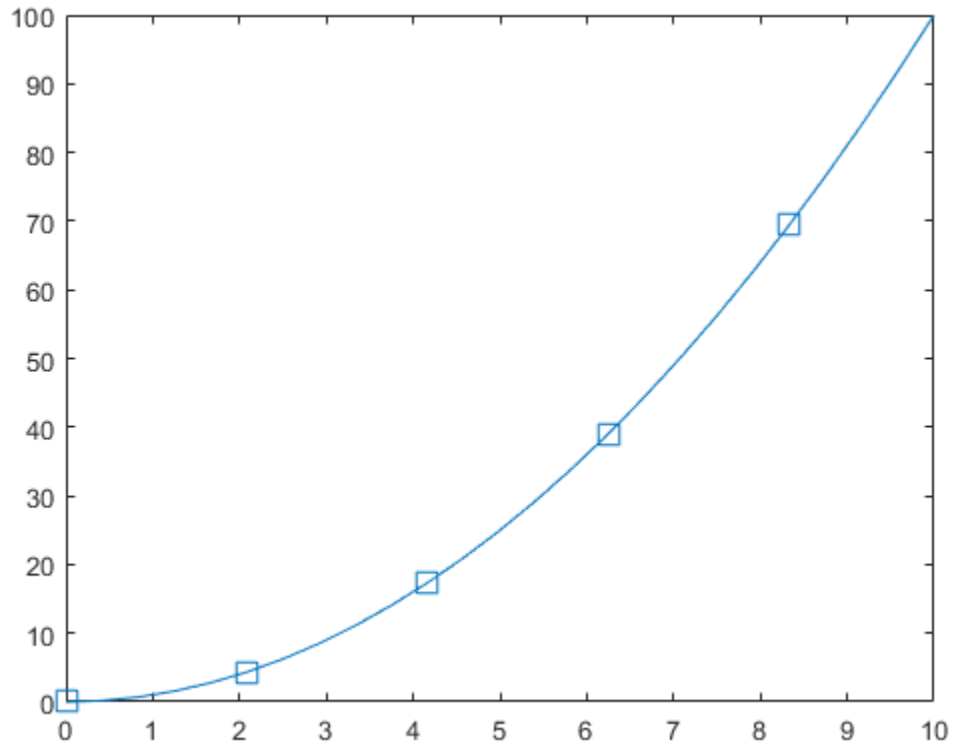
Reset the `MarkerIndices` property to the default value, which is a vector of all index values from 1 to the number of data points.

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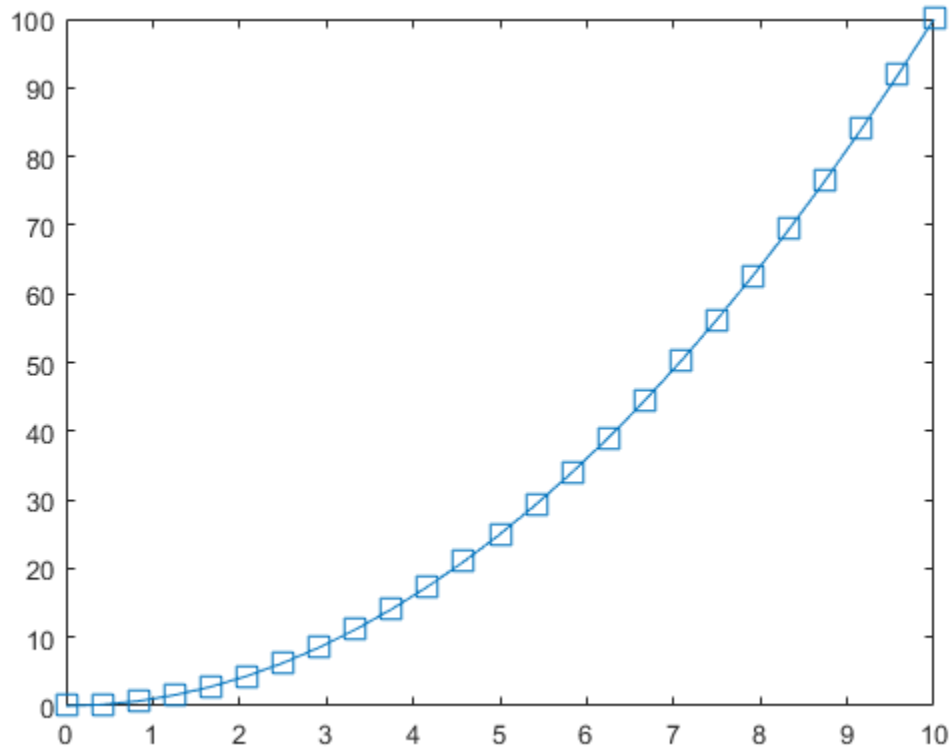
Supported Marker Symbols

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
'|'	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)

Value	Description
`'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
Line

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);
```
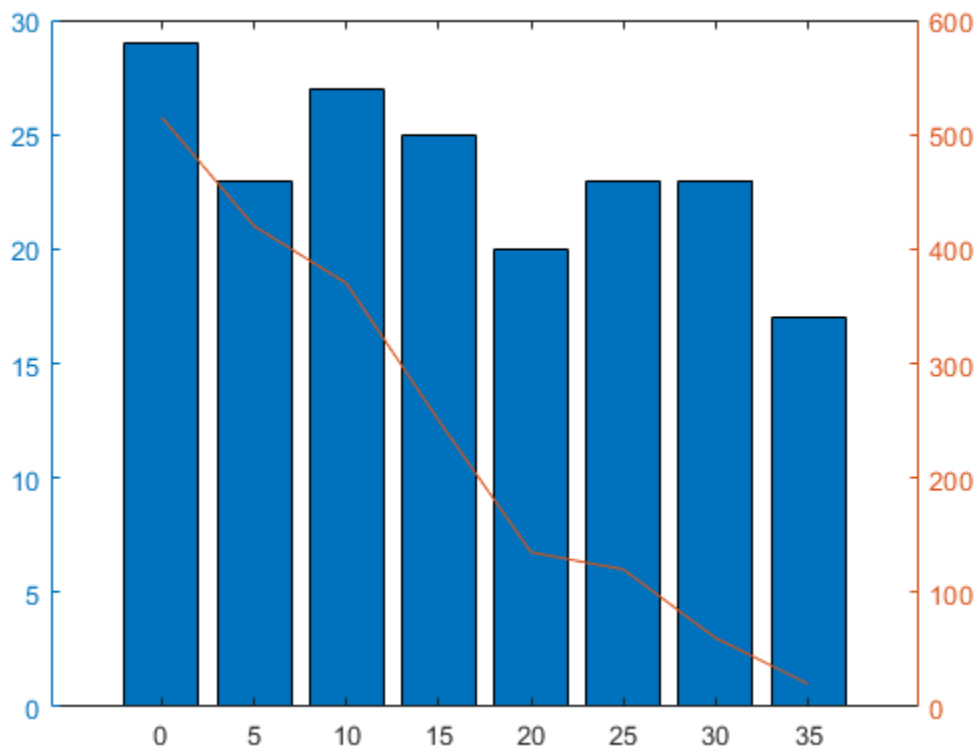

Add a title and axis labels to the chart.

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

1-27
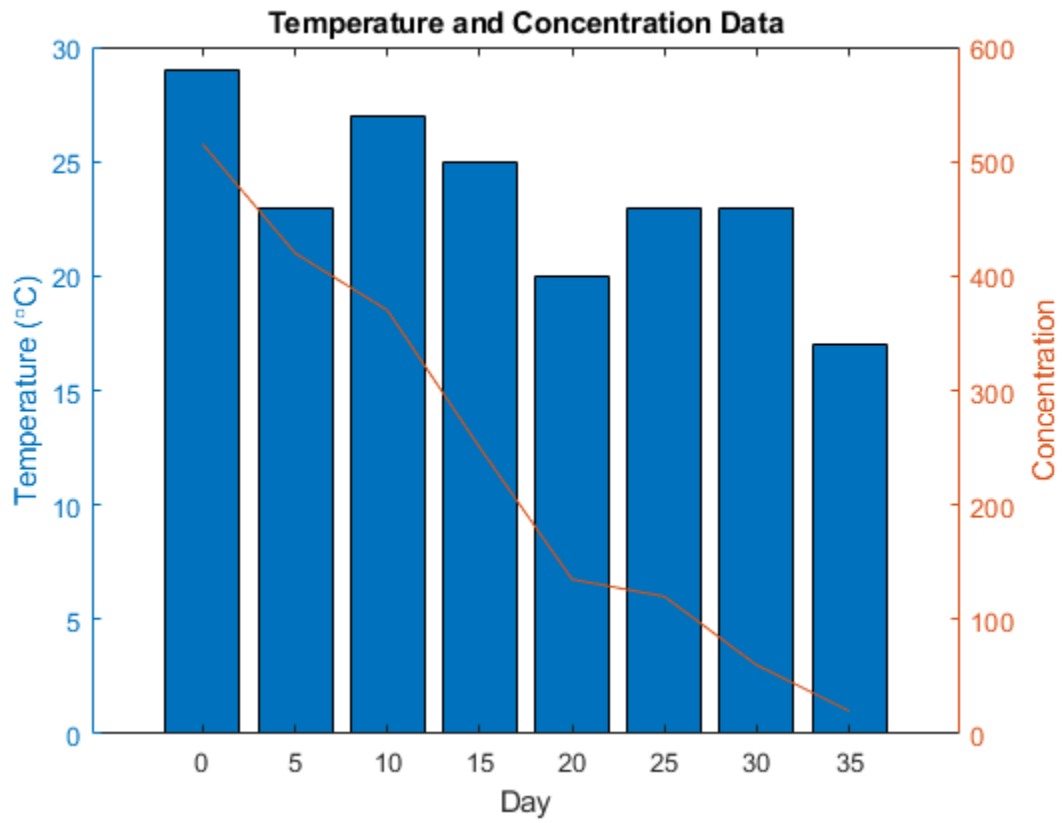
Change the width of the chart line and change the bar colors.

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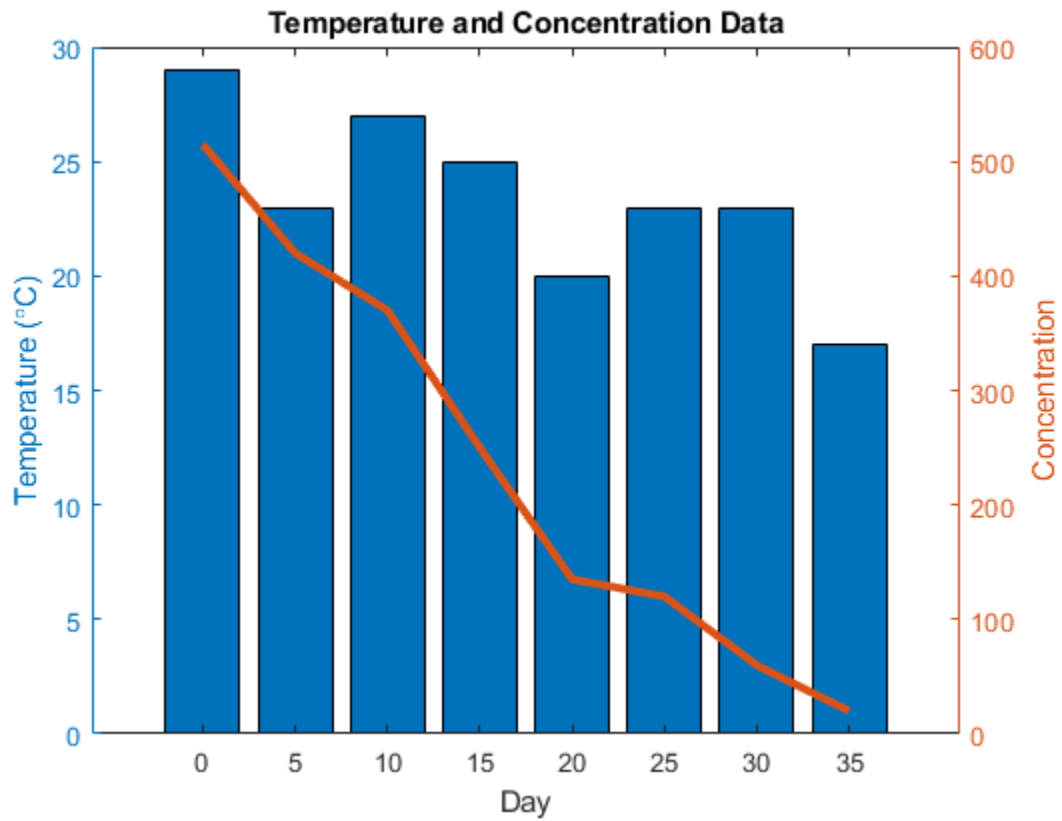
See Also

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

Properties
Bar | Line

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)
```
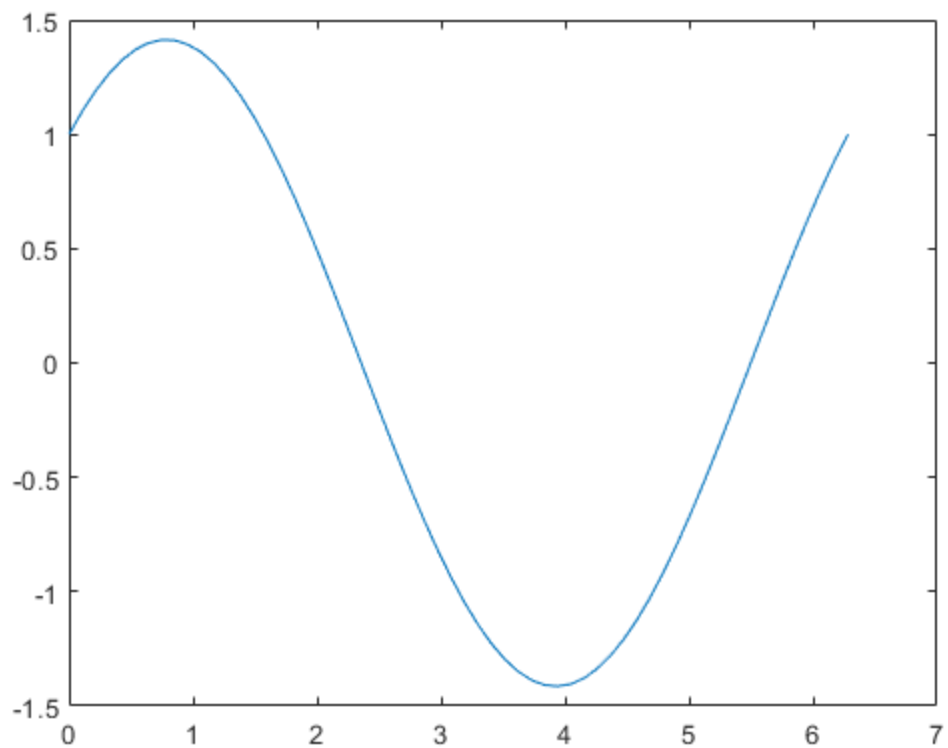

Add two stem plots to the axes. Prevent new plots from replacing existing plots using `hold on`.

```
hold on
stem(x,a)
stem(x,b)
hold off
```
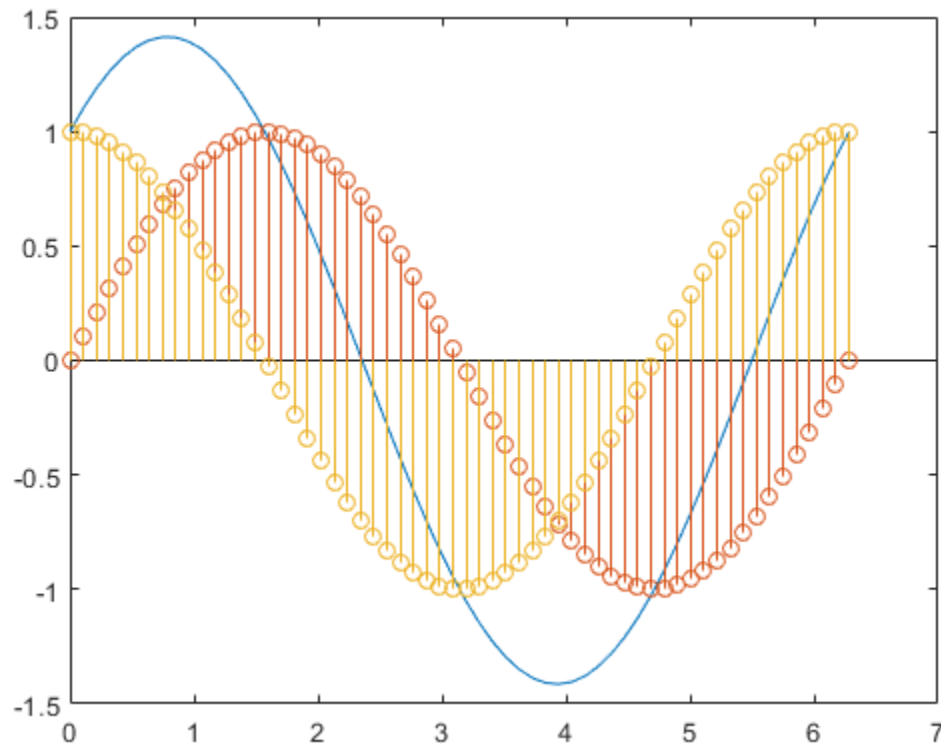
Add a title, axis labels, and a legend. Specify the legend descriptions in the order that you create the plots.

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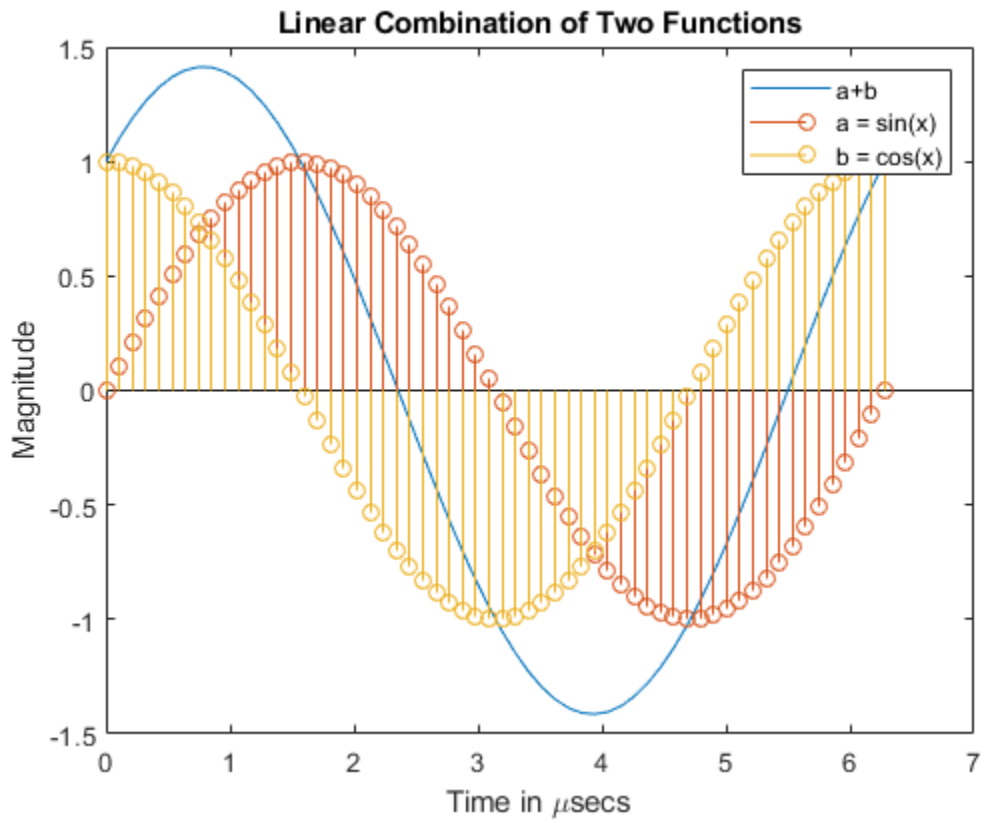
See Also
hold | plot | stem

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
```
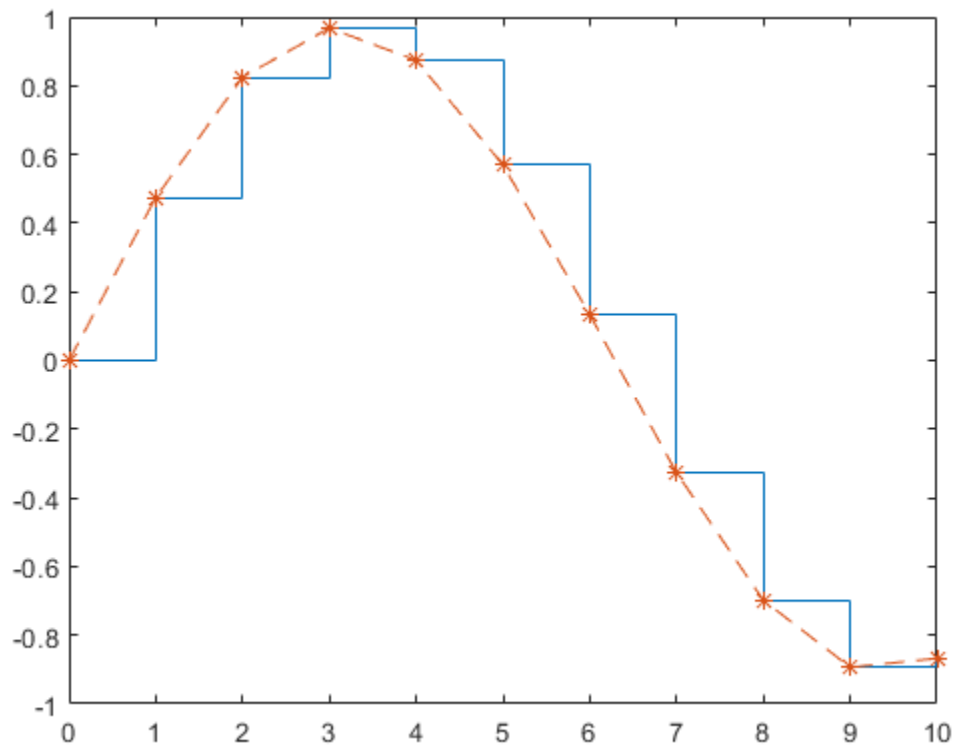

Use the `axis` function to set the axis limits. Label the *x*-axis and add a title to the graph.

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

Stairstep plot of $e^{-(\alpha*t)} \sin\beta*t$

$t = 0:10$
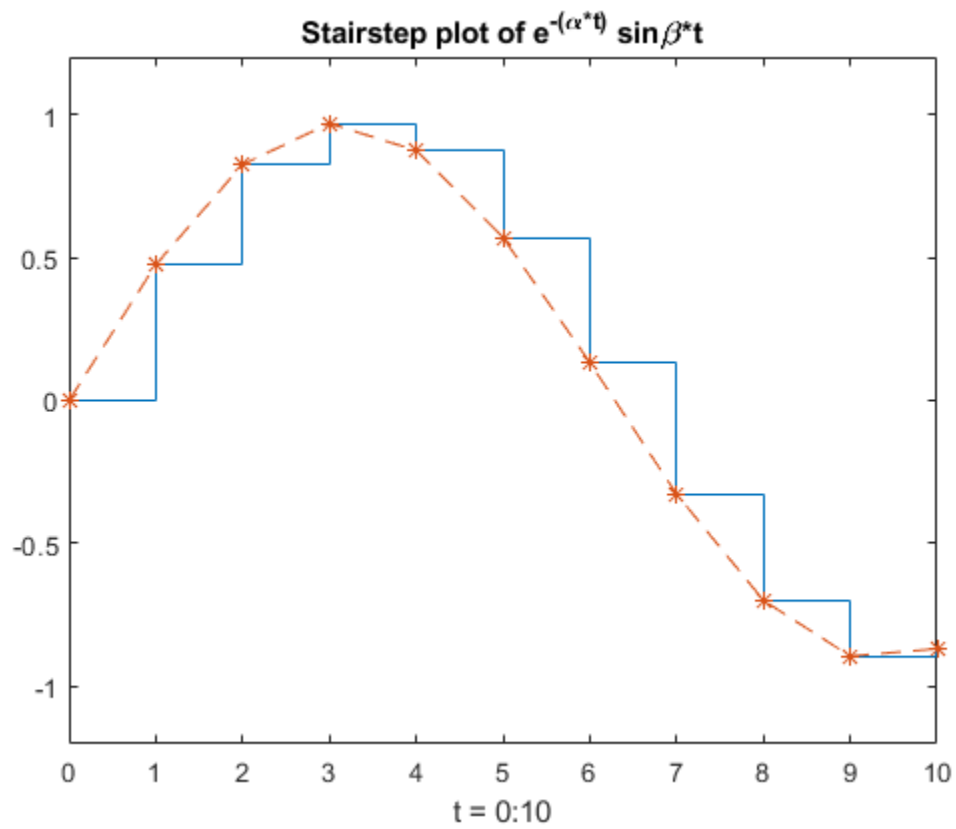
See Also
axis | plot | stairs

Line Plot with Confidence Bounds

Create a plot with confidence bounds using the `fill` function to draw the confidence bounds and the `plot` function to draw the data points. Use dot notation syntax `object.PropertyName` to customize the look of the plot.

```
x = 0:0.2:10;
y = besselj(0, x);

xconf = [x x(end:-1:1)] ;
yconf = [y+0.15 y(end:-1:1)-0.15];

figure
p = fill(xconf,yconf,'red');
p.FaceColor = [1 0.8 0.8];
p.EdgeColor = 'none';

hold on
plot(x,y,'ro')
hold off
```
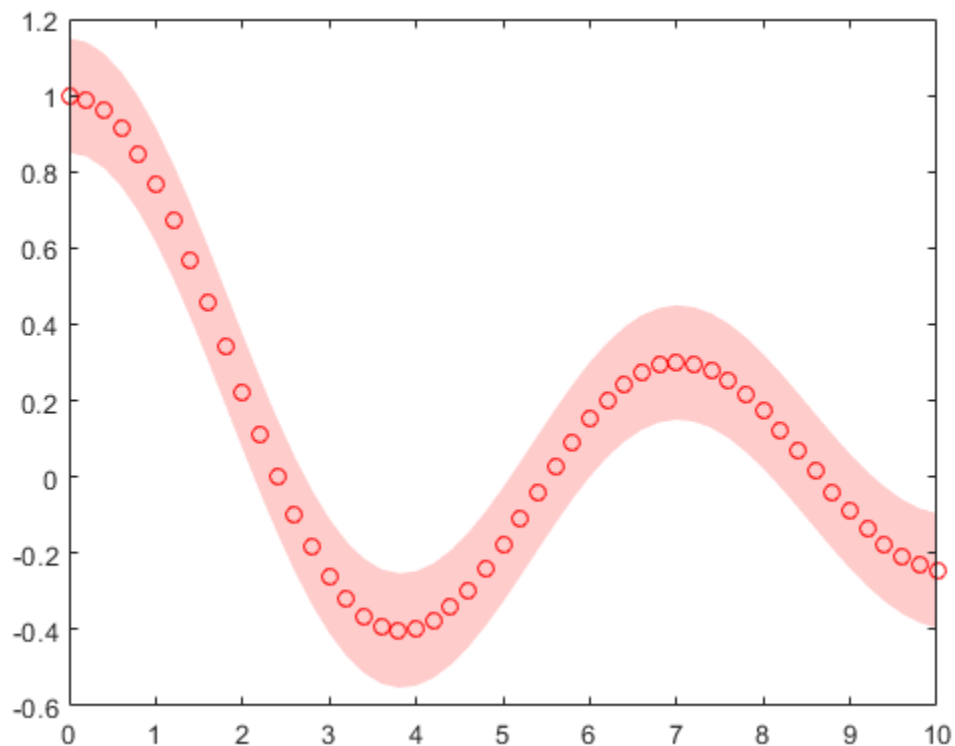
Plot Imaginary and Complex Data

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')
```
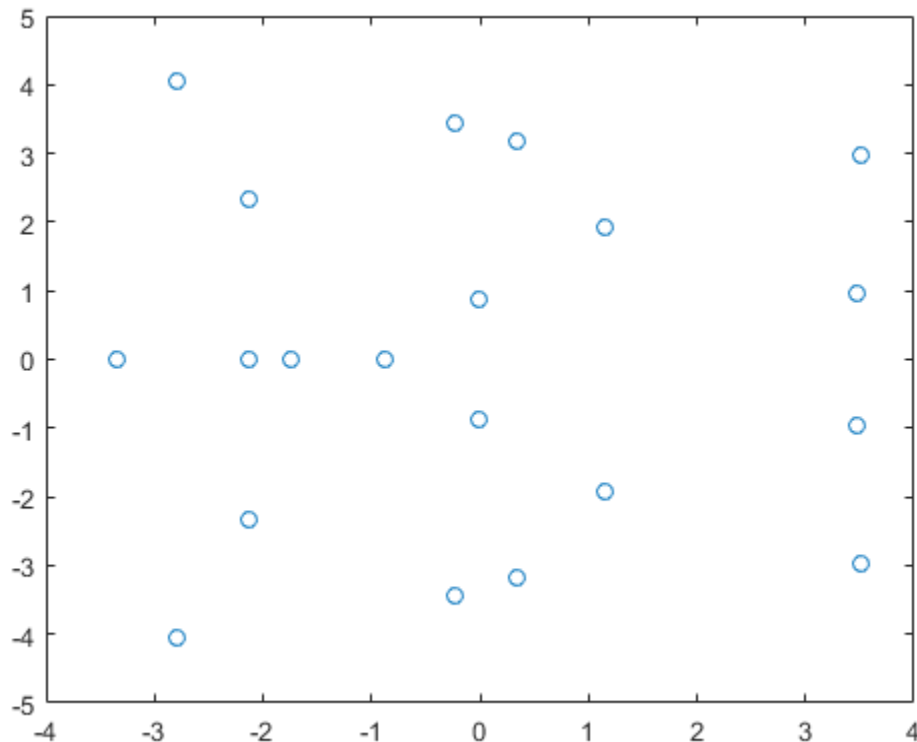

Plot Multiple Complex Inputs

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')
```
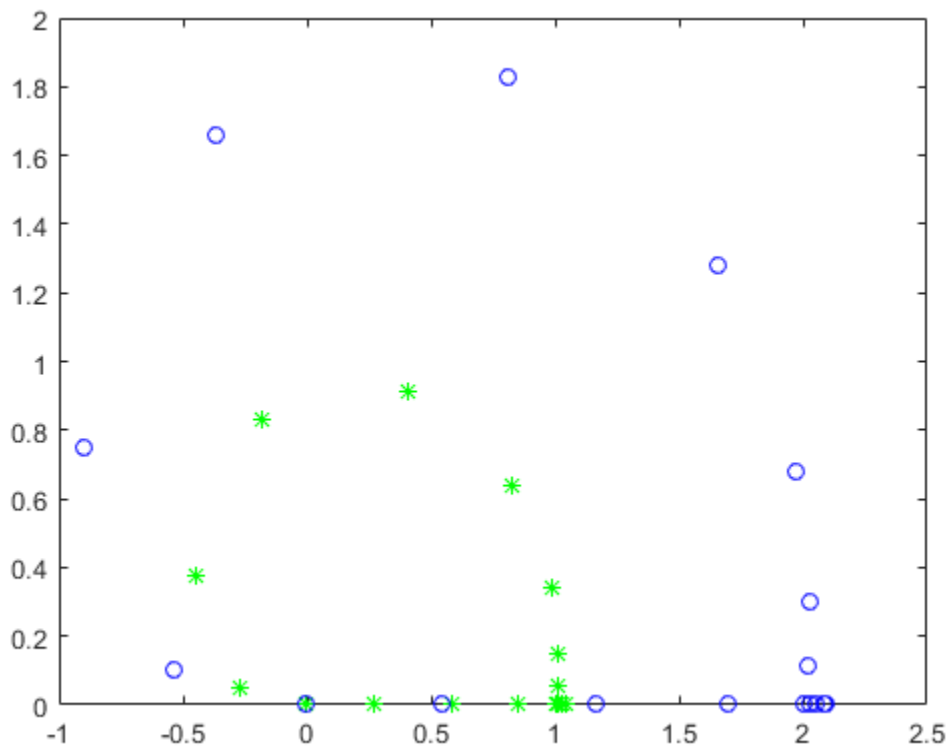
See Also
imag | plot | real

Pie Charts, Bar Plots, and Histograms

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)
```
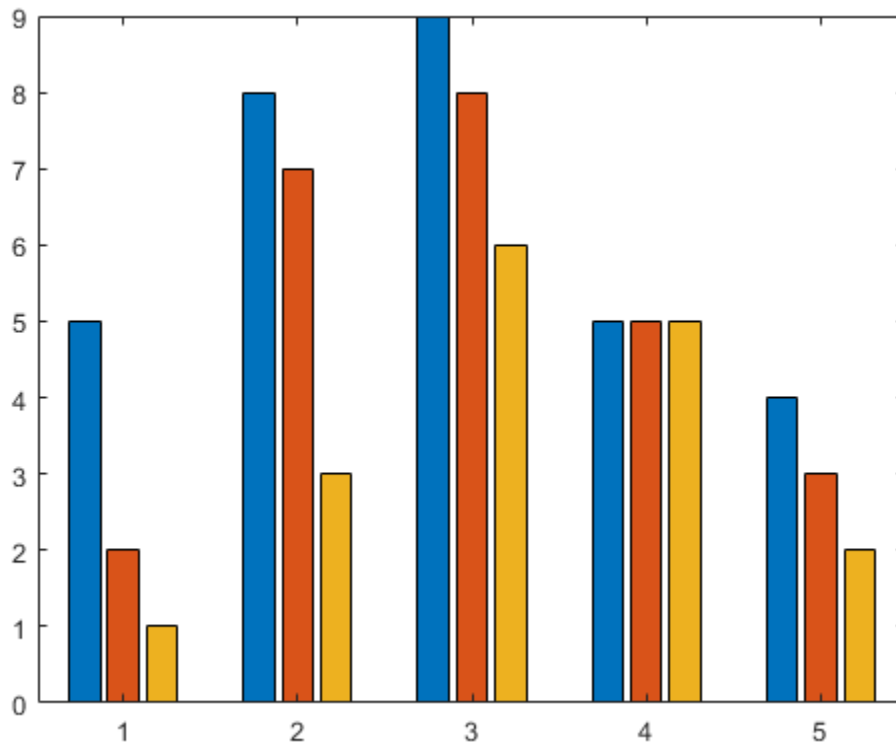

To stack the elements in a row, specify the `stacked` option for the `bar` function.

```
figure
bar(Y,'stacked')
```
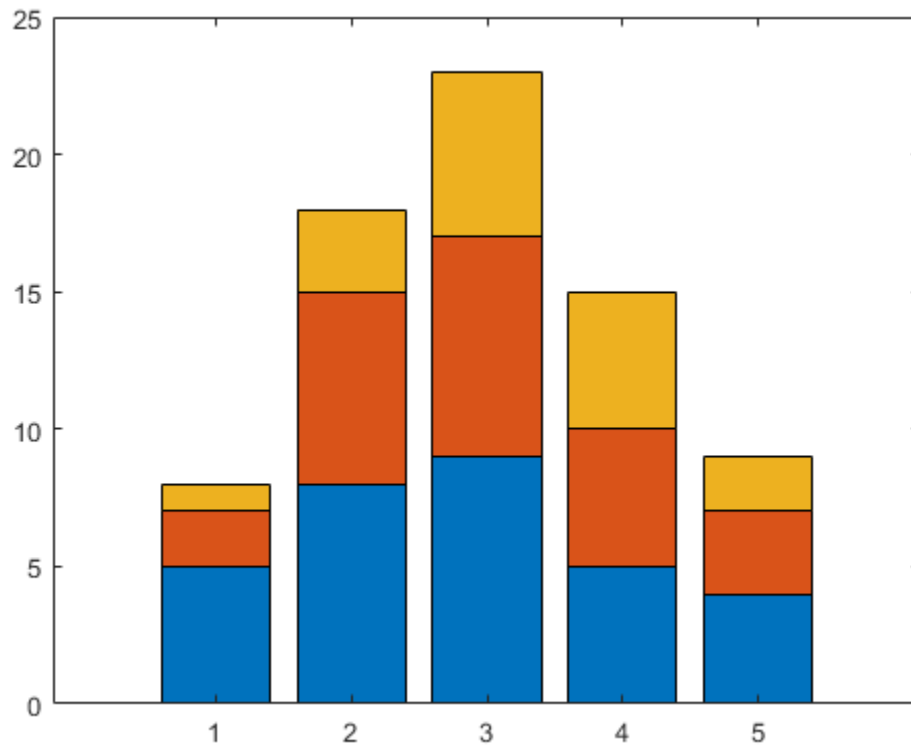
2-D Horizontal Bar Graph

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)
```
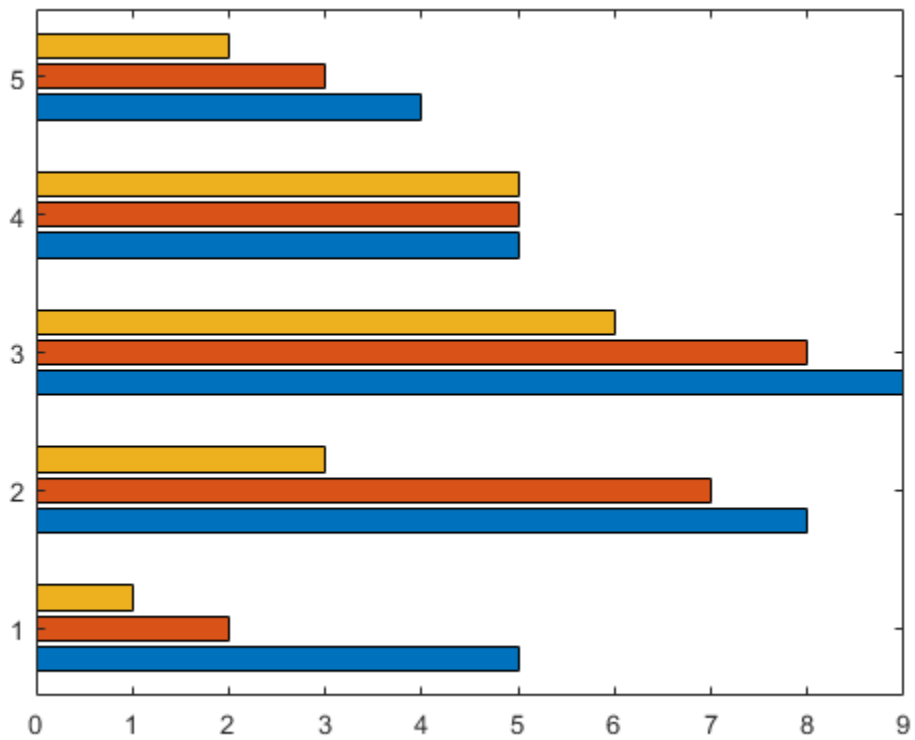
3-D Bar Graph

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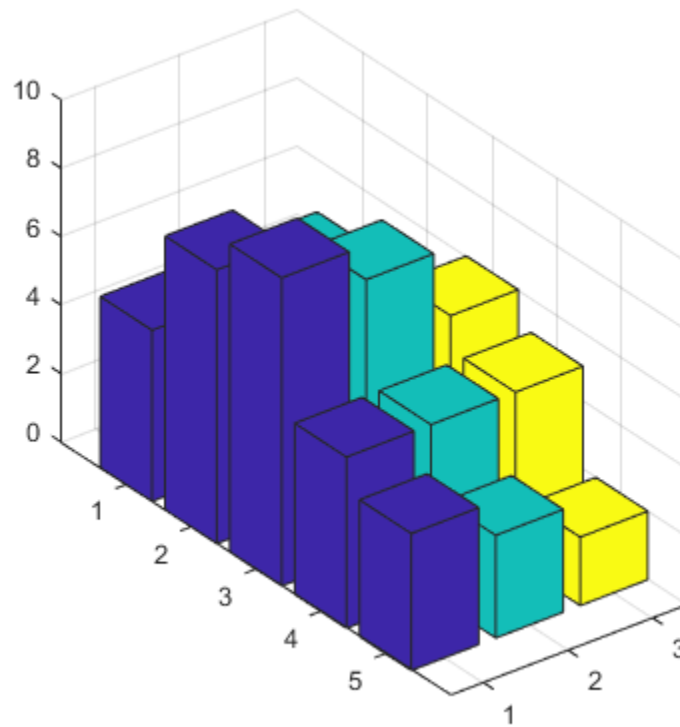
To stack the elements in a row, specify the `stacked` option for the `bar3` function.

```
figure
bar3(Y,'stacked')
```
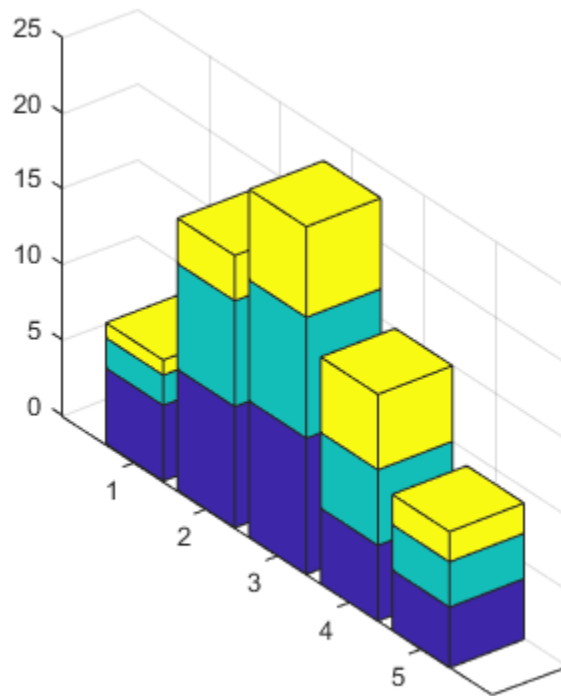
3-D Horizontal Bar Graph

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)
```
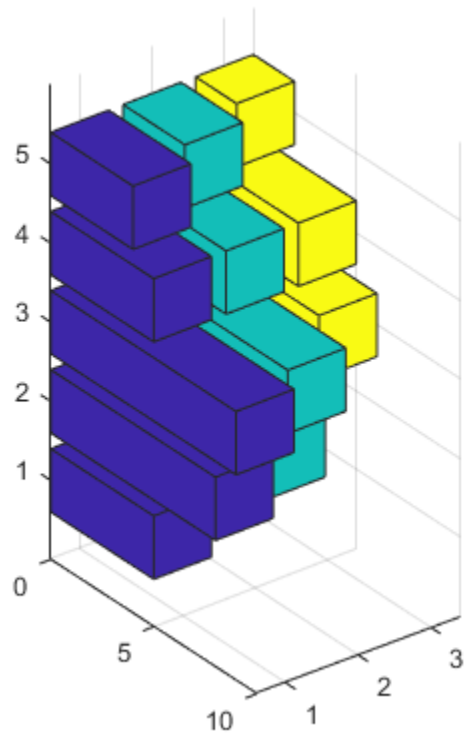
See Also
bar | bar3 | bar3h | barh

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);
```
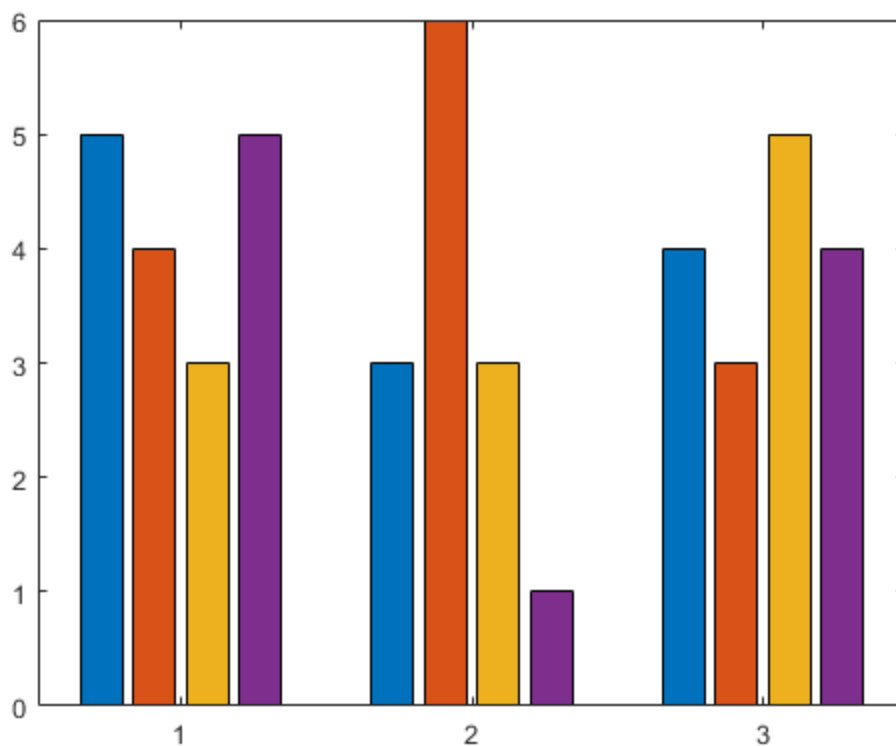

All bar series in a graph share the same baseline. Change the value of the baseline to 2 by setting the `BaseValue` property for any of the bar series. Use dot notation to set properties.

```
b(1).BaseValue = 2;
```
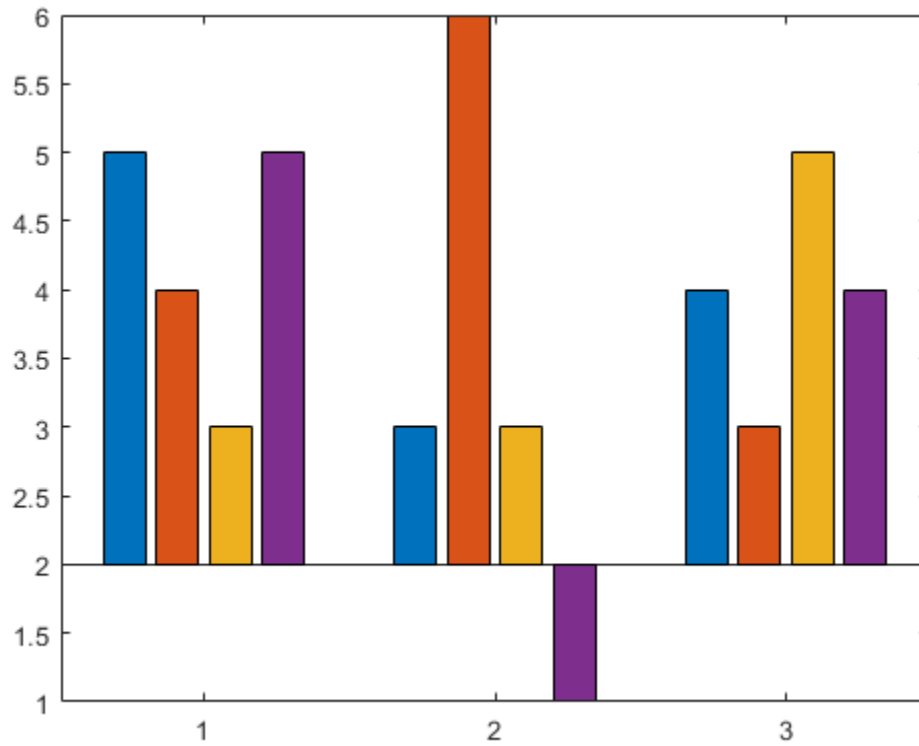
Change the baseline to a thick, red dotted line.

```
b(1).BaseLine.LineStyle = ':';
b(1).BaseLine.Color = 'red';
b(1).BaseLine.LineWidth = 2;
```
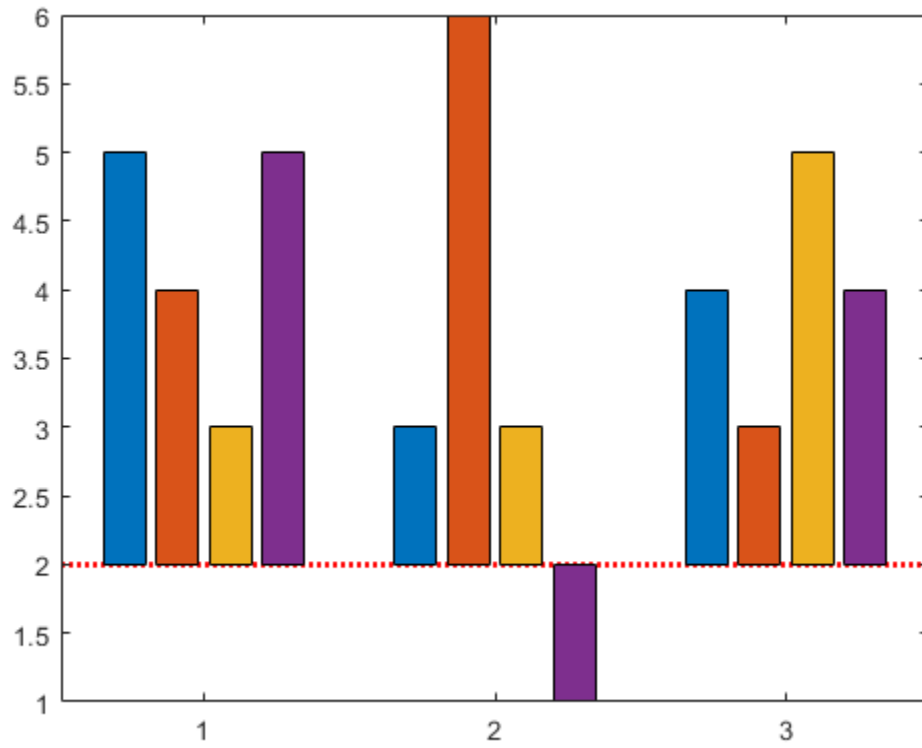
See Also
bar | barh

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])
```
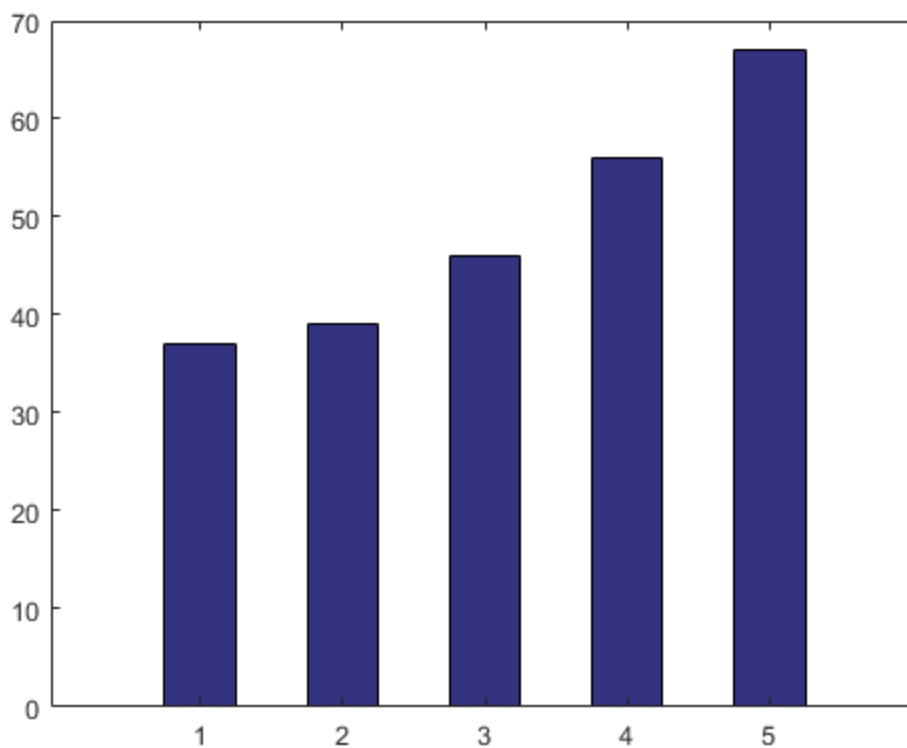

Plot a second bar graph over the first bar graph. Use the `hold` function to retain the first graph. Set the bar width to .25 so that the bars use 25% of the available space. Specify a different RGB color value for the bar color.

```
temp_low = [22 24 32 41 50];
w2 = .25;
hold on
bar(x,temp_low,w2,'FaceColor',[0 0.7 0.7])
hold off
```
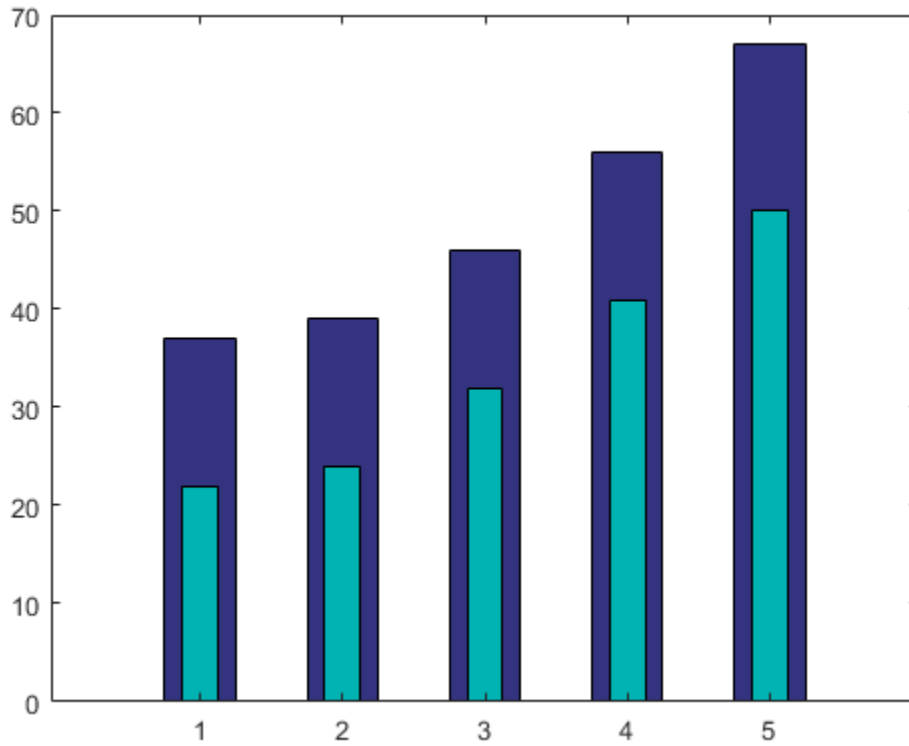
Add grid lines, a *y*-axis label, and a legend in the upper left corner. Specify the legend descriptions in the order that you create the graphs.

```
grid on
ylabel('Temperature (\circF)')
legend({'Average High','Average Low'},'Location','northwest')
```
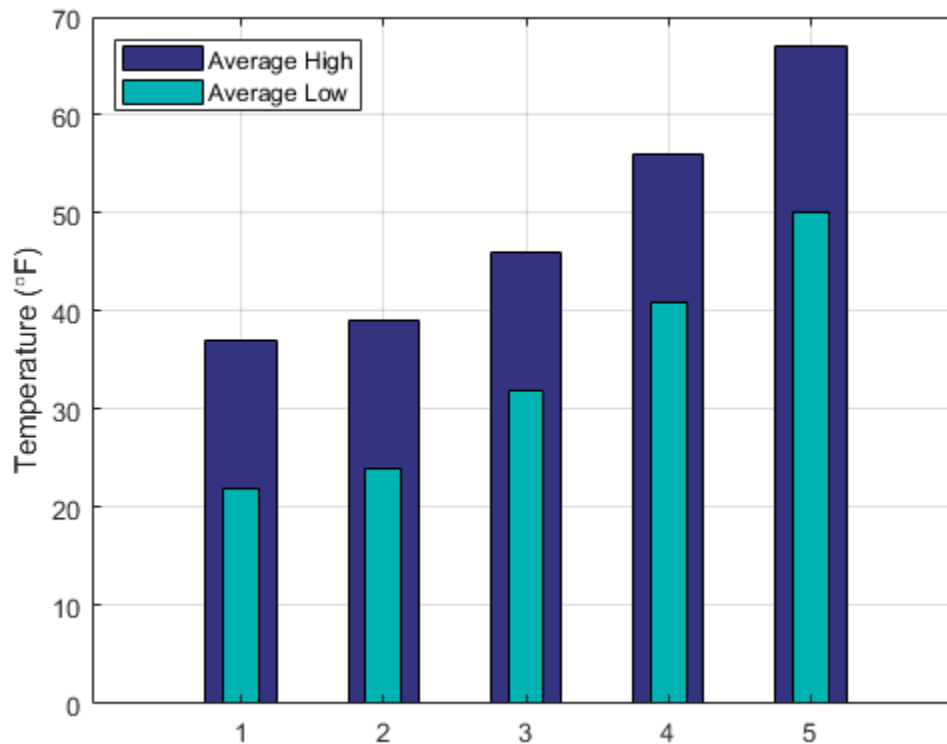
Specify the *x*-axis tick labels by setting the `XTick` and `XTickLabel` properties of the axes object. The `XTick` property specifies tick value locations along the *x*-axis. The `XTickLabel` property specifies the text to use at each tick value. Rotate the labels using the `XTickLabelRotation` property. Use dot notation to set properties.

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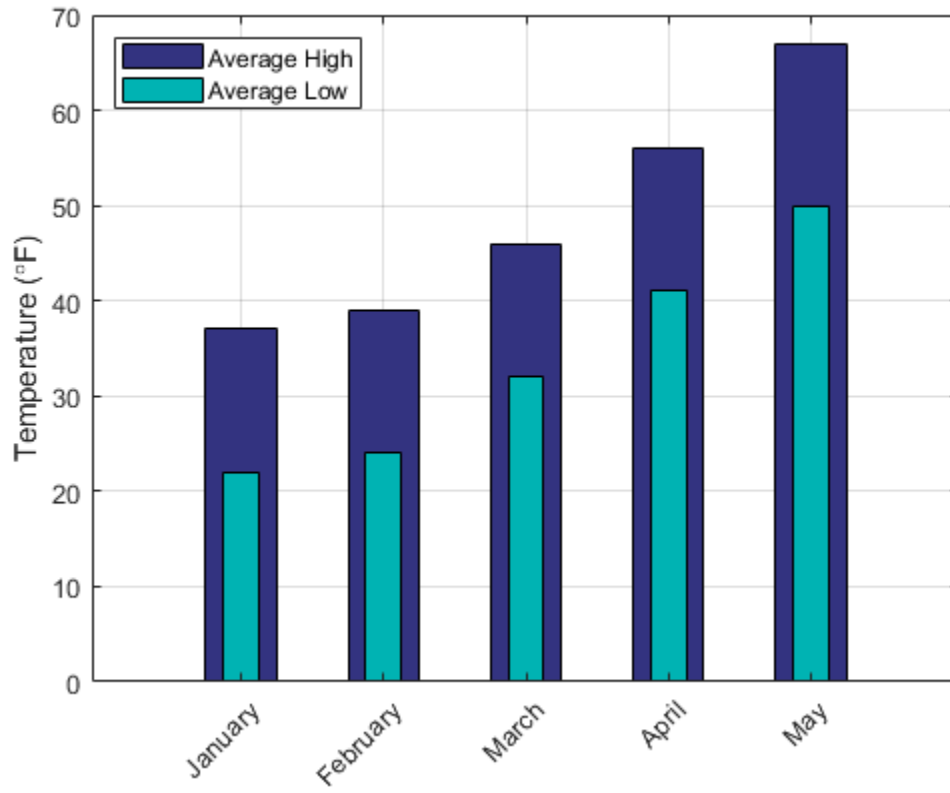
See Also
bar | barh | hold

Bar Chart with Error Bars

Create a bar chart with error bars using both the `bar` and `errorbar` functions.

```
x = 1:13;
data = [37.6 24.5 14.6 18.1 19.5 8.1 28.5 7.9 3.3 4.1 7.9 1.9 4.3]';
errhigh = [2.1 4.4 0.4 3.3 2.5 0.4 1.6 0.8 0.6 0.8 2.2 0.9 1.5];
errlow  = [4.4 2.4 2.3 0.5 1.6 1.5 4.5 1.5 0.4 1.2 1.3 0.8 1.9];

bar(x,data)

hold on

er = errorbar(x,data,errlow,errhigh);
er.Color = [0 0 0];
er.LineStyle = 'none';

hold off
```
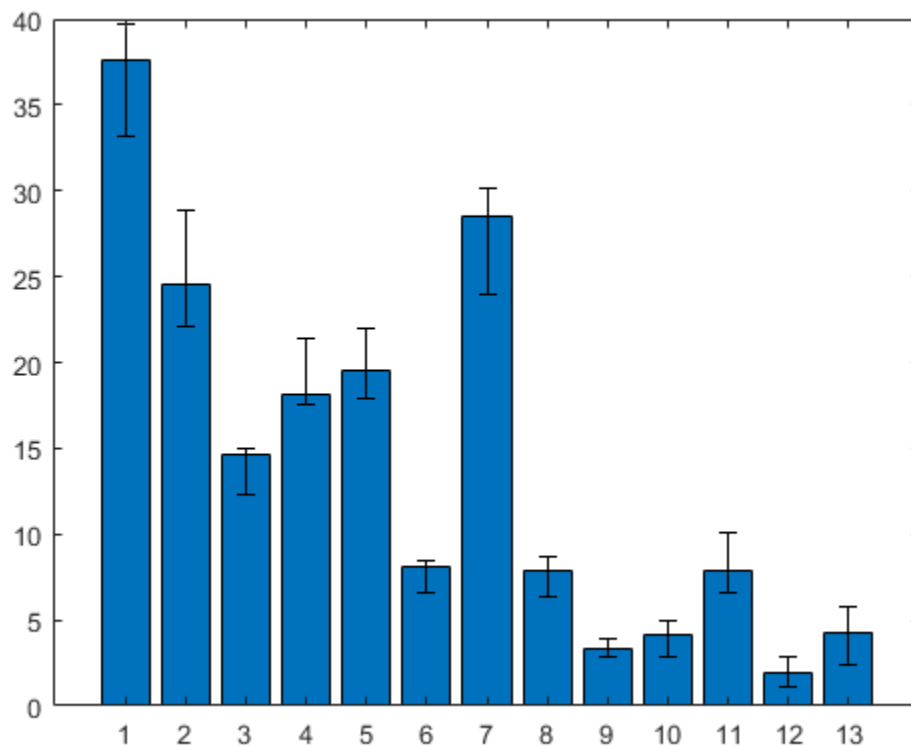

See Also
`bar | errorbar | hold`

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
```
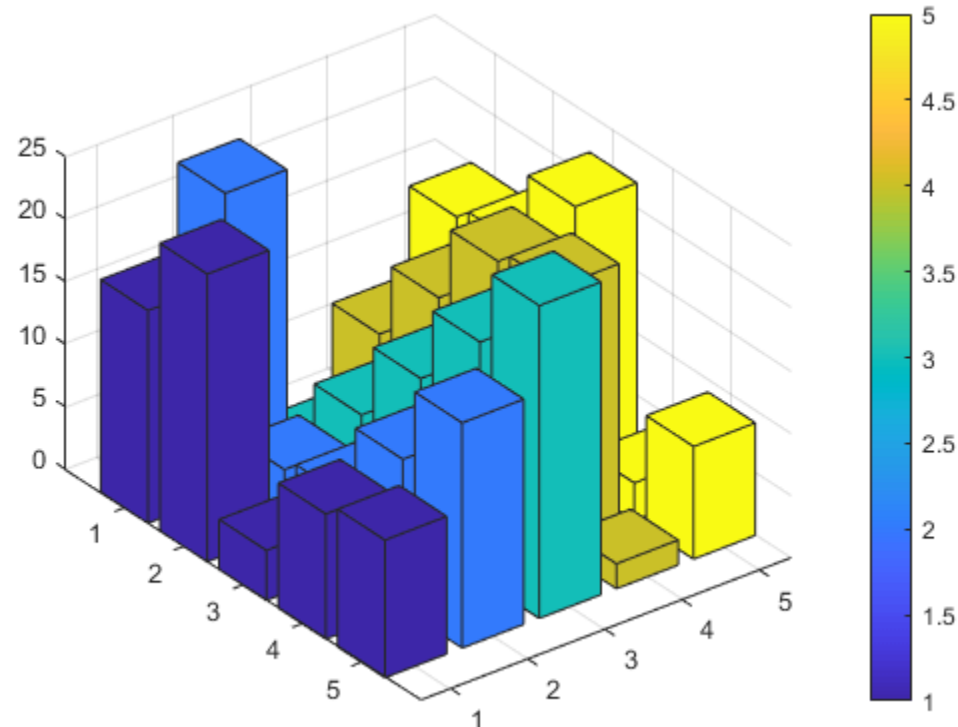

For each surface object, get the array of *z*-coordinates from the `ZData` property. Use the array to set the `CData` property, which defines the vertex colors. Interpolate the face colors by setting the `FaceColor` properties of the surface objects to `'interp'`. Use dot notation to query and set properties.

```
for k = 1:length(b)
    zdata = b(k).ZData;
    b(k).CData = zdata;
    b(k).FaceColor = 'interp';
end
```
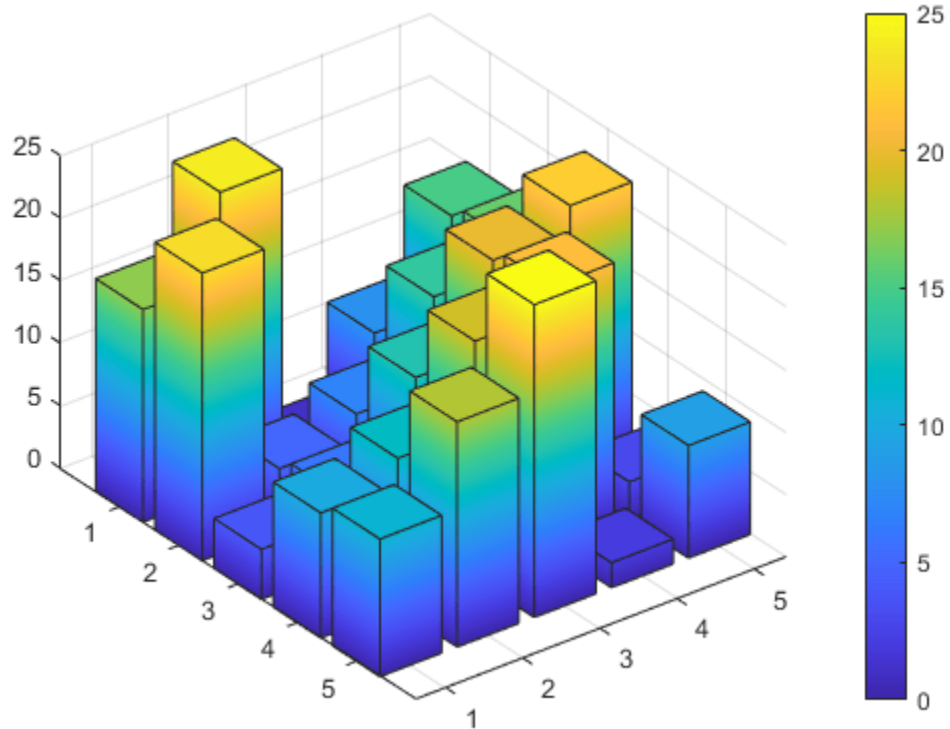
The height of each bar determines its color. You can estimate the bar heights by comparing the bar colors to the colorbar.

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])
```
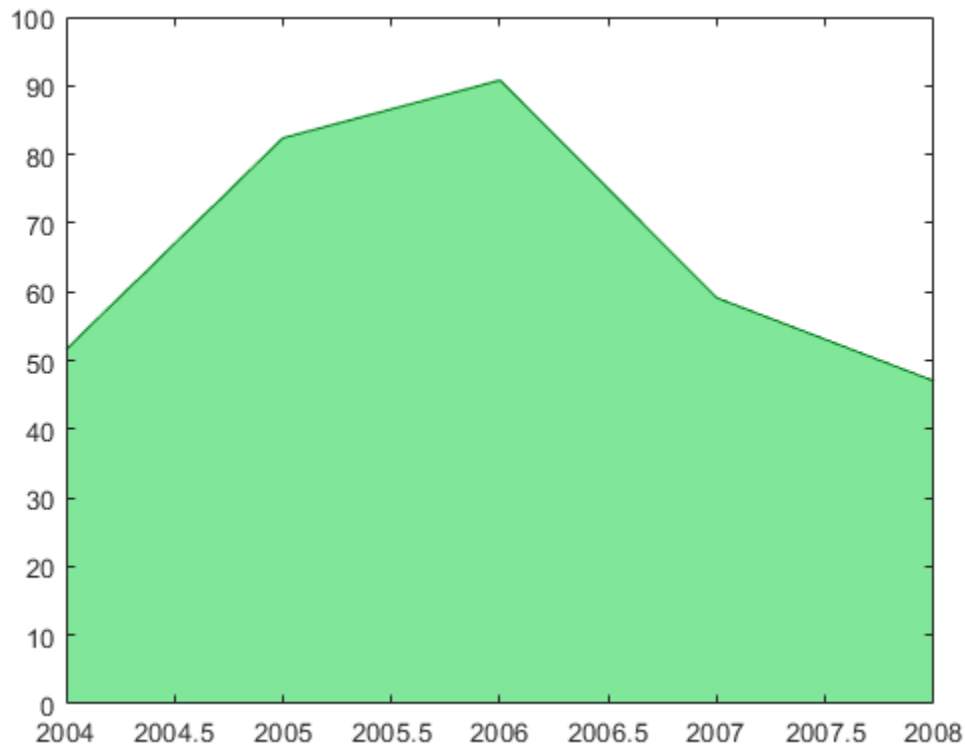

Use the `hold` command to prevent a new graph from replacing the existing graph. Plot a second area graph of `expenses`. Then, set the `hold` state back to `off`.

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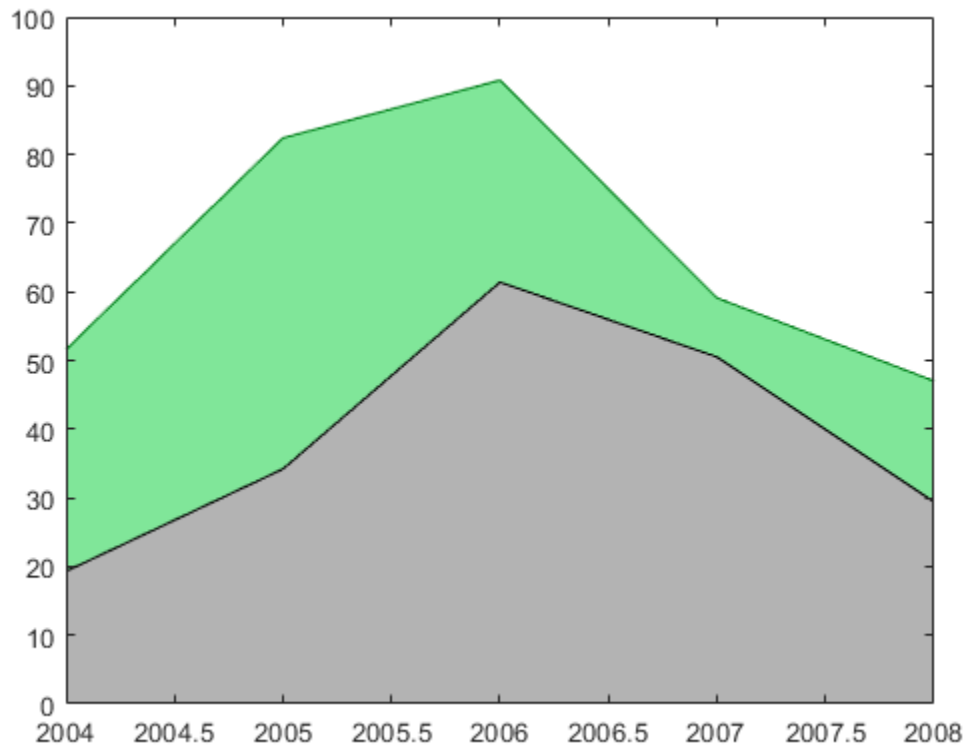
Add Grid Lines

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. Use dot notation to set properties.

```
ax = gca; % current axes
ax.XTick = years;
ax.XGrid = 'on';
ax.Layer = 'top';
```
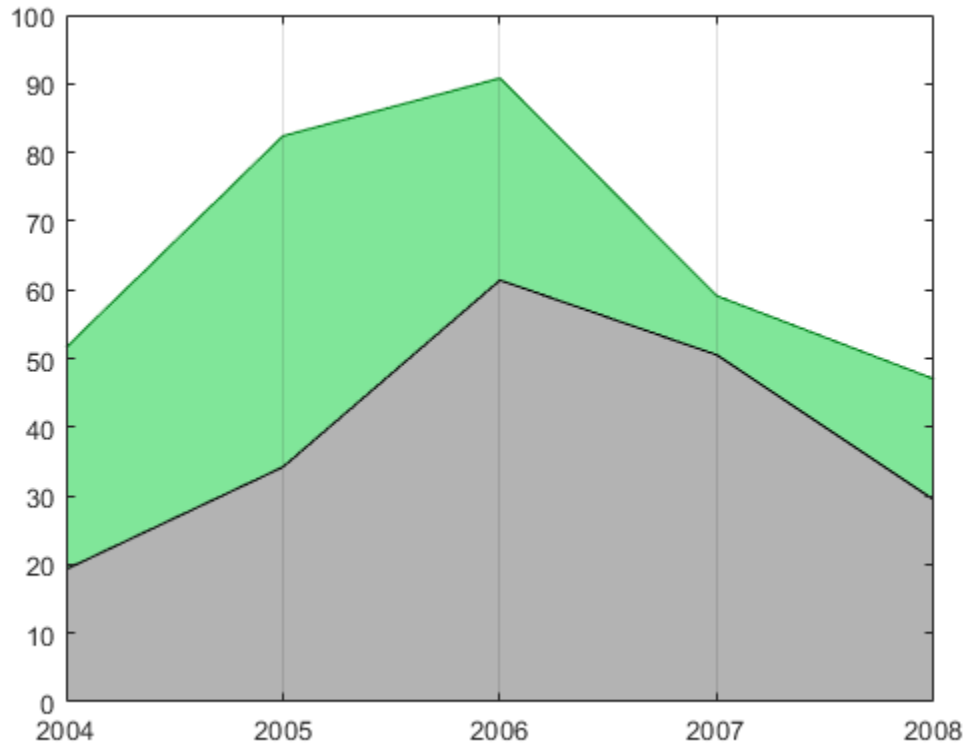
Add Title, Axis Labels, and Legend

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')
```
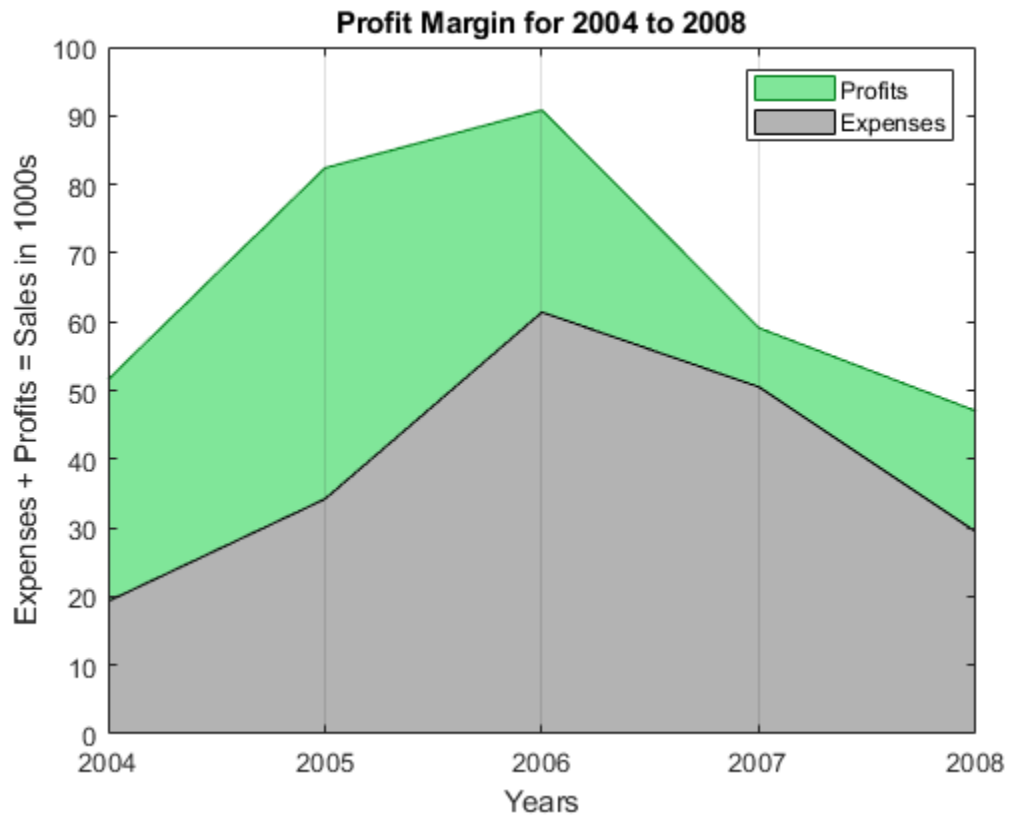
See Also
area | hold | legend

Offset Pie Slice with Greatest Contribution

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

Set up a three-column array, X, so that each column contains yearly sales data for a specific product over a 5-year period.

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

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

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

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

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

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

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

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

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

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

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

Sales Contributions of Three Products
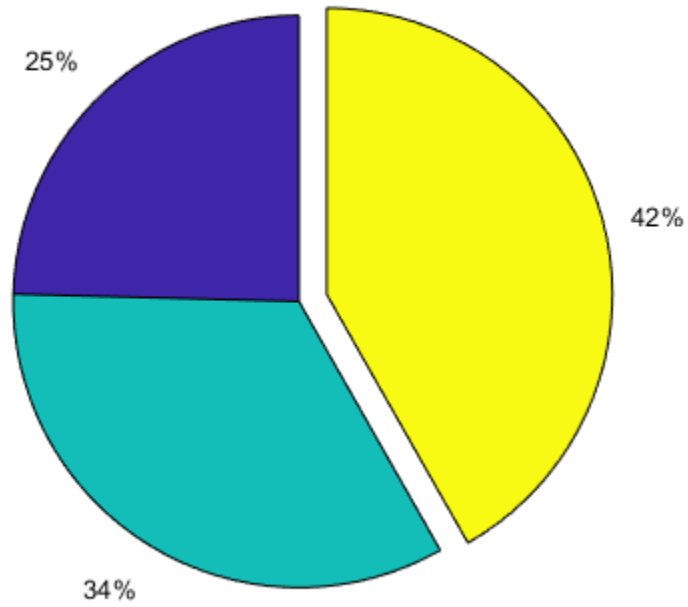
See Also

`max` | `pie` | `zeros`

Related Examples

* "Add Legend to Pie Chart" on page 2-24

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)
```
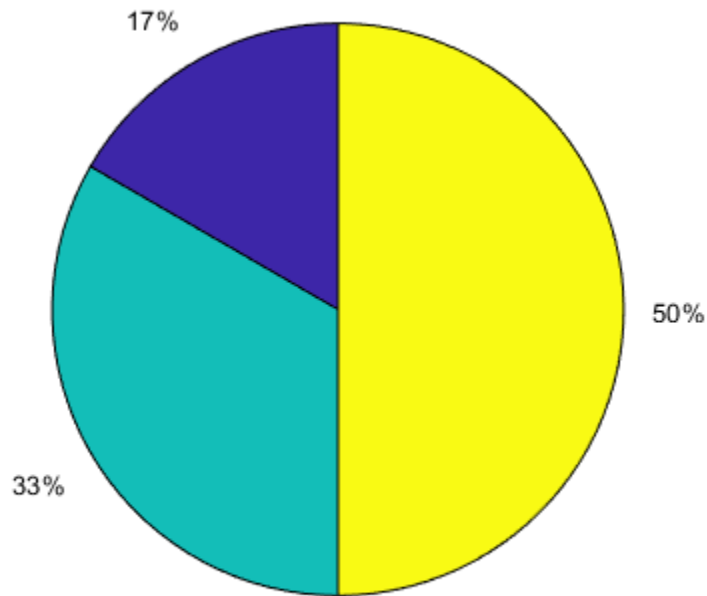

Specify the description for each pie slice in the cell array `labels`. Specify the descriptions in the order that you specified the data in x.

```
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')
```
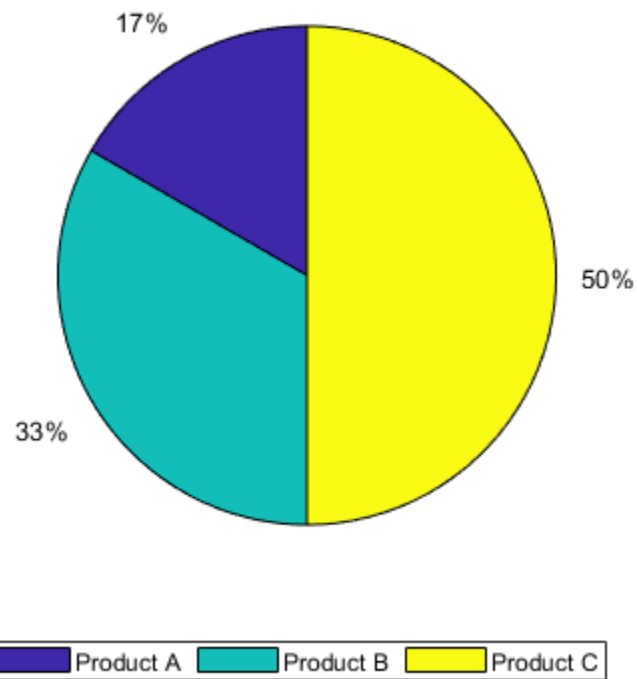
See Also

`legend` | `pie`

Related Examples

* "Offset Pie Slice with Greatest Contribution" on page 2-22

Label Pie Chart With Text and Percentages

When you create a pie chart, MATLAB labels each pie slice with the percentage of the whole that slice represents. You can change the labels to show different text.

Simple Text Labels

Create a pie chart with simple text labels.

```
x = [1,2,3];
pie(x,{'Item A','Item B','Item C'})
```


Labels with Percentages and Text

Create a pie chart with labels that contain custom text and the precalculated percent values for each slice.

Create the pie chart and specify an output argument, p, to contain the text and patch objects created by the `pie` function. The `pie` function creates one text object and one patch object for each pie slice.

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

Get the percent contributions for each pie slice from the `String` properties of the text objects.Then, specify the text that you want in the cell array `txt`. Concatenate the text with the associated percent values in the cell array `combinedtxt`.

```
pText = findobj(p,'Type','text');
percentValues = get(pText,'String');
txt = {'Item A: ';'Item B: ';'Item C: '};
combinedtxt = strcat(txt,percentValues);
```

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

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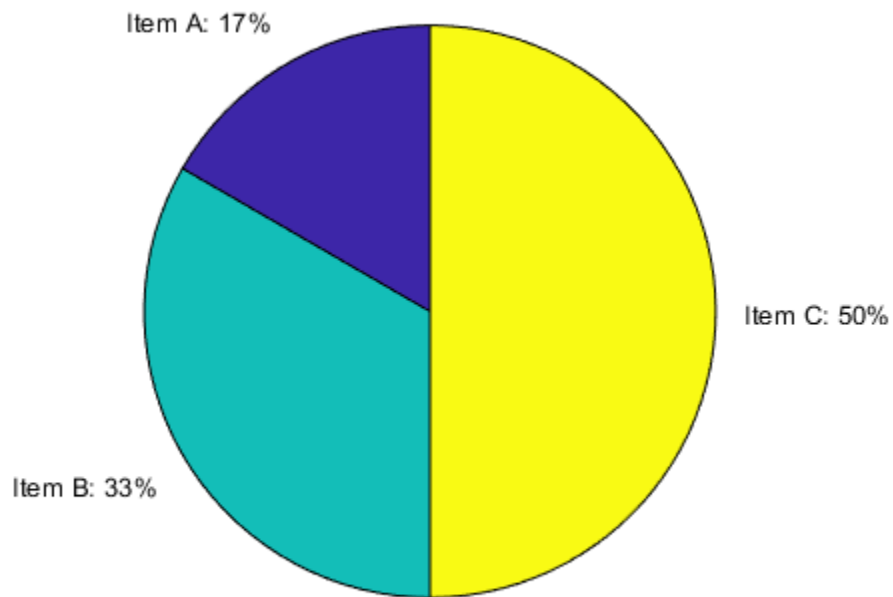

See Also

cell2mat | findobj | pie

Related Examples

- "Add Legend to Pie Chart" on page 2-24

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')
```
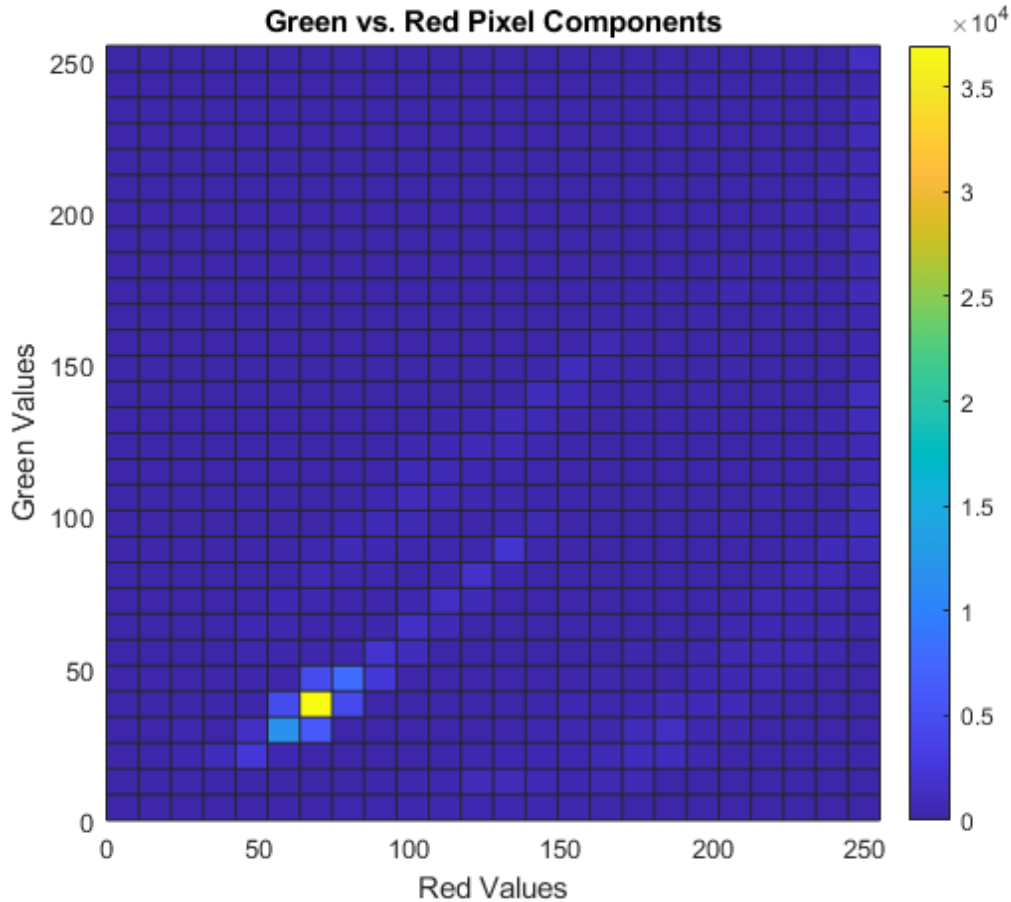

The histogram is heavily weighted towards the bottom of the color scale because there are a few bins with very large counts. This results in most of the bins displaying as the first color in the colormap, blue. Without additional detail it is hard to draw any conclusions about which color is more dominant.

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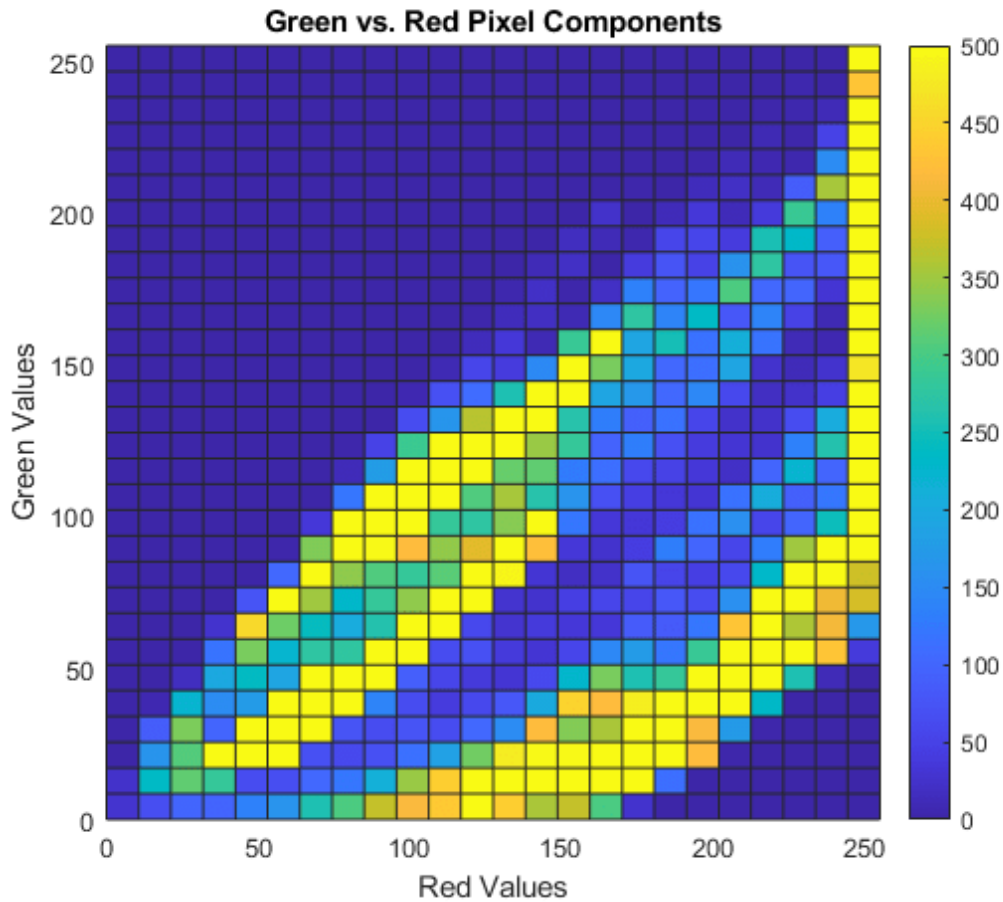
Use a similar method to compare the dominance of red vs. blue and green vs. blue.

```
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];
```
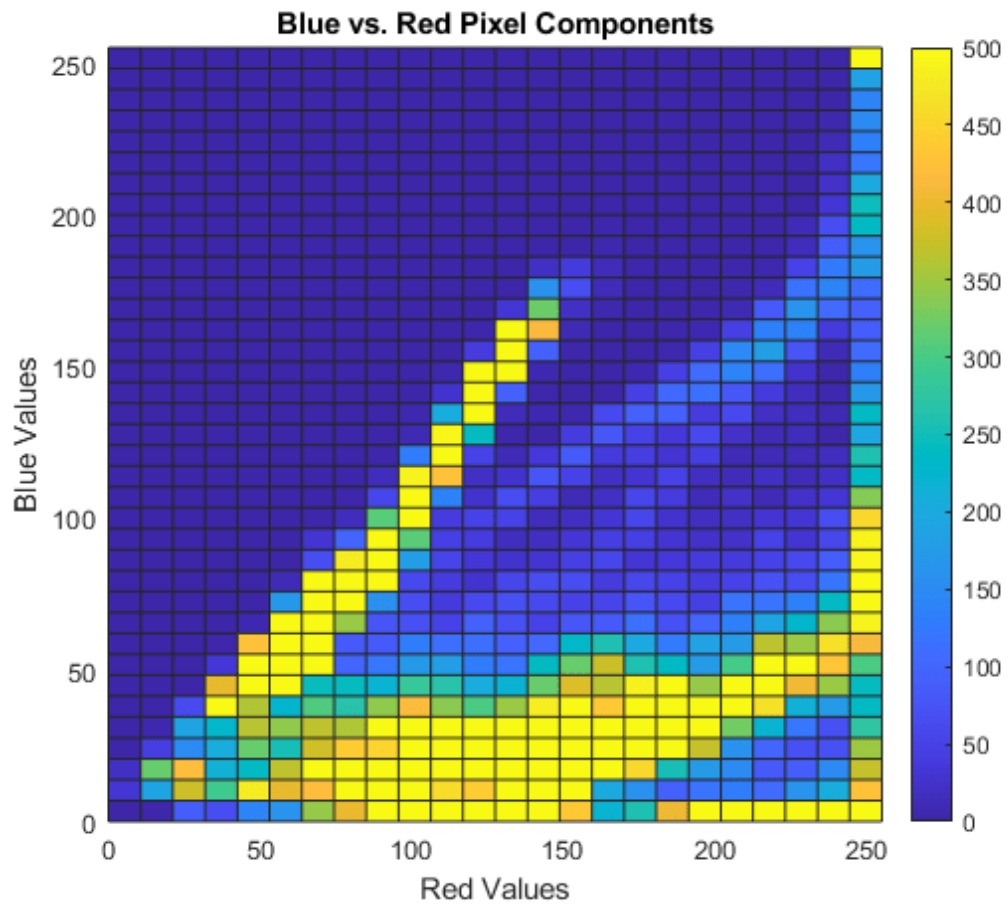
```
histogram2(g,b,'DisplayStyle','tile','ShowEmptyBins','on',...
    'XBinLimits',[0 255],'YBinLimits',[0 255]);
axis equal
colorbar
xlabel('Green Values')
ylabel('Blue Values')
title('Green vs. Blue Pixel Components')
ax = gca;
ax.CLim = [0 500];
```
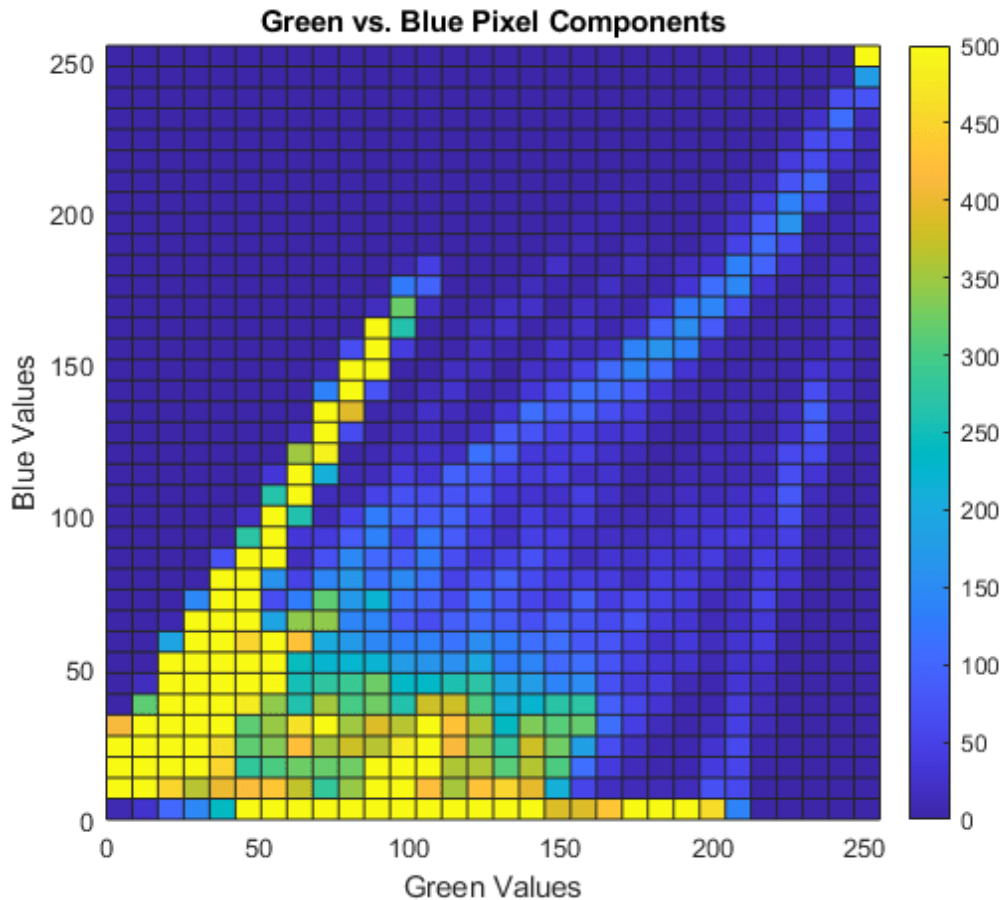
In each case, blue is the least dominant color signal. Looking at all three histograms, red appears to be the dominant color.

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])
```

Color histogram in RGB color space
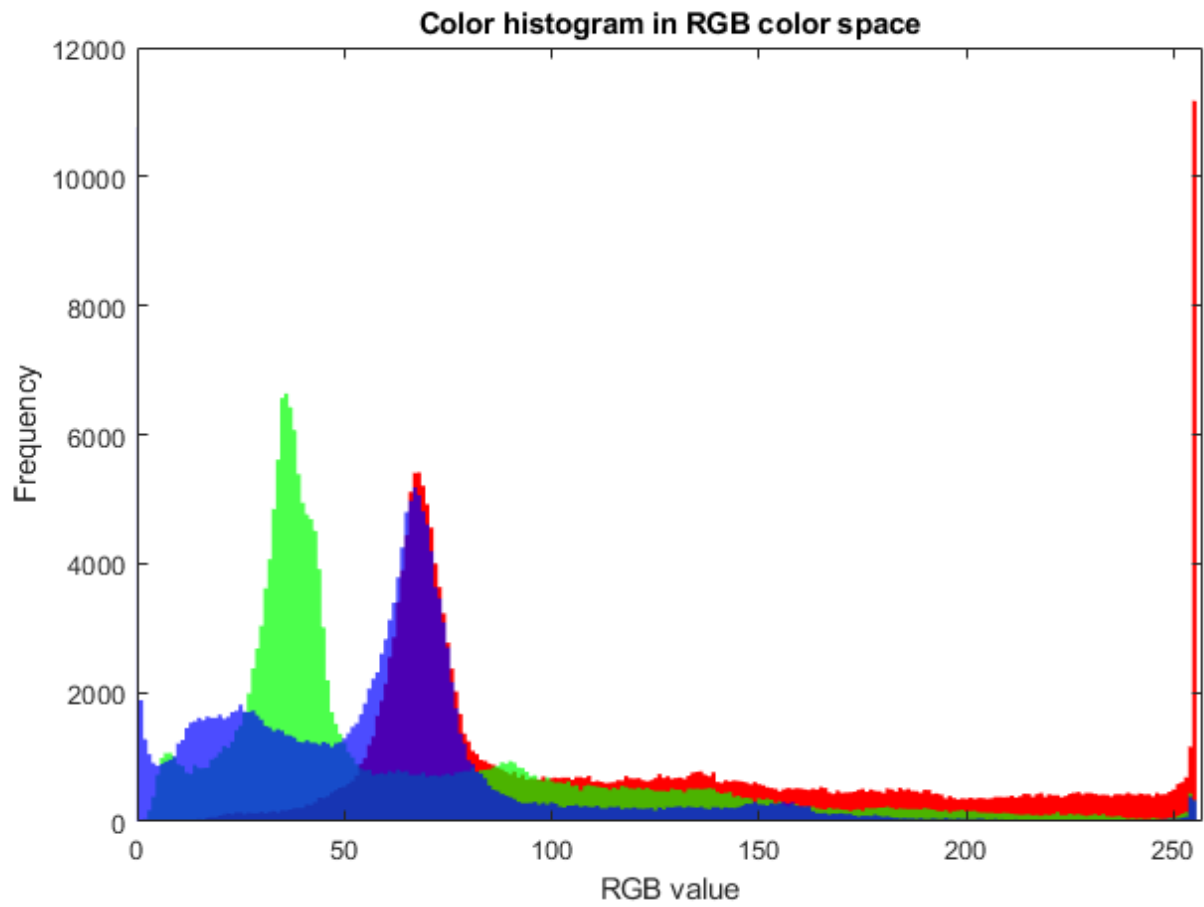
See Also

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	Cause
SouthWest	2002-02-01 12:18	458.98	1.8202e+06	2002-02-07 16:50	winter storm
SouthEast	2003-01-23 00:49	530.14	2.1204e+05	NaT	winter storm
SouthEast	2003-02-07 21:15	289.4	1.4294e+05	2003-02-17 08:14	winter storm
West	2004-04-06 05:44	434.81	3.4037e+05	2004-04-06 06:10	equipment fault
MidWest	2002-03-16 06:18	186.44	2.1275e+05	2002-03-18 23:23	severe storm
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	equipment fault
West	2002-06-06 19:28	311.86	NaN	2002-06-07 00:51	equipment fault
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	equipment fault

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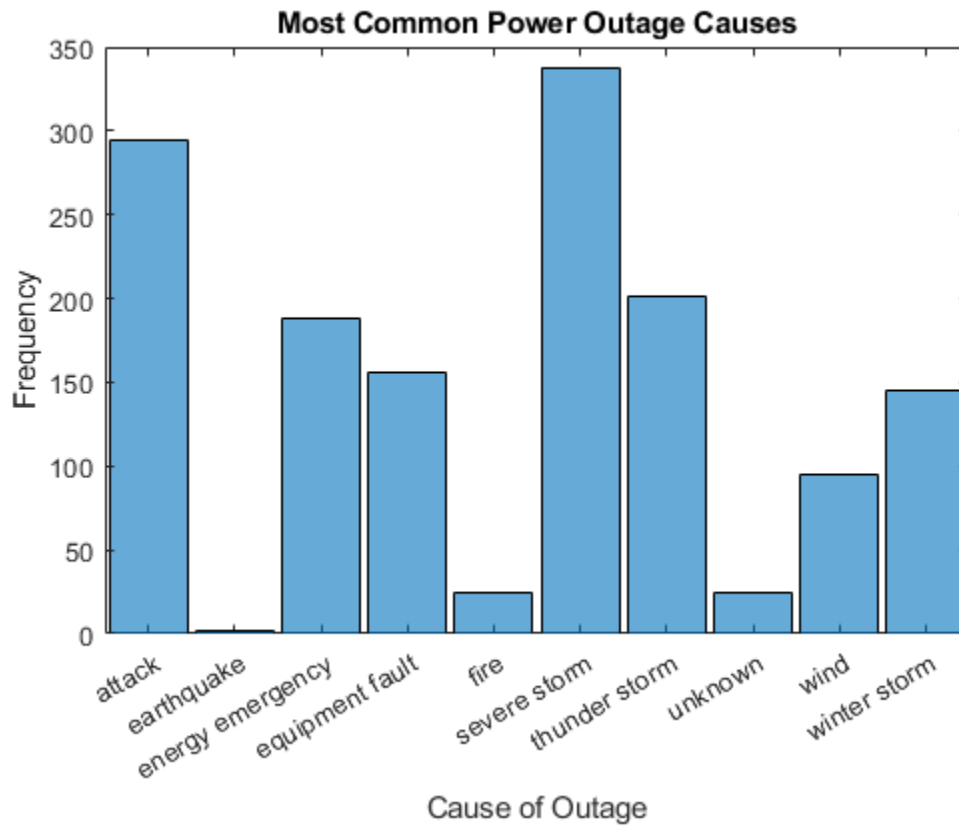
Change the normalization of the histogram to use the `'probability'` normalization, which displays the relative frequency of each outage cause.

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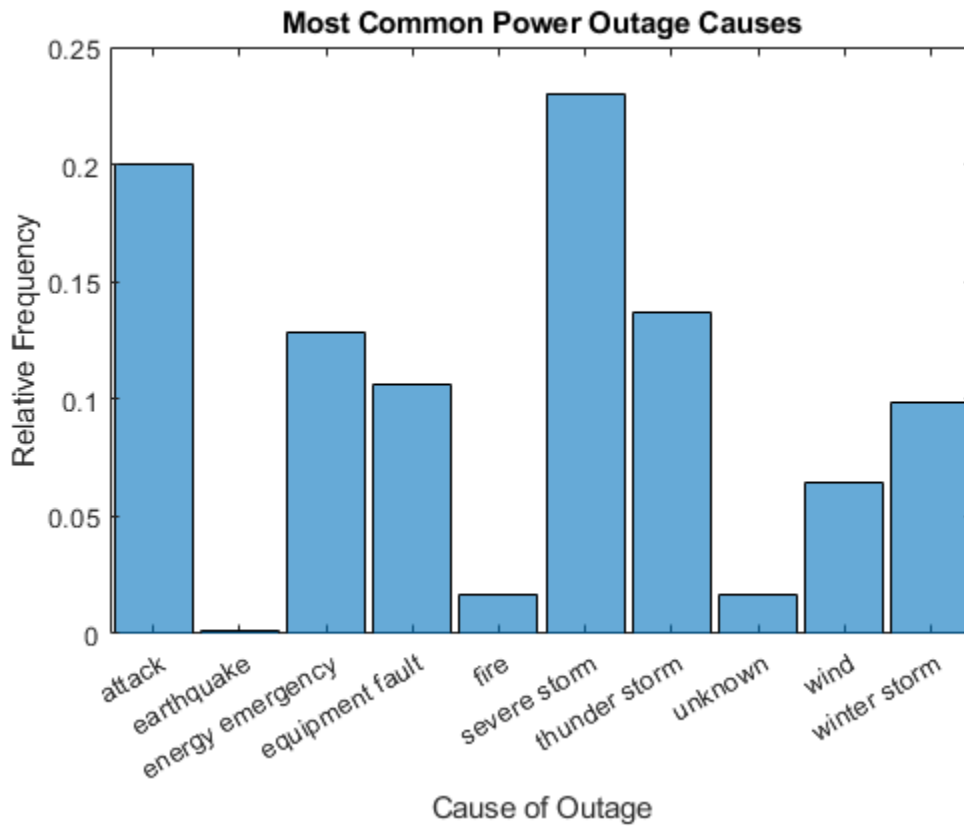
Change Display Order

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

```
h.DisplayOrder = 'descend';
```
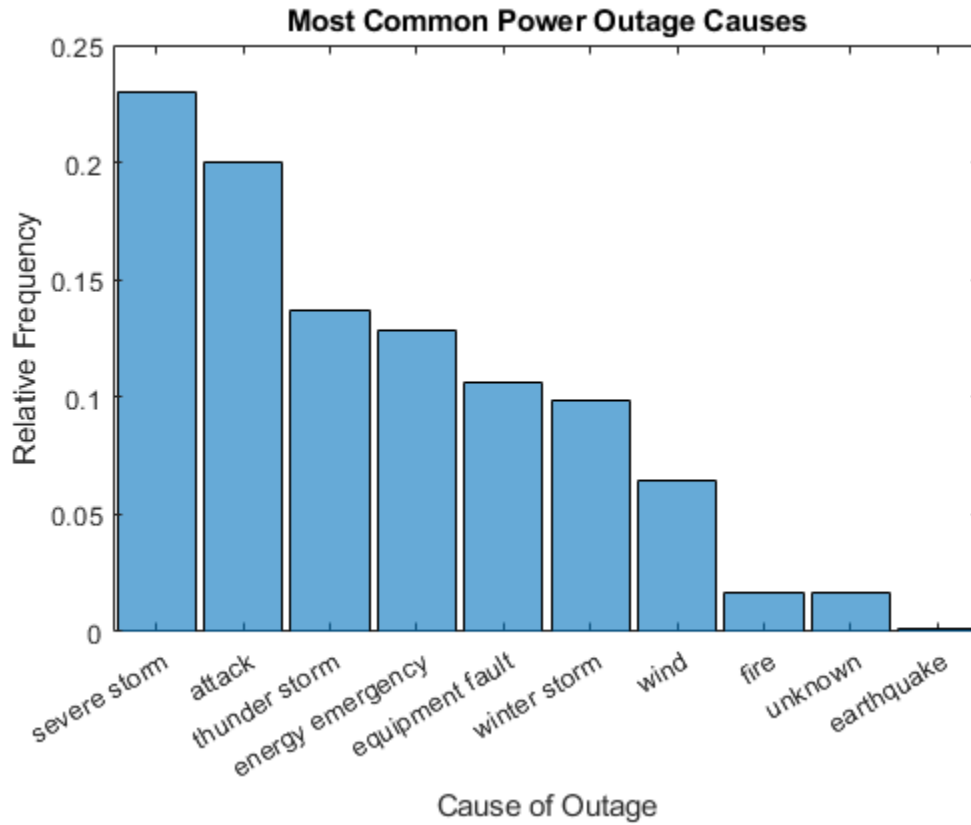
Truncate Number of Bars Displayed

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;
```

Most Common Power Outage Causes
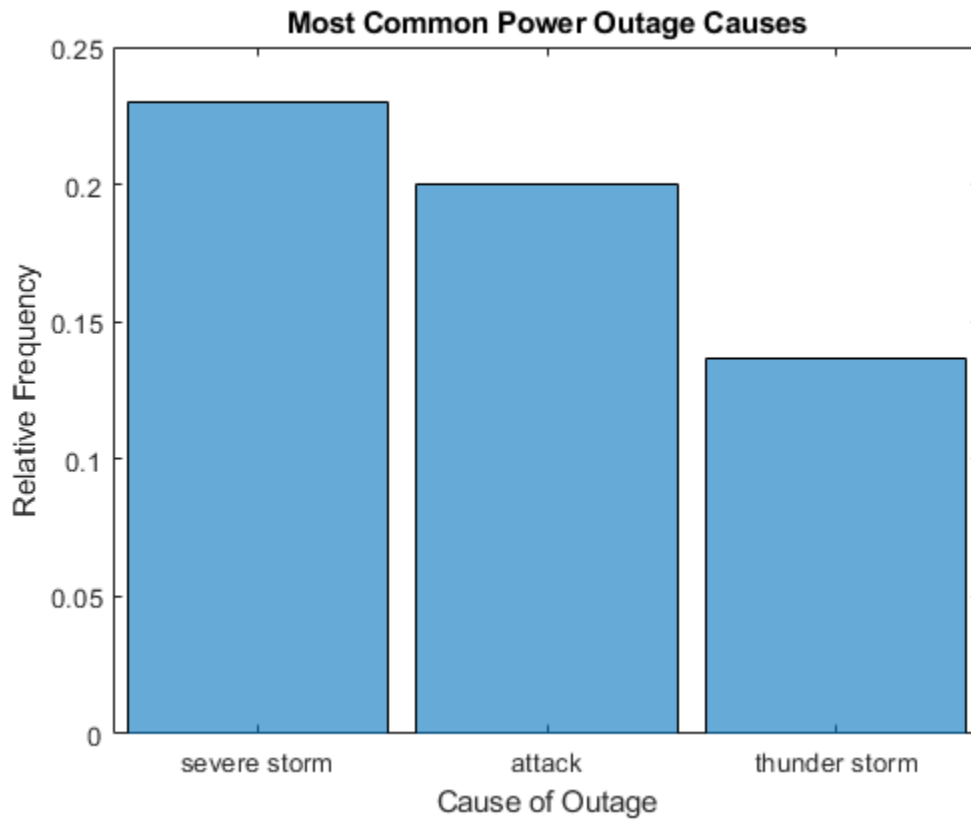
Summarize Excluded Data

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

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

Most Common Power Outage Causes
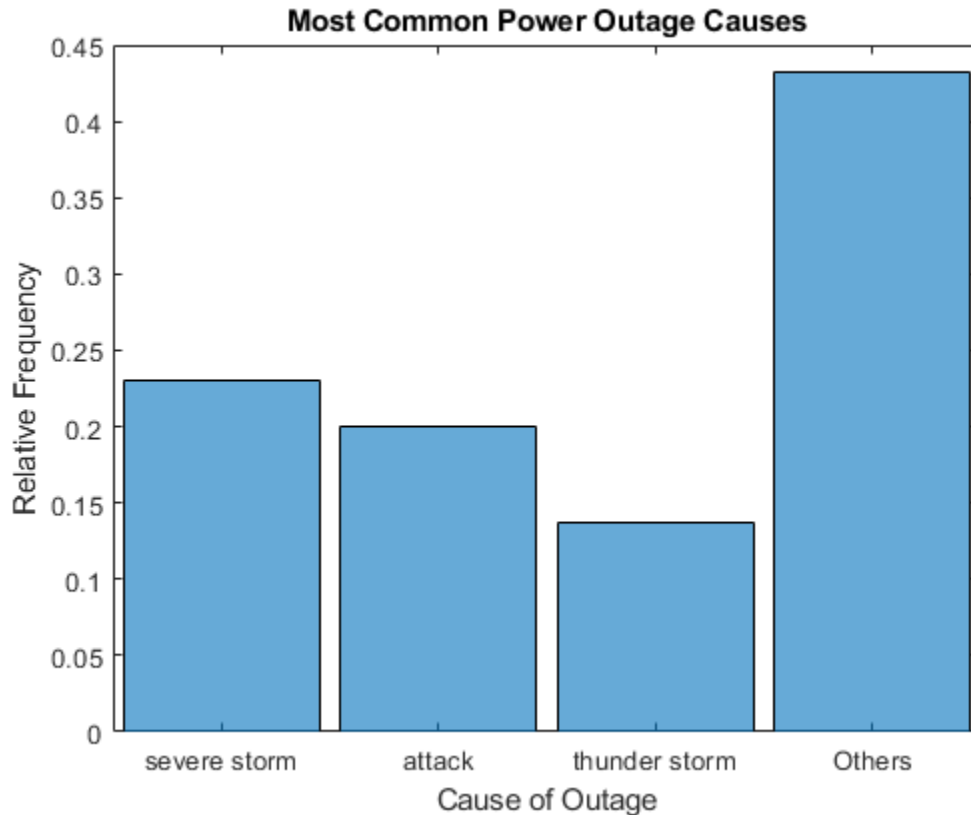
Limit Normalization to Display Data

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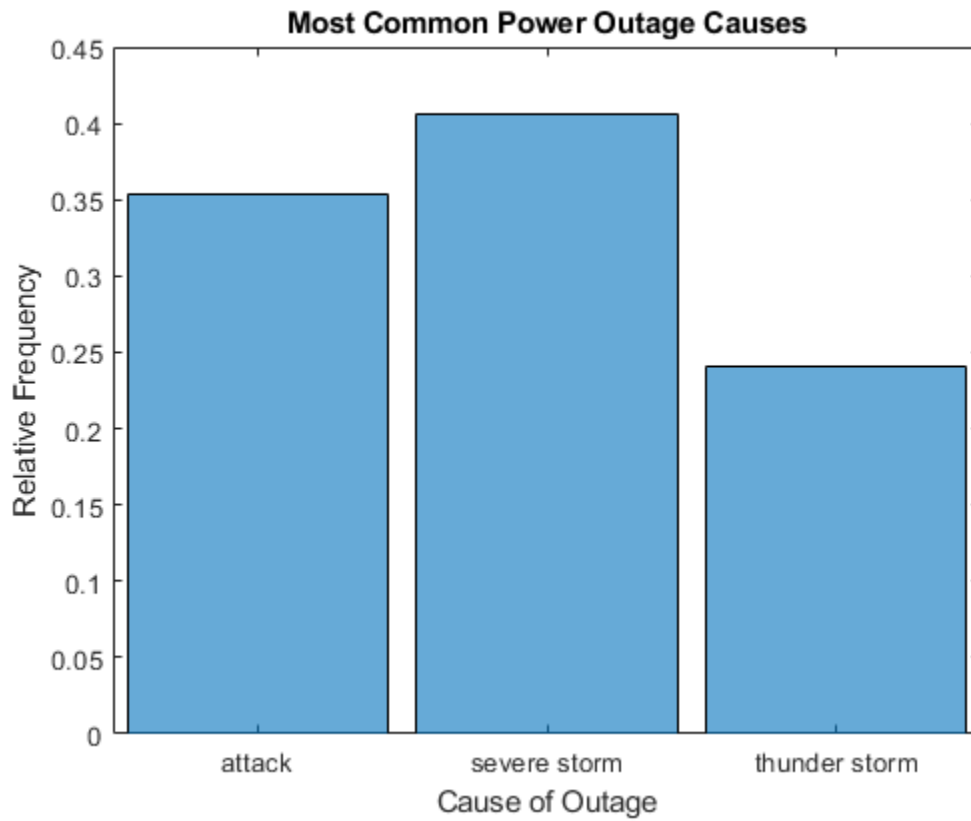
The normalization is now based only on the three remaining categories, so the three bars add to 1.

See Also

`categorical` | `histogram` | histogram

Replace Discouraged Instances of hist and histc

In this section...
"Old Histogram Functions (hist, histc)" on page 2-41
"Recommended Histogram Functions" on page 2-41
"Differences Requiring Code Updates" on page 2-41

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 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 2-46. --- **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],'BinMeth` ---

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

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`
		
		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) max(`

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.

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

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, 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','integers'`

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

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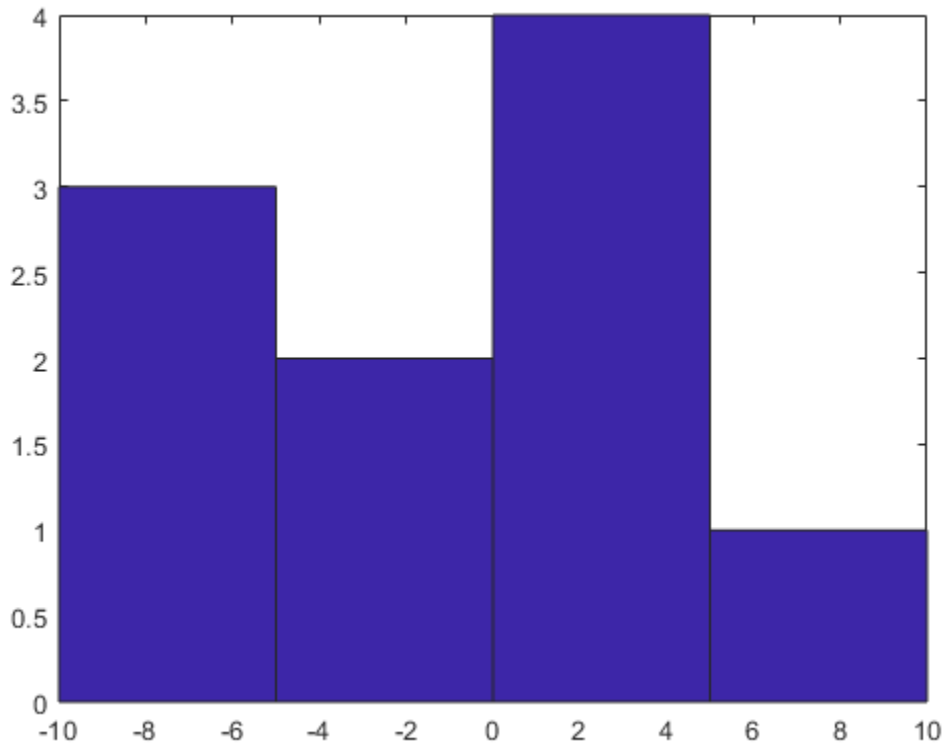
To convert the bin centers into bin edges, calculate the midpoint between consecutive values in centers. This method reproduces the results of hist for both uniform and nonuniform bin widths.

```
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 = *1×5*

```
 -10.0000   -5.0000    0.0000    5.0000   10.0000
```

Now, use histogram with the bin edges.

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

3
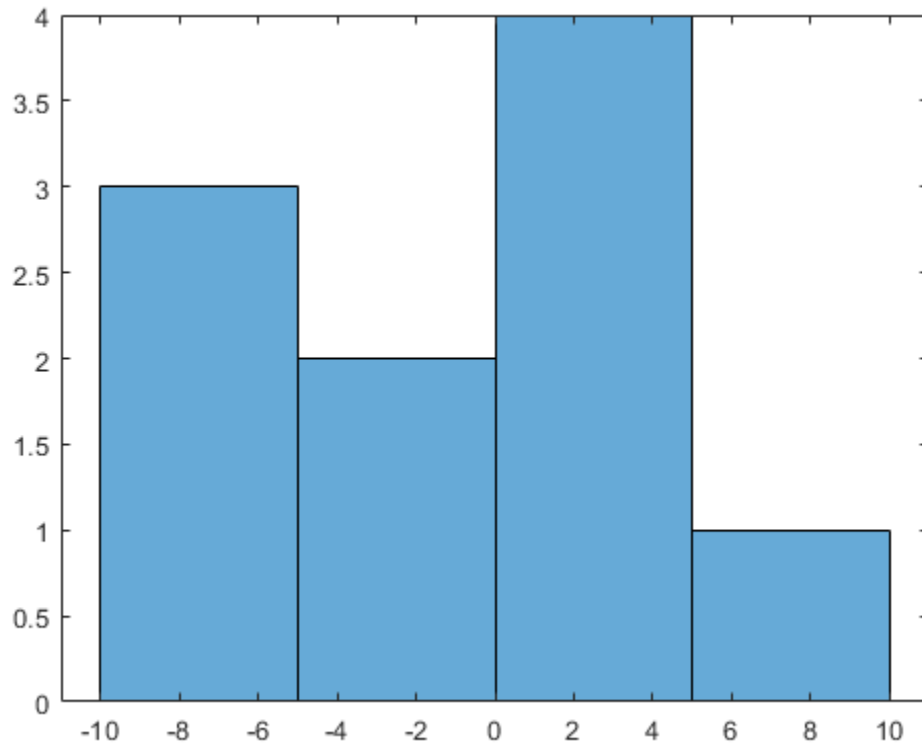
Polar Plots

Plotting in Polar Coordinates

These examples show how to create line plots, scatter plots, and histograms in polar coordinates. They also show how to annotate and change axes limits on polar plots.

Create Polar Line Plot

Visualize the radiation pattern from an antenna in polar coordinates. Load the file `antennaData.mat`, which contains the variables `theta` and `rho`. The variable `rho` is a measure of how intensely the antenna radiates for each value of `theta`. Visualize this radiation pattern by plotting the data in polar coordinates using the `polarplot` function.

```
load('antennaData.mat')
```

```
figure
polarplot(theta,rho)
```
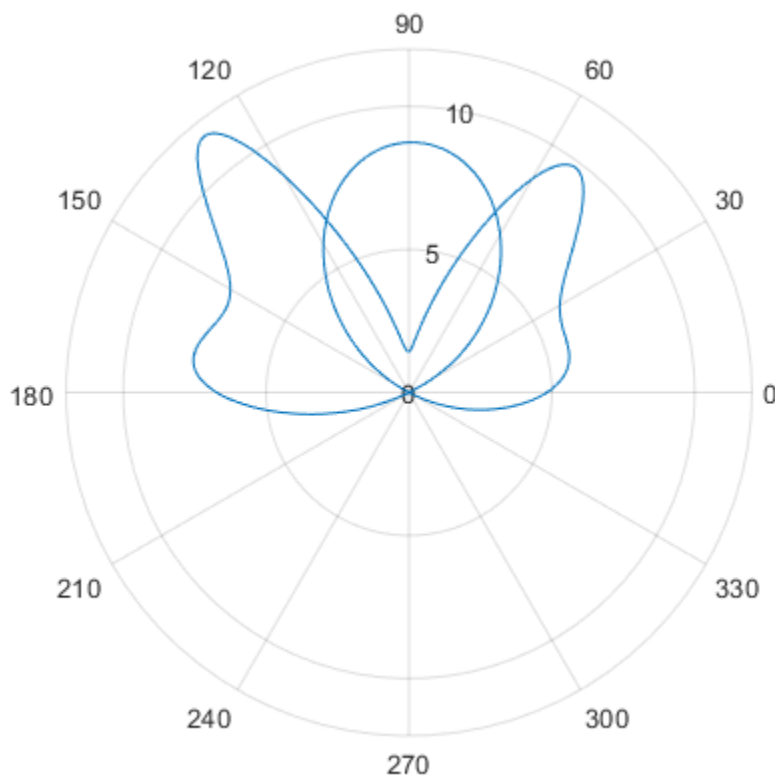

Multiple Polar Line Plots

Use `hold on` to retain the current polar axes and plot additional data using `polarplot`.

```
rng('default')
noisy = rho + rand(size(rho));
hold on
polarplot(theta,noisy)
hold off
```
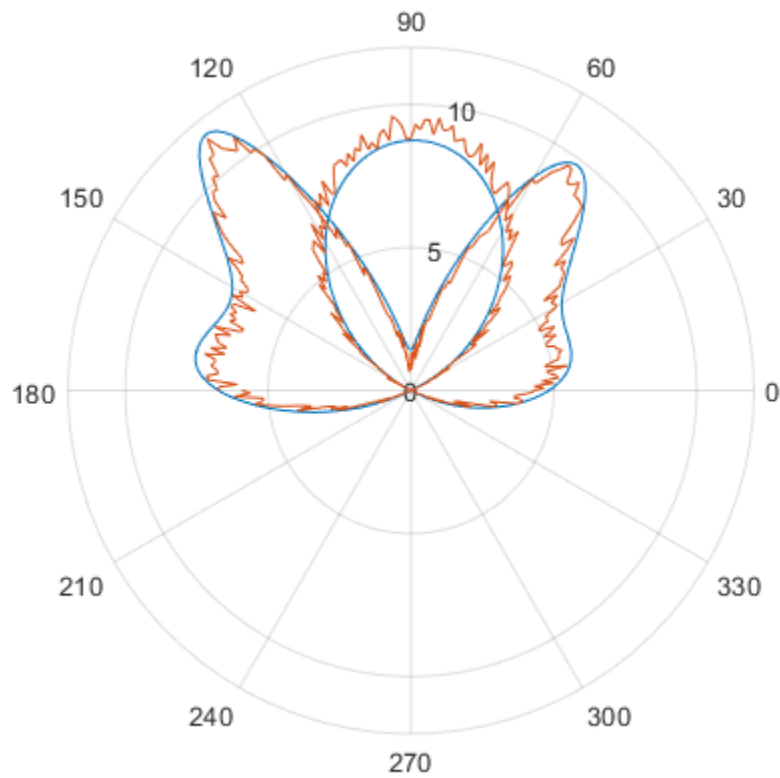
Annotating Polar Plots

Use annotation functions such as `legend` and `title` to label polar plots like other visualization types.

```
legend('Original','With Noise')
title('Antenna Radiation Pattern')
```

Antenna Radiation Pattern
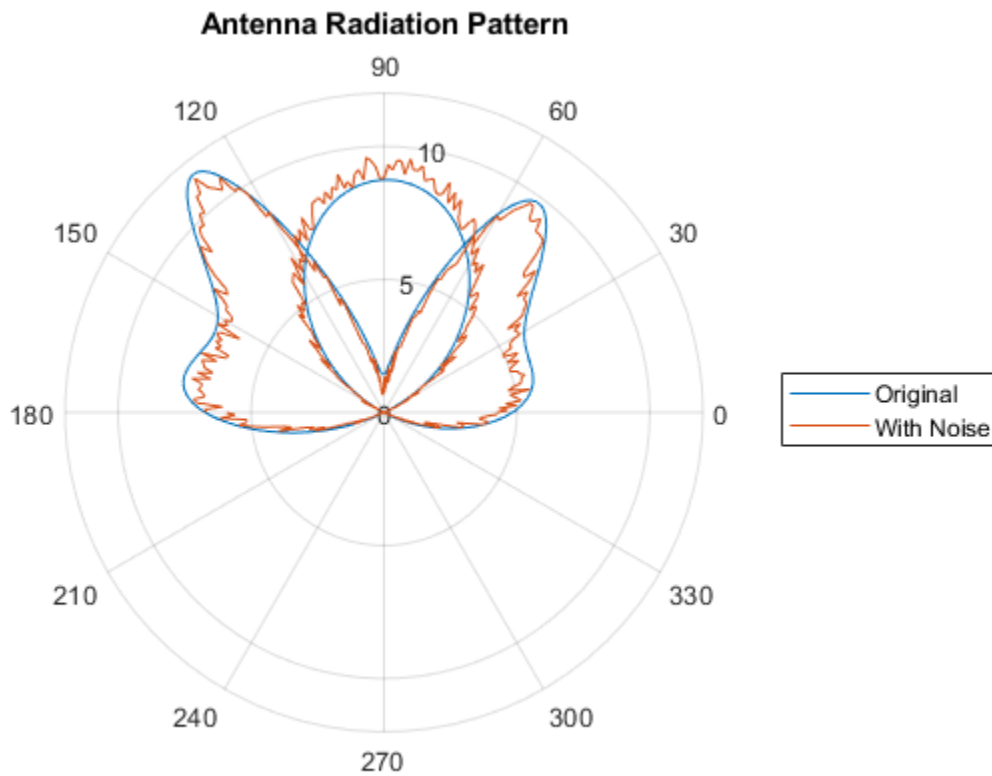
Change Polar Axes Limits

By default, negative values of the radius are plotted as positive values in the polar plot. Use `rlim` to adjust the *r*-axis limit to include negative values.

```
rmin = min(rho);
rmax = max(rho);
rlim([rmin rmax])
```

Antenna Radiation Pattern
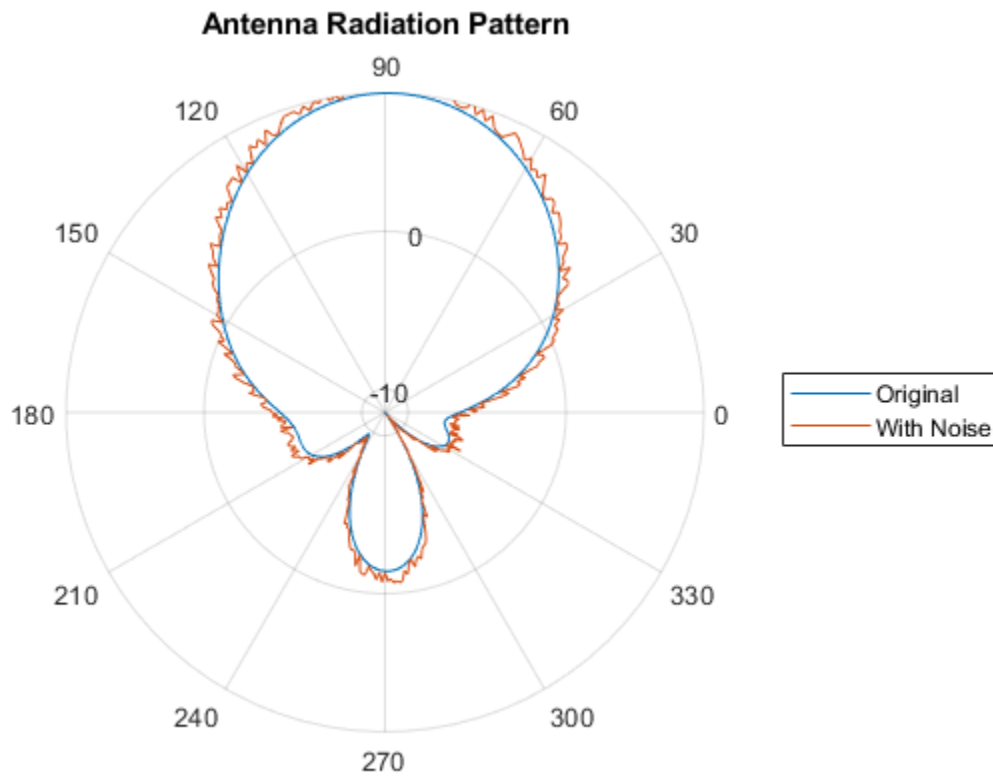
Change the *theta*-axis limits to 0 and 180 with `thetalim`.

```
thetalim([0 180])
```

Antenna Radiation Pattern
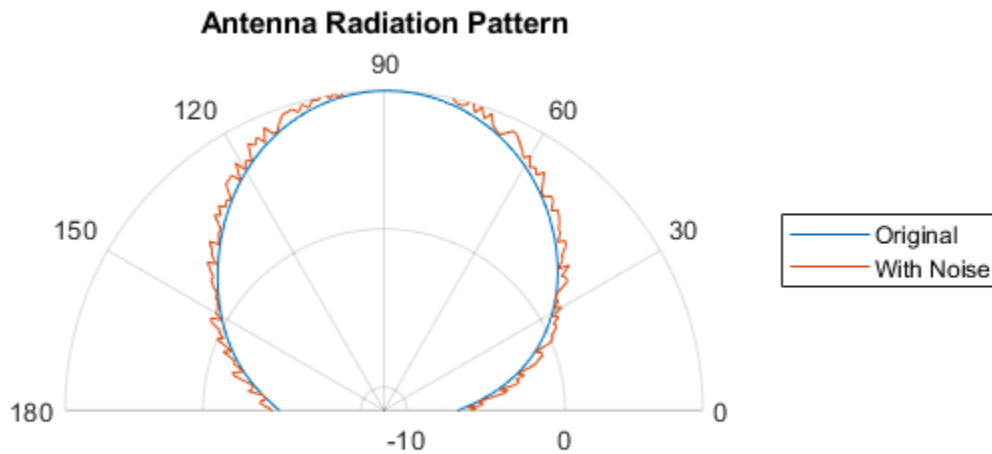
Create Polar Scatter Plot

Plot wind velocity data in polar coordinates. Load the file `windData.dat`, which includes the variables `direction`, `speed`, `humidity`, and `C`. Visualize the wind patterns by plotting the data in polar coordinates using the `polarscatter` function.

```
load('windData.mat')
polarscatter(direction,speed)
```
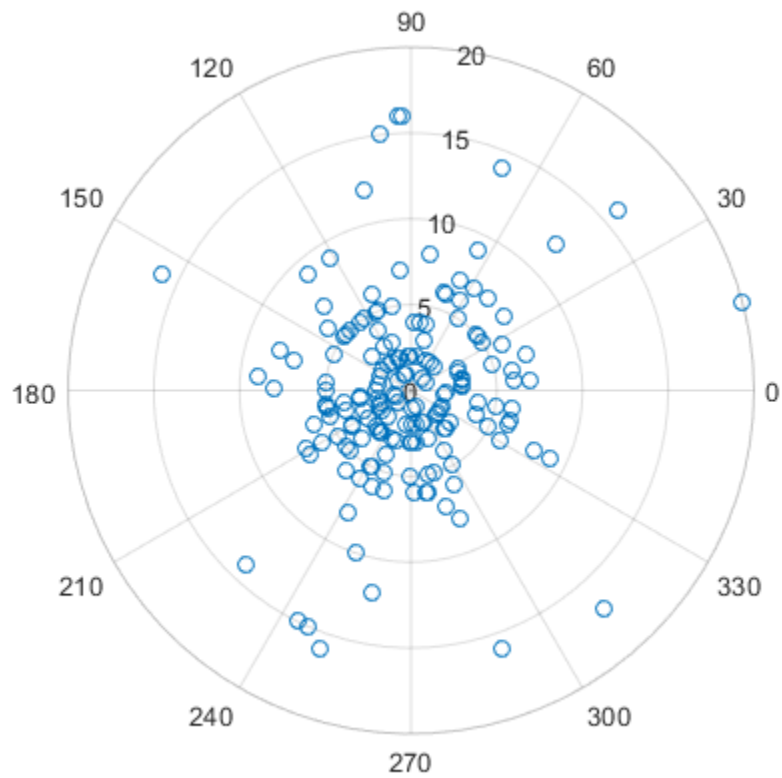
Include a third data input to vary marker size and represent a third dimension.

```
polarscatter(direction,speed,humidity)
```
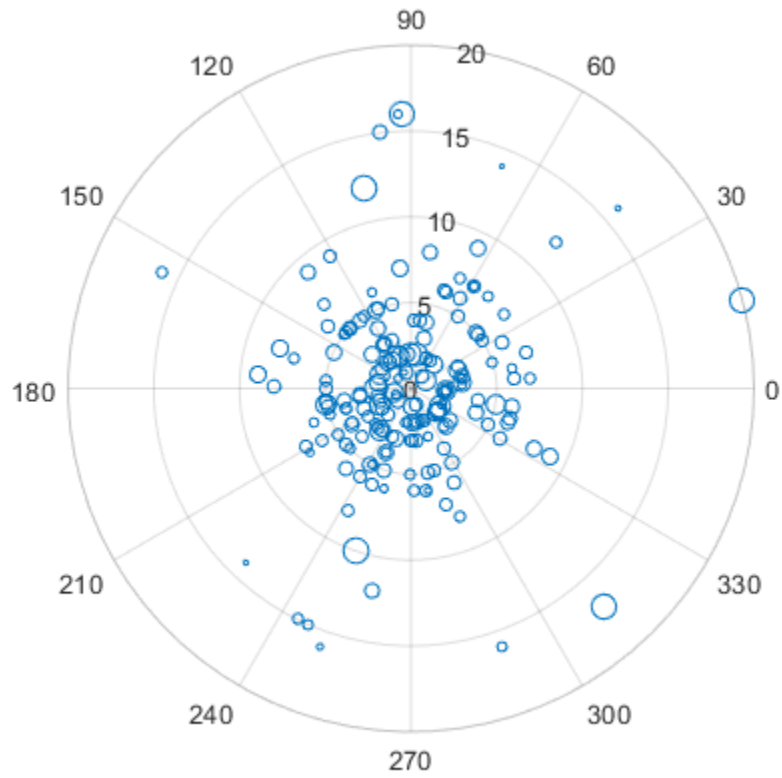
Use formatting inputs to adjust marker display properties.

```
polarscatter(direction,speed,humidity,C,'filled')
```
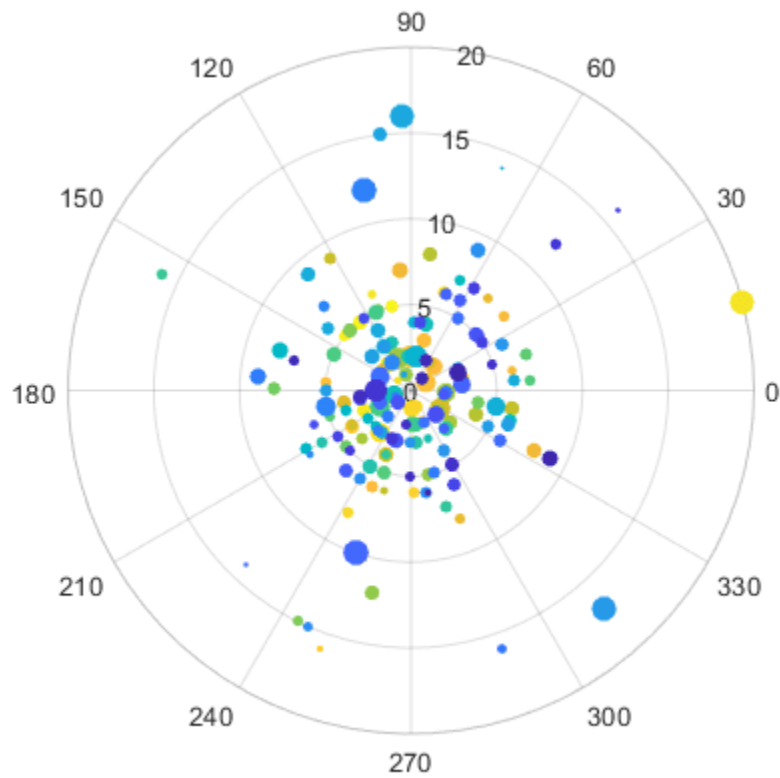
Create Polar Histogram Plot

Visualize the data using the `polarhistogram` function, which produces a visual representation known as a wind rose.

```
polarhistogram(direction)
```
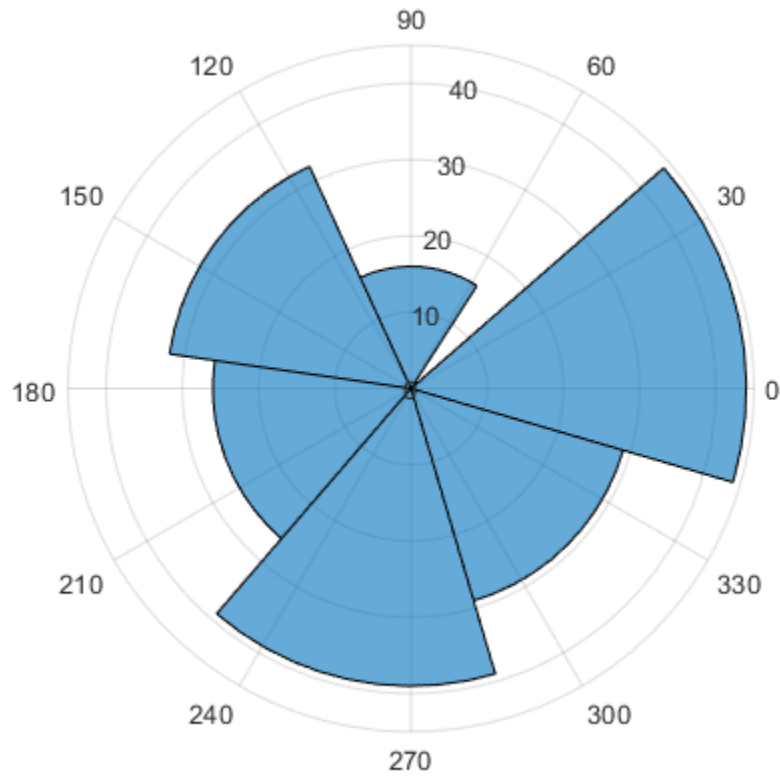
Specify a bin determination algorithm. The `polarhistogram` function has a variety of bin number and bin width determination algorithms to choose from within the `BinMethod` field.

```
polarhistogram(direction,'BinMethod','sqrt')
```
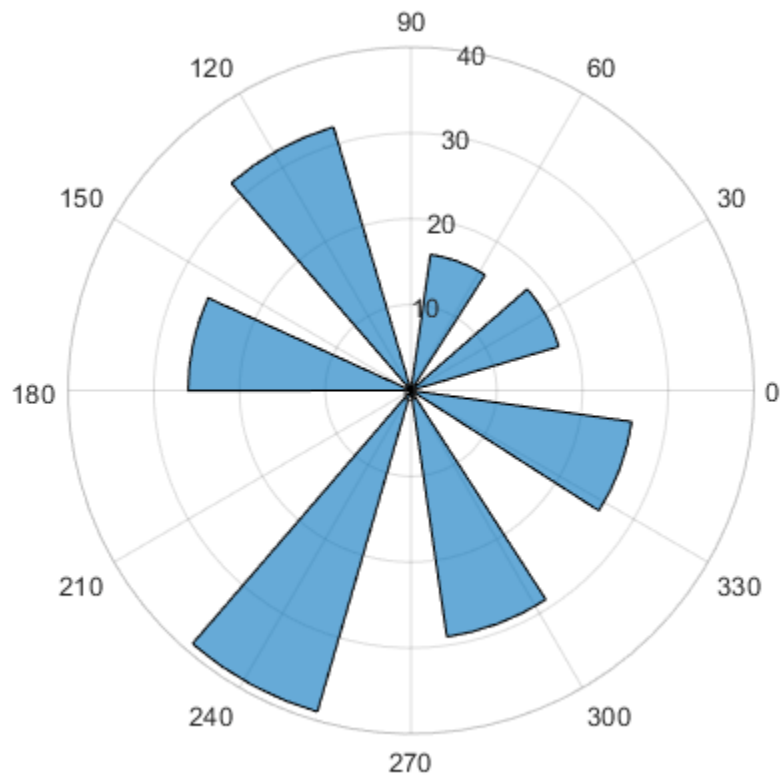
Specify the number of bins and the bin width.

```
polarhistogram(direction,24,'BinWidth',.5)
```
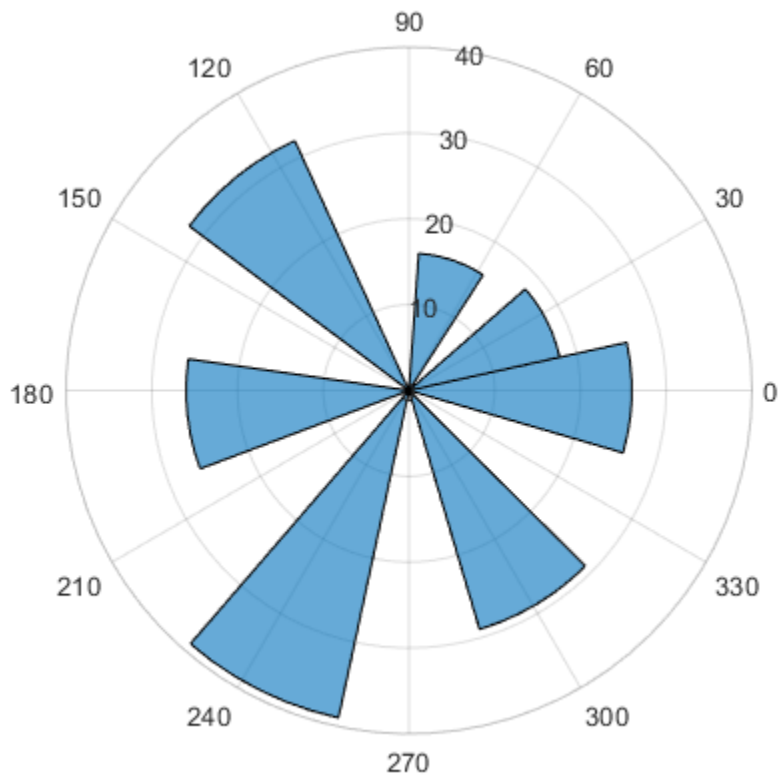
Specify a normalization method and adjust the display style to exclude any fill.

```
polarhistogram(direction,'Normalization','pdf','DisplayStyle','stairs')
```
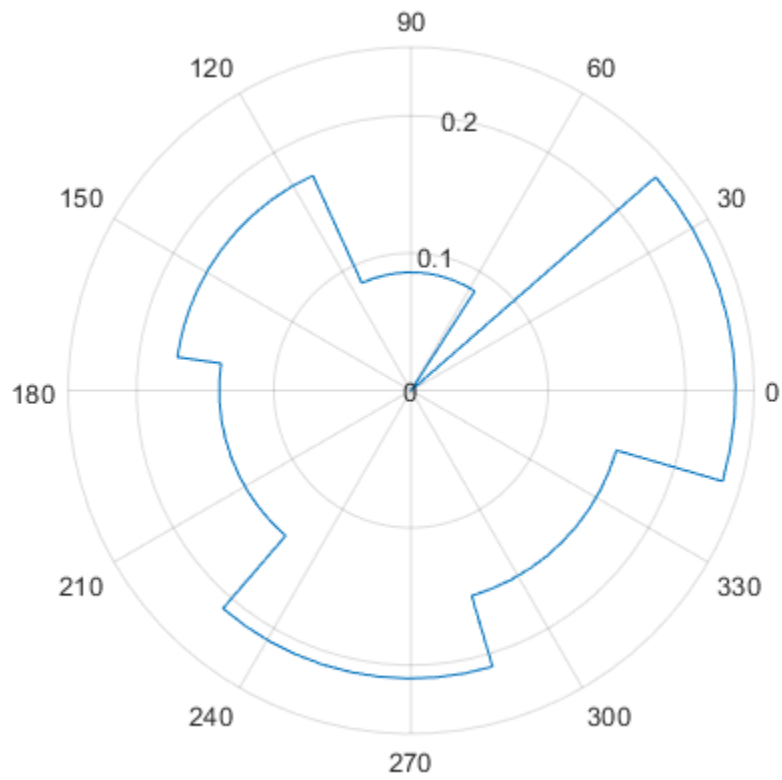
See Also
PolarAxes | polarplot | rticklabels | rticks | thetaticklabels | thetaticks

Customize Polar Axes

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')
```
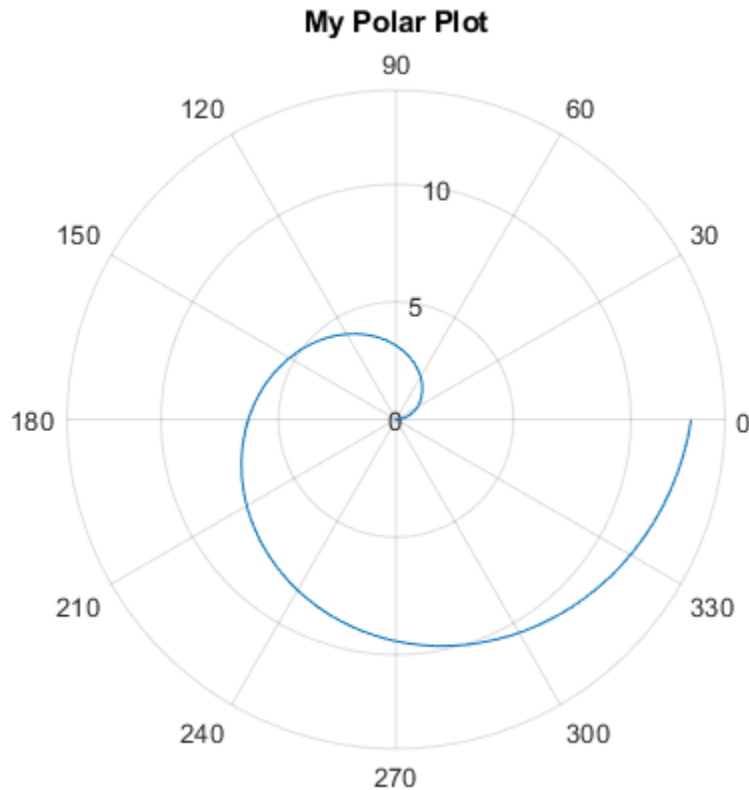

Customize Polar Axes Using Properties

When you create a polar plot, MATLAB creates a `PolarAxes` object. `PolarAxes` objects have properties that you can use to customize the appearance of the polar axes, such as the font size, color, or ticks. For a full list, see PolarAxes Properties.

Access the PolarAxes object using the `gca` function, such as `pax = gca`. Then, use `pax` with dot notation to set properties, such as `pax.FontSize = 14`.

```
pax = gca
```

```
pax =
  PolarAxes (My Polar Plot) with properties:

            ThetaLim: [0 360]
               RLim: [0 14]
      ThetaAxisUnits: 'degrees'
            ThetaDir: 'counterclockwise'
   ThetaZeroLocation: 'right'

  Show all properties
```

```
pax.FontSize = 14;
```
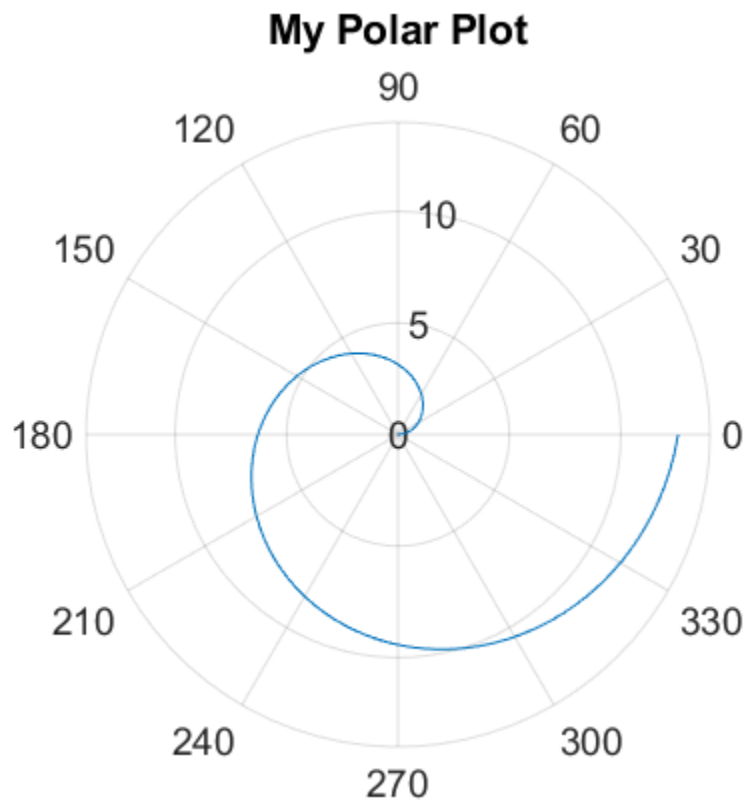

theta-Axis Tick Values

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

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

My Polar Plot
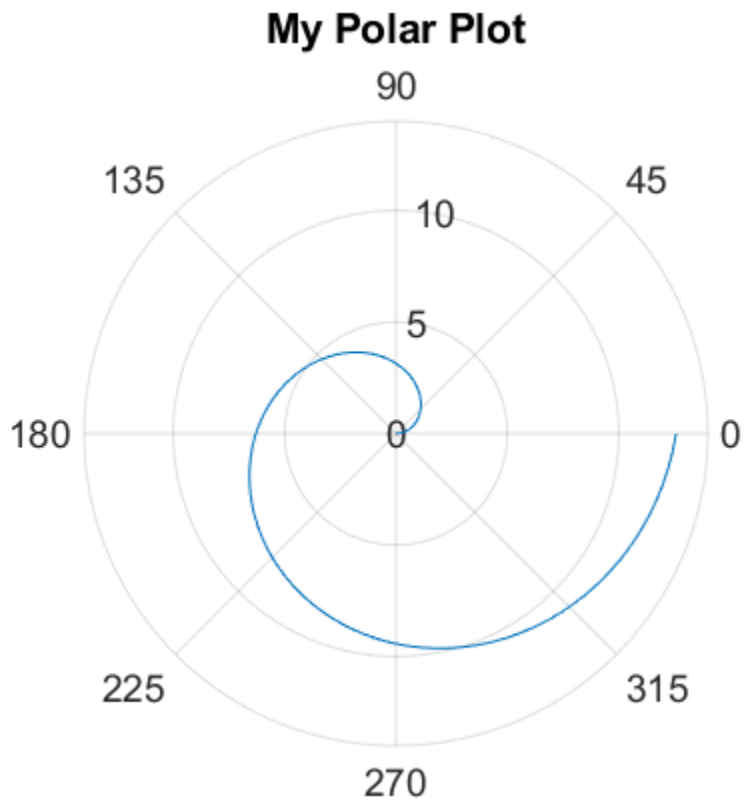
Display the *theta*-axis values in radians instead of degrees by setting the `ThetaAxisUnits` property.

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

My Polar Plot
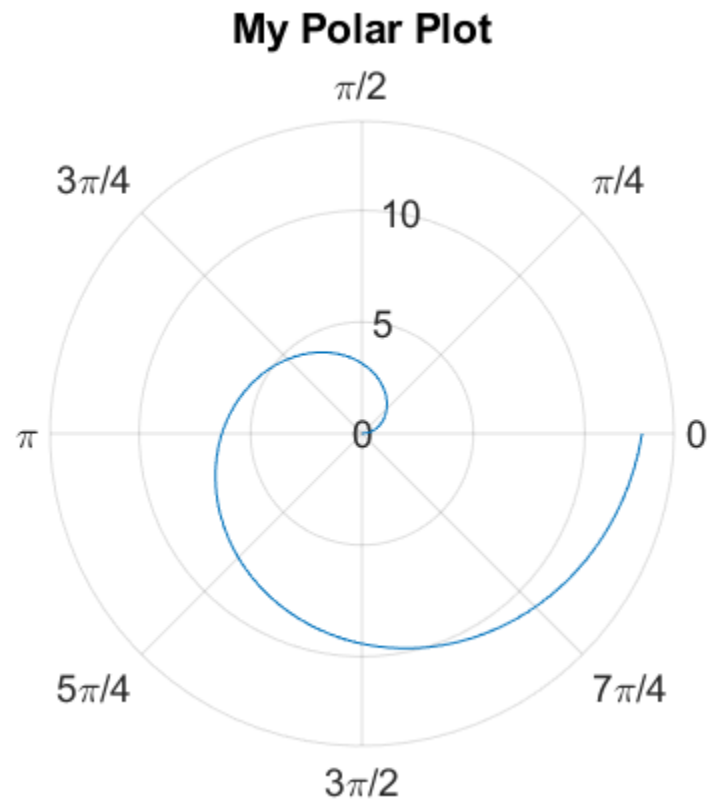
Modify the *theta*-axis so that it increases in a clockwise direction. Also, rotate the *theta*-axis so that the zero reference angle is on the left side.

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

My Polar Plot
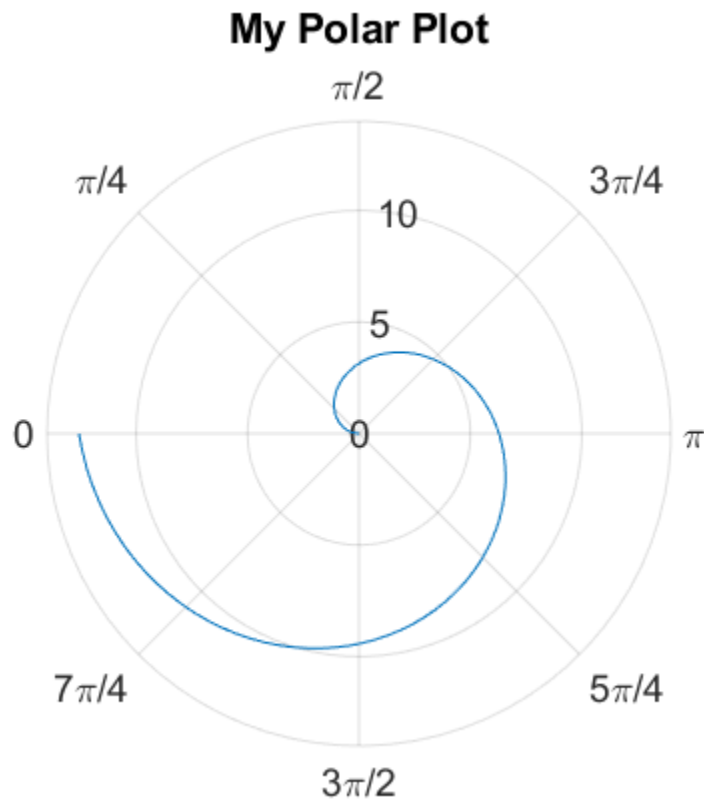
r-Axis Limits, Tick Values, and Labels

Change the limits of the *r*-axis so that the values range from -5 to 15. Display lines at the values -2, 3, 9, and 15. Then, change the labels that appear next to each 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'})
```

My Polar Plot
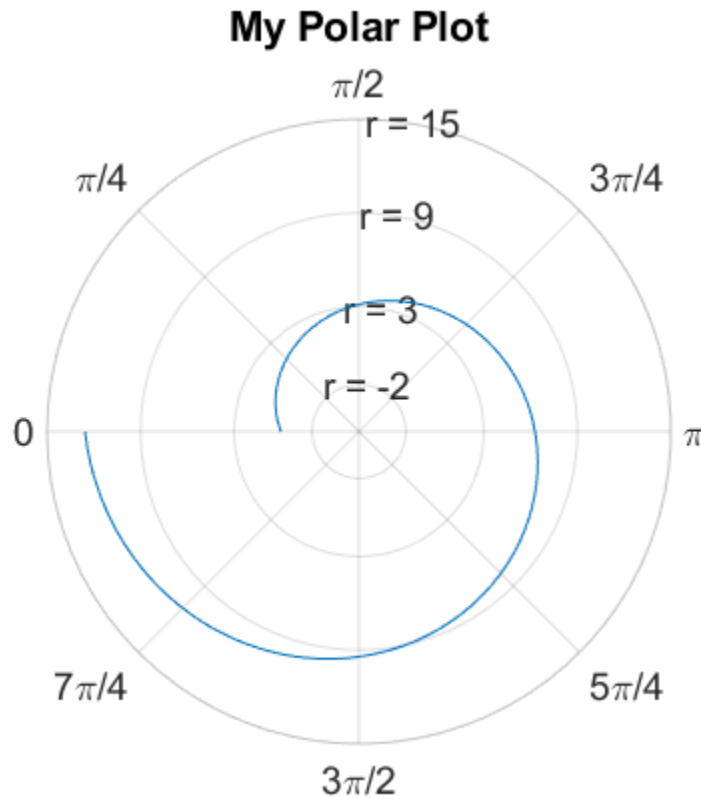
Grid Line and Label Colors

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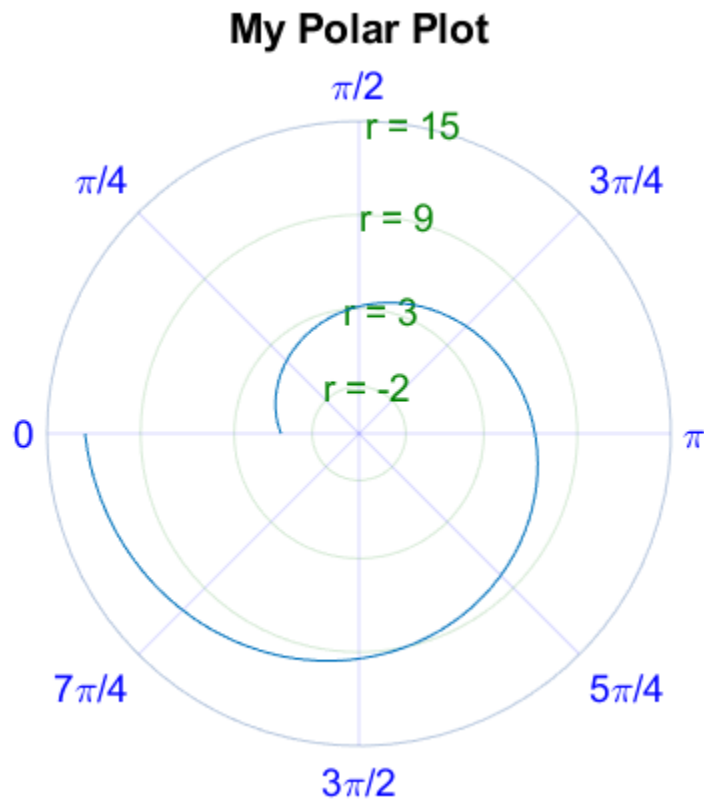
Change the color of all the grid lines without affecting the labels by setting the `GridColor` property.

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

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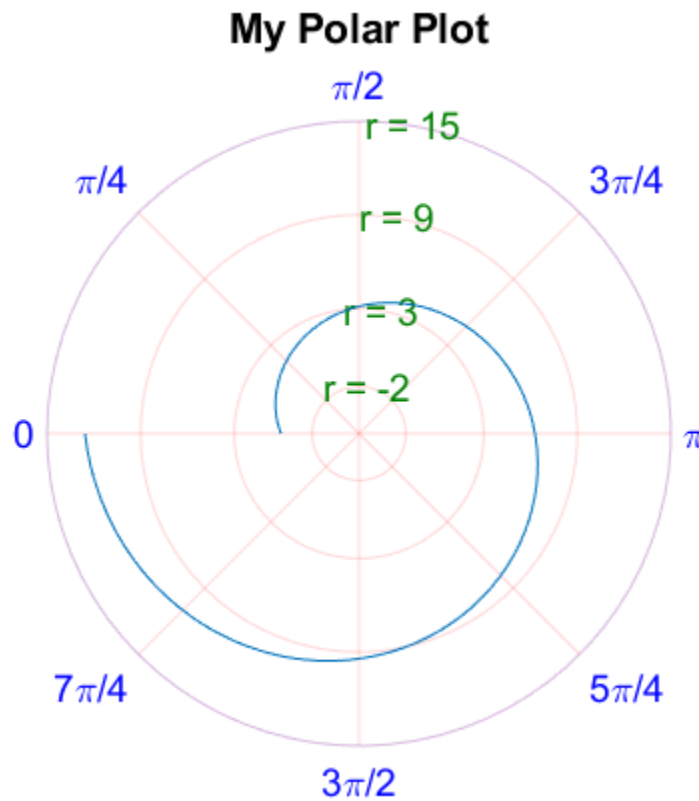
When you specify the `GridColor` property, the `ThetaColor` and `RColor` properties no longer affect the grid lines. If you want the `ThetaColor` and `RColor` properties to affect the grid lines, then set the `GridColorMode` property back to `'auto'`.

See Also

PolarAxes | `polarplot` | `rticklabels` | `rticks` | `thetaticklabels` | `thetaticks`

Related Examples

- "Compass Labels on Polar Axes" on page 3-22

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')
```
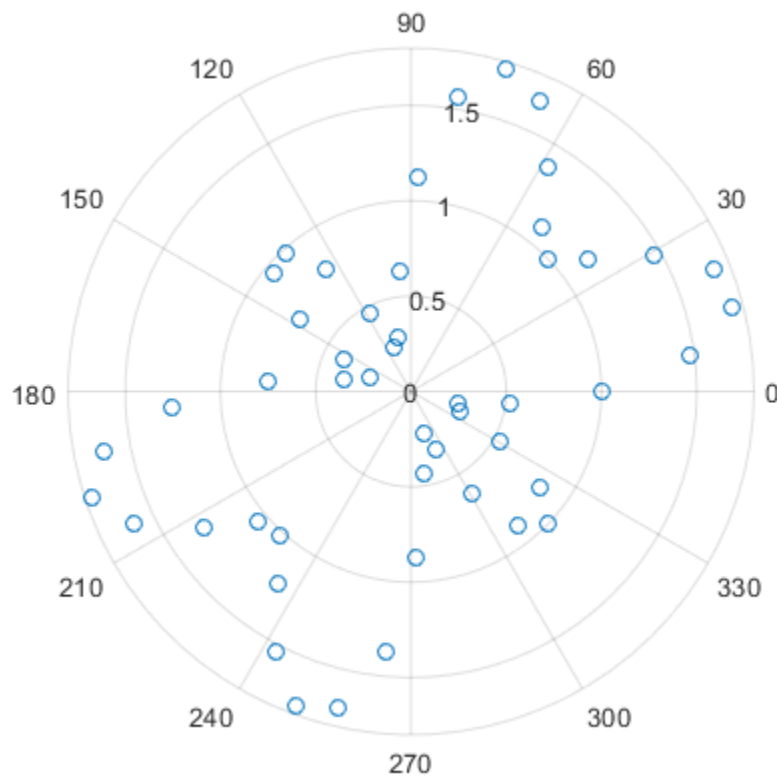

Use `gca` to access the polar axes object. Specify the angles at which to draw grid lines by setting the `ThetaTick` property. Then, specify the label for each grid line by setting the `ThetaTickLabel` property.

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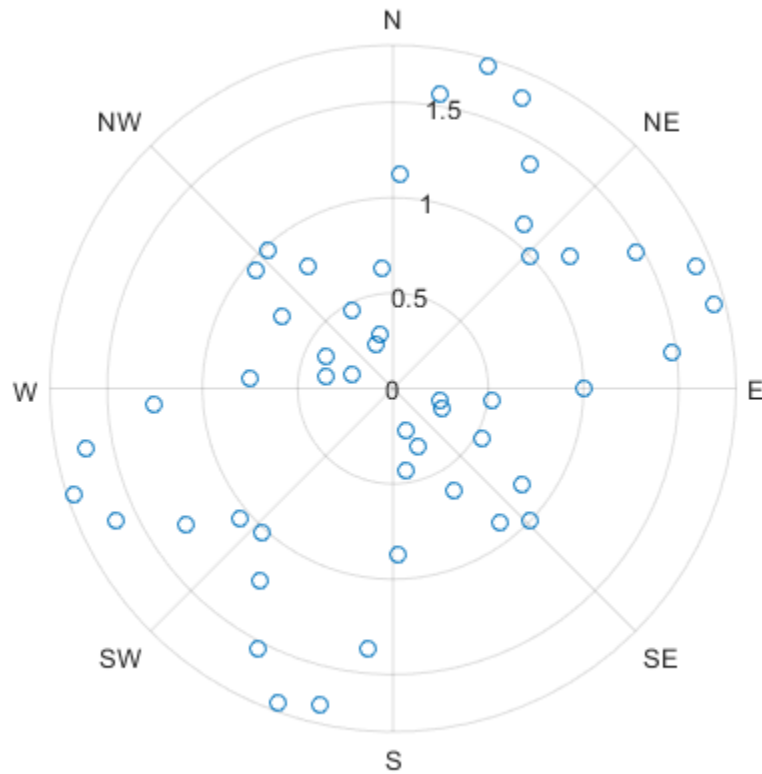
See Also

PolarAxes | polarplot | rticklabels | rticks | thetaticklabels | thetaticks

Related Examples

• "Customize Polar Axes" on page 3-14

4

Contour Plots

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)')
```


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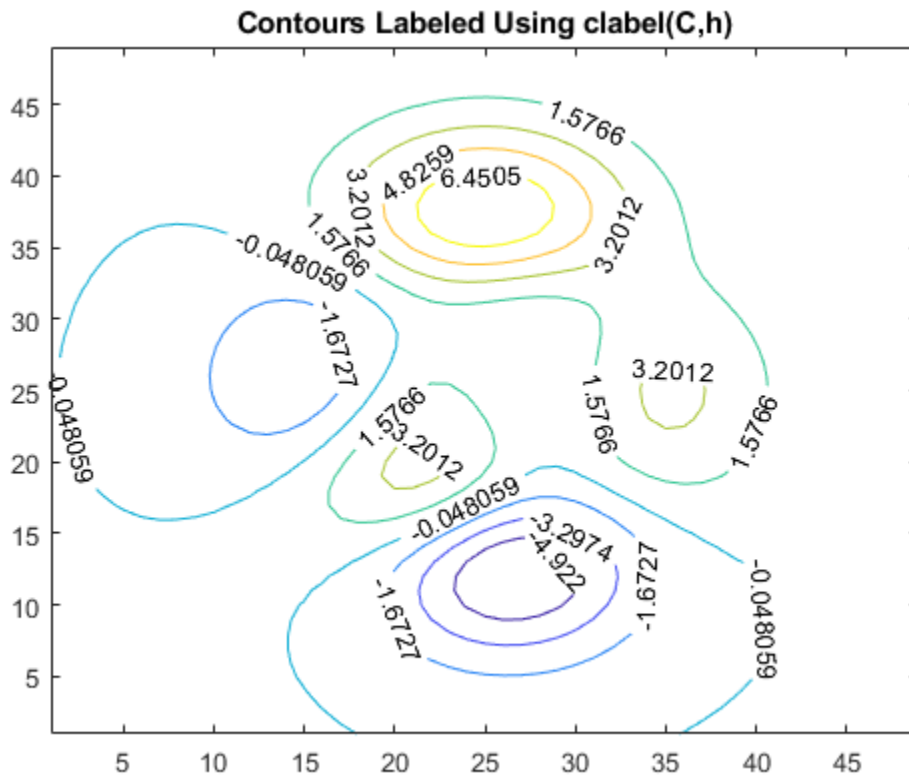
To interactively select the contours to label using the mouse, pass the `manual` option to `clabel`, for example, `clabel(C,h,'manual')`. This command displays a crosshair cursor when the mouse is within the figure. Click the mouse to label the contour line closest to the cursor.

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')
```
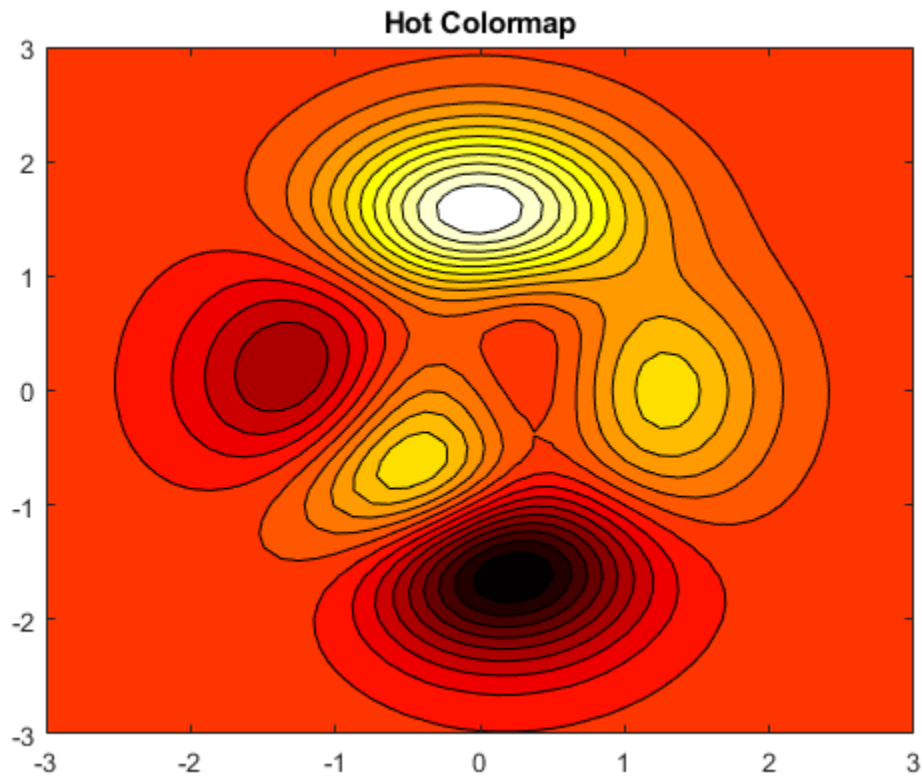

Control Mapping of Data Values to Colormap

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')
```
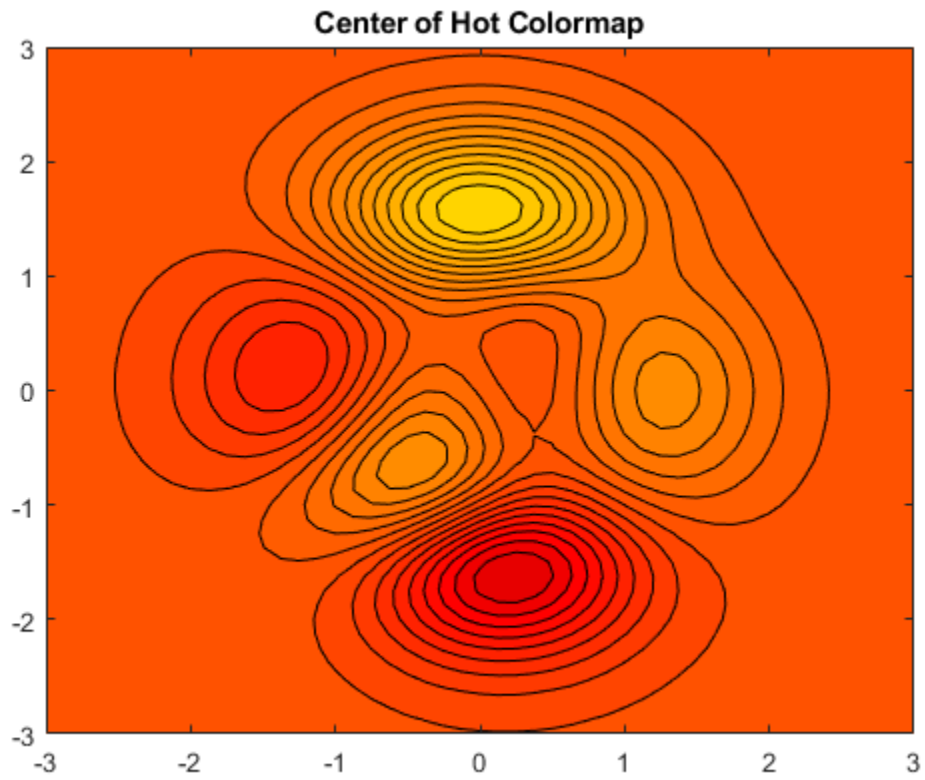
See Also
caxis | colormap | contourf

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)
```
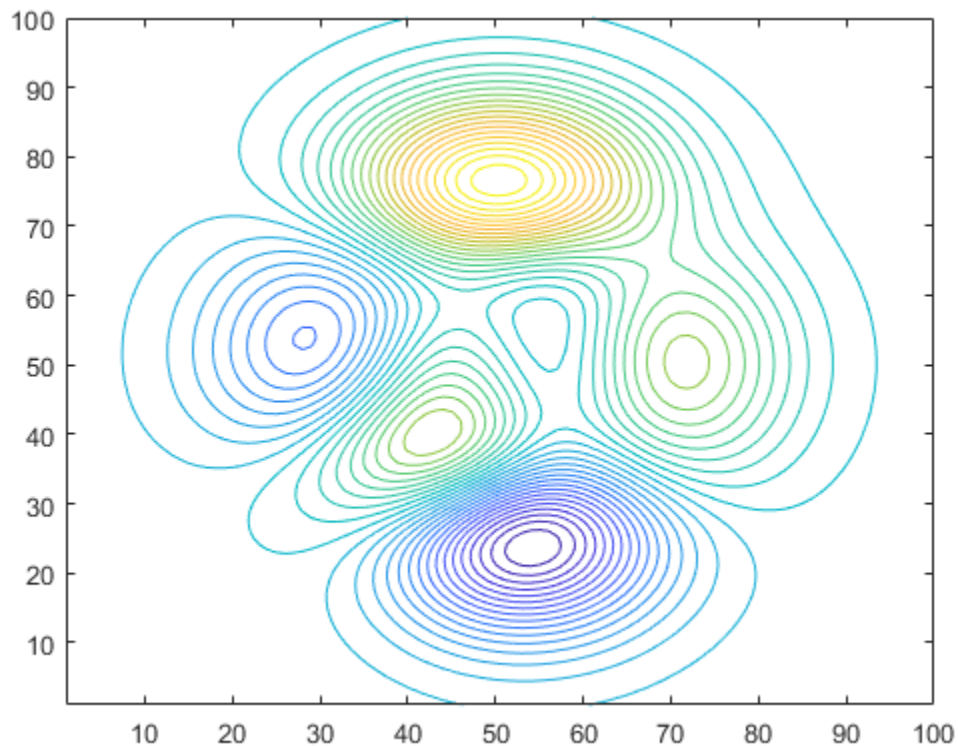

Define `zindex` as a vector of integer values between `zmin` and `zmax` indexed by 2.

```
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
```
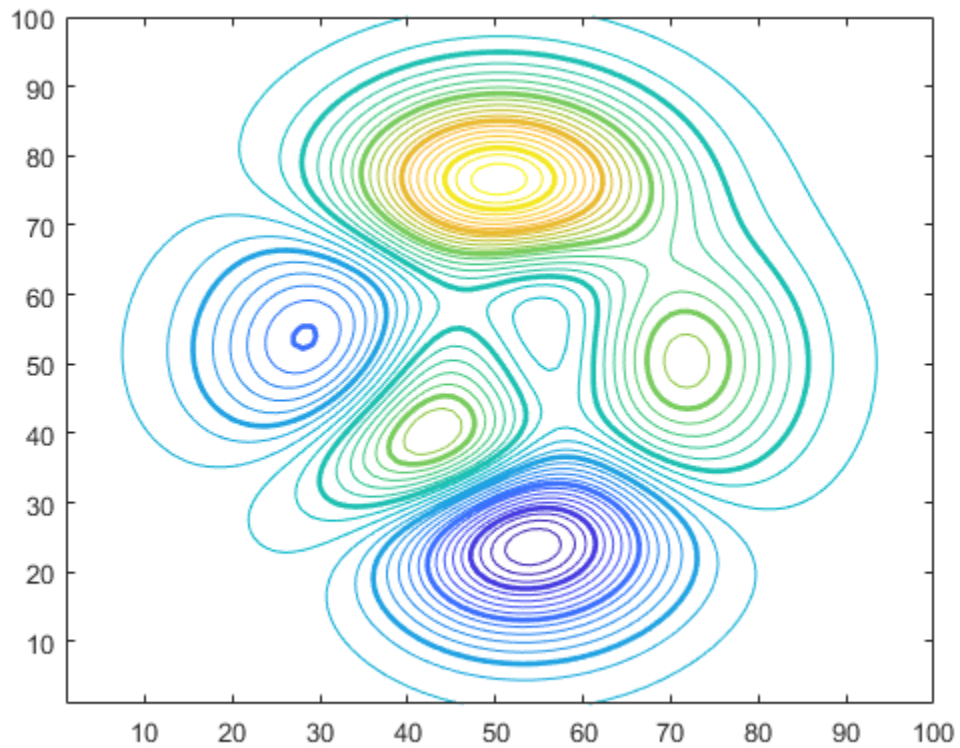

See Also
ceil | contour | floor | hold | max | min

Combine Contour Plot and Quiver Plot

Display contour lines and gradient vectors on the same plot.

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)
```
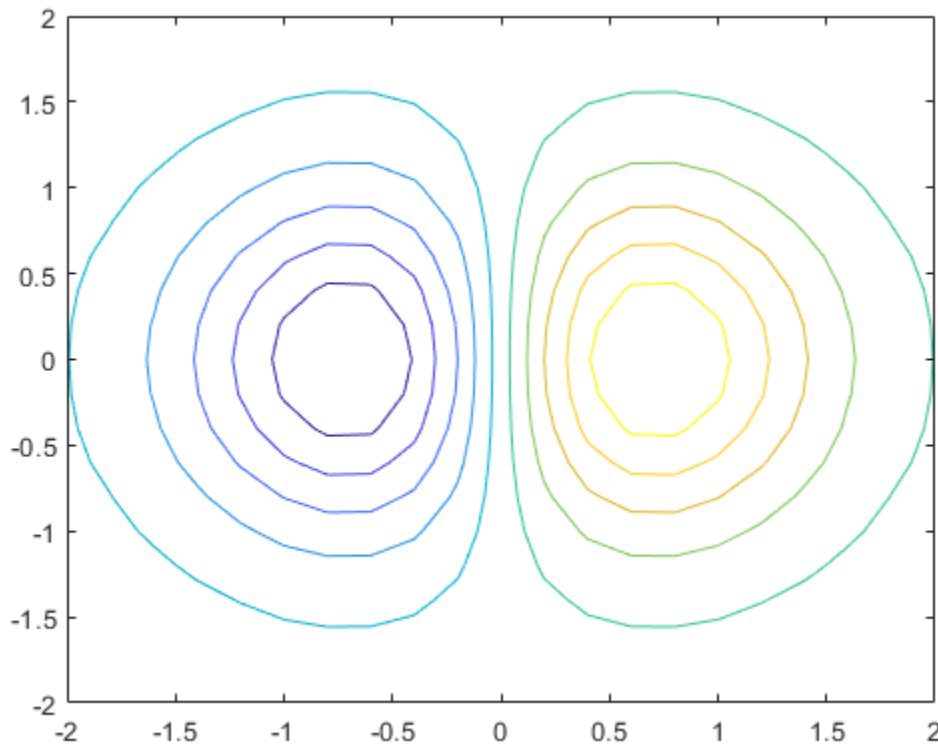

Calculate the 2-D gradient of Z using the `gradient` function. The `gradient` function returns U as the gradient in the *x*-direction and V as the gradient in the *y*-direction. Display arrows indicating the gradient values using the `quiver` function.

```
[U,V] = gradient(Z,0.2,0.2);
hold on
quiver(X,Y,U,V)
hold off
```
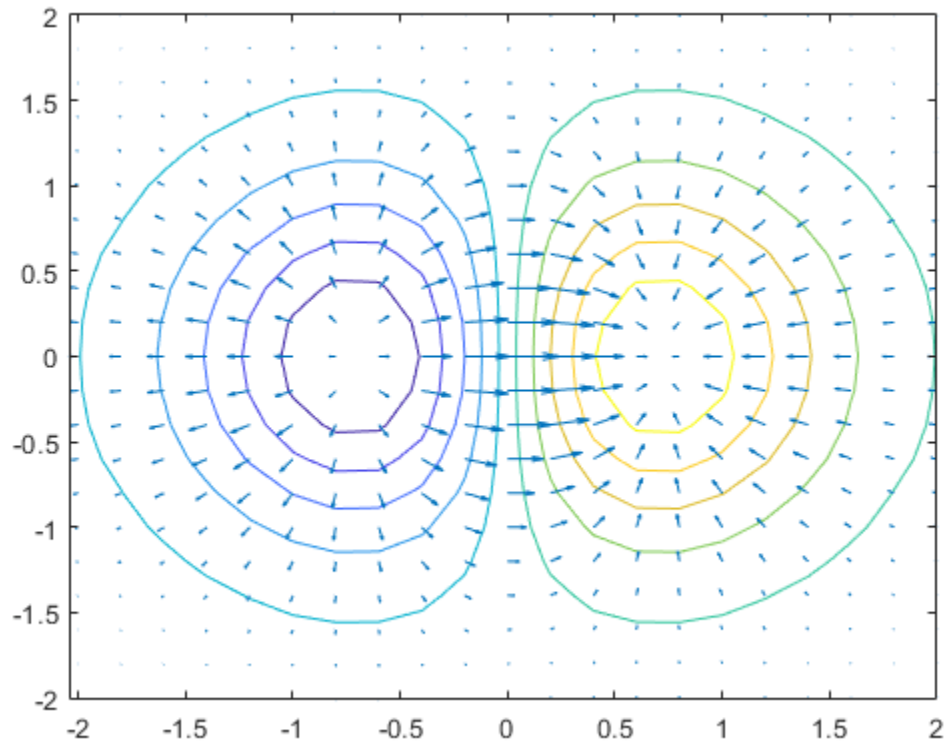
See Also

contour | hold

Contour Plot with Major and Minor Grid Lines

You can create a contour plot with emphasis on selected contour lines by splitting the data and creating two overlapping contour plots.

For example, create a contour plot of the `peaks` function where the even numbered contours lines are solid and the odd numbered contour lines are dotted. Plot one contour for the even numbered levels. Then, overlay a second contour plot with the odd numbered levels drawn with a dotted line.

```
major = -6:2:8;
minor = -5:2:7;
[cmajor,hmajor] = contour(peaks,'LevelList',major);
clabel(cmajor,hmajor)

hold on
[cminor,hminor] = contour(peaks,'LevelList',minor);
hminor.LineStyle = ':';
hold off
```
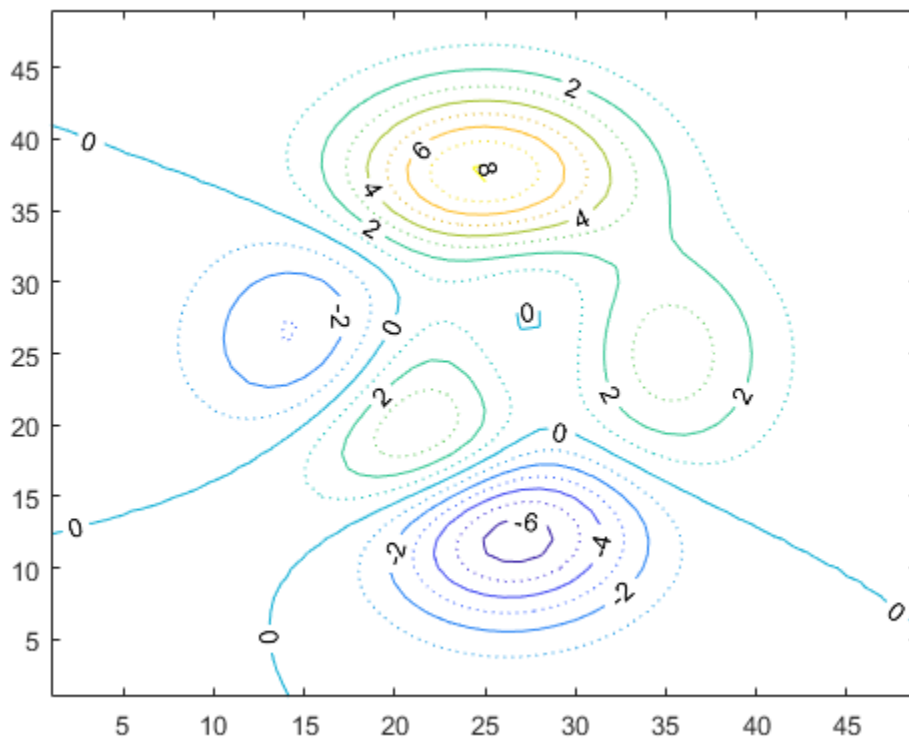

See Also

clabel | contour | contourf | hold

Specialized Charts

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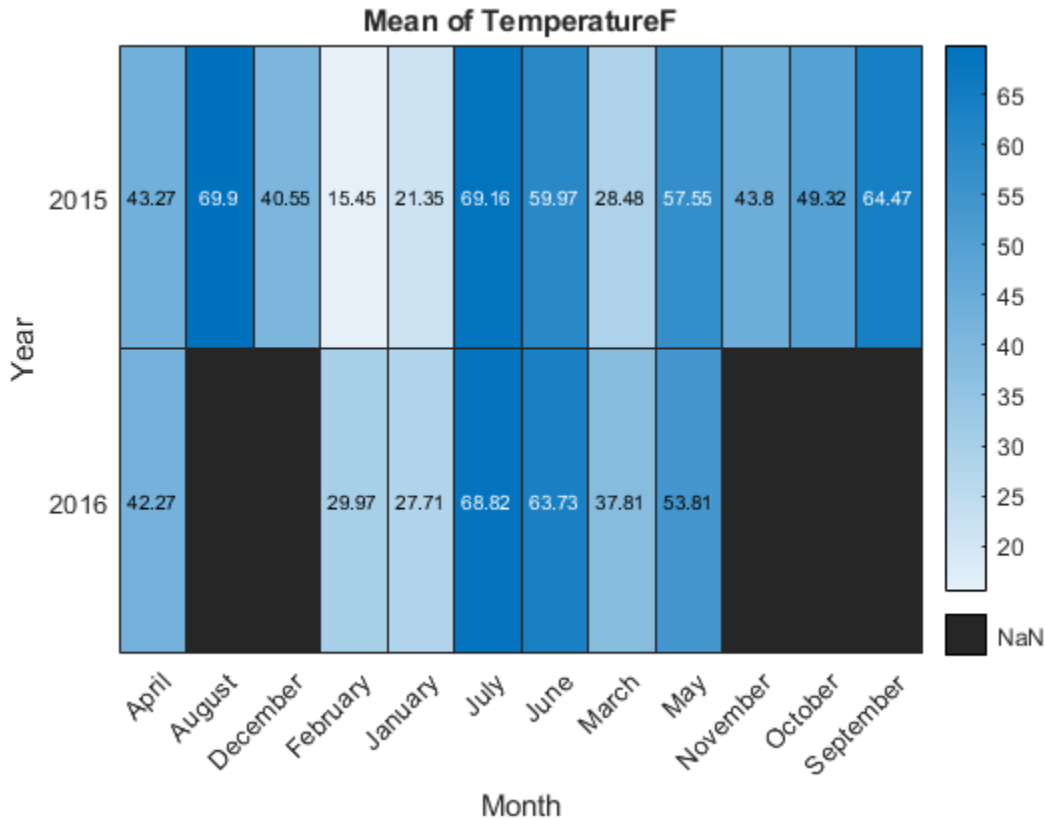
By default, MATLAB calculates the color data as the average temperature for each month. However, you can change the calculation method by setting the `ColorMethod` property.

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 or by setting `HeatmapChart` properties.

To use categorical arrays, first change the data in the `Month` column of the table from a cell array to a categorical array. Then use the `reordercats` function to reorder the categories. You can apply these functions to the table in the workspace (`tbl`) or 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.

```
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);
```

Mean of TemperatureF
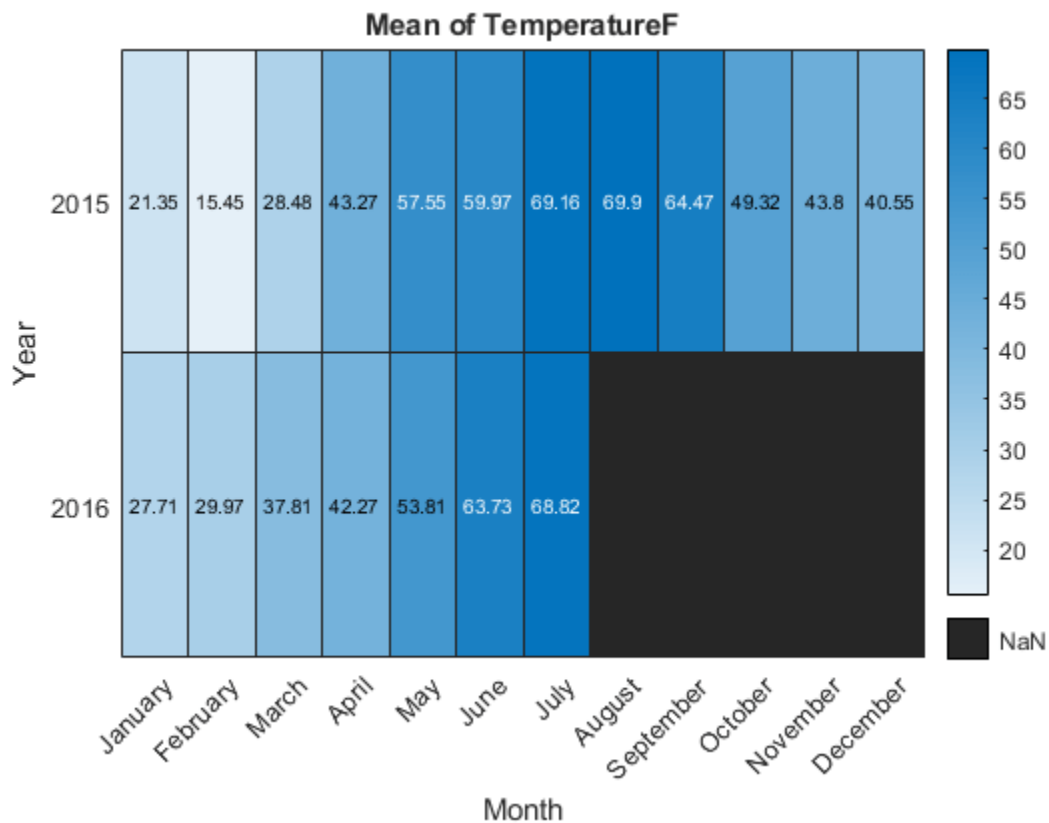
Similarly, you can add, remove, or rename the heatmap labels using the `addcats`, `removecats`, or `renamecats` functions for categorical arrays.

Alternatively, you can reorder the values along an axis using the `XDisplayData` and `YDisplayData` properties of the `HeatmapChart` object.

```
h.XDisplayData = {'January','February','March','April','May','June','July',...
    'August','September','October','November','December'};
```
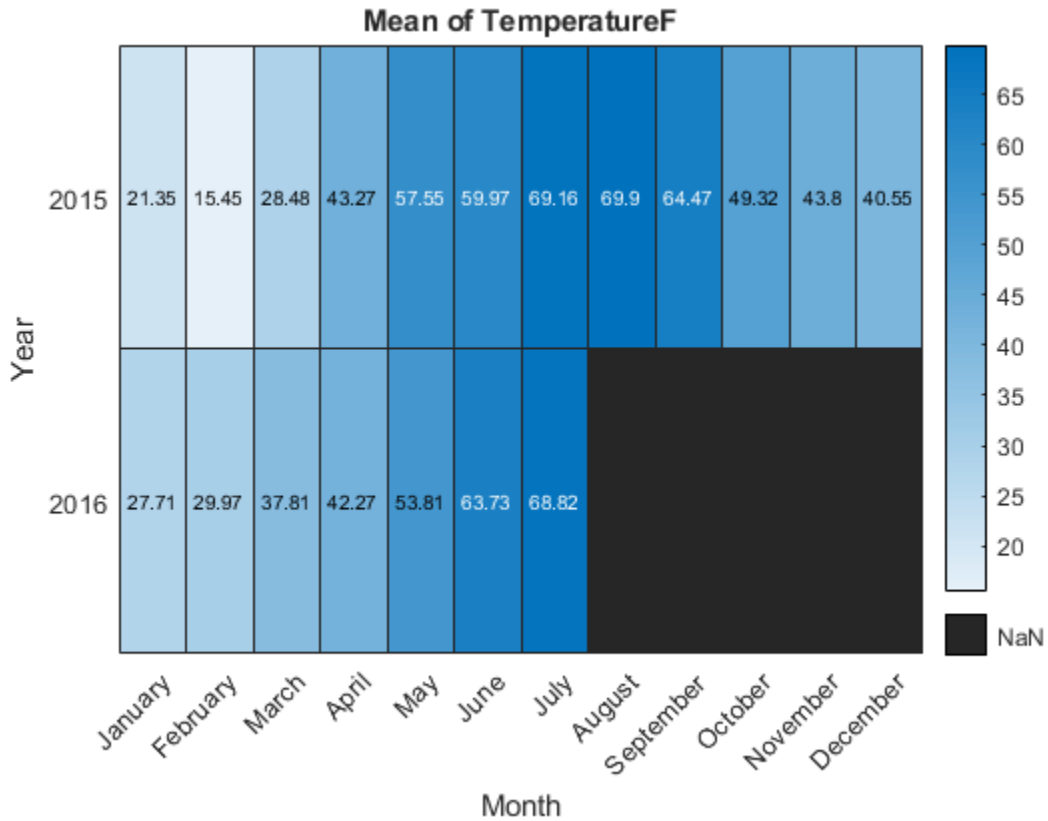
Modify Title and Axis Labels

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;
```

Average Temperatures
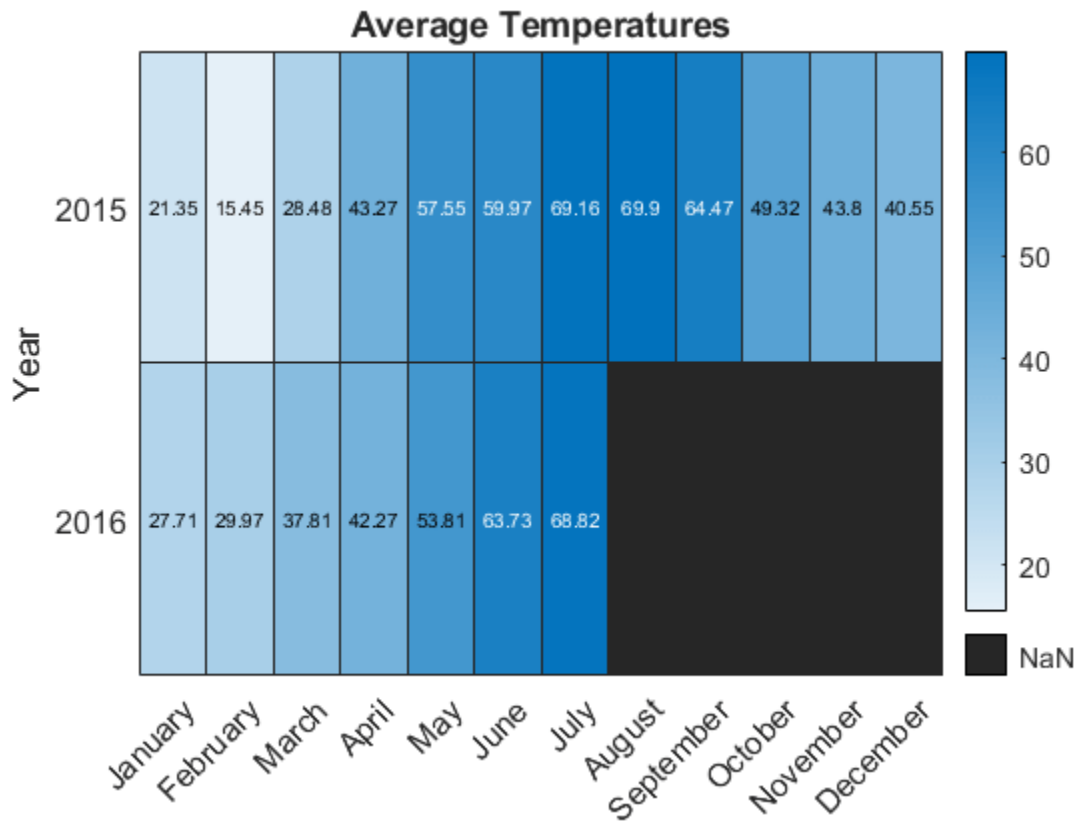
Modify Appearance of Missing Data Cells

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';
```

Average Temperatures
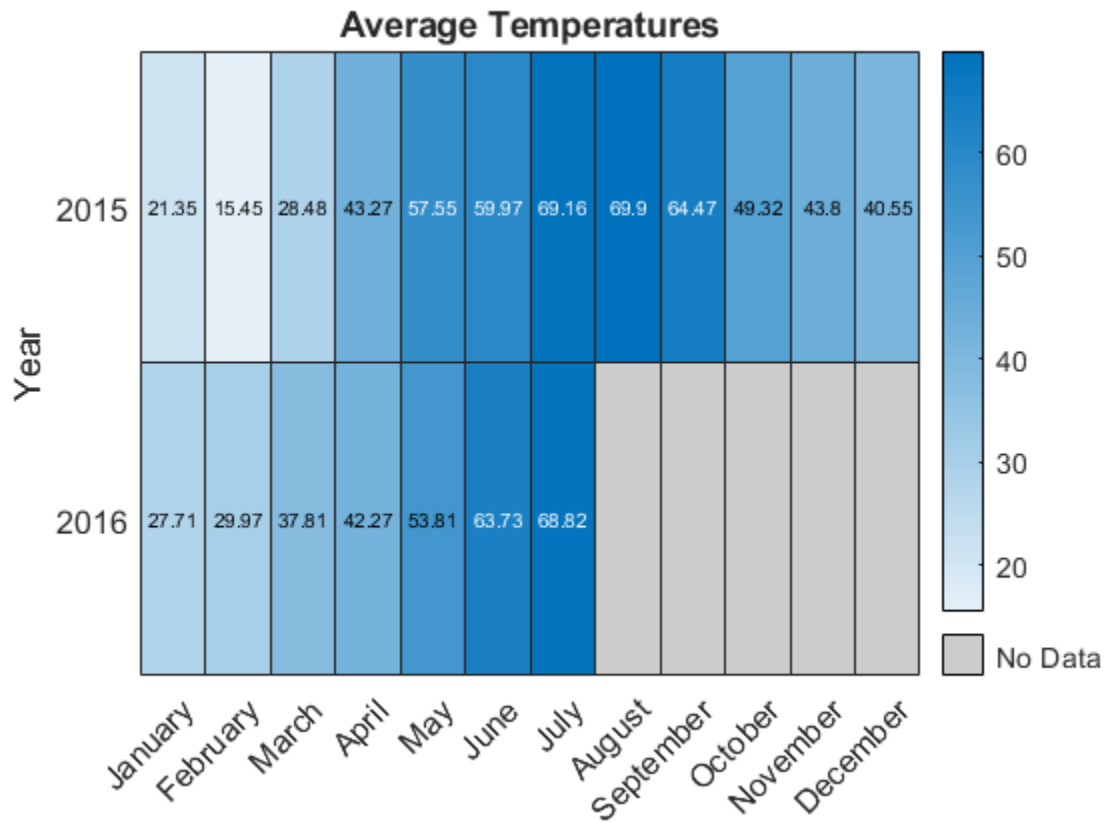
Remove Colorbar

Remove the colorbar by setting the `ColorbarVisible` property.

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

Average Temperatures

Year	January	February	March	April	May	June	July	August	September	October	November	December
2015	21.35	15.45	28.48	43.27	57.55	59.97	69.16	69.9	64.47	49.32	43.8	40.55
2016	27.71	29.97	37.81	42.27	53.81	63.73	68.82					
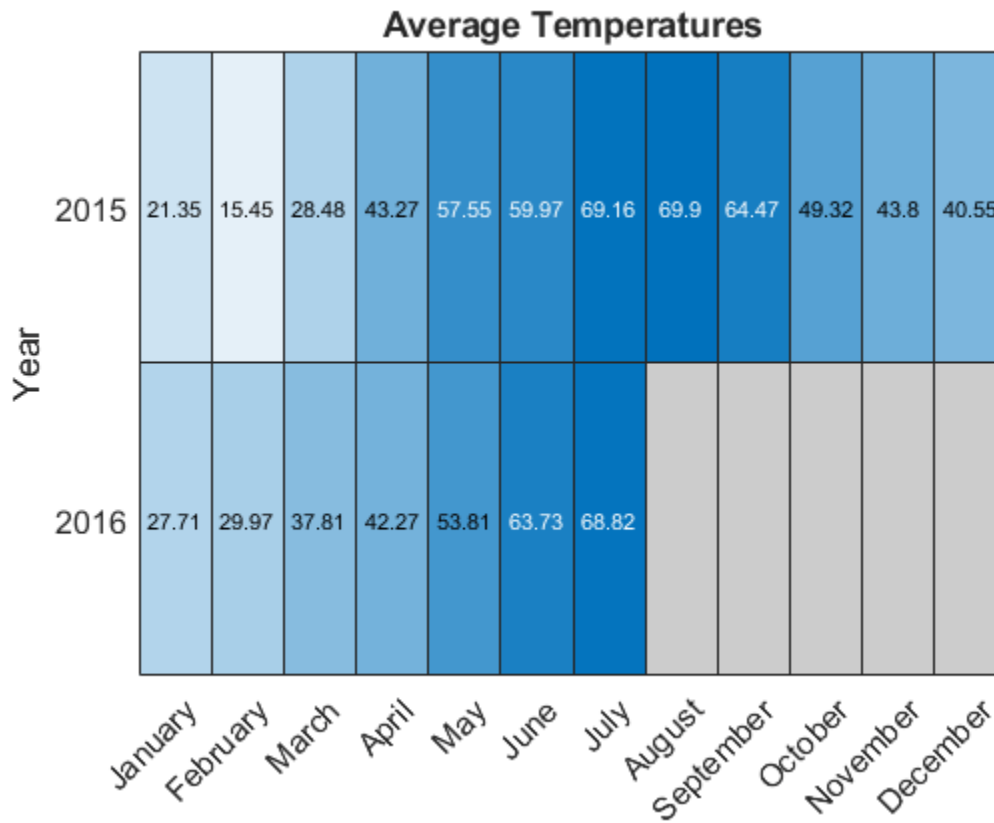
Format Cell Text

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';
```

Average Temperatures
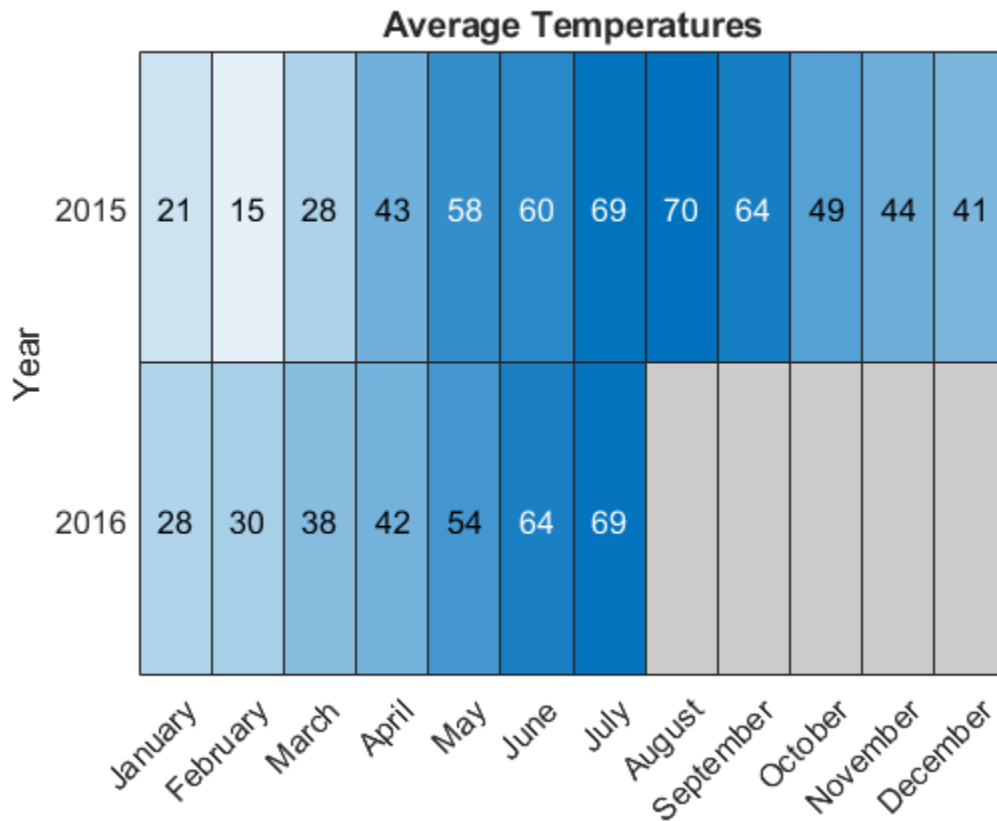
Add or Remove Values Along Axis

Show only the first month of each quarter by setting the XDisplayData property. Add the year 2017 along the *y*-axis by setting the YDisplayData property. Set these properties to a subset, superset, or permutation of the values in XData or YData, respectively.

```
h.XDisplayData = {'January','April','July','October'};
h.YDisplayData = {'2015','2016','2017'};
```

Average Temperatures
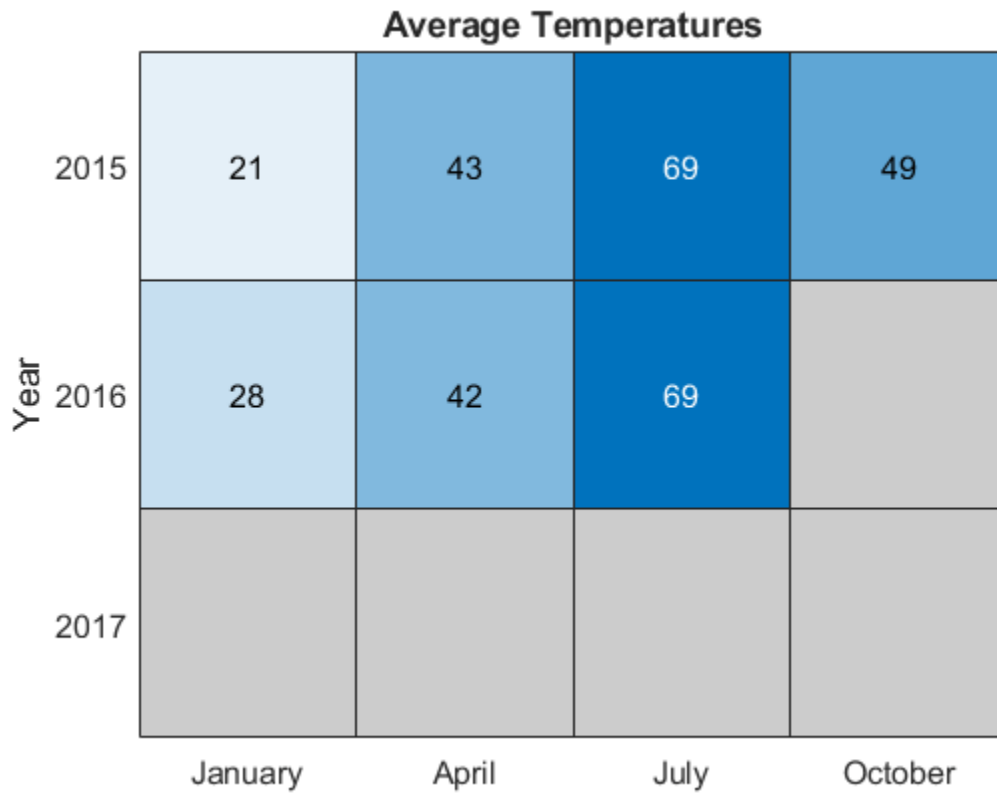
Since there is no data associated with the year 2017, the heatmap cells use the missing data color.

See Also

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

Properties
HeatmapChart

Create Word Cloud from String Arrays

This example shows how to create a word cloud from plain text by reading it into a string array, preprocessing it, and passing it to the `wordcloud` function. If you have Text Analytics Toolbox™ installed, then you can create word clouds directly from string arrays. For more information, see `wordcloud` (Text Analytics Toolbox) (Text Analytics Toolbox).

Read the text from Shakespeare's Sonnets with the `fileread` function.

```
sonnets = fileread('sonnets.txt');
sonnets(1:135)

ans =
    'THE SONNETS

     by William Shakespeare




        I

        From fairest creatures we desire increase,
        That thereby beauty's rose might never die,'
```

Convert the text to a string using the `string` function. Then, split it on newline characters using the `splitlines` function.

```
sonnets = string(sonnets);
sonnets = splitlines(sonnets);
sonnets(10:14)

ans = 5x1 string
    "  From fairest creatures we desire increase,"
    "  That thereby beauty's rose might never die,"
    "  But as the riper should by time decease,"
    "  His tender heir might bear his memory:"
    "  But thou, contracted to thine own bright eyes,"
```

Replace some punctuation characters with spaces.

```
p = ["." "?" "!" "," ";" ":"];
sonnets = replace(sonnets,p," ");
sonnets(10:14)

ans = 5x1 string
    "  From fairest creatures we desire increase "
    "  That thereby beauty's rose might never die "
    "  But as the riper should by time decease "
    "  His tender heir might bear his memory "
    "  But thou  contracted to thine own bright eyes "
```

Split `sonnets` into a string array whose elements contain individual words. To do this, join all the string elements into a 1-by-1 string and then split on the space characters.

```
sonnets = join(sonnets);
sonnets = split(sonnets);
sonnets(7:12)

ans = 6x1 string
    "From"
    "fairest"
    "creatures"
    "we"
    "desire"
    "increase"
```

Remove words with fewer than five characters.

```
sonnets(strlength(sonnets)<5) = [];
```

Convert `sonnets` to a categorical array and then plot using `wordcloud`. The function plots the unique elements of `C` with sizes corresponding to their frequency counts.

```
C = categorical(sonnets);
figure
wordcloud(C);
title("Sonnets Word Cloud")
```
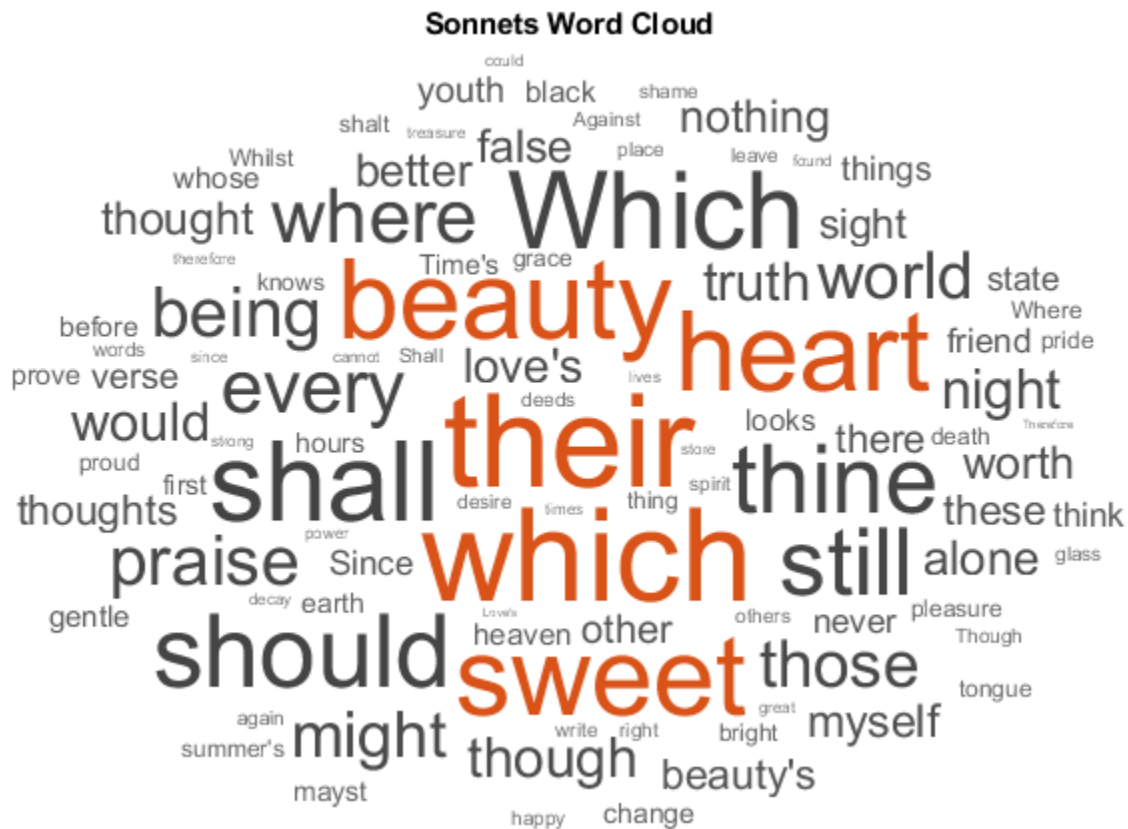
See Also

WordCloudChart Properties | `wordcloud`

Explore Table Data Using Parallel Coordinates Plot

This example shows how to import a file into MATLAB® as a table, create a parallel coordinates plot from the tabular data, and modify the appearance of the plot.

Parallel coordinates plots are useful for visualizing tabular or matrix data with multiple columns. The rows of the input data correspond to lines in the plot, and the columns of the input data correspond to coordinates in the plot. You can group the lines in the plot to better see trends in your data.

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 few rows.

```
tbl = readtable('TemperatureData.csv');
head(tbl)
```

ans=*8×4 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
2015	{'January'}	6	12
2015	{'January'}	7	10
2015	{'January'}	8	4

Create Basic Parallel Coordinates Plot

Create a parallel coordinates plot from the first few rows of the table. Each line in the plot corresponds to a single row in the table. By default, `parallelplot` displays all the coordinate variables in the table, in the same order as they appear in the table. The software displays the coordinate variable names below their corresponding coordinate rulers.

The plot shows that the first eight rows of the table provide temperature data for the first eight days in January 2015. For example, the eighth day was the coldest of the eight days, on average.

```
parallelplot(head(tbl))
```
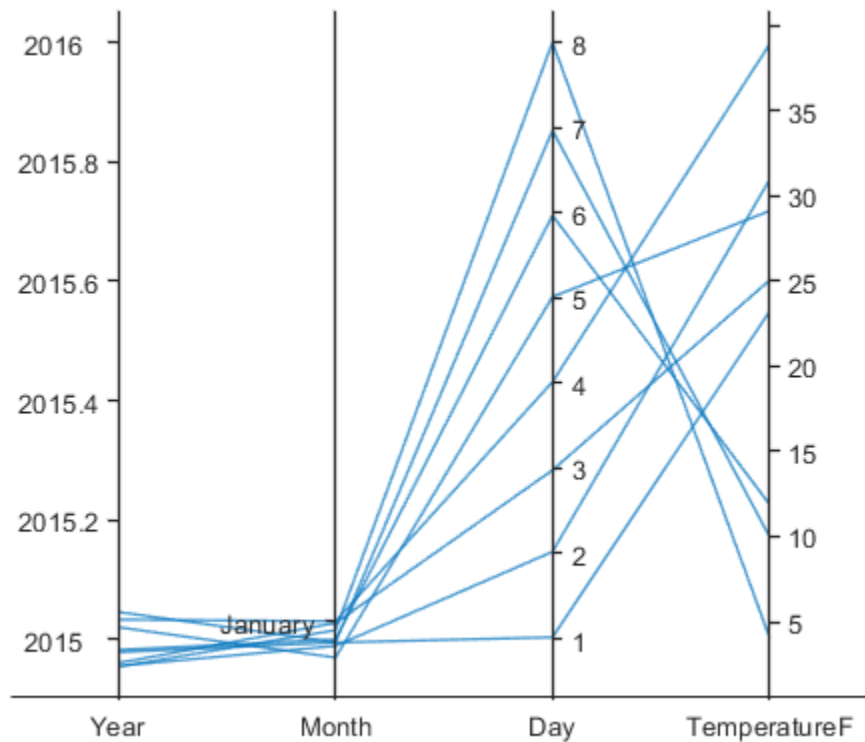
To help you interpret the plot, MATLAB randomly jitters plot lines by default so that they are unlikely to overlap perfectly along coordinate rulers. For example, although the first eight observations have the same `Year` and `Month` values, the plot lines are not flush with the `2015` tick mark along the `Year` coordinate ruler or the `January` tick mark along the `Month` coordinate ruler. Although jittering affects all coordinate variables, it is often more noticeable along categorical coordinate rulers because it depends on the distance between tick marks. You can control the amount of jittering in the plot by setting the `Jitter` property.

Notice that some of the tick marks along the `Year` coordinate ruler are meaningless decimal values. To ensure that tick marks along a coordinate ruler correspond only to meaningful values, convert the variable to a categorical variable by using the `categorical` function.

```
tbl.Year = categorical(tbl.Year);
```

Now create a parallel coordinates plot from the entire table. Assign the `ParallelCoordinatesPlot` object to the variable `p`, and use `p` to modify the plot after you create it. For example, add a title to the plot using the `Title` property.

```
p = parallelplot(tbl)

p =
  ParallelCoordinatesPlot with properties:

            SourceTable: [565x4 table]
    CoordinateVariables: {'Year'  'Month'  'Day'  'TemperatureF'}
          GroupVariable: ''
```

Show all properties

```
p.Title = 'Temperature Data';
```

Temperature Data
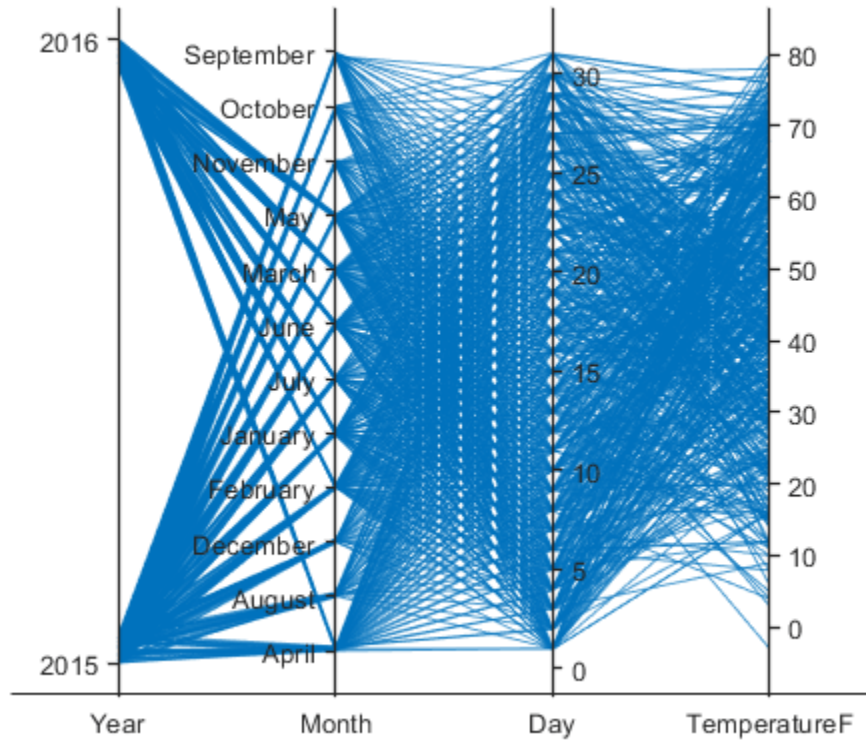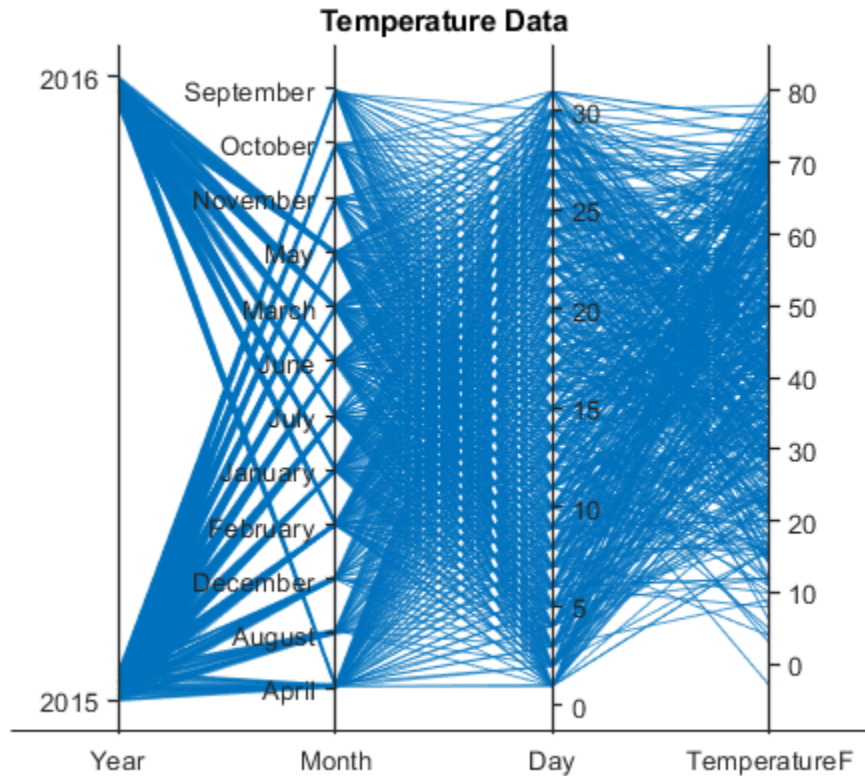
Group Plot Lines

Group the lines in the plot according to the `Year` values by setting the `GroupVariable` property. By default, MATLAB adds a legend to the plot. You can remove the legend by setting the `LegendVisible` property to `'off'`.

```
p.GroupVariable = 'Year';
```
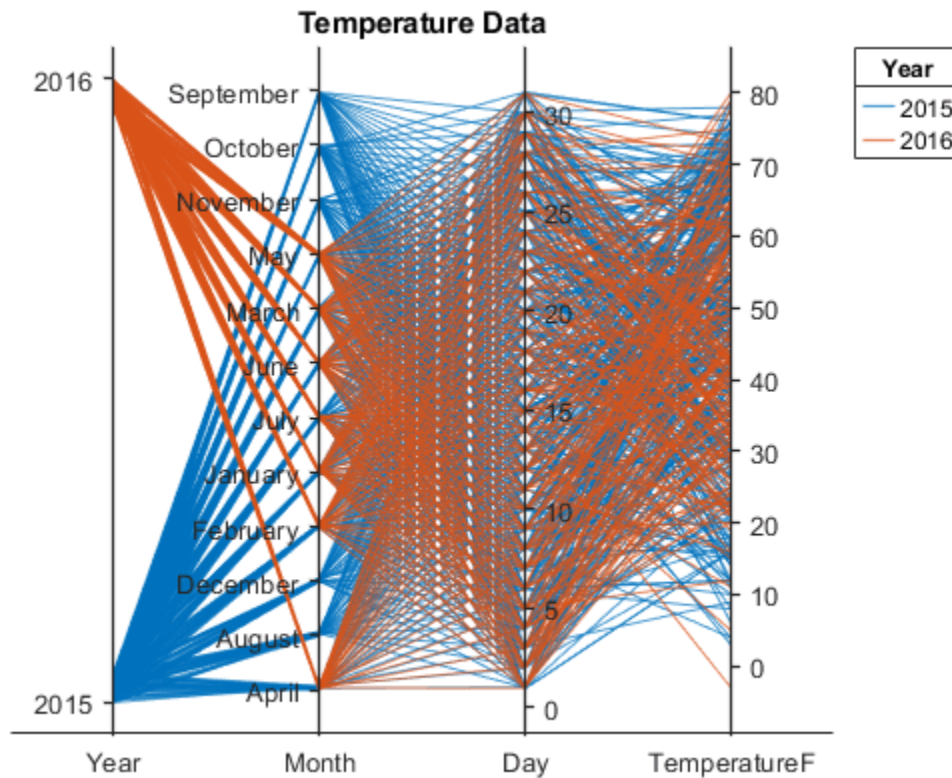
Rearrange Coordinate Variables Interactively

Rearrange coordinate variables interactively to compare them more easily and decide which variables to keep in your plot.

Open your plot in a figure window. Click a coordinate tick label and drag the associated coordinate ruler to the location of your choice. The software outlines the selected coordinate ruler in a black rectangle. For example, you can click the Month coordinate tick label and drag the coordinate ruler to the right. You can then easily compare Month and TemperatureF values.
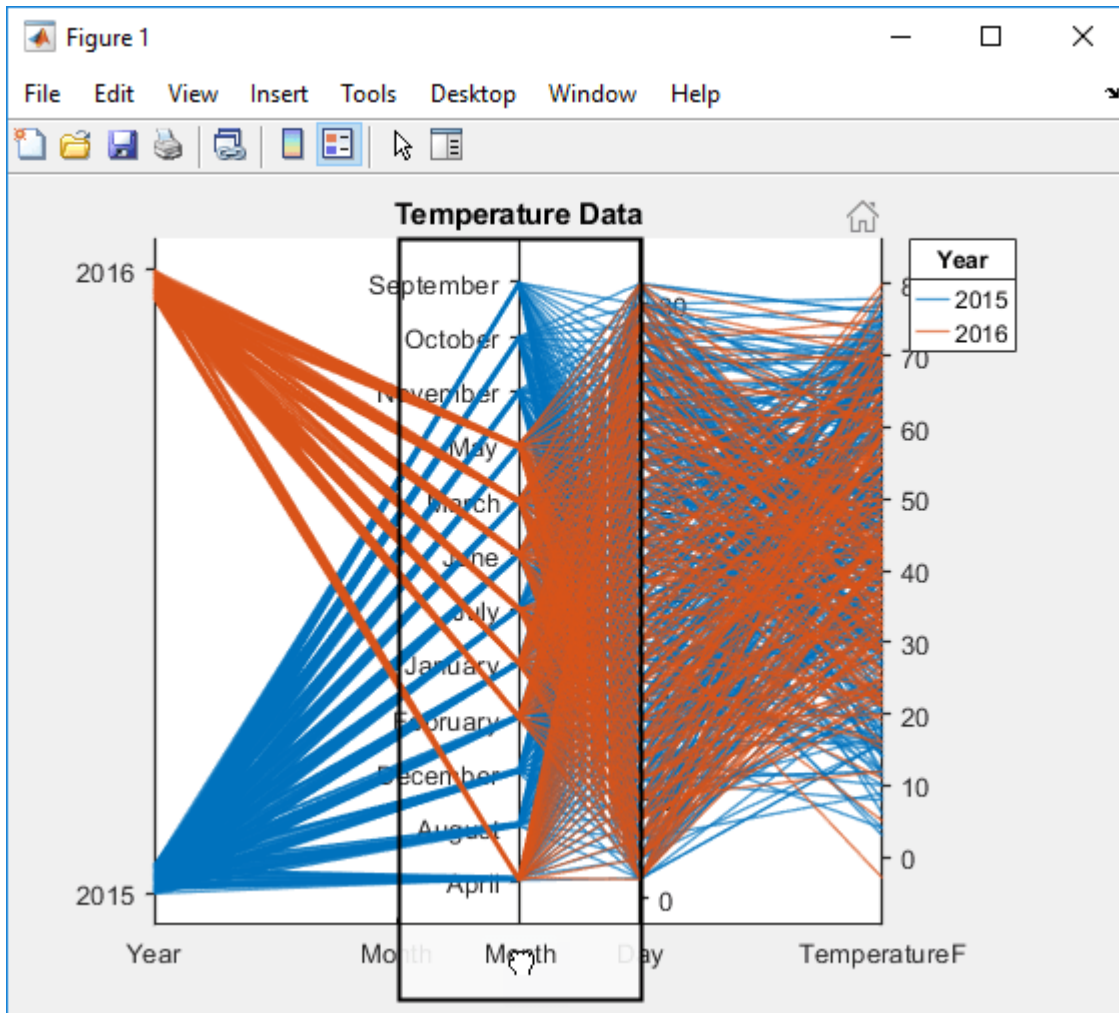
When you rearrange coordinate variables interactively, the software updates the associated `CoordinateTickLabels`, `CoordinateVariables`, and `CoordinateData` properties of the plot.

For more interactivity options, see "Tips".

Select Subset of Coordinate Variables

Display a subset of the coordinate variables in `p.SourceTable` and specify their order in the plot by setting the `CoordinateVariables` property of `p`.

In particular, remove the `Day` variable from the plot, and display the `TemperatureF` variable, which is in the fourth column of the source table, as the second coordinate in the plot.

```
p.CoordinateVariables = [1 4 2];
```
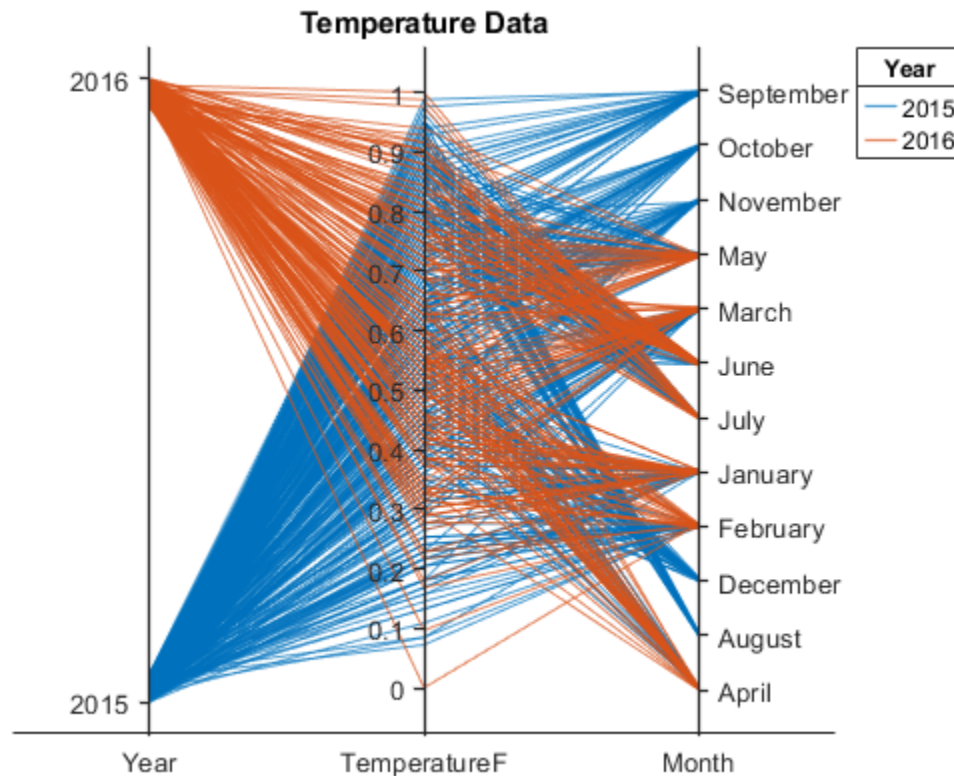
Alternatively, you can set the `CoordinateVariables` property by using a string or cell array of variable names or a logical vector with `true` elements for the selected variables.

Modify Categories in Coordinate Variable

Display a subset of the categories in `Month` and change the category order along the coordinate ruler in the plot.

Because some months have data for only one of the two years, remove the rows in the source table corresponding to those unique months. MATLAB updates the plot as soon as you change the source table.

```
uniqueMonth = {'September','October','November','December','August'};
uniqueMonthIdx = ismember(p.SourceTable.Month,uniqueMonth);
p.SourceTable(uniqueMonthIdx,:) = [];
```
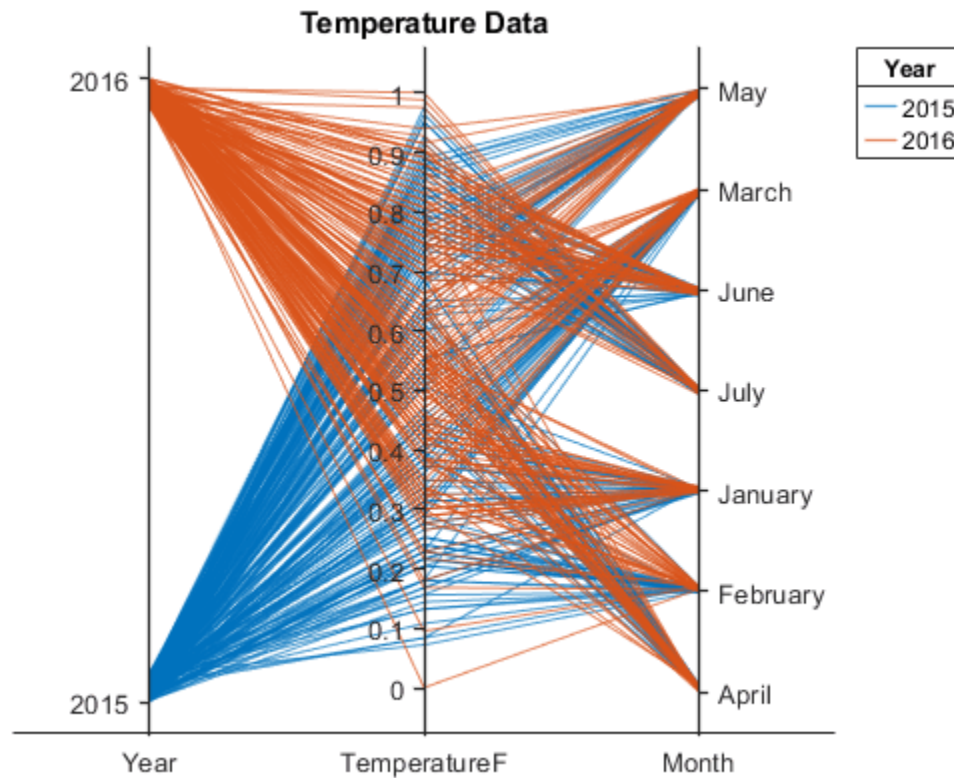
Arrange the months in chronological order along the `Month` coordinate ruler by updating the source table.

```
categoricalMonth = categorical(p.SourceTable.Month);
newOrder = {'January','February','March','April','May','June','July'};
orderMonth = reordercats(categoricalMonth,newOrder);
p.SourceTable.Month = orderMonth;
```
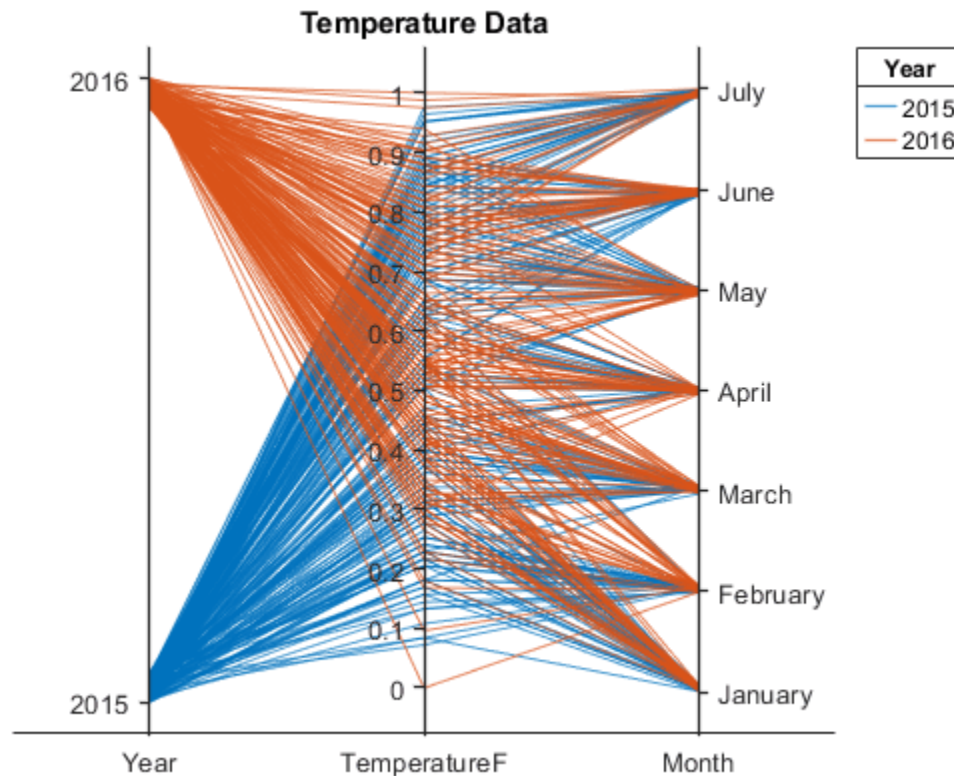
Temperature Data

Group Plot Lines Using Binned Values

To better visualize the range of temperatures during each month, bin the temperature data by using `discretize` and group the lines in the plot using the binned values. Check the minimum and maximum temperatures in the source table. Set the bin edges such that they include these values.

```
min(p.SourceTable.TemperatureF)
```

ans = -3

```
max(p.SourceTable.TemperatureF)
```

ans = 80

```
binEdges = [-3 10:10:80];
bins = {'00s+/-','10s','20s','30s','40s','50s','60s','70s+'};
groupTemperature = discretize(p.SourceTable.TemperatureF,binEdges,'categorical',bins);
```

Add the binned temperatures to the source table. Group the lines in the plot according to the binned temperature data.

```
p.SourceTable.GroupTemperature = groupTemperature;
p.GroupVariable = 'GroupTemperature';
```
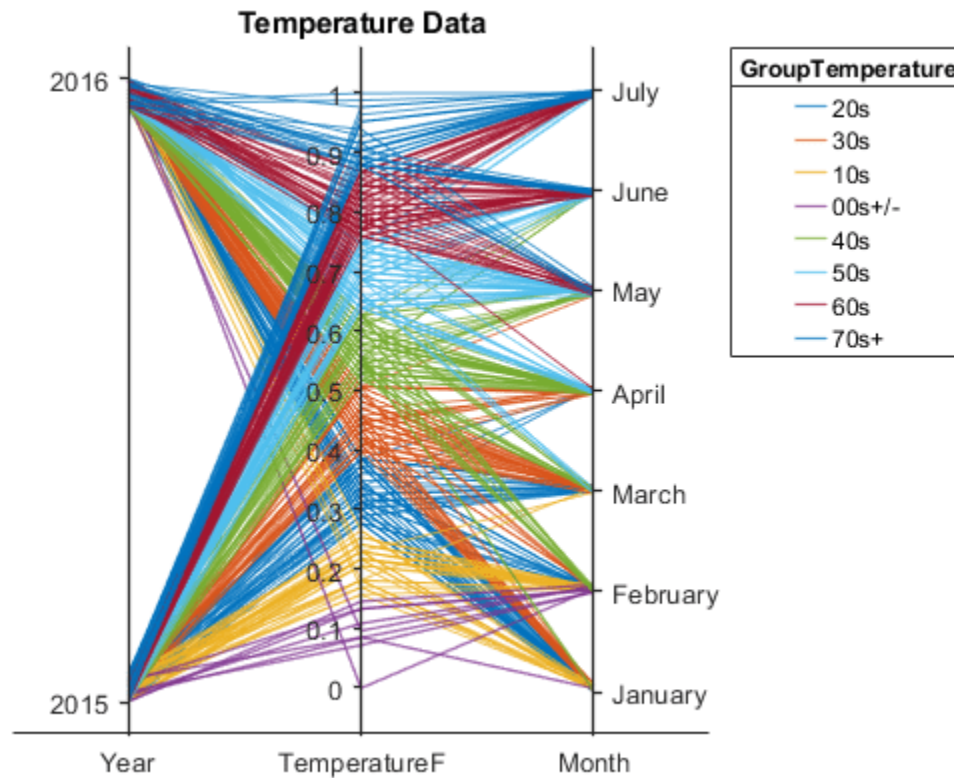
Because `GroupTemperature` includes more than seven categories, some of the groups have the same color in the plot. Assign distinct colors to every group by setting the `Color` property.

```
p.Color = jet(8);
```

Temperature Data
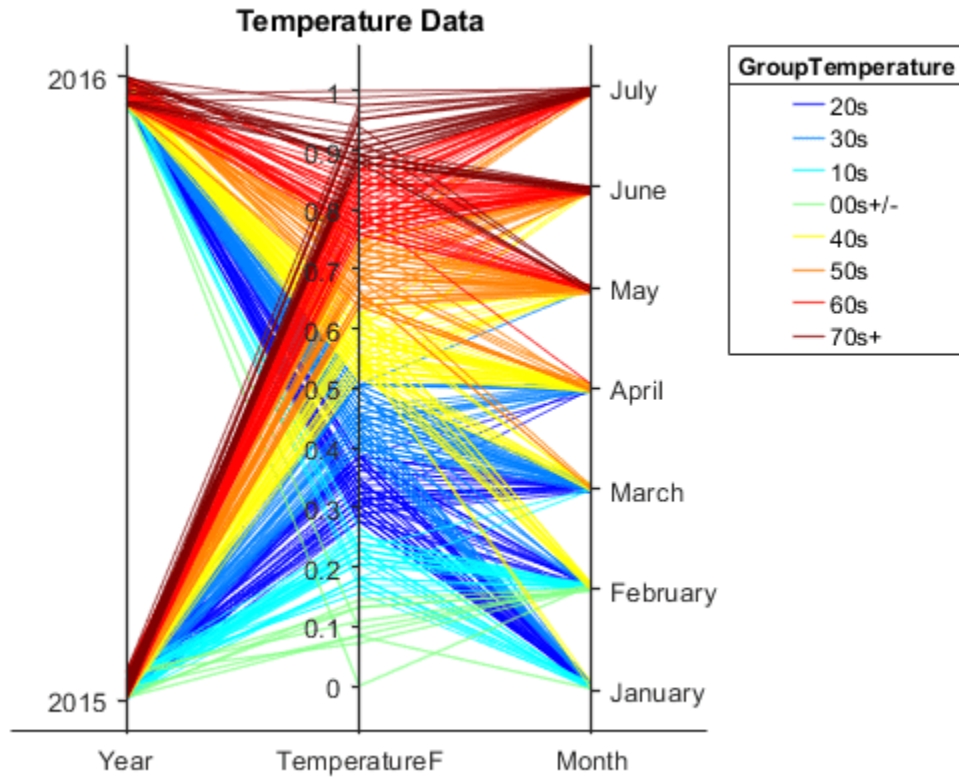
See Also

Functions
categorical | discretize | parallelplot | readtable | reordercats | table

Properties
ParallelCoordinatesPlot

Geographic Axes and Charts

Create Maps Using Latitude and Longitude Data

If you have data that is associated with specific geographic locations, use a geographic axes or chart to visualize your data on a map and provide visual context. For example, if you have data that describes the occurrences of tsunamis around the world, plot the data in a geographic axes where a marker indicates the location of each occurrence on a map. These examples show how to create line plots, scatter plots, bubble charts, and density plots in geographic coordinates.

Create Geographic Line Plot

Draw a line on a map between Seattle and Anchorage. Specify the latitude and longitude for each city, then plot the data using the `geoplot` function. Customize the appearance of the line using the line specification `'-*'`. Adjust the latitude and longitude limits of the map using `geolimits`. Change the basemap using the `geobasemap` function.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;

geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],'-*')
geolimits([45 62],[-149 -123])
geobasemap streets
```
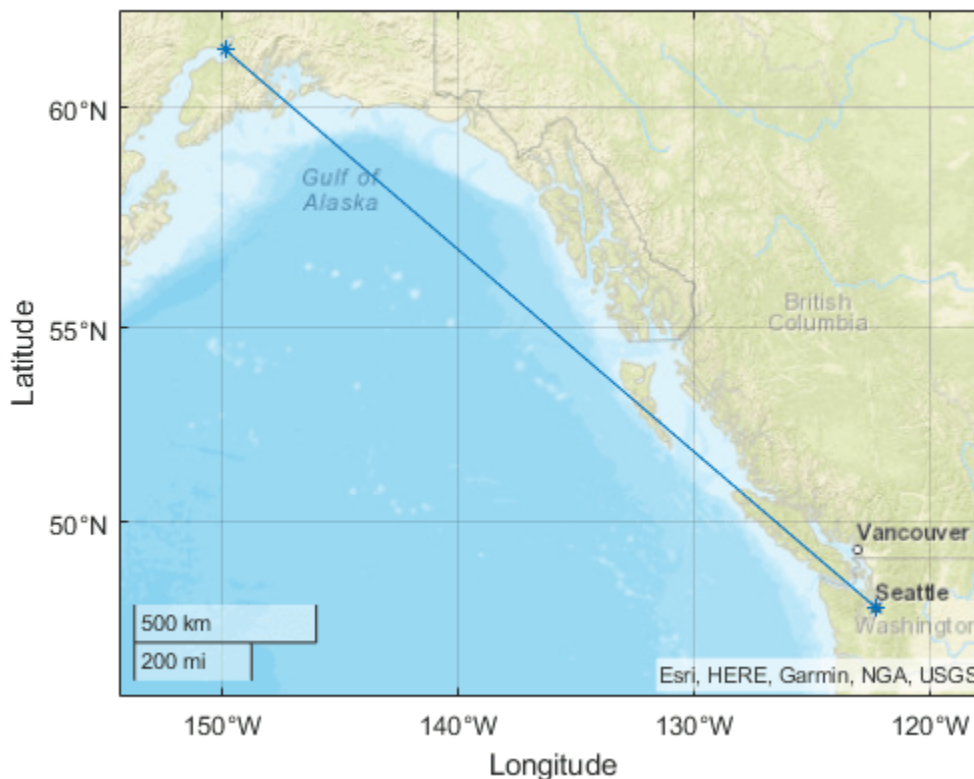
Create Geographic Scatter Plot

Create latitude and longitude positions and define values at each point. Plot the values on a map using the `geoscatter` function. The example specifies the triangle as the marker, with size and color representing variations in the values.

```
lon = (-170:10:170);
lat = 50 * cosd(3*lon);
A = 101 + 100*(sind(2*lon));
C = cosd(4*lon);

geoscatter(lat,lon,A,C,'^')
```
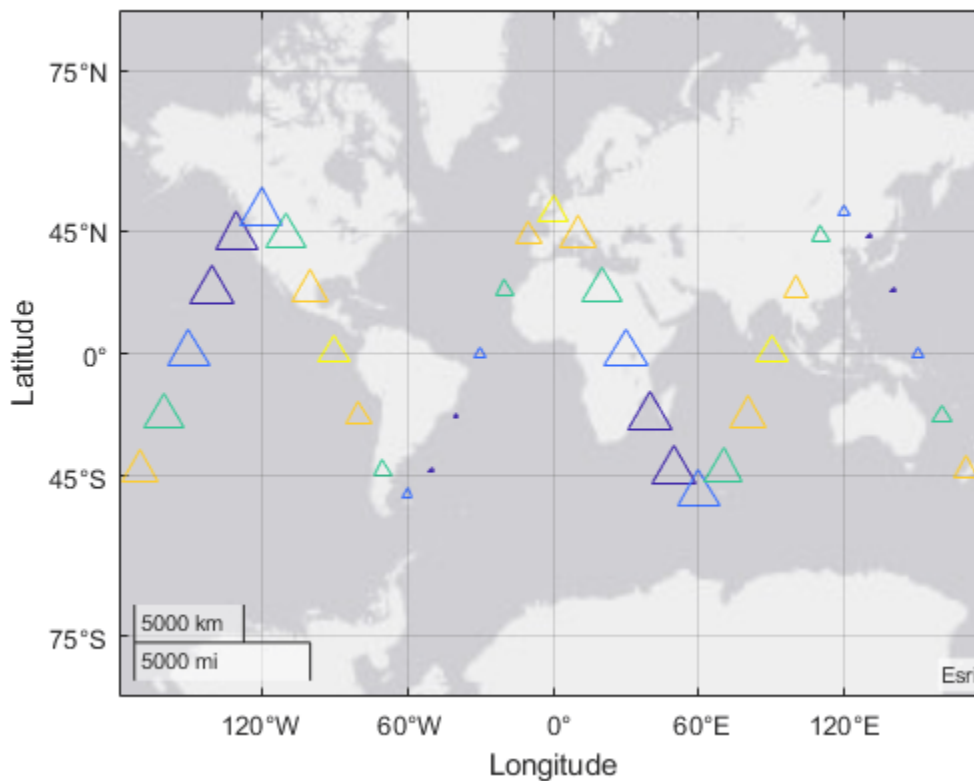

Create Geographic Bubble Chart

Create a table from tsunami data. Define one value as a categorical value. Plot the data on a map using the `geobubble` function. The example uses the size of the bubble to indicate the height of the tsunami wave and color to indicate the cause of the tsunami.

```
tsunamis = readtable('tsunamis.xlsx');
tsunamis.Cause = categorical(tsunamis.Cause);
figure
gb = geobubble(tsunamis,'Latitude','Longitude', ...
        'SizeVariable','MaxHeight','ColorVariable','Cause');
geolimits([10 65],[-180 -80])
title 'Tsunamis in North America';
gb.SizeLegendTitle = 'Maximum Height';
geobasemap colorterrain
```
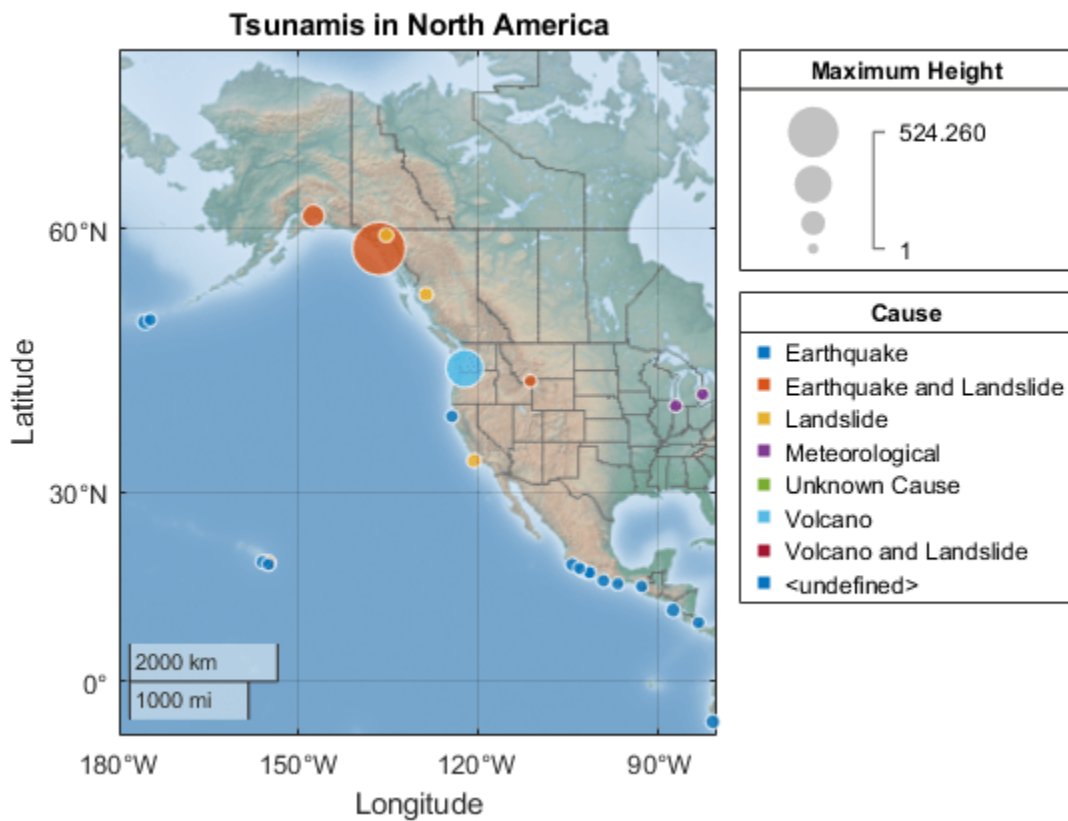
Create Geographic Density Plot

Create a table from tsunami data. Plot the data using the `geodensityplot` function.

```
tsunamis = readtable('tsunamis.xlsx');
lat = tsunamis.Latitude;
lon = tsunamis.Longitude;
weights = tsunamis.MaxHeight;

geodensityplot(lat,lon,weights)
geolimits([41.2 61.4],[-148.6 -107.0])
geobasemap topographic
```
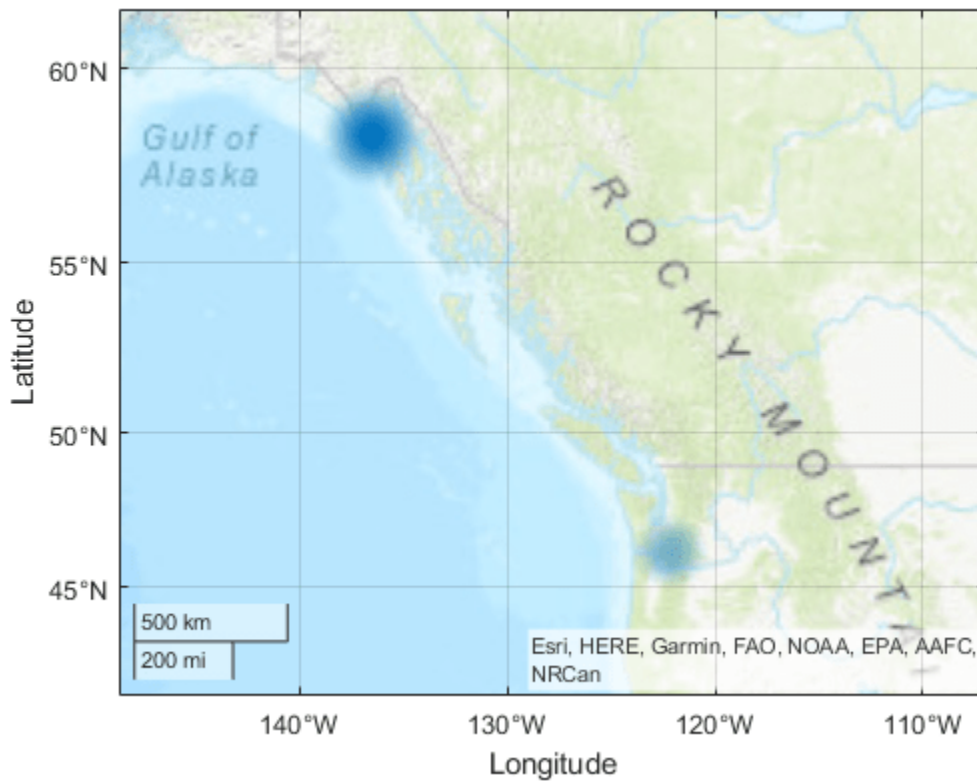
See Also

Functions
geoaxes | geobubble | geodensityplot | geoplot | geoscatter

Properties
GeographicBubbleChart Properties

Related Examples

- "Use Geographic Bubble Chart Properties" on page 6-27
- "Access Basemaps for Geographic Axes and Charts" on page 6-35
- "Create Geographic Bubble Chart from Tabular Data" on page 6-41

Pan and Zoom Behavior in Geographic Axes and Charts

The basemap in a geographic axes or chart is *live*, that is, you can pan the basemap, to view other geographic locations, or zoom in and out on the map to view regions in more detail. The map updates as you pan and zoom. On geographic axes and charts, pan and zoom capabilities are enabled by default.

To pan the basemap in a geographic axes or chart, use the arrow keys or move the cursor over the map and click and drag the basemap. You can pan the map in the horizontal direction continuously— longitude wraps. Panning in the vertical direction stops just beyond 85 degrees, north and south.

To zoom in and out on the map in a geographic axes or chart, you can use the scroll wheel, trackpad, or the **Plus** and **Minus** keys on the keyboard.

You can also zoom in, zoom out, or restore the original view of the map by using the axes toolbar. When you move the cursor over the map, the axes toolbar appears. When you move the cursor away from the map, the axes toolbar disappears.
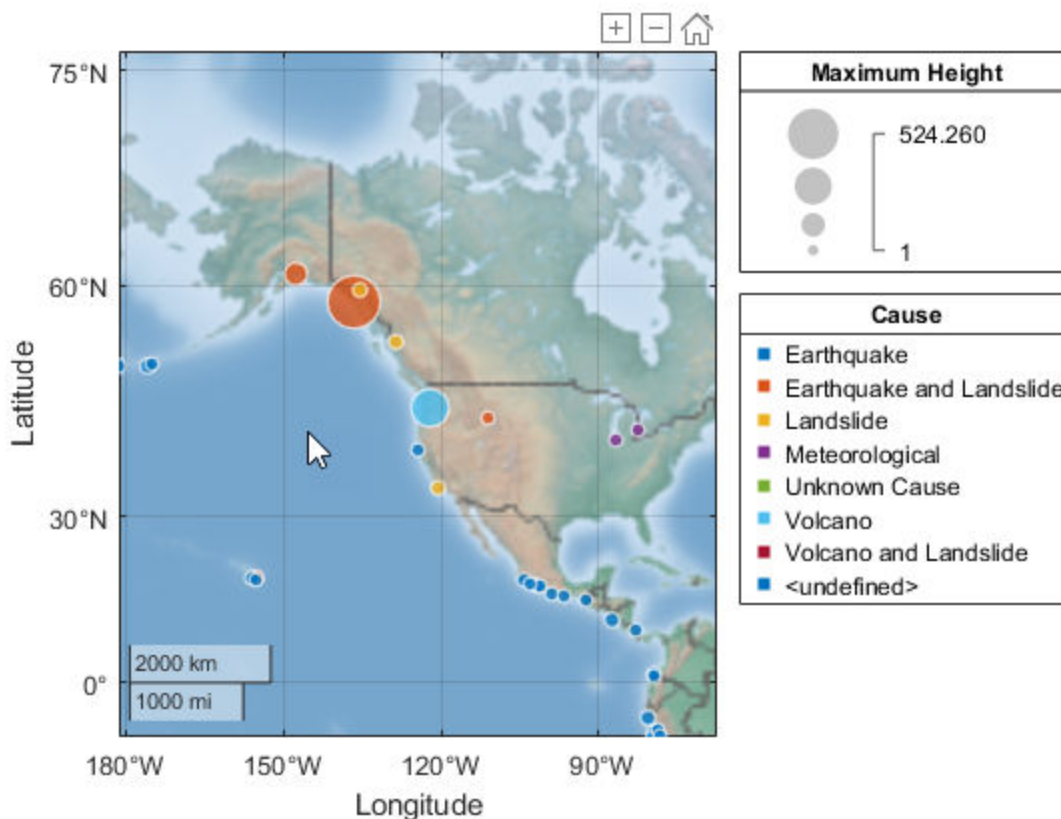
See Also

geoaxes | geobubble | geodensityplot | geoplot | geoscatter

Related Examples

- "Use Geographic Bubble Chart Properties" on page 6-27

Geographic Bubble Charts Overview

Using a map as a background, the geographic bubble chart plots your data as filled, colored circles, called bubbles, at locations on the map specified by longitude and latitude. You can use the size and color of the bubbles to indicate data values at these locations.

Suppose that you have data that describes the occurrences of tsunamis around the world. Plot the data in a geographic bubble chart where the bubbles mark each occurrence on a map, called a basemap. You can use bubble size to indicate the height of the wave and bubble color to indicate the cause. With the map as background, you can immediately see tsunami occurrences and their severity. Plotting the data on a map is an effective way to visualize your data.

A geographic bubble chart includes these components (shown in the following figure):

Geographic Bubble Chart Components

Component	Description
Basemap	The map over which the geographic bubble chart plots the data. For more information, see "Access Basemaps for Geographic Axes and Charts" on page 6-35.
Bubbles	Symbols that mark map locations and communicate other information through their size and color.
Data Tips	Small windows that pop open containing information about the bubble, such as latitude and longitude.
Decorations	Descriptive visual elements of the chart, such as latitude and longitude grids, and a scale bar, which shows how distances are represented on the map. The chart updates these elements as you zoom in and out on the map. Use geographic bubble chart properties to control the visibility of these elements, such as the `ScalebarVisible` property.
Legends	Displays of tabular information that explain the meaning of bubble size and bubble color. For more information, see "Geographic Bubble Chart Legends" on page 6-10.
Title	Text displays at the top of the chart, similar to any MATLAB figure. You can specify this using the geographic bubble chart `Title` property or the `title` command.
Axes Toolbar	Set of controls that let you zoom in or out on the map, or return to the original view of the map. For more information, see "Pan and Zoom Behavior in Geographic Axes and Charts" on page 6-6.
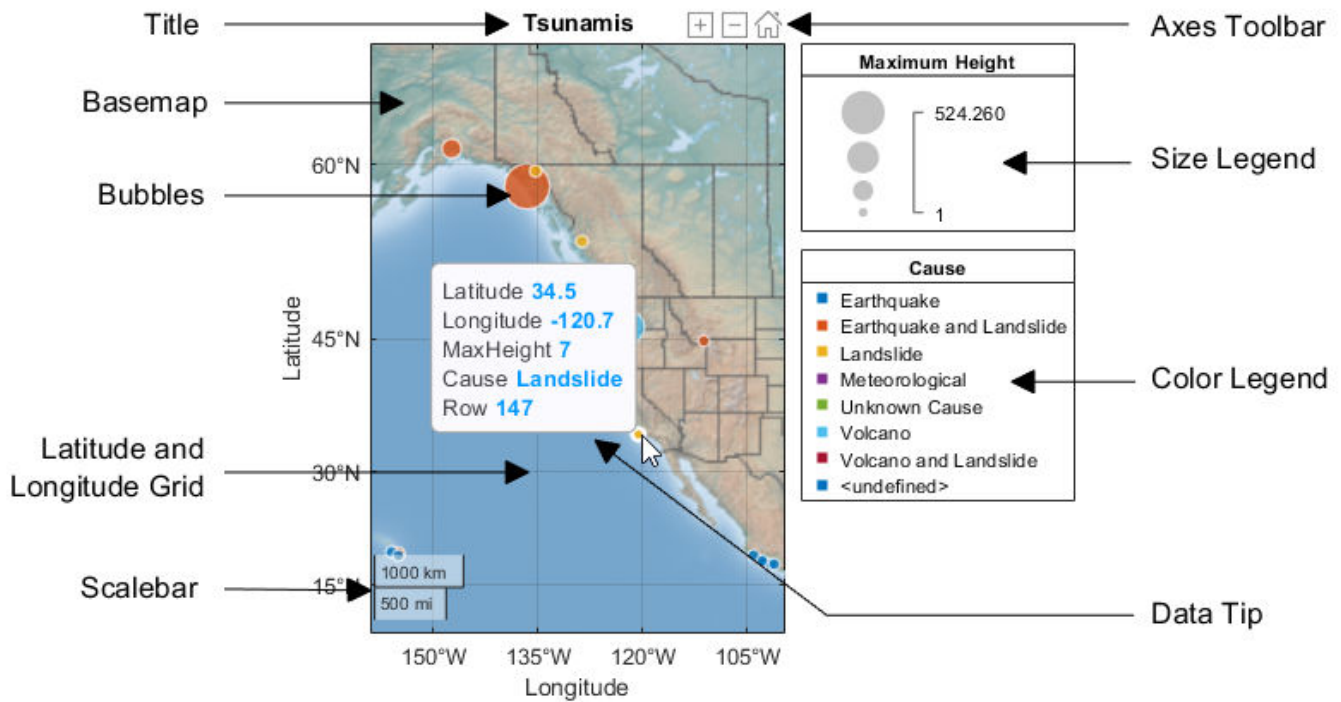
See Also

GeographicBubbleChart Properties | `geobubble`

Related Examples

- "Use Geographic Bubble Chart Properties" on page 6-27
- "Access Basemaps for Geographic Axes and Charts" on page 6-35
- "Create Geographic Bubble Chart from Tabular Data" on page 6-41

Geographic Bubble Chart Legends

When you create a geographic bubble chart with `SizeData`, the chart includes a size legend that explains how the bubble sizes represent the data. The legend includes a sampling of four bubble sizes, smallest to largest. You can specify the widths of the smallest and largest bubbles using the `BubbleWidthRange` property. The legend labels the smallest and largest bubbles in the legend with their associated numeric values. The legend gets these values from the `SizeLimits` property. If you are specifying `SizeData` directly, the legend has no title. You can specify a title for the legend using the `SizeLegendTitle` property. If you are specifying a table variable for size data, the legend uses the variable name as the size legend title. The legend includes a sampling of four bubble sizes, smallest to largest. The widths of the smallest and largest bubbles can be specified using the `BubbleWidthRange` property. The legend labels the smallest and largest bubbles in the legend with their associated numeric values.

Similarly, if you create a geographic bubble chart with `ColorData`, the chart includes a color legend that shows how bubble colors map to your categorical data. The legend includes all the colors, labeled with their associated category. If you are specifying `ColorData` directly, the legend has no title. You can specify a title for the legend using the `ColorLegendTitle` property. If you are specifying a table variable for color data, the legend uses the variable name as the color legend title.

The following illustration shows the geographic bubble chart size and color legends.
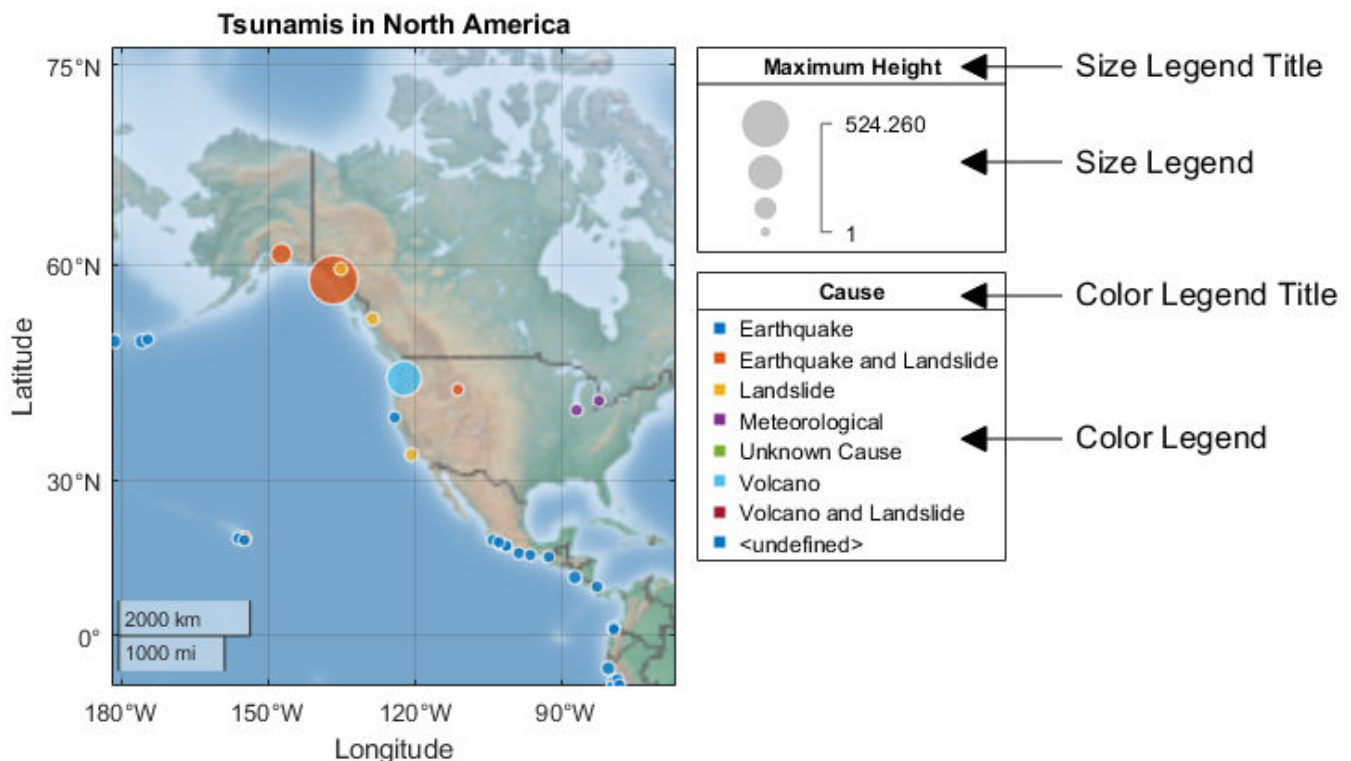
See Also

GeographicBubbleChart Properties | `geobubble`

Related Examples

- "Use Geographic Bubble Chart Properties" on page 6-27
- "Access Basemaps for Geographic Axes and Charts" on page 6-35
- "Create Geographic Bubble Chart from Tabular Data" on page 6-41

View Cyclone Track Data in Geographic Density Plot

This example shows how to view cyclone tracking data in a geographic density plot. The data records observations of cyclones over an 11 year period, between 2007-2017.

Load the cyclone track data and display the first few rows. The data, produced by the Japan Meteorological Agency, records the location, pressure (in hPa), and wind speed (knots) of cyclones at six-hour intervals. Each row in the table represents the record of an observation of a particular cyclone, identified by a name and an ID number.

```
load cycloneTracks
head(cycloneTracks)
```

```
ans =

  8×8 table

    ID       Name              Time           Grade    Latitude    Longitude    Pressure    W:
    ___    _____    _____    _____    _____    _____    _____    _

    701    "KONG-REY"    30-Mar-2007 12:00:00      2        5.8         158.2        1008
    701    "KONG-REY"    30-Mar-2007 18:00:00      2        6.3         158.1        1004
    701    "KONG-REY"    31-Mar-2007 00:00:00      2        6.7         157.9        1006
    701    "KONG-REY"    31-Mar-2007 06:00:00      2        7.1         156.6        1004
    701    "KONG-REY"    31-Mar-2007 12:00:00      2        7.6         155.5        1004
    701    "KONG-REY"    31-Mar-2007 18:00:00      2        8.8         154.2        1002
    701    "KONG-REY"    01-Apr-2007 00:00:00      3        9.7         152.8        1000
    701    "KONG-REY"    01-Apr-2007 06:00:00      3       10.2         152.1         996
```

To understand the data, plot the tracks of three cyclones, using the `geoplot` function. Get the data records for three cyclones, identified by ID number and name. Each observation record provides the latitude and longitude. Plot all the three cyclone tracks on one map by turning `hold` on.

```
figure
latMalakas = cycloneTracks.Latitude(cycloneTracks.ID == 1012);
lonMalakas = cycloneTracks.Longitude(cycloneTracks.ID == 1012);
geoplot(latMalakas,lonMalakas,'.-')
geolimits([0 60],[100 180])
hold on
latMegi = cycloneTracks.Latitude(cycloneTracks.ID == 1013);
lonMegi = cycloneTracks.Longitude(cycloneTracks.ID == 1013);
geoplot(latMegi,lonMegi,'.-')
latChaba = cycloneTracks.Latitude(cycloneTracks.ID == 1014);
lonChaba = cycloneTracks.Longitude(cycloneTracks.ID == 1014);
geoplot(latChaba,lonChaba,'.-')
```
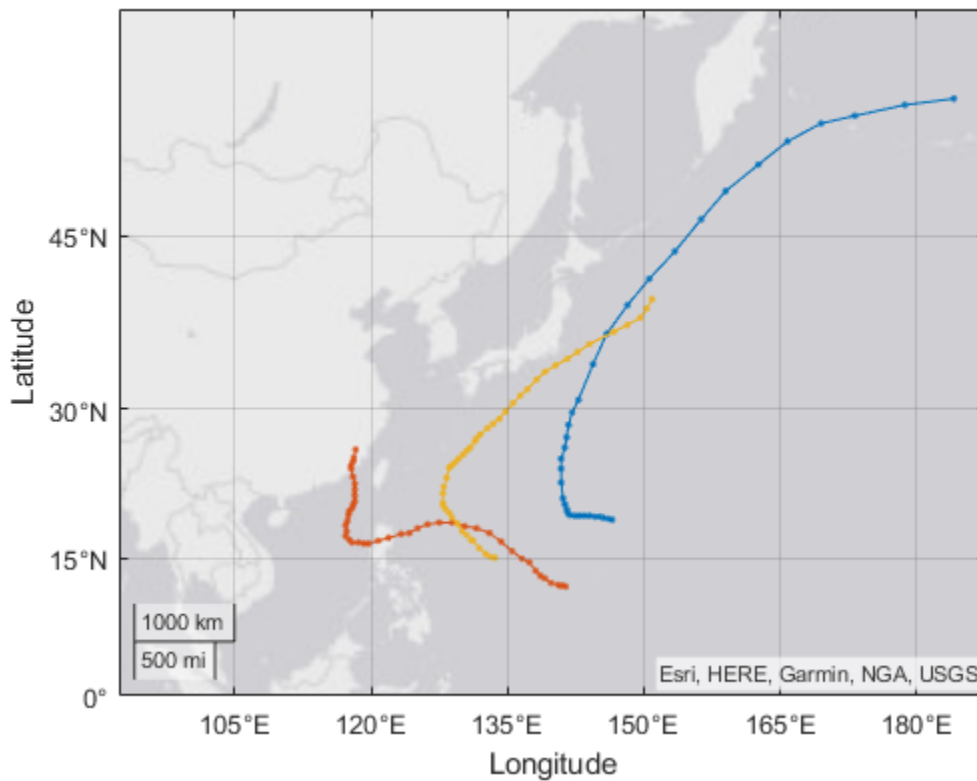
View the density of all cyclones tracked over this 11-year period using `geodensityplot`. In this plot, instead of a seeing the track of a particular cyclone, view all the records at every point for all the cyclones. `geodensityplot` calculates a cumulative probability distribution surface using contributions from the individual locations. The surface transparency varies with density.

```
figure
latAll = cycloneTracks.Latitude;
lonAll = cycloneTracks.Longitude;
geodensityplot(latAll,lonAll)
```
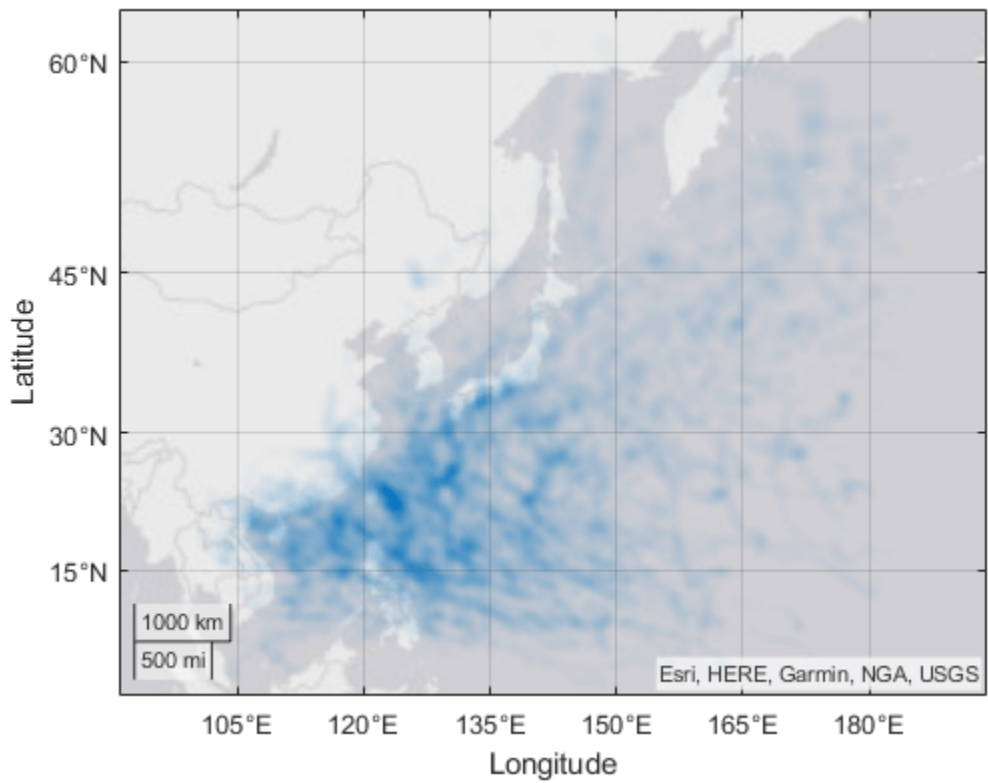
By default, `geodensityplot` uses a single color to represent all density values, using transparency to represent density variation. You can also use multiple colors with `geodensityplot` to represent areas of varying density. To do this, set the `'FaceColor'` property.

```
geodensityplot(latAll,lonAll,'FaceColor','interp')
```
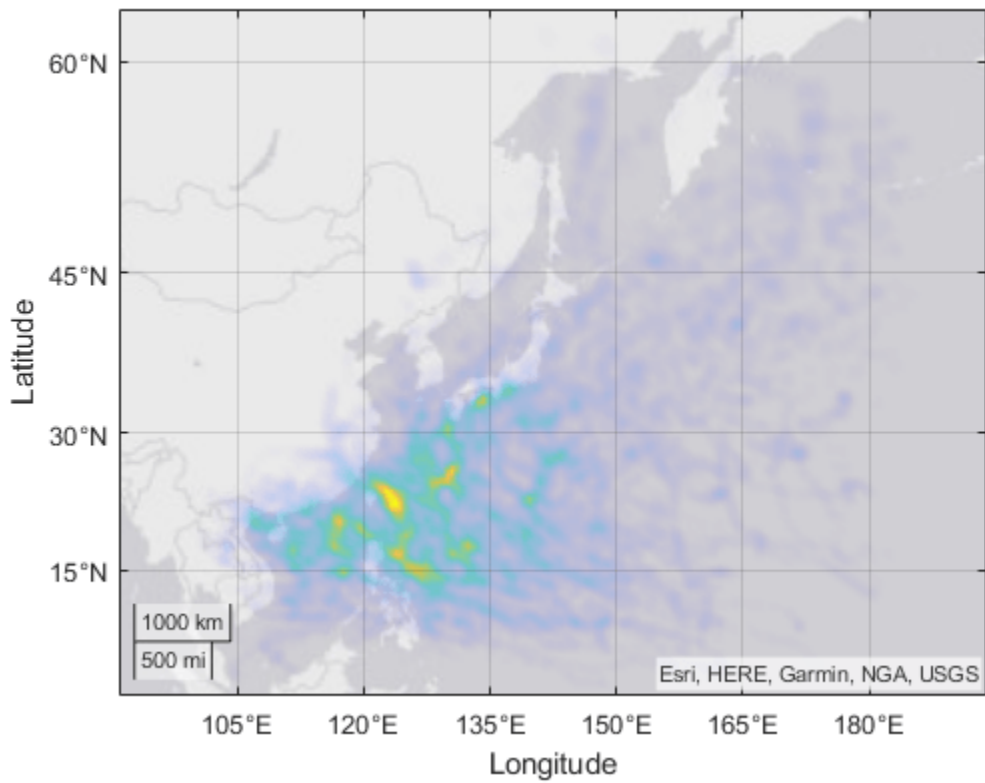
A density plot can apply weights to individual data points. The weights multiply the contribution of individual points to the density surface.

```
windspeedAll = cycloneTracks.WindSpeed;
geodensityplot(latAll,lonAll,windspeedAll,'FaceColor','interp')
```
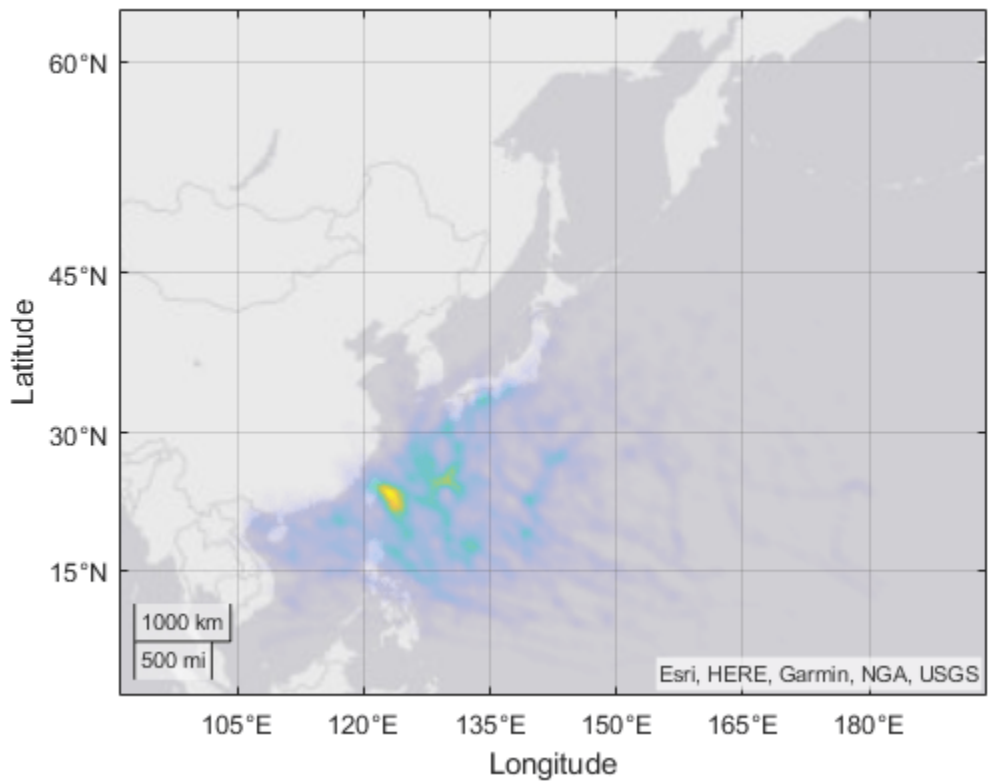
Reference: This cyclone track data was modified for use in this example by MathWorks from the RSMC Best Track Data by the Japan Meteorological Agency (https://www.jma.go.jp/jma/jma-eng/jma-center/rsmc-hp-pub-eg/RSMC_HP.htm).

See Also

DensityPlot Properties | `geodensityplot`

Related Examples

- "Access Basemaps for Geographic Axes and Charts" on page 6-35

View Density of Cellular Tower Placement

This example shows how to use a geographic density plot to view the density of cellular tower placement in California.

Load Cellular Tower Placement Data

Load a table of cellular tower placement data into the workspace and view the first few rows. The table includes fields that identify the location of the cellular tower by latitude and longitude, and identify the type of tower.

```
load cellularTowers
head(cellularTowers)
```

```
ans=8×10 table
    ID      Latitude    Longitude         City             County        State    NEPA    SUPSTRU
    ____    _____    _____    _____    _____    _____    ____    _____

    2166     37.119     -121.83      "MORGAN HILL"       SANTA CLARA      CA       N        46.9
    2167     37.346     -121.63      "SAN JOSE"          SANTA CLARA      CA       N         2.4
    2168     37.158     -121.98      "REDWOOD ESTATES"   SANTA CLARA      CA       N        24.7
    2169     37.366     -122.14      "LOS ALTOS HILLS"   SANTA CLARA      CA       N        18.3
    2170     37.402     -122.18      "STANFORD"          SANTA CLARA      CA       N         6.4
    2171     37.258     -122.03      "SARATOGA"          SANTA CLARA      CA       N        10.1
    2172     37.434     -121.89      "MILPITAS"          SANTA CLARA      CA       N        17.1
    2173     37.446     -121.89      "MILPITAS"          SANTA CLARA      CA       N        19.5
```

View the Data as a Geographic Scatter Plot

Plot the cellular tower data using the `geoscatter` function. In the plot, there are clear areas around San Francisco where the number of towers are too dense to be represented using a scatter plot.

```
geoscatter(cellularTowers.Latitude, cellularTowers.Longitude, '.')
text(gca,37.75,-122.75,'San Francisco','HorizontalAlignment','right')
```
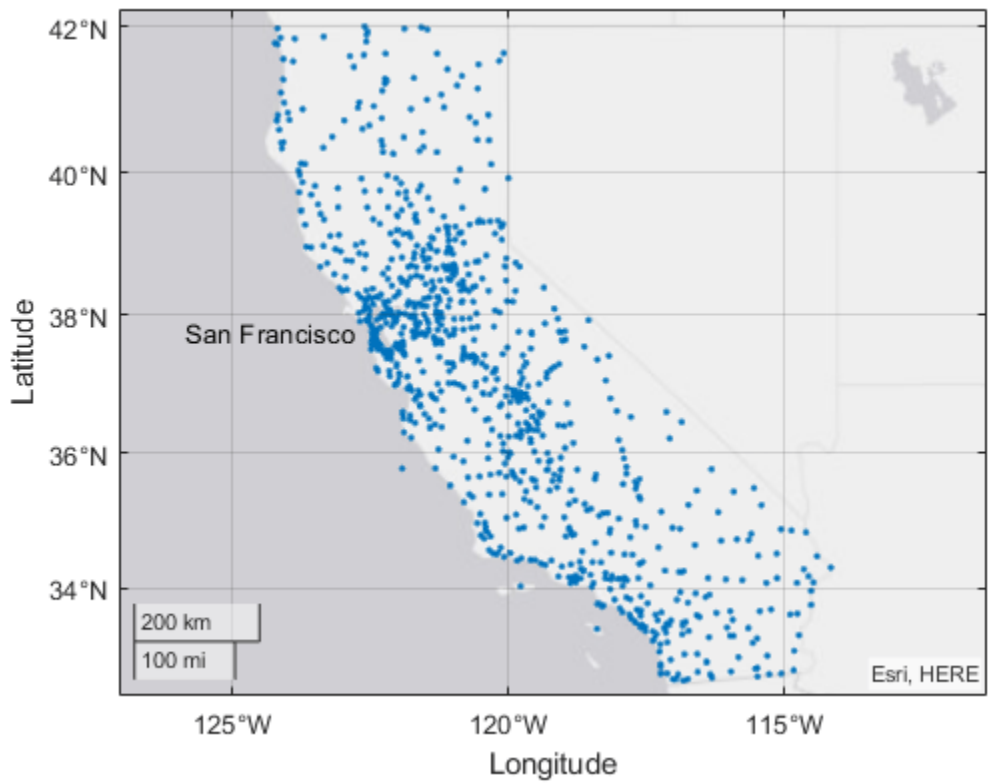
View the Data as a Geographic Density Plot

The dense area of towers in the San Francisco area can be shown using `geodensityplot`.

```
geodensityplot(cellularTowers.Latitude, cellularTowers.Longitude)
text(gca,37.75,-122.75,'San Francisco','HorizontalAlignment','right')
```
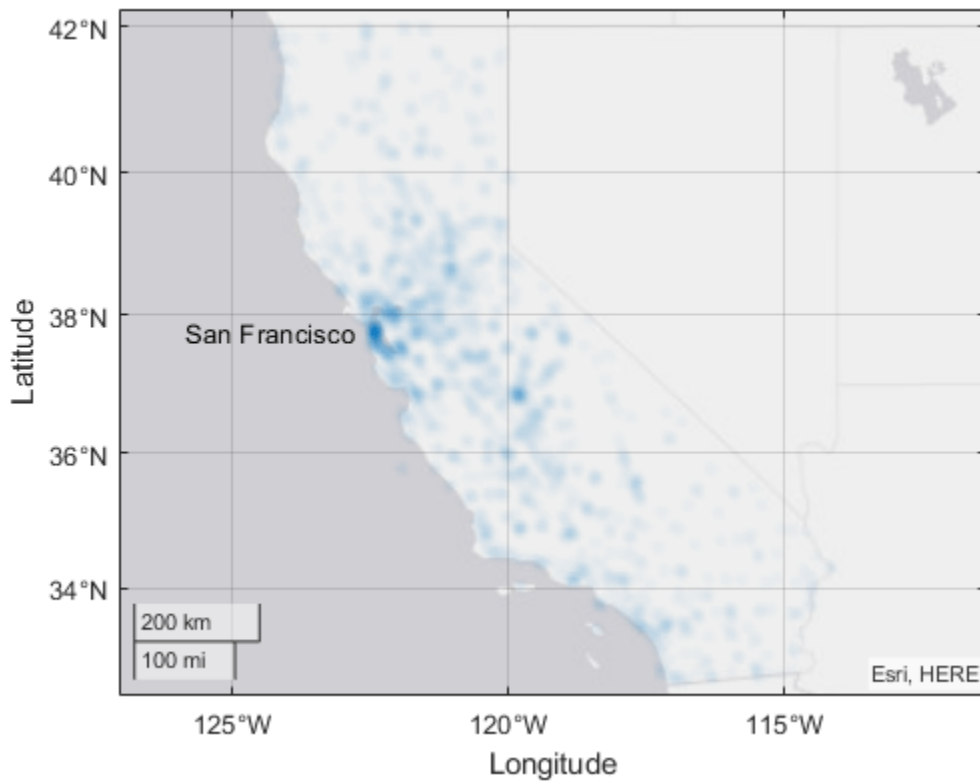
Create a Density Plot Specifying the Radius

When you create a geographic density plot, by default, the density plot automatically selects a radius value, using the latitude and longitude data. Use the `Radius` property to manually select a radius in meters.

```
radiusInMeters = 50e3; % 50 km
geodensityplot(cellularTowers.Latitude, cellularTowers.Longitude,'Radius',radiusInMeters)
```
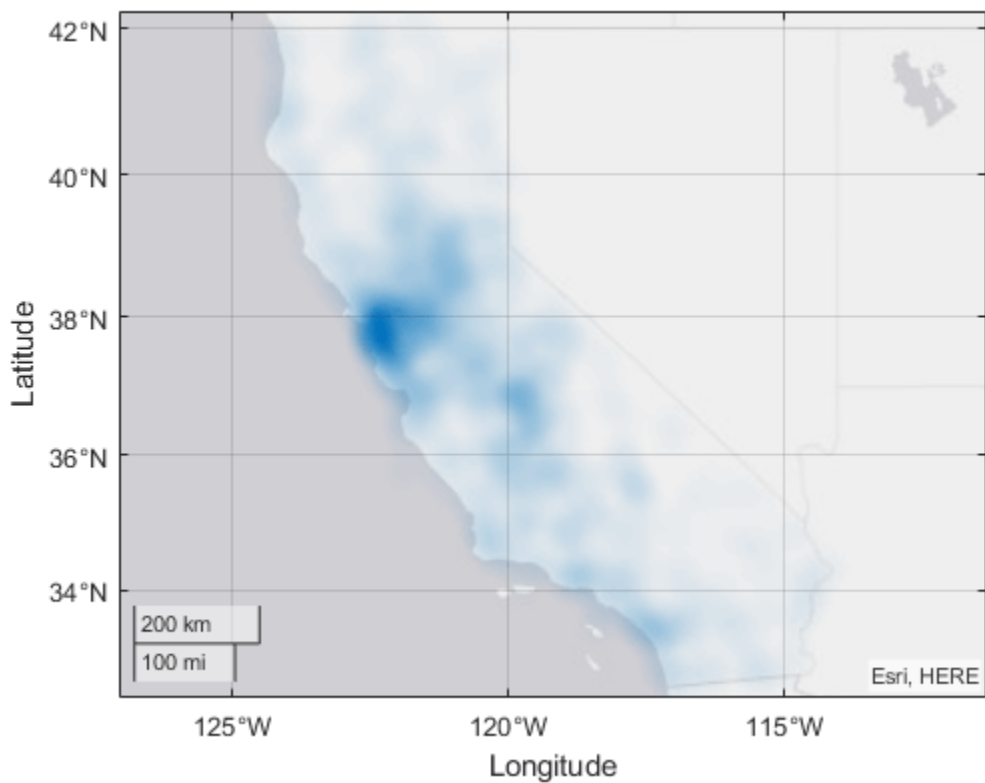
Use axes properties to adjust transparency

When set to `'interp'`, the density plot's `FaceAlpha` and `FaceColor` properties use the `Alphamap` and `Colormap` properties of the underlying geographic axes, respectively. Changing the `Alphamap` changes the mapping of the density values to color intensities.

```
geodensityplot(cellularTowers.Latitude, cellularTowers.Longitude)
alphamap(normalize((1:64).^0.5,'range'))
```
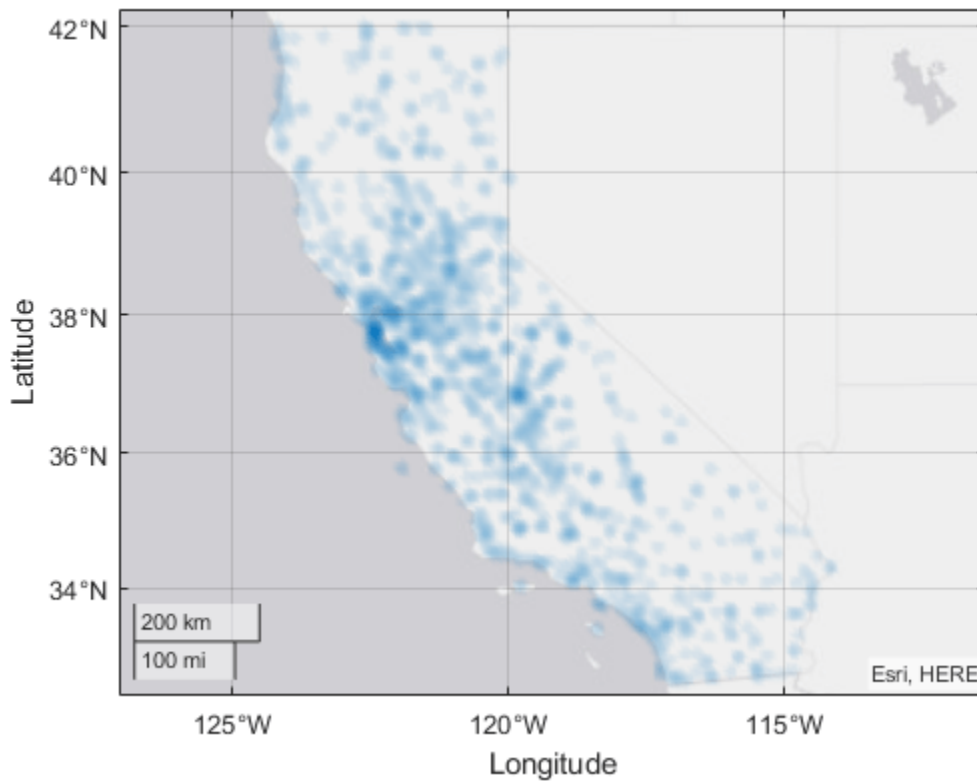
The `AlphaScale` property on the geographic axes can also be used to alter the transparency. This property is particularly useful when trying to show where any density is found, rather than highlighting the most dense areas.

```
figure
dp = geodensityplot(cellularTowers.Latitude, cellularTowers.Longitude)

dp =
  DensityPlot with properties:

        FaceColor: [0 0.4470 0.7410]
        FaceAlpha: 'interp'
     LatitudeData: [1×1193 double]
    LongitudeData: [1×1193 double]
       WeightData: [1×0 double]
           Radius: 1.8291e+04

  Show all properties


gx = gca

gx =
  GeographicAxes with properties:

     Basemap: 'streets-light'
    Position: [0.1300 0.1100 0.7750 0.8150]
       Units: 'normalized'
```

6-21

```
Show all properties
```

```
gx.AlphaScale = 'log';
```
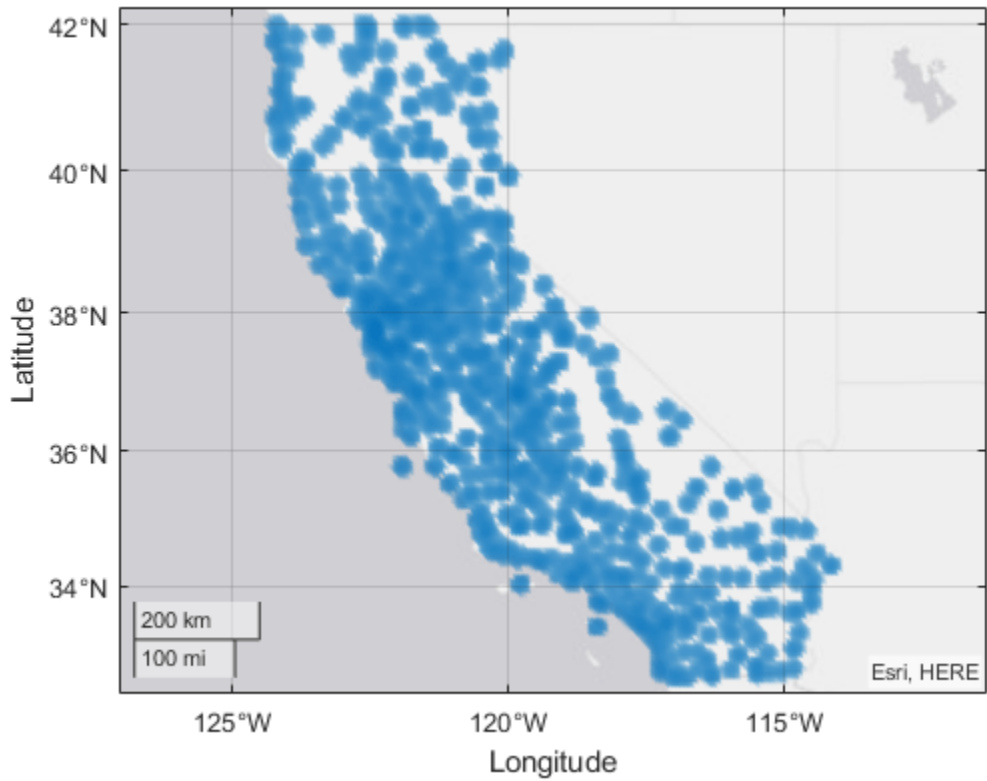

Use `DensityPlot` Object Properties to Specify Color

Add color.

```
dp.FaceColor = 'interp';
colormap hot
```
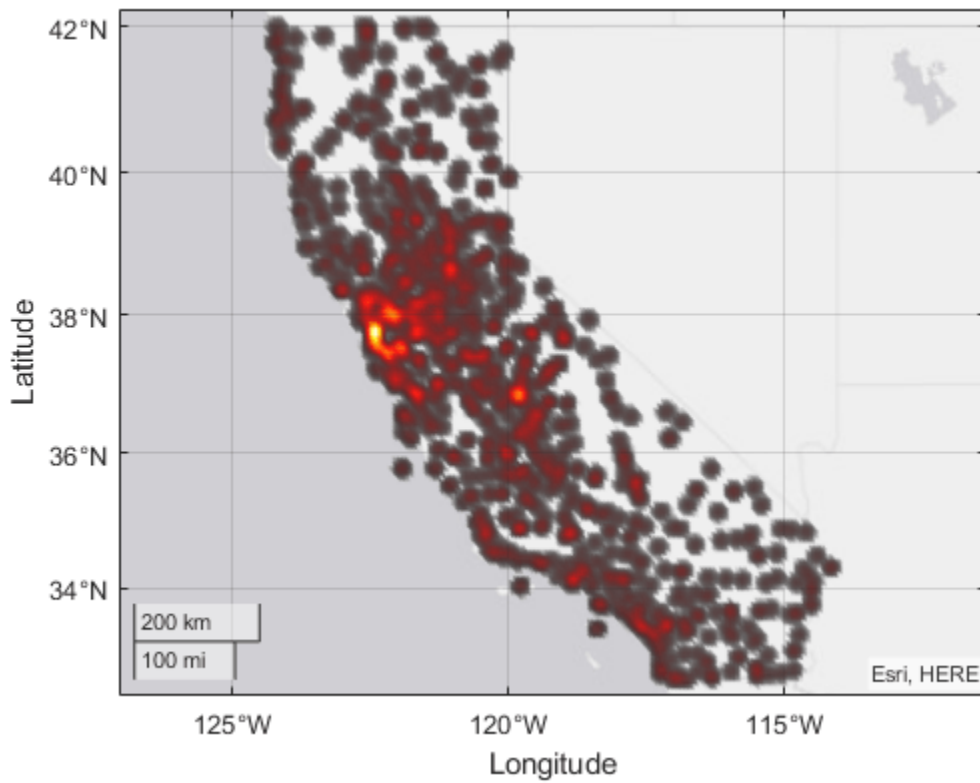
See Also

DensityPlot Properties | `geodensityplot`

Related Examples

- "Access Basemaps for Geographic Axes and Charts" on page 6-35

Customize Layout of Geographic Axes

Customize the layout of a geographic axes by modifying its properties.

Plot a straight line between two points on a map. Specify the endpoints of the line using the coordinates of Seattle and Anchorage. Specify latitude and longitude in degrees.

```
latSeattle = 47.62;
lonSeattle = -122.33;
latAnchorage = 61.20;
lonAnchorage = -149.9;
```

Plot the data using `geoplot`. Customize the appearance of the line using the line specification `'b:'`. Adjust the latitude and longitude limits of the map using `geolimits`.

```
geoplot([latSeattle latAnchorage],[lonSeattle lonAnchorage],'b:')
geolimits([45 62],[-149 -123])
```
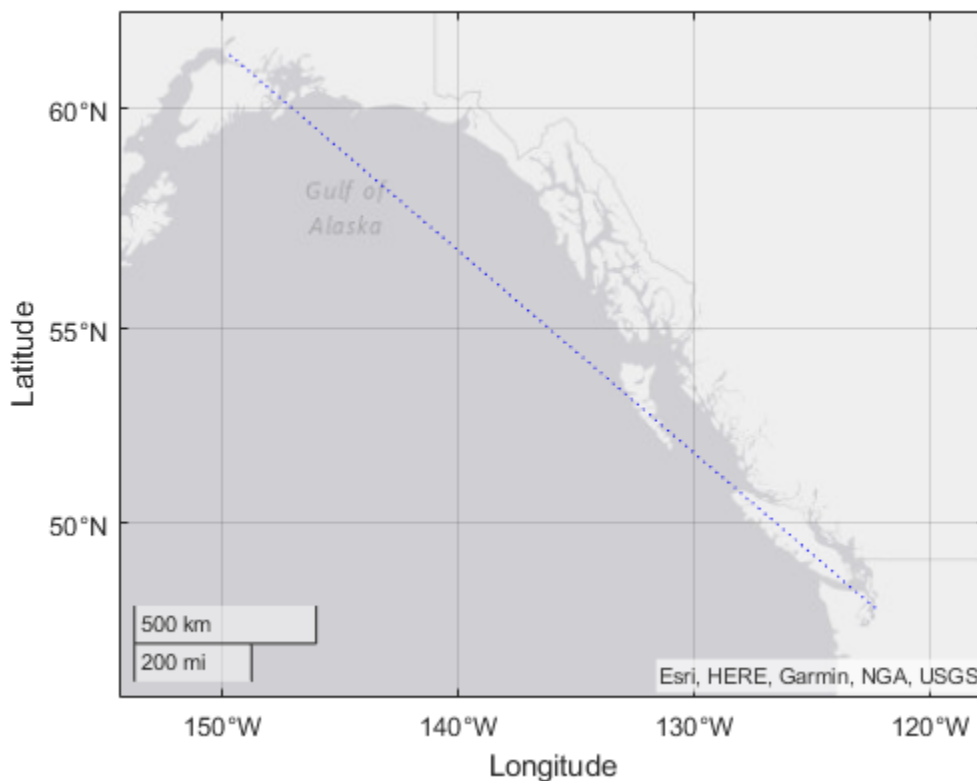

Customize the layout of the axes. Turn off the grid, stretch the grid to take up the entire figure, and turn off tick marks by modifying the `Grid`, `Position`, and `TickDir` properties.

```
gx = gca;
gx.Grid = 'off';
gx.TickDir = 'out';
gx.Position = gx.OuterPosition;
```
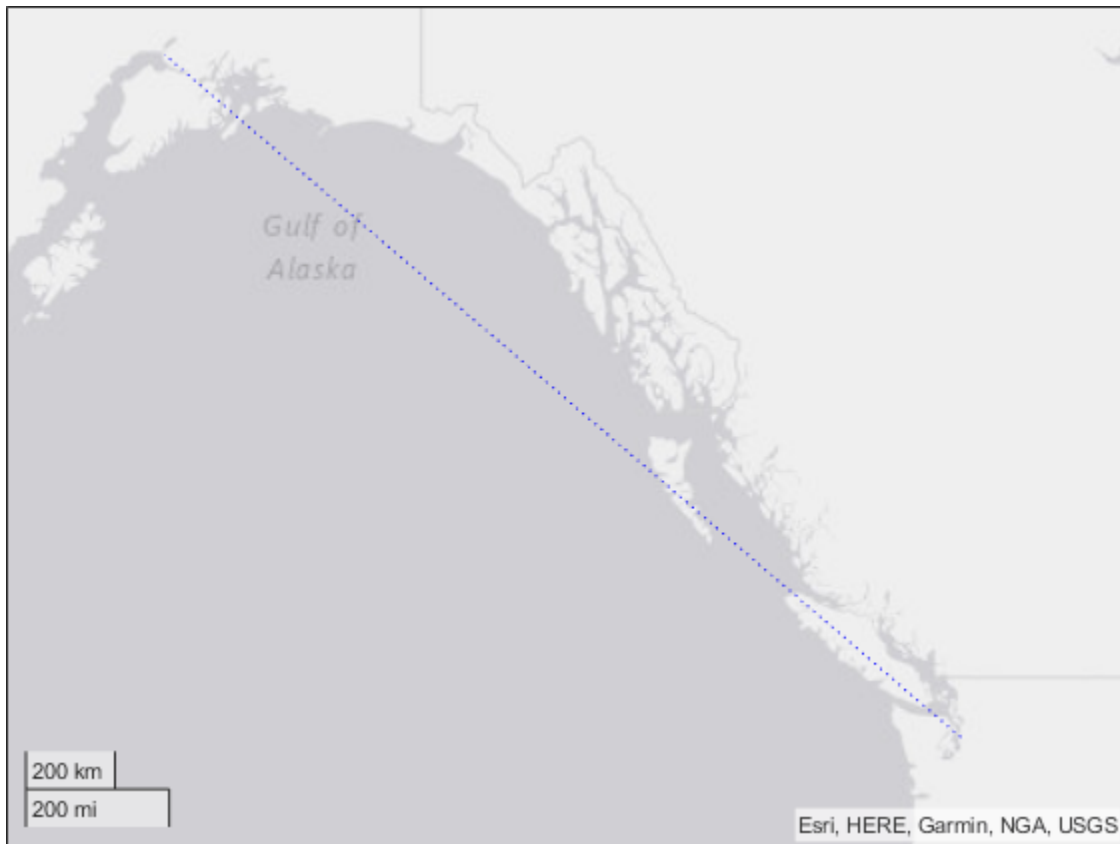
See Also
geoplot

Related Examples
*   "Access Basemaps for Geographic Axes and Charts" on page 6-35

Deploy Geographic Axes and Charts

You can use MATLAB Compiler™ to deploy MATLAB applications that use geographic axes or charts. Depending on which basemaps you use, map interactions in your deployed application are the same as in a MATLAB session; that is, the map is "live" and can be panned and zoomed.

App Usage	Behavior
The deployed application uses only the `'darkwater'` basemap.	The `'darkwater'` basemap is included with MATLAB and does not require Internet access.
The deployed application offers a choice of basemaps.	The application accesses basemaps over the Internet.
The deployed application offers a choice of basemaps and does not require an Internet connection.	You must download the MATLAB Basemap Data add-ons and include them in the deployed application package. For more information about downloading basemaps, see "Access Basemaps for Geographic Axes and Charts" on page 6-35.

Note By default, the deployment tool pre-selects all of your downloaded MATLAB Basemap Data add-ons for inclusion in the deployed application package. Do not leave them all selected. Choose only the basemaps that you want users of your application to see. Including all the MATLAB Basemap Data add-ons in your deployed application package can create a file that exceeds file system limits.

See Also

geoaxes | geobubble | geodensityplot | geoplot | geoscatter

Related Examples

- "Access Basemaps for Geographic Axes and Charts" on page 6-35

Use Geographic Bubble Chart Properties

In this section...
"Control Bubble Size" on page 6-27
"Control Bubble Color" on page 6-29

This topic describes some common tasks you can perform using geographic bubble charts properties.

Control Bubble Size

You can use the size of the bubbles in a geographic bubble chart to communicate a quantifiable aspect of your data. For example, for Lyme disease sample data, you can use bubble size to visualize the number of cases in each county in New England. The following properties of the geographic bubble chart work together to control the size of the bubbles on the chart:

- `SizeData`
- `SizeVariable`
- `SizeLimits`
- `BubbleWidthRange`

The `SizeData` property specifies the data that you want to plot on the chart. `SizeData` must be a vector of numeric data the same size as the latitude and longitude vectors, or a scalar. Another way to specify size data is to pass a table as the first argument to `geobubble` and specify the name of a table variable to use for size data. You use the `SizeVariable` property to specify this table variable. When you use a table variable to specify size data, `geobubble` stores the values of this variable in the `SizeData` property and sets the property to read-only. If you do not specify `SizeData`, `geobubble` plots the geographic locations on a map using bubbles that are all the same size.

`geobubble` determines the size (diameter) of each bubble by linearly scaling the `SizeData` values between the limits set by the `BubbleWidthRange` property. `BubbleWidthRange` is a two-element vector that specifies the smallest bubble diameter and the largest bubble diameter in points. By default, `BubbleWidthRange` sets the range of bubble diameters between 5 points and 20 points. You can specify a bubble diameter as small as 1 point and as large as 100 points.

Use the `SizeLimits` property to control the mapping between `SizeData` and `BubbleWidthRange`. By default, the `SizeLimits` property specifies the extremes of your data range. For example, the `SizeLimits` default for the Lyme disease sample data is: `[0 514]` when the Cases2010 variable is used as the `SizeVariable`.

When you specify size data, the geographic bubble chart includes a legend that describes the mapping of bubble sizes to your data. `geobubble` uses the values in the `SizeLimits` property as upper and lower bounds of the legend. When you specify a table variable, `geobubble` uses the variable name as the title of the size legend.

Make Bubbles Smaller in Geographic Bubble Charts

This example shows how to reduce the size of the bubbles in a geographic bubble chart using the `BubbleWidthRange` property. (You can also reduce overlapping by resizing the geographic bubble chart figure.)

Read Lyme disease sample data into the workspace.

```
counties = readtable('counties.xlsx');
```

Create a geographic bubble chart using the latitude, longitude, and occurrence data from the table. Adjust the limits of the chart using the `geolimits` function.

```
gb = geobubble(counties,'Latitude','Longitude','SizeVariable','Cases2010');
geolimits(gb,[41 47],[-75 -66])
```
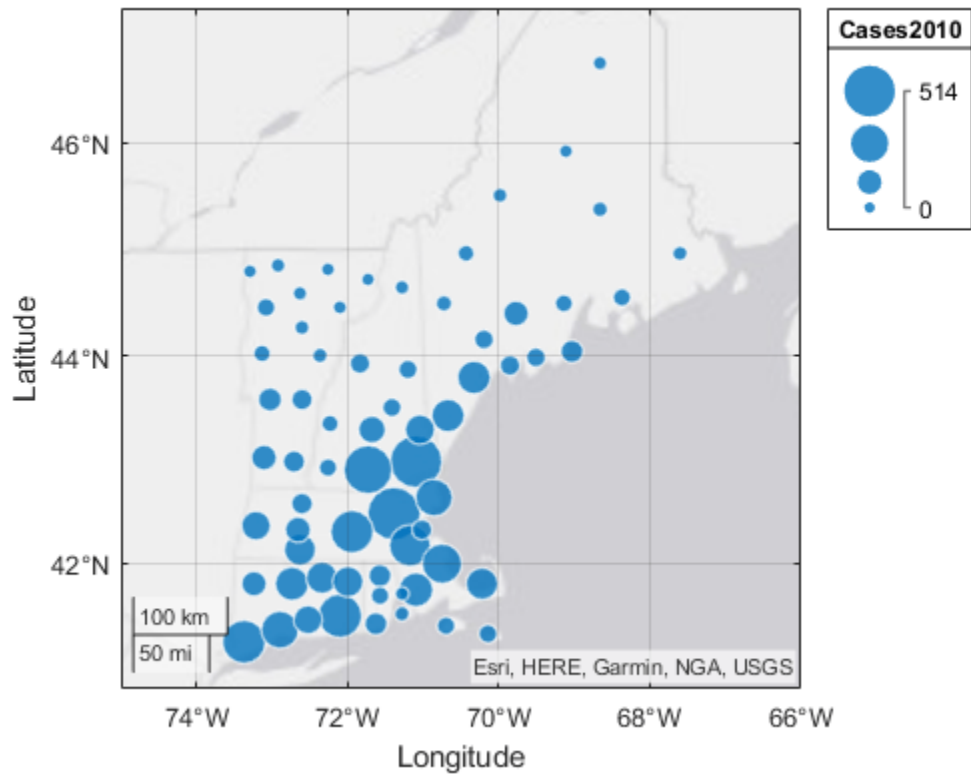

View the values of the `SizeData` and `SizeLimits` properties of the geographic bubble chart.

```
size_data_values = gb.SizeData;
size_data_values(1:15)
```

ans = *15×1*

```
   331
   187
    88
   125
   240
   340
   161
   148
    38
     4
     ⋮
```

```
size_limits = gb.SizeLimits
```

```
size_limits = 1×2

     0    514
```

Make the bubbles smaller to avoid overlapping using the `BubbleWidthRange` property. First view the initial setting of the property.

```
default_width_range = gb.BubbleWidthRange

default_width_range = 1×2

     5    20
```

```
gb.BubbleWidthRange = [4 15];
```
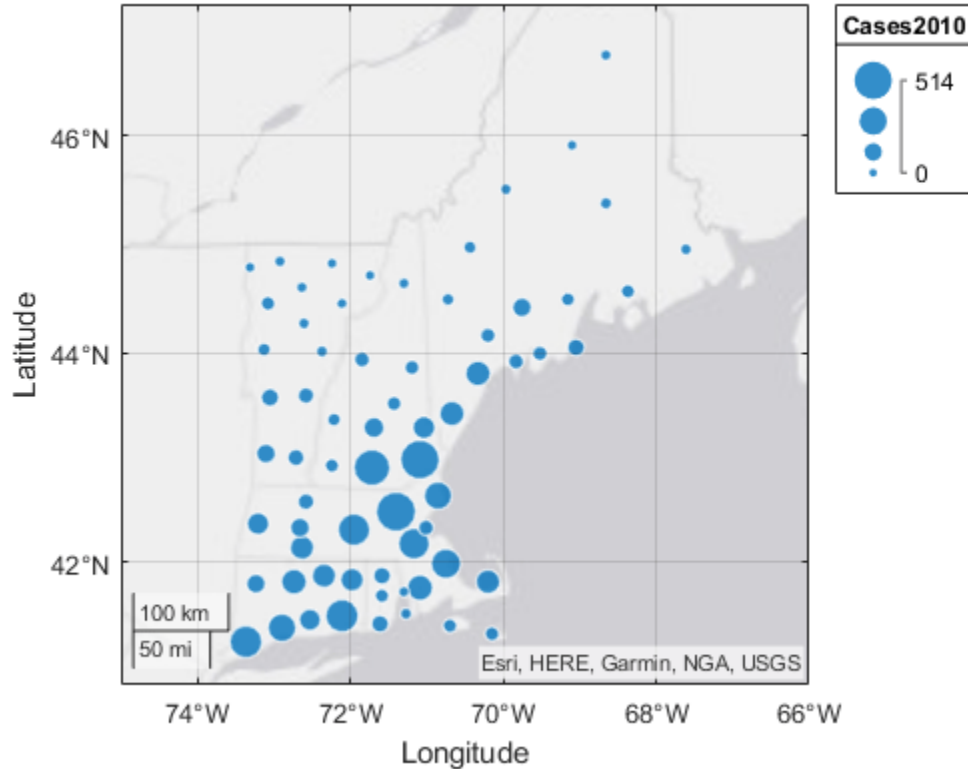

Control Bubble Color

You can use the color of the bubbles in a geographic bubble chart to code them according to data category. For example, in the Lyme disease sample data, you can characterize the severity of Lyme disease in each county in New England as high, medium, or low. The following properties of the geographic bubble chart work together to control the color of the bubbles on the chart:

- `ColorData`

- `ColorVariable`
- `BubbleColorList`

The `ColorData` property specifies the data that you want to control the color of the bubbles in your chart. `ColorData` must be a vector of categorical data, the same size as latitude and longitude. Another way to specify color data is to pass a table as the first argument to `geobubble` and specify the name of a table variable to use for color data. You use the `ColorVariable` property to specify this table variable. `geobubble` stores the values of the table variable in the `ColorData` property and sets the property to read-only.

If your data does not initially include a `categorical` variable, you can create one. For example, the Lyme disease sample data does not include a categorical variable. One way to create a variable of this type is to use the `discretize` function. Take the occurrences data, `cases2010`, and create three categories based on the number of occurrences, describing them as low, medium, or high. The following code creates a categorical variable named `Severity` from the occurrence data.

```
Severity = discretize(counties.Cases2010,[0 50 100 550],...
'categorical', {'Low', 'Medium', 'High'});
```

The `BubbleColorList` property controls the colors used for the bubbles in a geographic bubble chart. The value is an *m*-by-3 array where each row is an RGB color triplet. By default, `geobubble` uses a set of seven colors. If you have more than seven categories, the colors repeat cyclically. To change the colors used, use one of the other MATLAB colormap functions, such as `parula` or `jet`, or specify a custom list of colors.

See Also

GeographicBubbleChart Properties | `discretize` | `geobubble` | `geolimits`

Related Examples

- "Deploy Geographic Axes and Charts" on page 6-26
- "Access Basemaps for Geographic Axes and Charts" on page 6-35
- "Geographic Bubble Charts Overview" on page 6-8
- "Create Geographic Bubble Chart from Tabular Data" on page 6-41

Specify Map Limits with Geographic Axes

A geographic axes or chart sets the latitude and longitude limits of the basemap to encompass all the points in your data. These map limits do not change when you resize the chart by resizing the figure window except to adapt to changes in the axes or chart aspect. The map limits do change when you zoom in or out or pan. The geographic axes and charts support properties related to map limits. Some are read-only properties that are for informational use.

- `LatitudeLimits` - Returns the current latitude limits (read-only).
- `LongitudeLimits` - Returns the current longitude limits (read-only).
- `MapCenter` - Return or set the current map center point.
- `ZoomLevel` - Return or set the current map zoom level.

A convenient way to get the current latitude and longitude limits is to call the `geolimits` function. You can also use the `geolimits` function to set the latitude and longitude limits. Use the `geolimits` function when you want to create a geographic axes or chart with the same map limits as an existing axes or chart. Retrieve the limits of the existing axes or chart and use `geolimits` to set the limits of the new axes or chart.

Note You can specify latitudes outside the approximate limits [-85 85], beyond which the basemap tiles do not extend. However these values typically are not visible unless you control the map extent using the `MapCenter` and `ZoomLevel` properties. Also, data points very close to 90 degrees and -90 degrees can never be seen, because they map to infinite or near-infinite values in the vertical direction.

Display Several Geographic Bubble Charts Centered Within Specified Limits

This example shows how to create two geographic bubble charts with the same map limits.

Read Lyme Disease sample data into the workspace.

```
counties = readtable('counties.xlsx');
```

Create a geographic bubble chart that plots the occurrences of Lyme disease in New England counties.

```
gb = geobubble(counties,'Latitude','Longitude','SizeVariable','Cases2010');
```
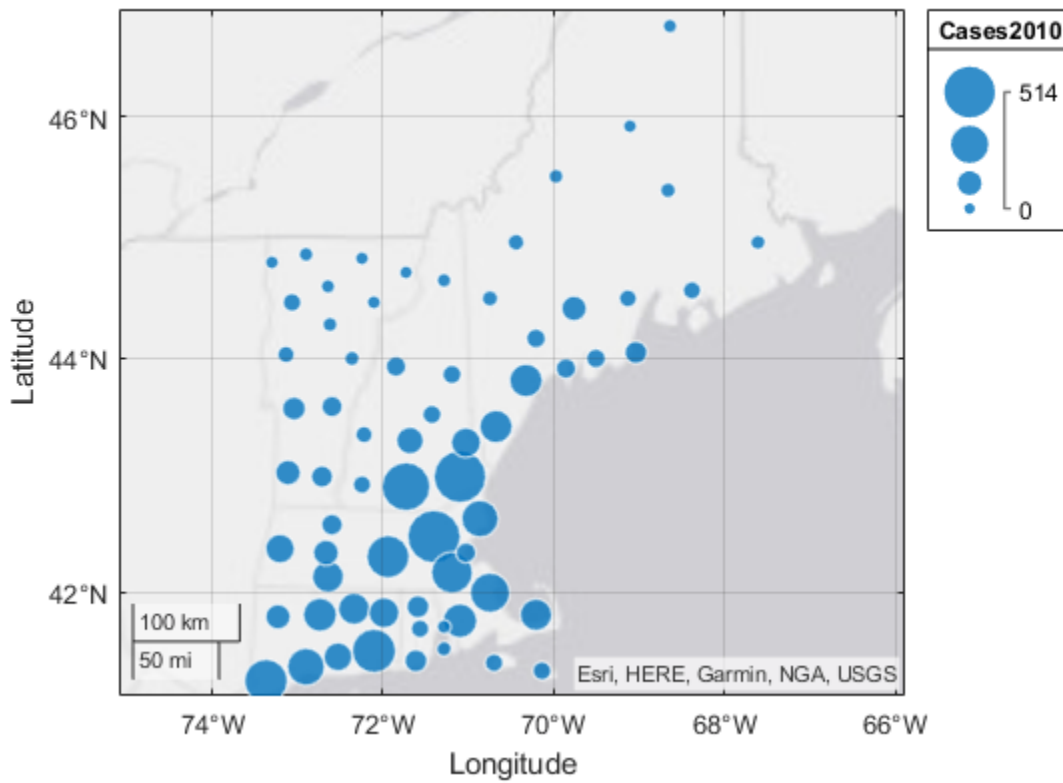
Pan and zoom the map until you see only the states in northern New England: Vermont, New Hampshire, and Maine.
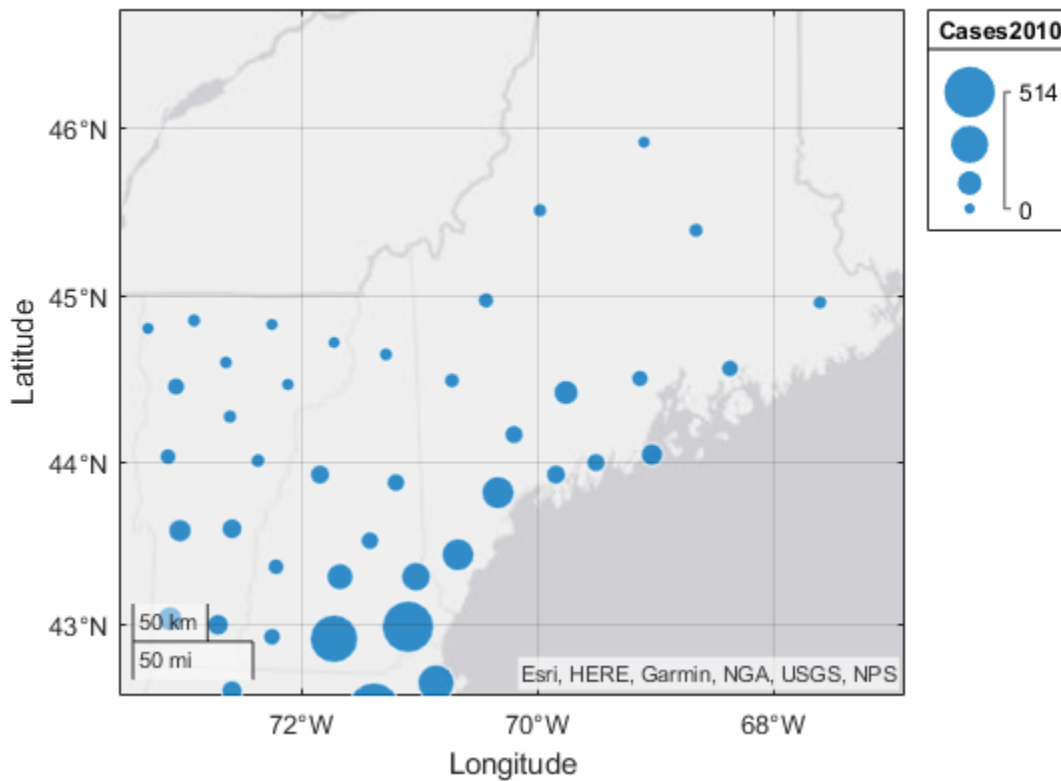
Get the new limits of the map using the command `[nlat nlon] = geolimits(gb)`. Get the new zoom level as well using the command `nzoom = gb.ZoomLevel`. Store the latitude, longitude, and zoom level of the new map limits.

```
nlat = [42.5577    46.6921];
nlon = [-73.5500   -66.8900];
nzoom = 6.3747;
```

Create another map with Lyme disease occurrence data for 2011 and set the map limits and zoom level to match the first chart.

```
figure
gb2 = geobubble(counties,'Latitude','Longitude','SizeVariable','Cases2011');
[n2lat n2lon] = geolimits(gb2,nlat,nlon);
gb2.ZoomLevel = nzoom;
```
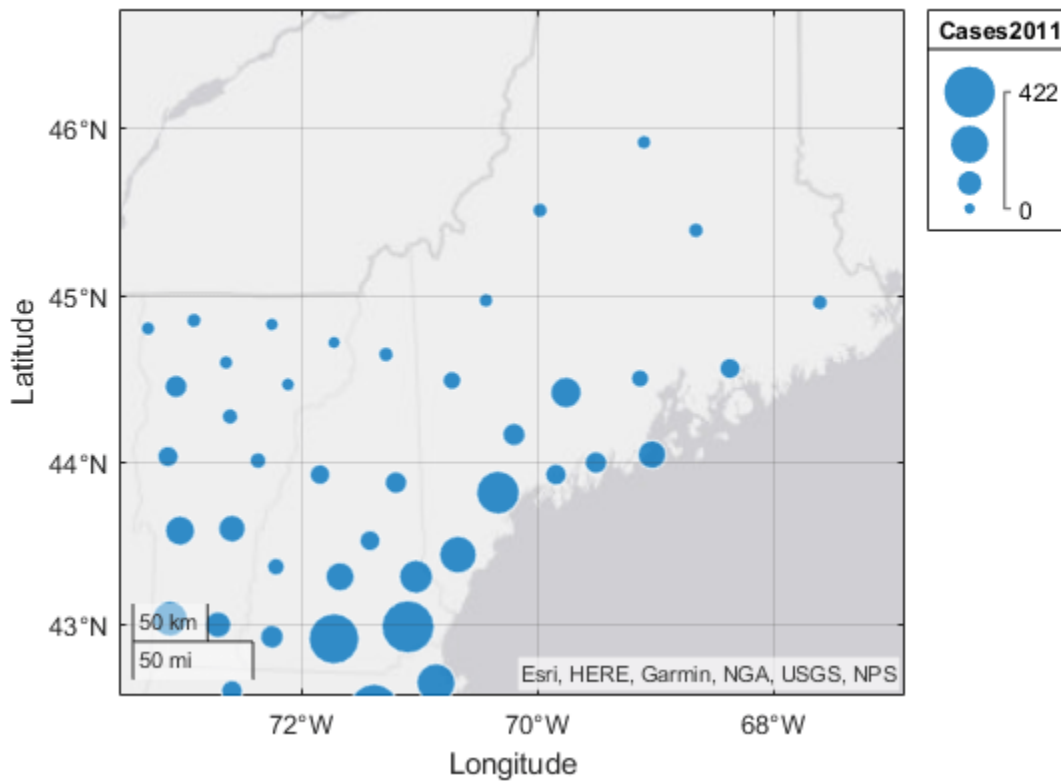
See Also

DensityPlot Properties | GeographicAxes Properties | GeographicBubbleChart Properties | geoaxes | geobubble | geodensityplot | geolimits | geoplot | geoscatter

Related Examples

- "Geographic Bubble Charts Overview" on page 6-8
- "Create Geographic Bubble Chart from Tabular Data" on page 6-41

Access Basemaps for Geographic Axes and Charts

MathWorks® offers a selection of basemaps for use with geographic axes and charts. The basemaps provide a variety of map options, including two-tone, color terrain, and high-zoom-level displays. Six of the basemaps are tiled data sets created using Natural Earth. Five of the basemaps are high-zoom-level maps hosted by Esri®. For more information about basemap options, see `geobasemap`.

To specify a basemap for your geographic axes or chart, you can either:

- Use the `geobasemap` function.
- Set the `Basemap` property of the `GeographicAxes` or `GeographicBubbleChart` object.

MATLAB includes one installed basemap, a two-tone map named `'darkwater'`. Use of this basemap does not require Internet access. Use of the other basemaps, including the default basemap `'streets-light'`, does require Internet access.

If you have trouble accessing basemaps over the Internet, check your proxy server settings. For more information about specifying proxy server settings, see "Use MATLAB Web Preferences For Proxy Server Settings".

If you do not have reliable access to the Internet, or want to improve map responsiveness, you can plot using the `'darkwater'` basemap or download a selection of basemaps onto your local system.

Display 'darkwater' on Geographic Plots

To display the `'darkwater'` basemap while plotting with functions such as `geoplot` and `geoscatter`, call `geobasemap`.

```
lat1 = [33.448 29.424 32.716 32.777 37.338];
lon1 = [-112.074 -98.494 -117.161 -96.797 -121.886];
geoscatter(lat1,lon1,'*')
geobasemap darkwater
```
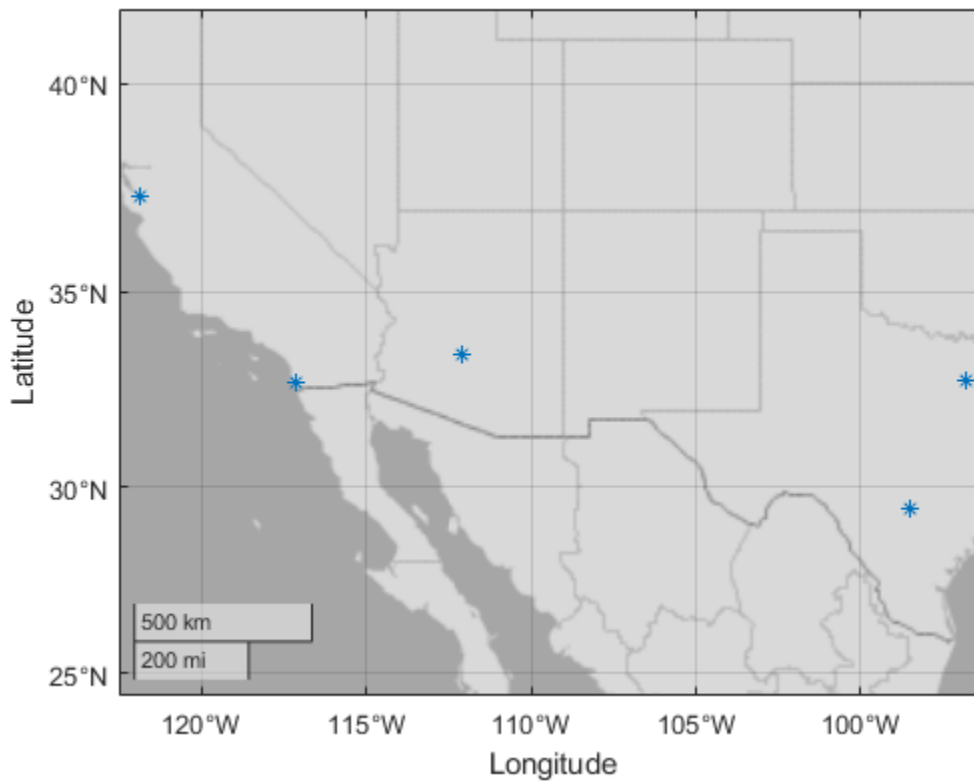
Alternatively, you can create a set of geographic axes and specify the `Basemap` name-value pair. To maintain the basemap, use the `hold on` command before plotting.

```
figure
lat2 = [40.713 34.052 41.878 29.760 39.952];
lon2 = [-74.006 -118.244 -87.630 -95.370 -75.165];
geoaxes('Basemap','darkwater')
hold on
geoscatter(lat2,lon2,'*')
```
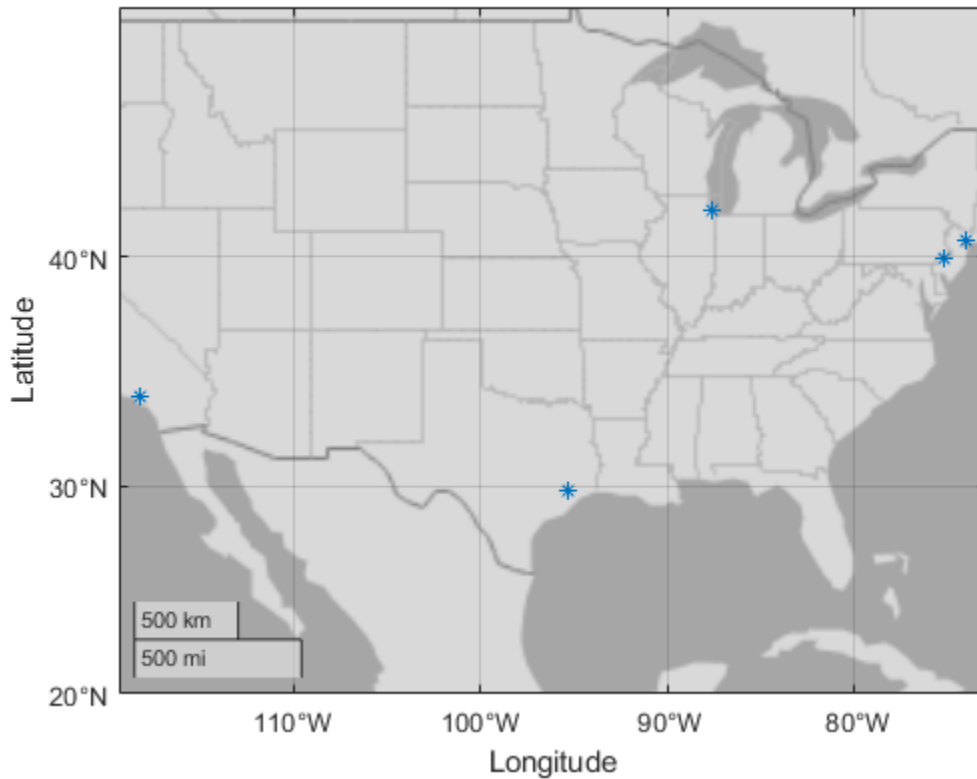
You can also change the default basemap for all plots created with `geoplot`, `geoscatter`, and `geodensityplot` during your MATLAB session.

```
set(groot,'defaultGeoaxesBasemap','darkwater')
```

Display 'darkwater' on Geographic Bubble Charts

To display `'darkwater'` on a geographic bubble chart, call `geobubble` using the `'Basemap'` name-value pair.

```
tsunamis = readtable('tsunamis.xlsx');
geobubble(tsunamis,'Latitude','Longitude','Basemap','darkwater')
```
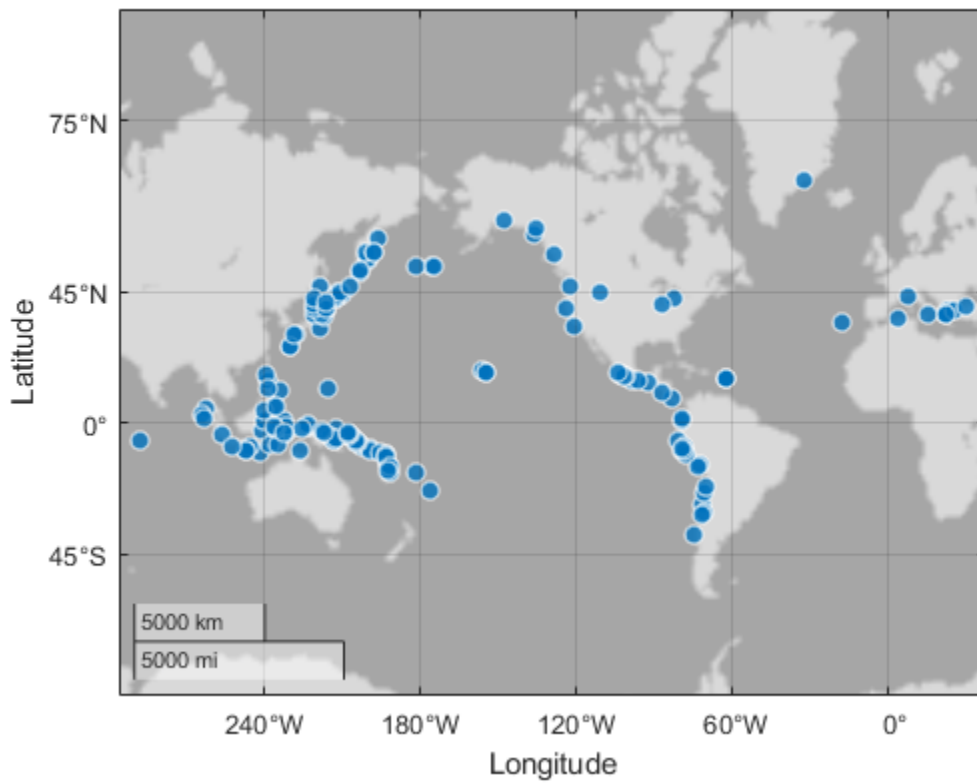
Alternatively, you can specify the basemap by using `geobasemap`.

```
counties = readtable('counties.xlsx');
geobubble(counties,'Latitude','Longitude')
geobasemap darkwater
```
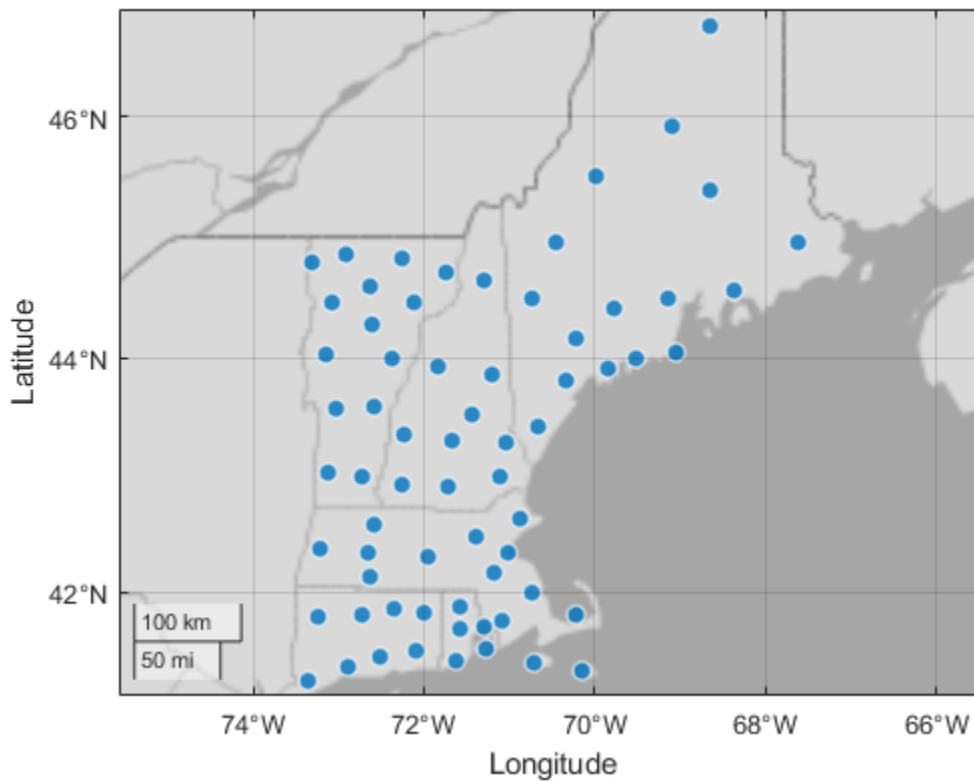
Download Basemaps

Download basemaps onto your local system using the Add-On Explorer. The five high-zoom-level basemaps provided by Esri are not available for download.

1 On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Add-Ons**.

2 In the Add-On Explorer, scroll to the **MathWorks Optional Features** section, and click **show all** to find the basemap packages. You can also search for the basemap add-ons by name (listed in the following table) or click **Optional Features** in **Filter by Type**.

3 Select the basemap data packages that you want to download.

Basemap Name	Basemap Data Package Name
'bluegreen'	MATLAB Basemap Data - bluegreen
'grayland'	MATLAB Basemap Data - grayland
'colorterrain'	MATLAB Basemap Data - colorterrain
'grayterrain'	MATLAB Basemap Data - grayterrain
'landcover'	MATLAB Basemap Data - landcover

Basemap Caching Behavior

When you access a basemap over the Internet, MATLAB improves performance by temporarily caching the basemap tiles. With this caching behavior, the program has to download each tile only once as you pan and zoom within the map. If you lose your connection to the Internet, you can still view parts of the map that you have already viewed, because the map tiles are stored locally.

When you are not connected to the Internet and you attempt to view a part of the map that you have not previously viewed, tiles for these areas are not in your cache. For the basemaps created using Natural Earth, the program replaces missing tiles with tiles from the `'darkwater'` basemap.

For the high-zoom-level basemaps provided by Esri, the program caches a limited number of tiles and the cached tiles expire after a limited time. If you attempt to view a region of a high-zoom-level basemap that is not cached, you see blank map tiles. The geographic chart does not use tiles from `'darkwater'` for these missing tiles.

See Also

Functions
geoaxes | geobasemap | geobubble | geoplot | geoscatter

Properties
GeographicAxes Properties | GeographicBubbleChart Properties

Related Examples

- "Geographic Bubble Charts Overview" on page 6-8

Create Geographic Bubble Chart from Tabular Data

Geographic bubble charts are a way to visualize data overlaid on a map. For data with geographic characteristics, these charts can provide much-needed context. In this example, you import a file into MATLAB® as a table and create a geographic bubble chart from the table variables (columns). Then you work with the data in the table to visualize aspects of the data, such as population size.

Import File as Table

Load the sample file `counties.xlsx`, which contains records of population and Lyme disease occurrences by county in New England. Read the data into a table using `readtable`.

```
counties = readtable('counties.xlsx');
```

Create Basic Geographic Bubble Chart

Create a geographic bubble chart that shows the locations of counties in New England. Specify the table as the first argument, `counties`. The geographic bubble chart stores the table in its `SourceTable` property. The example displays the first five rows of the table. Use the `'Latitude'` and `'Longitude'` columns of the table to specify locations. The chart automatically sets the latitude and longitude limits of the underlying map, called the *basemap*, to include only those areas represented by the data. Assign the `GeographicBubbleChart` object to the variable `gb`. Use `gb` to modify the chart after it is created.

```
figure
gb = geobubble(counties,'Latitude','Longitude');
```
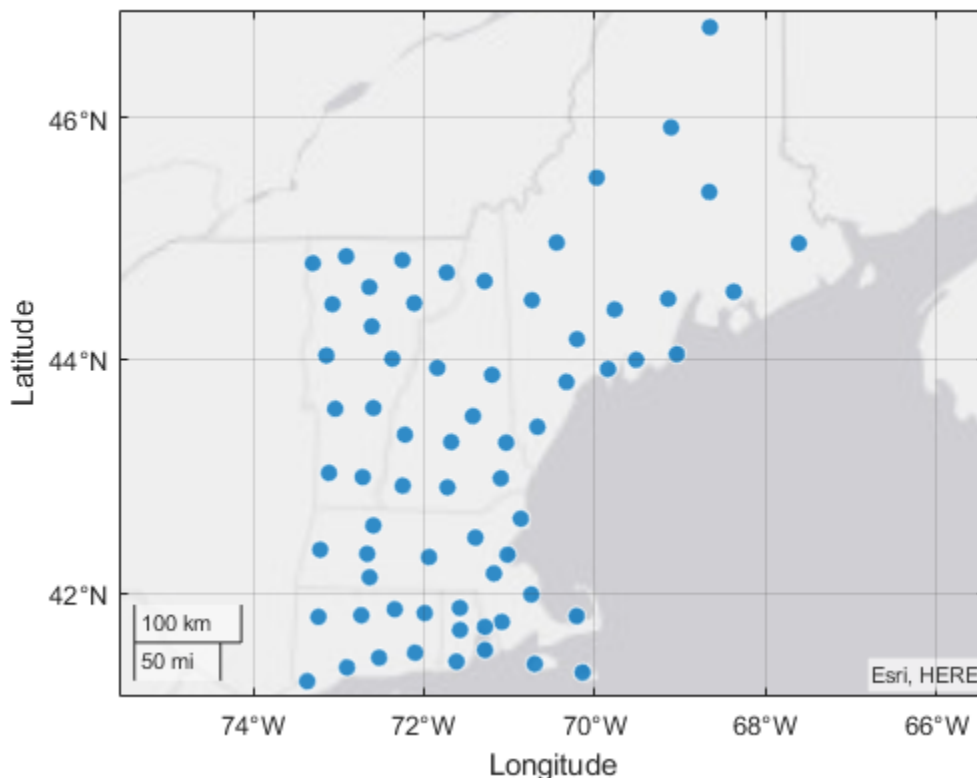
```
head(gb.SourceTable, 5)
```

```
ans=5×19 table
    FIPS      ANSICODE      Latitude    Longitude          CountyName            State         StateNam
    ____     _____     _____    _____     _____     _____     _____

    9001     2.1279e+05      41.228      -73.367     {'Fairfield County' }     {'CT'}     {'Connectic
    9003     2.1234e+05      41.806      -72.733     {'Hartford County'  }     {'CT'}     {'Connectic
    9005      2.128e+05      41.792      -73.235     {'Litchfield County'}     {'CT'}     {'Connectic
    9007      2.128e+05      41.435      -72.524     {'Middlesex County' }     {'CT'}     {'Connectic
    9009      2.128e+05       41.35        -72.9     {'New Haven County' }     {'CT'}     {'Connectic
```

You can pan and zoom in and out on the basemap displayed by the `geobubble` function.

Visualize County Populations on the Chart

Use bubble size (diameter) to indicate the relative populations of the different counties. Specify the `Population2010` variable in the table as the value of the `SizeVariable` parameter. In the resultant geographic bubble chart, the bubbles have different sizes to indicate population. The chart includes a legend that describes how diameter expresses size. Adjust the limits of the chart using `geolimits`.

```
gb = geobubble(counties,'Latitude','Longitude',...
                        'SizeVariable','Population2010');
geolimits([39.50 47.17],[-74.94 -65.40])
```
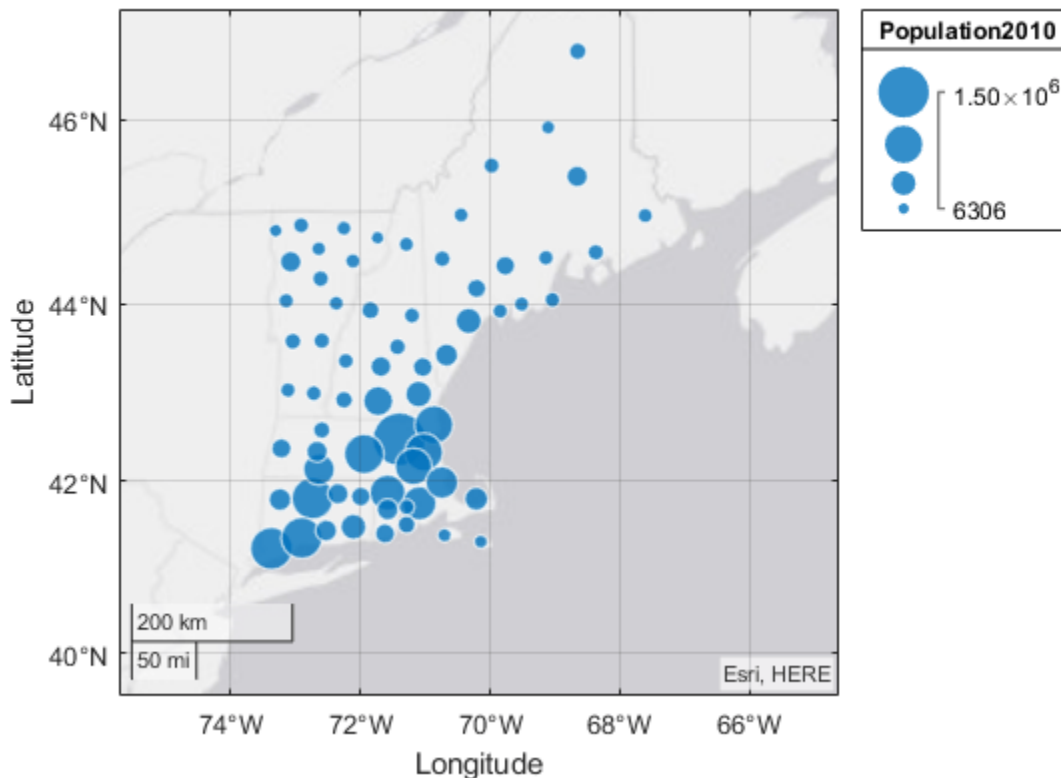

`geobubble` scales the bubble diameters linearly between the values specified by the `SizeLimits` property.

Visualize Lyme Disease Cases by County

Use bubble color to show the number of Lyme disease cases in a county for a given year. To display this type of data, the `geobubble` function requires that the data be a `categorical` value. Initially, none of the columns in the table are categorical but you can create one. For example, you can use the `discretize` function to create a categorical variable from the data in the `Cases2010` variable. The new variable, named `Severity`, groups the data into three categories: Low, Medium, and High. Use this new variable as the `ColorVariable` parameter. These changes modify the table stored in the `SourceTable` property, which is a copy of the original table in the workspace, `counties`. Making changes to the table stored in the `GeographicBubbleChart` object avoids affecting the original data.

```
gb.SourceTable.Severity = discretize(counties.Cases2010,[0 50 100 500],...
                          'categorical', {'Low', 'Medium', 'High'});
gb.ColorVariable = 'Severity';
```
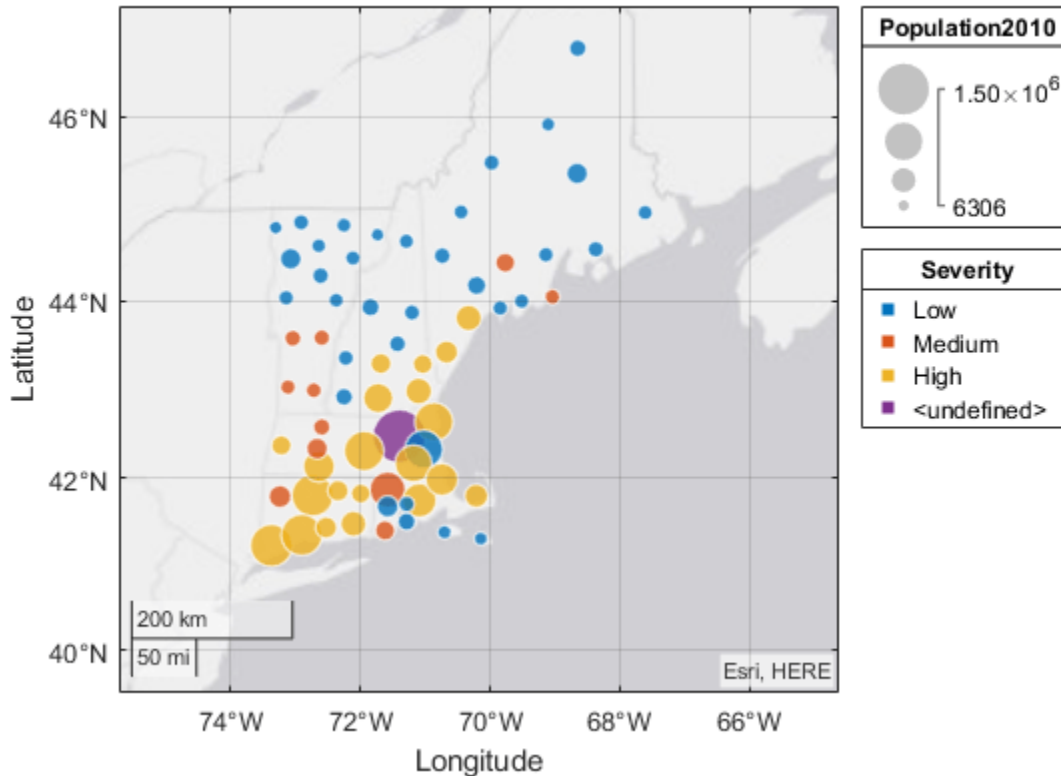

Handle Undefined Data

When you plot the severity information, a fourth category appears in the color legend: `undefined`. This category can appear when the data you cast to `categorical` contains empty values or values that are out of scope for the categories you defined. Determine the cause of the undefined `Severity` value by hovering your cursor over the undefined bubble. The data tip shows that the bubble represents values in the 33rd row of the Lyme disease table.

Check the value of the variable used for Severity, Cases2010, which is the 12th variable in the 33rd row of the Lyme disease table.

```
gb.SourceTable(33,12)
```

```
ans=table
    Cases2010

    _____

       514
```

The `High` category is defined as values between 100 and 500. However, the value of the Cases2010 variable is 514. To eliminate this undefined value, reset the upper limit of the High category to include this value. For example, use 5000.

```
gb.SourceTable.Severity = discretize(counties.Cases2010,[0 50 100 5000],...
                            'categorical', {'Low', 'Medium', 'High'});
```
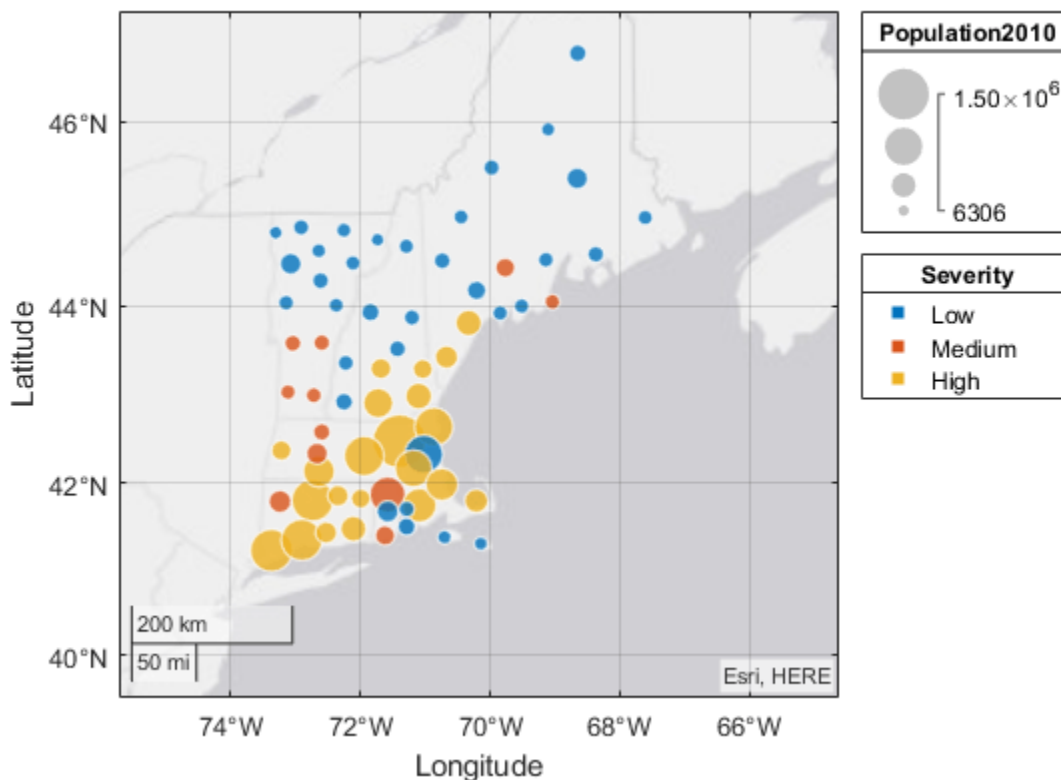

Unlike the color variable, when `geobubble` encounters an undefined number (NaN) in the size, latitude, or longitude variables, it ignores the value.

Choose Bubble Colors

Use a color gradient to represent the Low-Medium-High categorization. `geobubble` stores the colors as an *m*-by-3 list of RGB values in the `BubbleColorList` property.

```
gb.BubbleColorList = autumn(3);
```
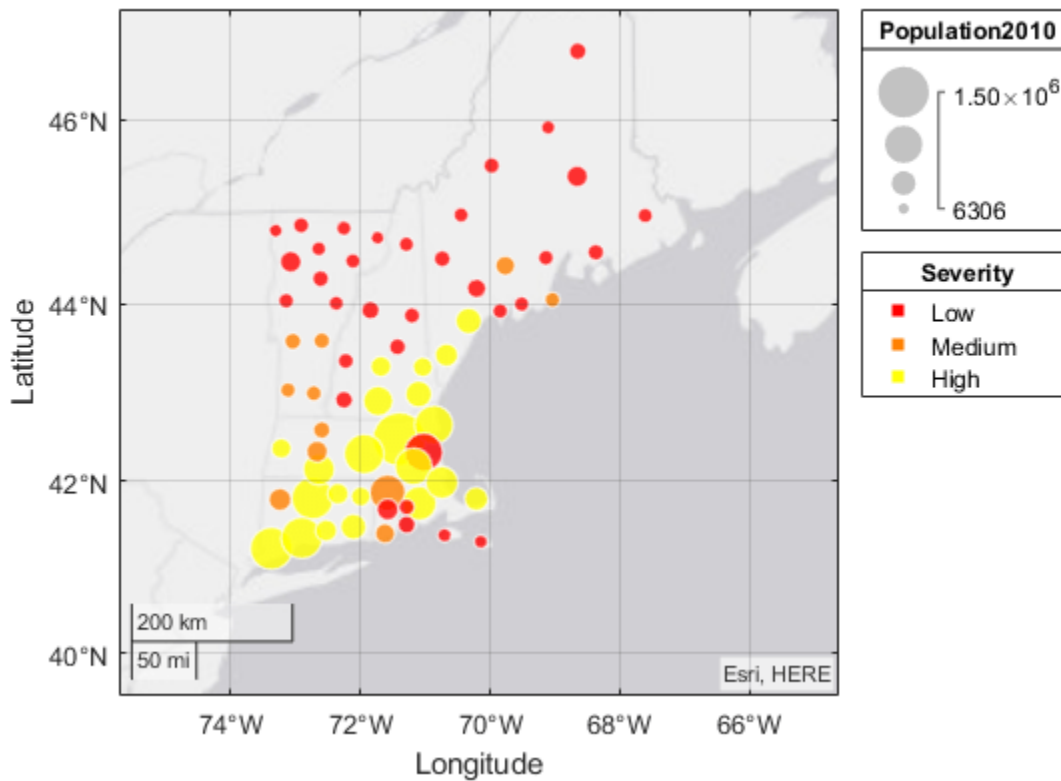
Reorder Bubble Colors

Change the color indicating high severity to be red rather than yellow. To change the color order, you can change the ordering of either the categories or the colors listed in the `BubbleColorList` property. For example, initially the categories are ordered Low-Medium-High. Use the `reordercats` function to change the categories to High-Medium-Low. The categories change in the color legend.

```
neworder = {'High','Medium','Low'};
gb.SourceTable.Severity = reordercats(gb.SourceTable.Severity,neworder);
```
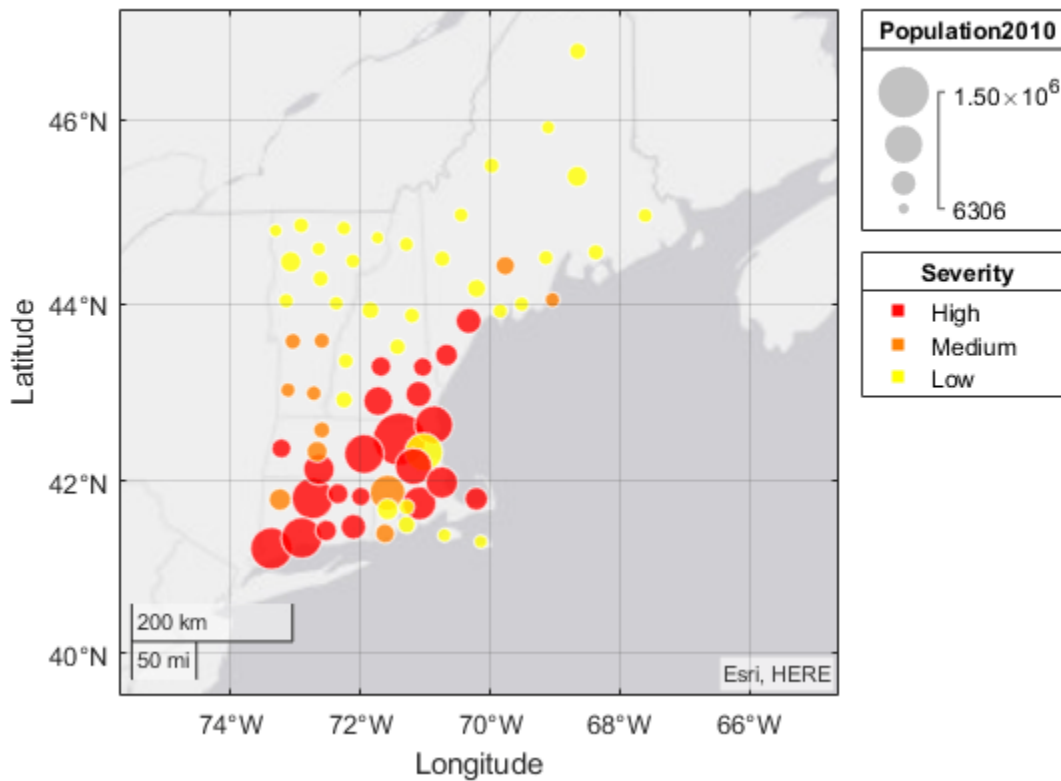
Adding Titles

When you display a geographic bubble chart with size and color variables, the chart displays a size legend and color legend to indicate what the relative sizes and colors mean. When you specify a table as an argument, `geobubble` automatically uses the table variable names as legend titles, but you can specify other titles using properties.

```
title 'Lyme Disease in New England, 2010'
gb.SizeLegendTitle = 'County Population';
gb.ColorLegendTitle = 'Lyme Disease Severity';
```
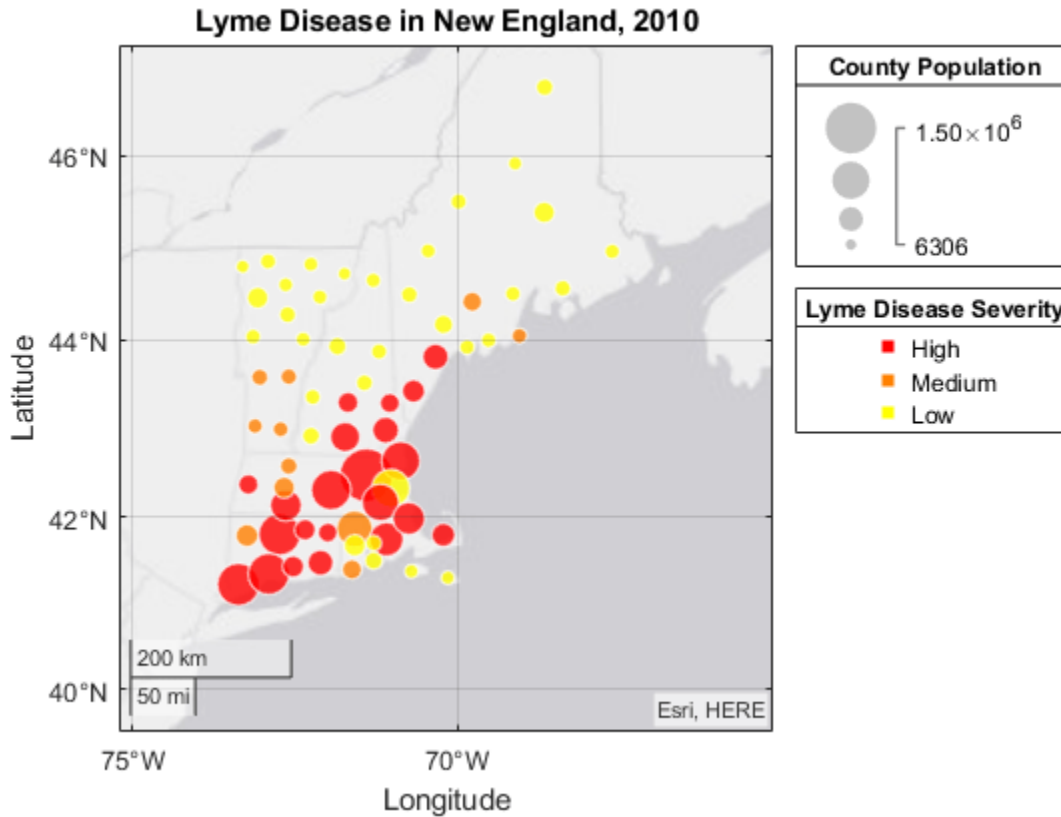
Refine Chart Data

Looking at the Lyme disease data, the trend appears to be that more cases occur in more densely populated areas. Looking at locations with the most cases *per capita* might be more interesting. Calculate the cases per 1000 people and display it on the chart.

```
gb.SourceTable.CasesPer1000 = gb.SourceTable.Cases2010 ./ gb.SourceTable.Population2010 * 1000;
gb.SizeVariable = 'CasesPer1000';
gb.SizeLegendTitle = 'Cases Per 1000';
```
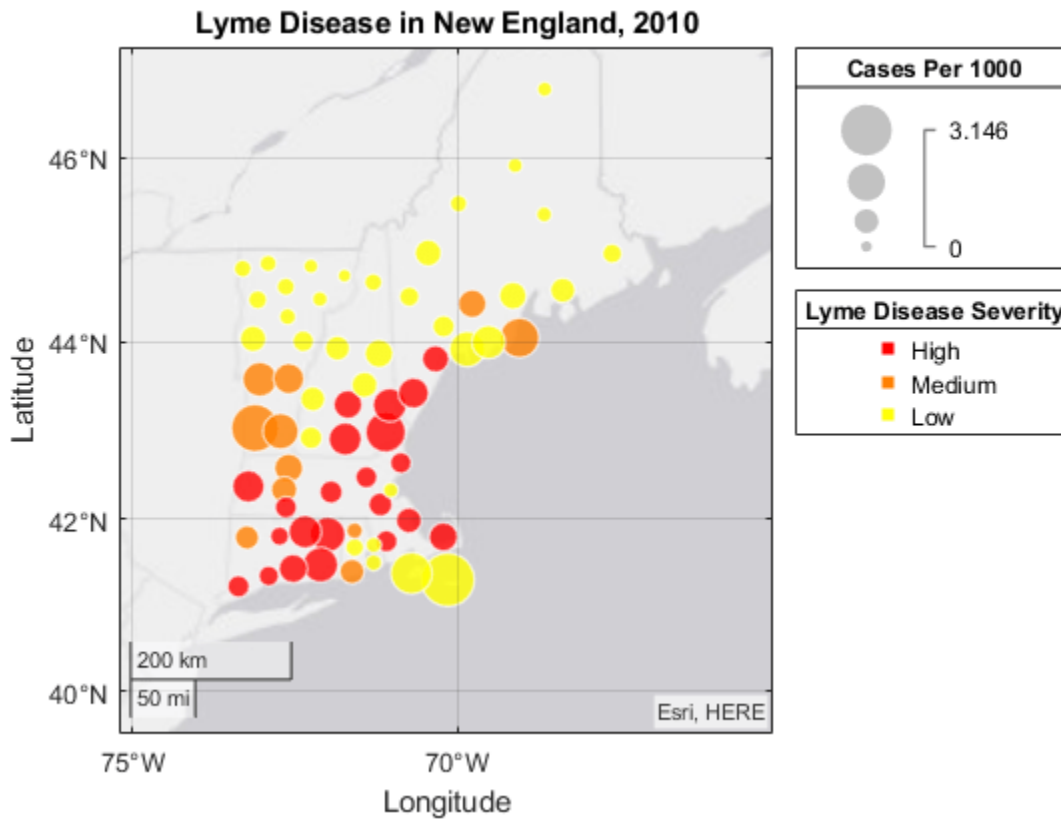
The bubble sizes now tell a different story than before. The areas with the largest populations tracked relatively well with the different severity levels. However, when looking at the number of cases normalized by population, it appears that the highest risk per capita has a different geographic distribution.

See Also

GeographicBubbleChart Properties | `categorical` | `discretize` | `geobubble` | `readtable` | `reordercats` | `table`

Related Examples

- "Use Geographic Bubble Chart Properties" on page 6-27
- "Deploy Geographic Axes and Charts" on page 6-26
- "Access Basemaps for Geographic Axes and Charts" on page 6-35
- "Geographic Bubble Charts Overview" on page 6-8

Animation

Animation Techniques

In this section...
"Updating the Screen" on page 7-2
"Optimizing Performance" on page 7-2

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".

See Also

Related Examples

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
```
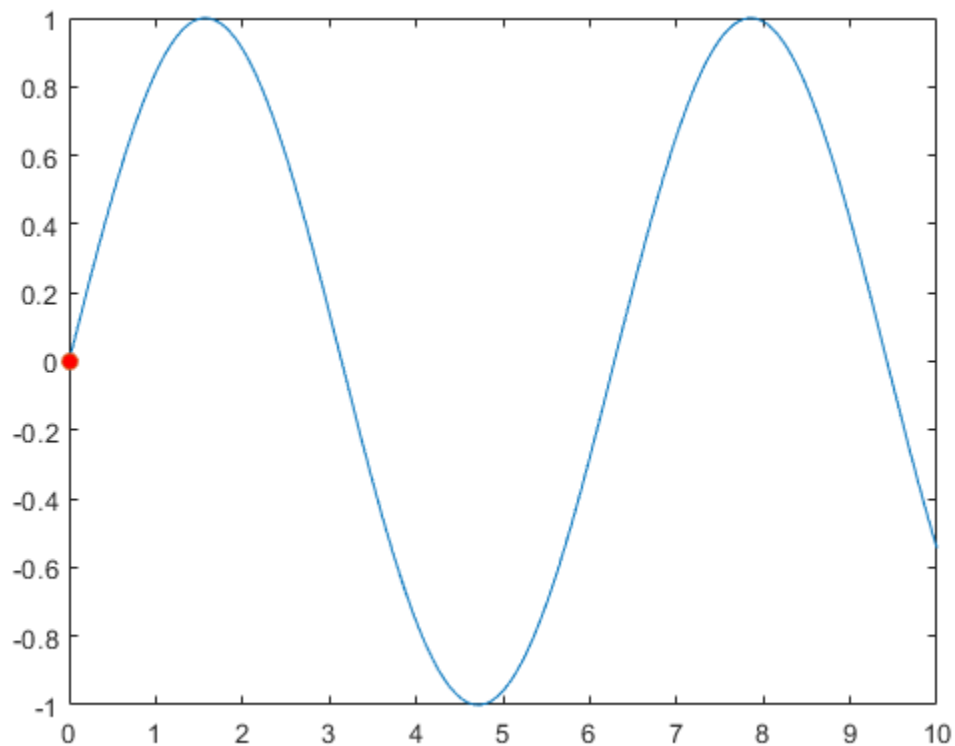

Move the marker along the line by updating the `XData` and `YData` properties in a loop. Use a `drawnow` or `drawnow limitrate` command to display the updates on the screen. `drawnow limitrate` is fastest, but it might not draw every frame on the screen. Use dot notation to set properties.

```
for k = 2:length(x)
    p.XData = x(k);
    p.YData = y(k);
    drawnow
end
```
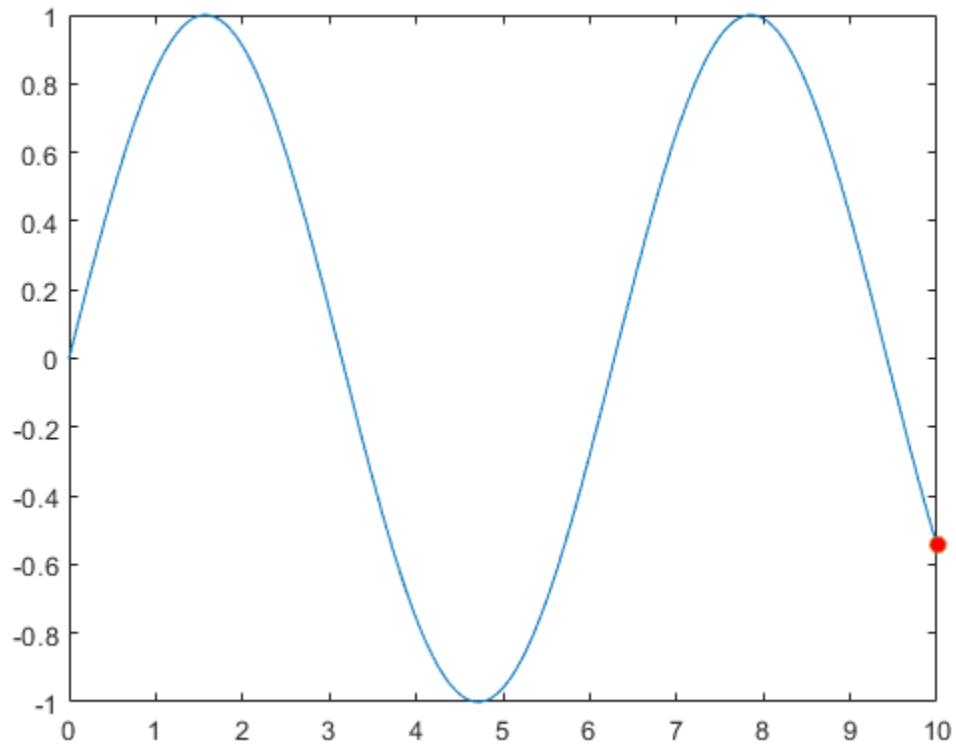
The animation shows the marker moving along the line.

See Also

`drawnow` | `linspace` | `plot`

Related Examples

- "Move Group of Objects Along Line" on page 7-5
- "Animate Graphics Object" on page 7-8
- "Record Animation for Playback" on page 7-12
- "Line Animations" on page 7-10

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
```
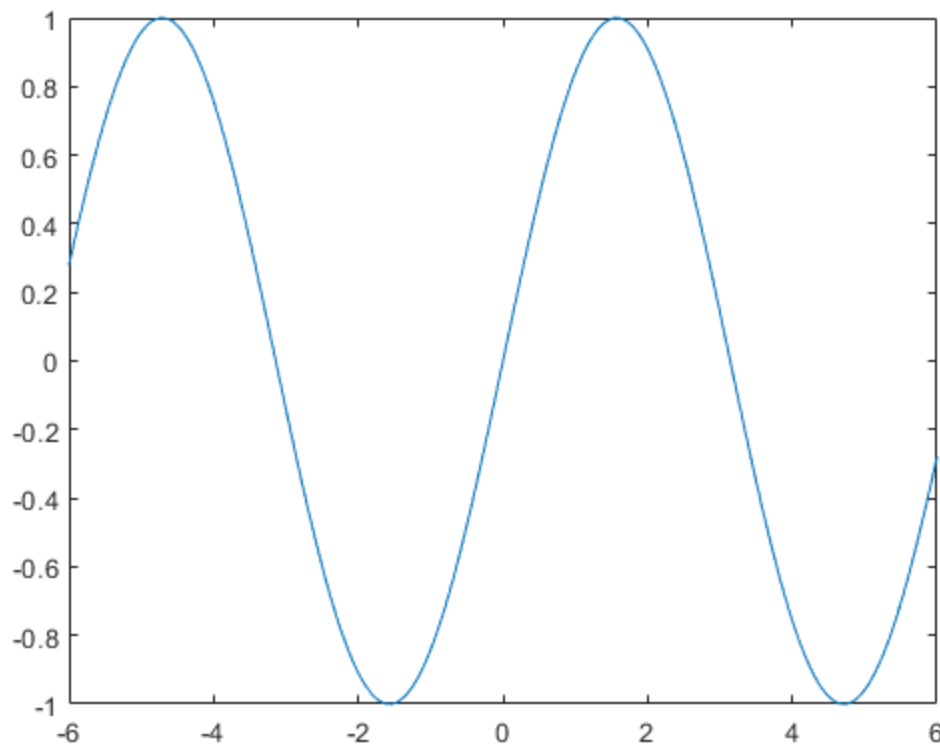

Create a transform object and set its parent to the current axes. Plot a marker and a text annotation at the beginning of the line. Use the `num2str` function to convert the *y*-value at that point to text. Group the two objects by setting their parents to the transform object.

```
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);
```
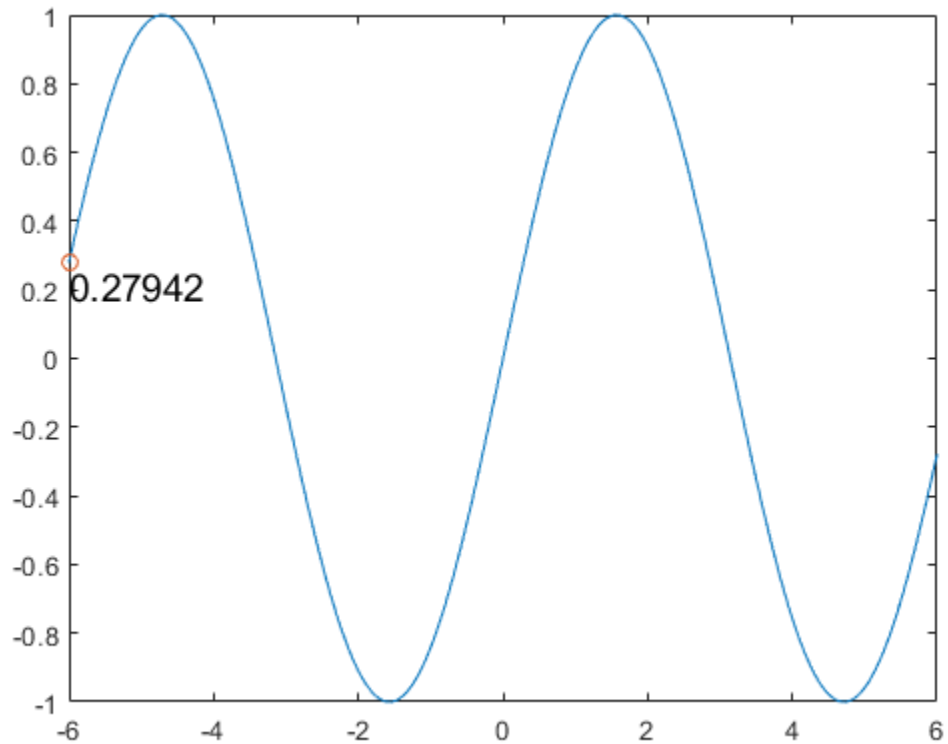
Move the marker and text to each subsequent point along the line by updating the `Matrix` property of the transform object. Use the x and y values of the next point in the line and the first point in the line to determine the transform matrix. Update the text to match the *y*-value as it moves along 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
```
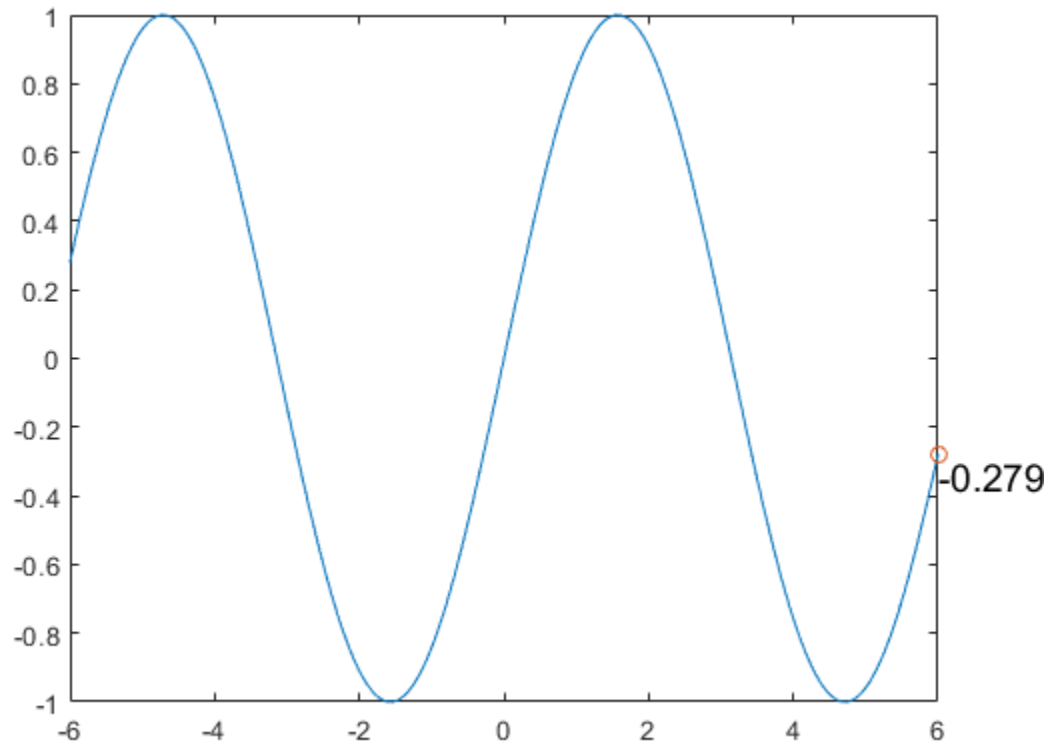
The animation shows the marker and text moving together along the line.

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 7-8
*   "Record Animation for Playback" on page 7-12
*   "Line Animations" on page 7-10

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
```
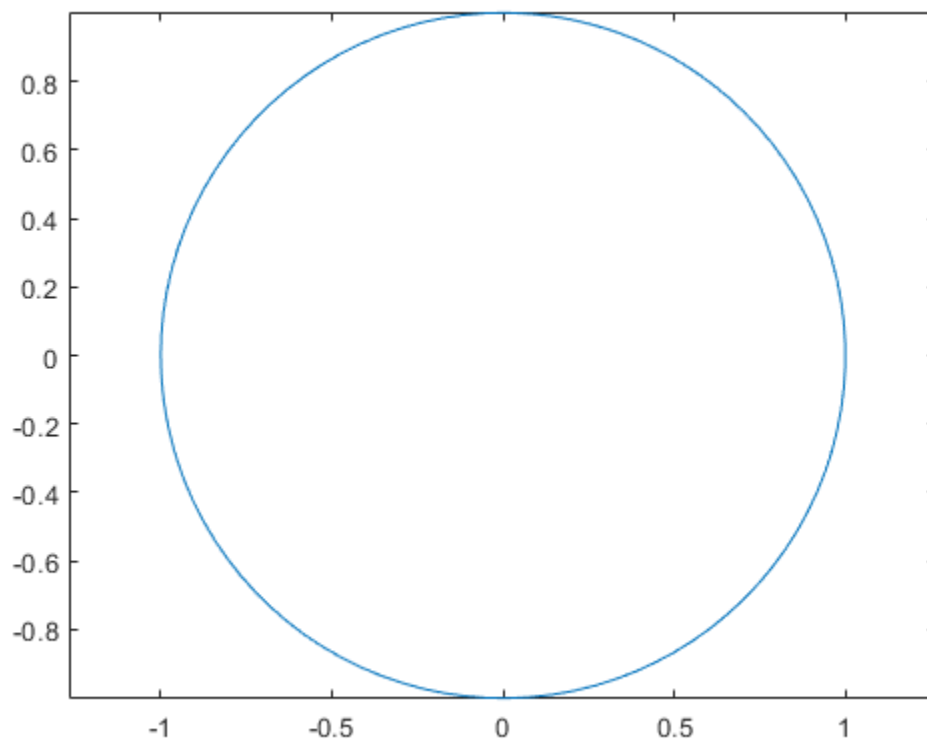

Use the `area` function to draw a flat triangle. Then, change the value of one of the triangle vertices using the (x,y) coordinates of the circle. Change the value in a loop to create an animation. Use a `drawnow` or `drawnow limitrate` command to display the updates after each iteration. `drawnow limitrate` 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
```
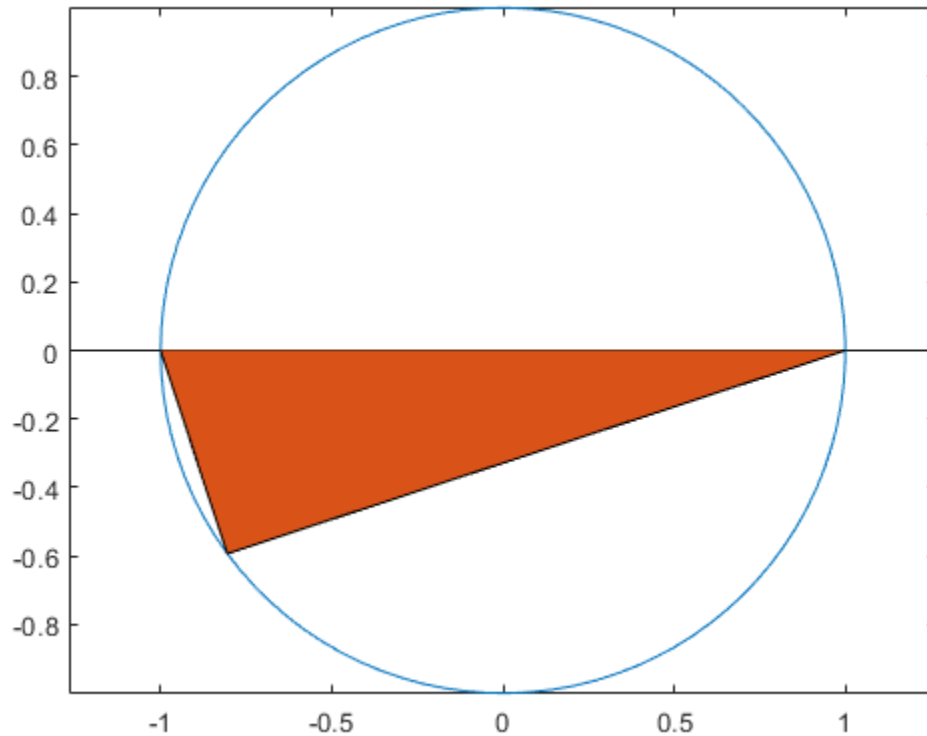
The animation shows the triangle looping around the inside of the circle.

See Also
area | axis | drawnow | hold | plot

Related Examples
- "Trace Marker Along Line" on page 7-3
- "Line Animations" on page 7-10
- "Record Animation for Playback" on page 7-12

More About
- "Animation Techniques" on page 7-2

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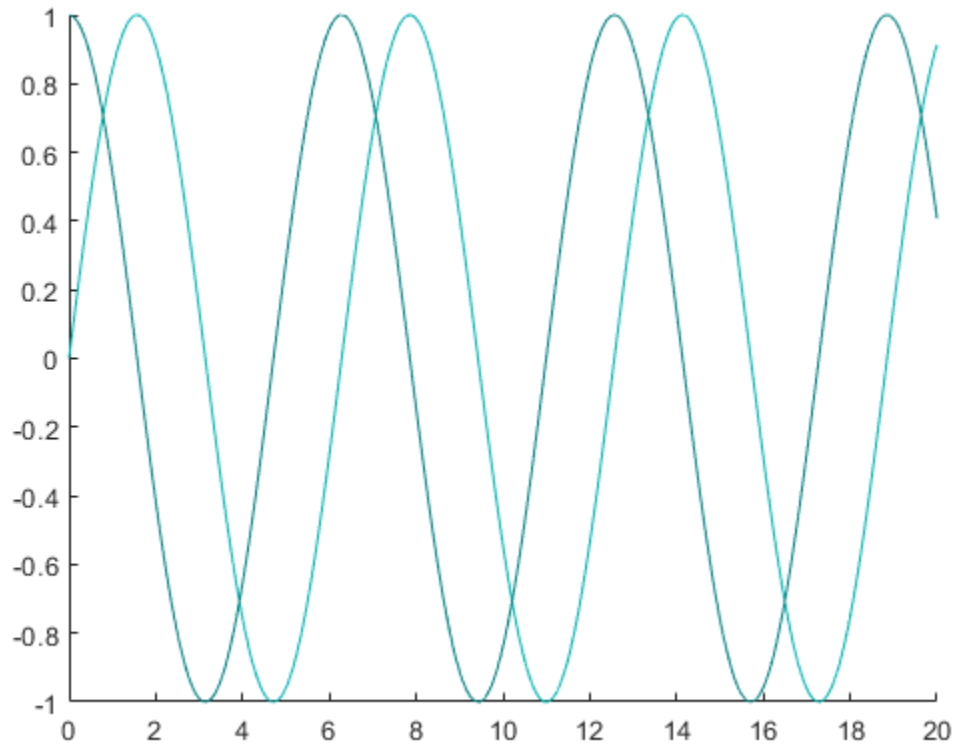
The animation shows two lines that grow as they accumulate data.

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 7-3
- "Move Group of Objects Along Line" on page 7-5
- "Record Animation for Playback" on page 7-12

More About
- "Animation Techniques" on page 7-2

Record Animation for Playback

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

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
```
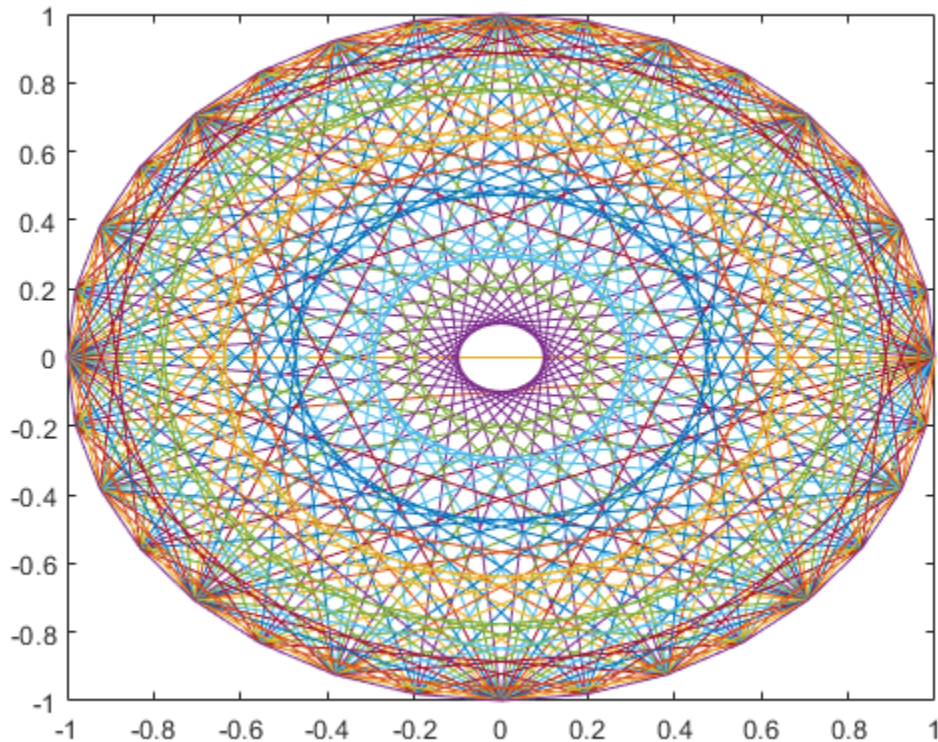

Play back the movie five times using the `movie` function.

```
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
```
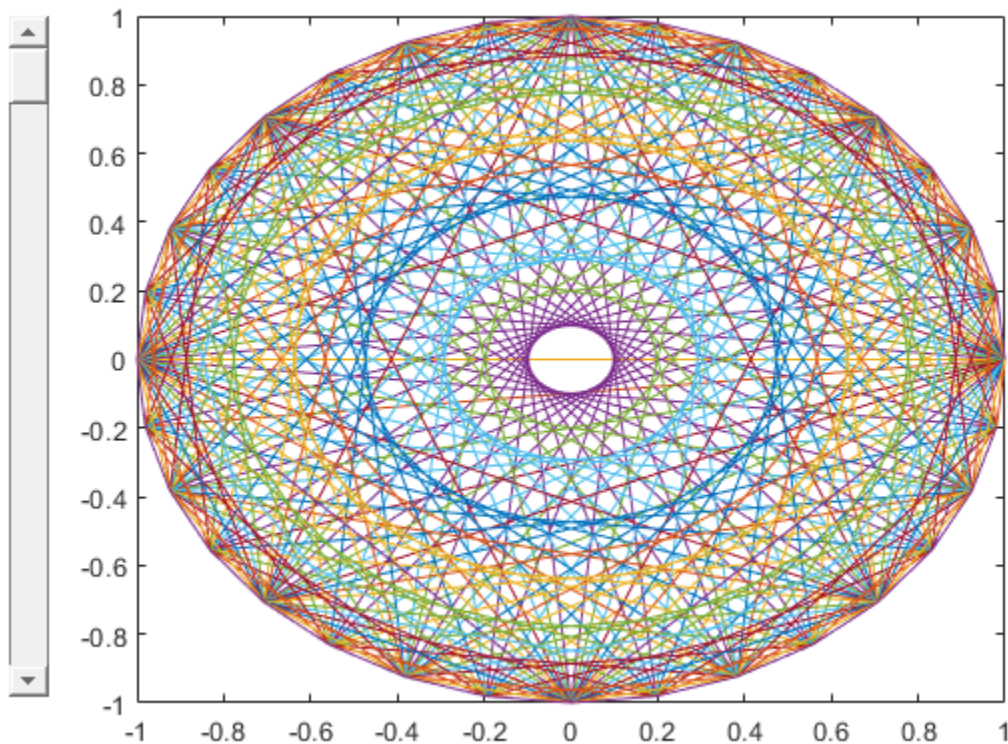

Play back the movie fives times. Movies play back within the current axes. Create a new figure and an axes to fill the figure window so that the movie looks like the original animation.

```
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 7-8
*   "Line Animations" on page 7-10

More About

Animating a Surface

This example shows how to animate a surface. Specifically, this example animates a spherical harmonic. Spherical harmonics are spherical versions of Fourier series and can be used to model the free oscillations of the Earth.

Define the Spherical Grid

Define a set of points on a spherical grid to calculate the harmonic.

```
theta = 0:pi/40:pi;                    % polar angle
phi = 0:pi/20:2*pi;                    % azimuth angle

[phi,theta] = meshgrid(phi,theta);     % define the grid
```

Calculate the Spherical Harmonic

Calculate the spherical harmonic with a degree of six, an order of one, and an amplitude of 0.5 on the surface of a sphere with a radius equal to five. Then, convert the values to Cartesian coordinates.

```
degree = 6;
order = 1;
amplitude = 0.5;
radius = 5;

Ymn = legendre(degree,cos(theta(:,1)));
Ymn = Ymn(order+1,:)';
yy = Ymn;

for kk = 2: size(theta,1)
    yy = [yy Ymn];
end

yy = yy.*cos(order*phi);

order = max(max(abs(yy)));
rho = radius + amplitude*yy/order;

r = rho.*sin(theta);     % convert to Cartesian coordinates
x = r.*cos(phi);
y = r.*sin(phi);
z = rho.*cos(theta);
```

Plot the Spherical Harmonic on the Surface of a Sphere

Using the surf function, plot the spherical harmonic on the surface of the sphere.

```
figure
s = surf(x,y,z);

light                % add a light
lighting gouraud     % preferred lighting for a curved surface
axis equal off       % set axis equal and remove axis
view(40,30)          % set viewpoint
camzoom(1.5)         % zoom into scene
```
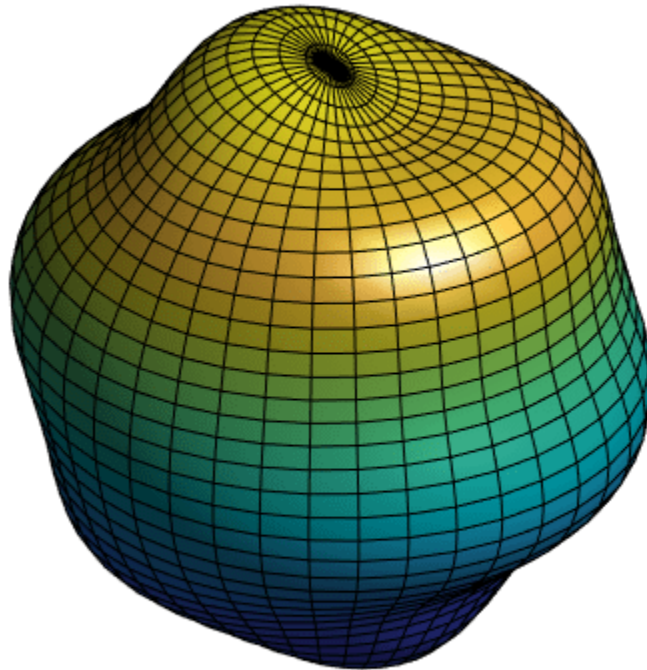
Animate the Surface

To animate the surface, use a for loop to change the data in your plot. To replace the surface data, set the XData, YData, and ZData properties of the surface to new values. To control the speed of the animation, use pause after updating the surface data.

```
scale = [linspace(0,1,20) linspace(1,-1,40)];    % surface scaling (0 to 1 to -1)

for ii = 1:length(scale)

    rho = radius + scale(ii)*amplitude*yy/order;

    r = rho.*sin(theta);
    x = r.*cos(phi);
    y = r.*sin(phi);
    z = rho.*cos(theta);

    s.XData = x;     % replace surface x values
    s.YData = y;     % replace surface y values
    s.ZData = z;     % replace surface z values

    pause(0.05)      % pause to control animation speed
end
```
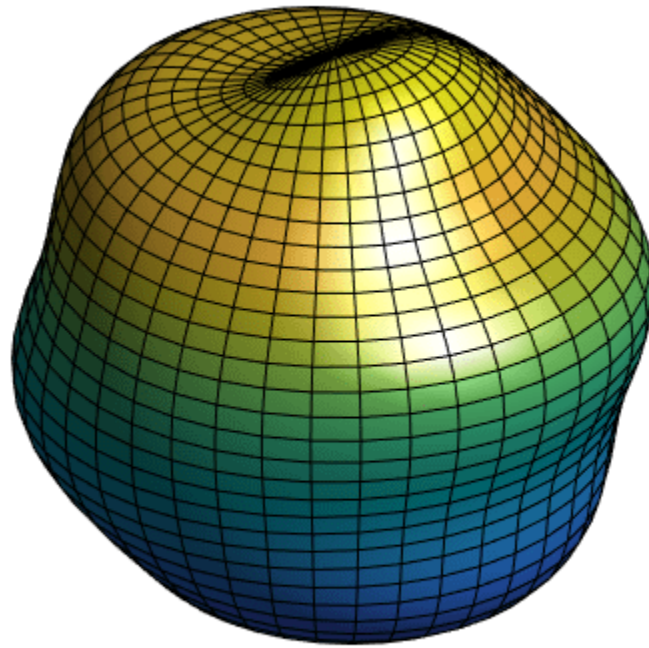
See Also
lighting | surf

Titles and Labels

Add Title and Axis Labels to Chart

This example shows how to add a title and axis labels to a chart by using the `title`, `xlabel`, and `ylabel` functions. It also shows how to customize the appearance of the axes text by changing the font size.

Create Simple Line Plot

Create x as 100 linearly spaced values between -2π and 2π. Create y1 and y2 as sine and cosine values of x. Plot both sets of data.

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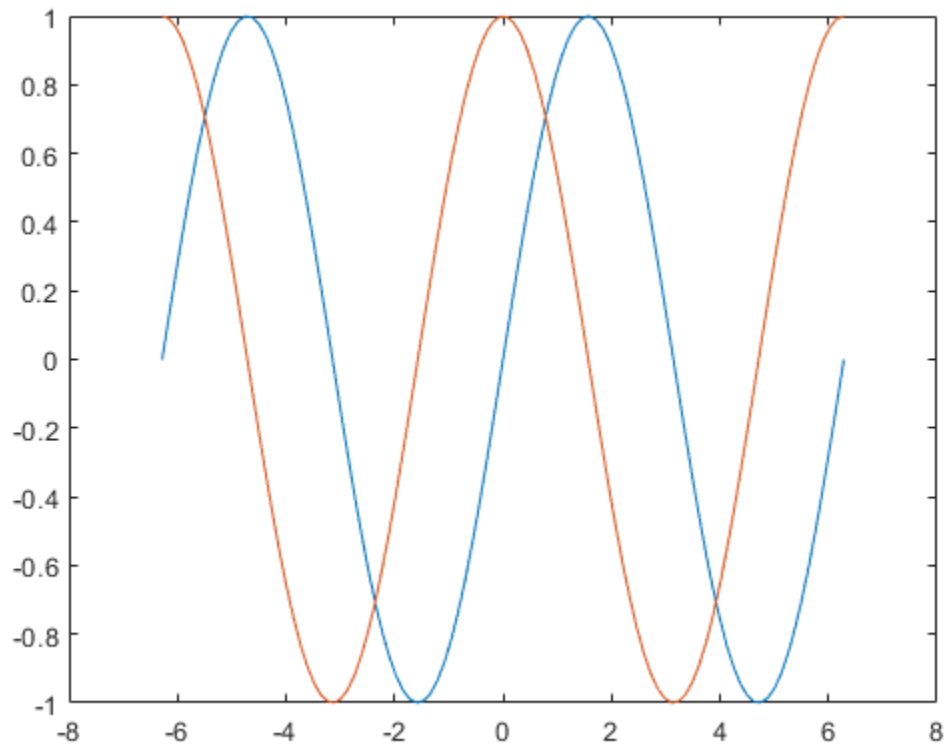

Add Title

Add a title to the chart by using the `title` function. To display the Greek symbol π, use the TeX markup, `\pi`.

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

Line Plot of Sine and Cosine Between -2π and 2π
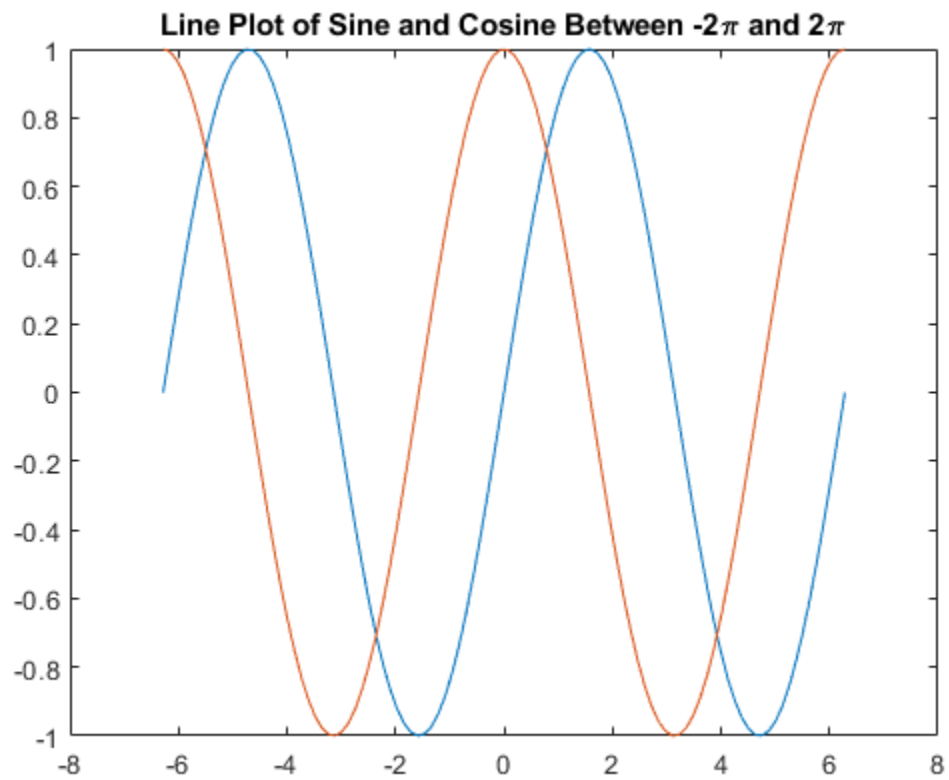
Add Axis Labels

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

```
xlabel('-2\pi < x < 2\pi')
ylabel('Sine and Cosine Values')
```

Line Plot of Sine and Cosine Between -2π and 2π
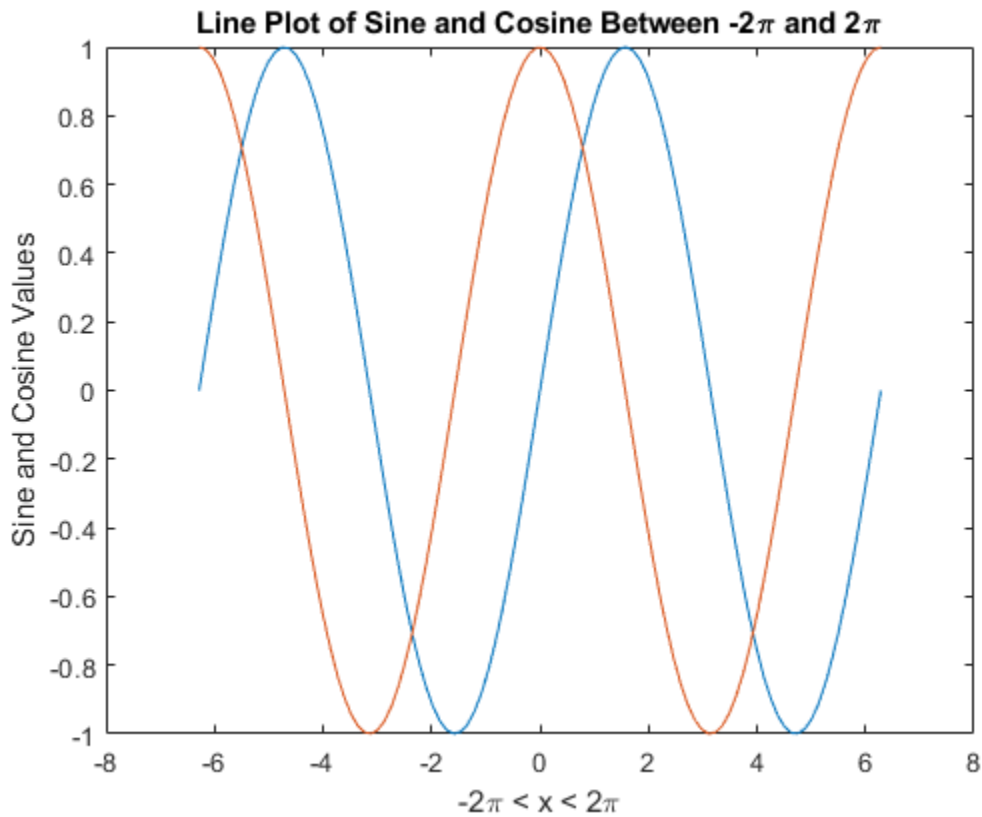
Add Legend

Add a legend to the graph that identifies each data set using the `legend` function. Specify the legend descriptions in the order that you plot the lines. Optionally, specify the legend location using one of the eight cardinal or intercardinal directions, in this case, `'southwest'`.

```
legend({'y = sin(x)','y = cos(x)'},'Location','southwest')
```
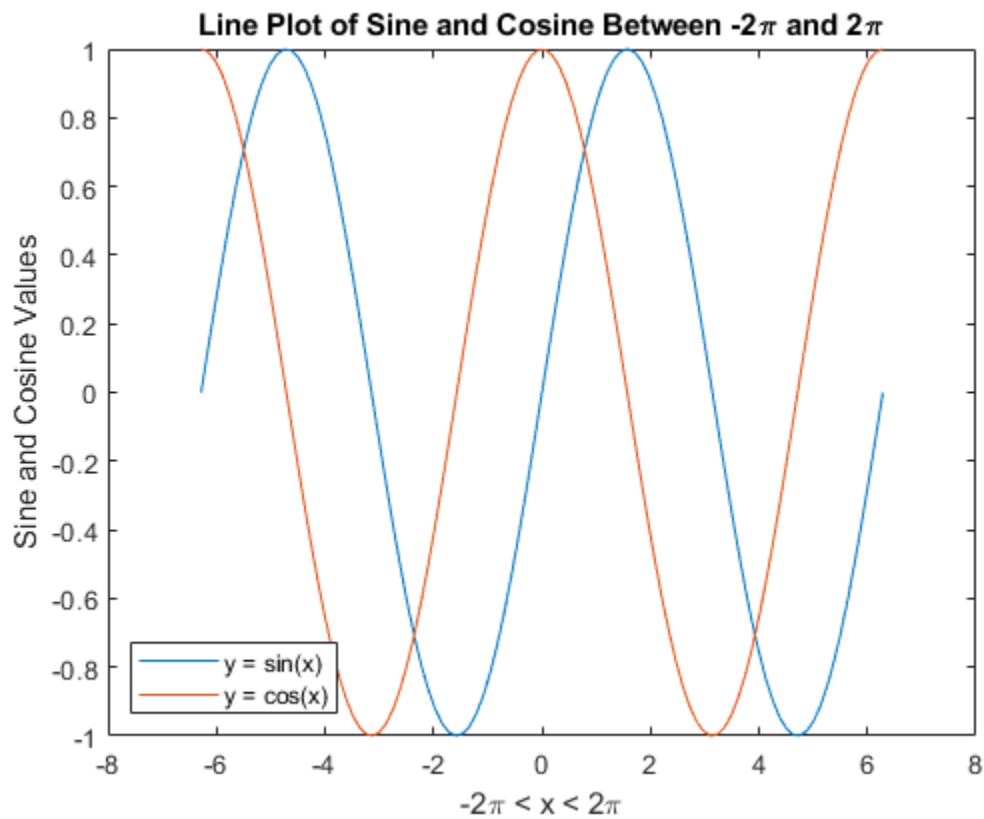
Change Font Size

Axes objects have properties that you can use to customize the appearance of the axes. For example, the FontSize property controls the font size of the title, labels, and legend.

Access the current Axes object using the gca function. Then use dot notation to set the FontSize property.

```
ax = gca;
ax.FontSize = 13;
```

Line Plot of Sine and Cosine Between -2π and 2π
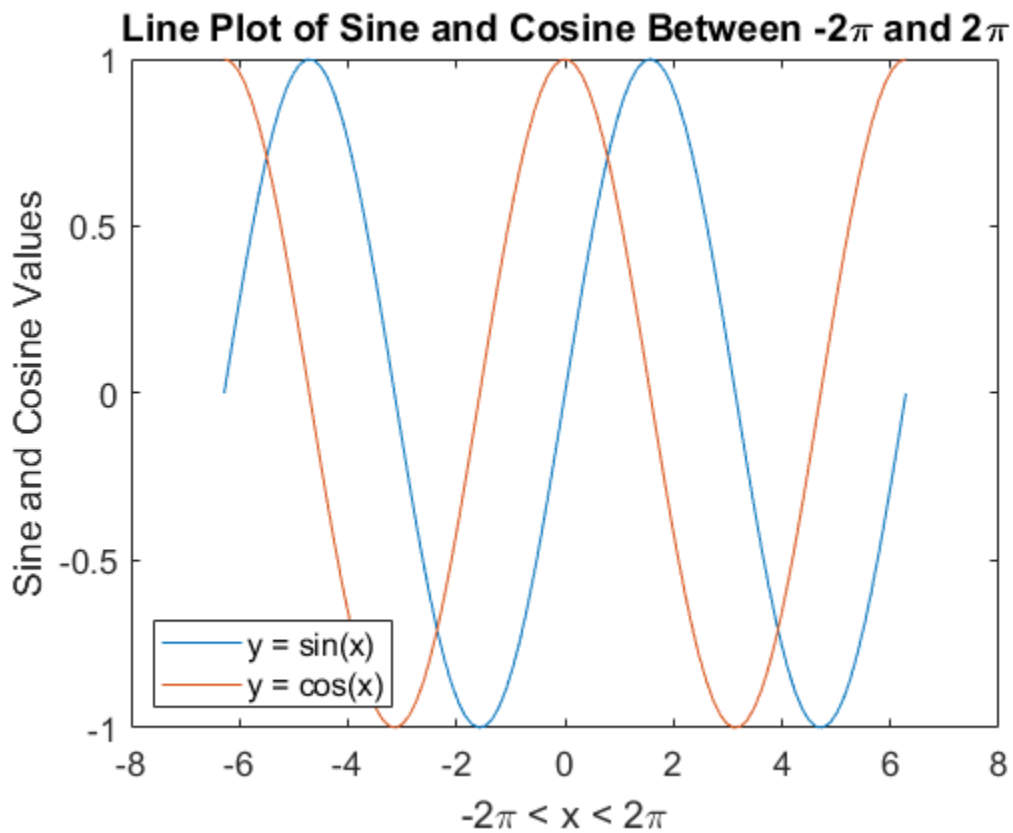
Title with Variable Value

Include a variable value in the title text by using the `num2str` function to convert the value to text. You can use a similar approach to add variable values to axis labels or legend entries.

Add a title with the value of $\sin(\pi)/2$.

```
k = sin(pi/2);
title(['sin(\pi/2) = ' num2str(k)])
```
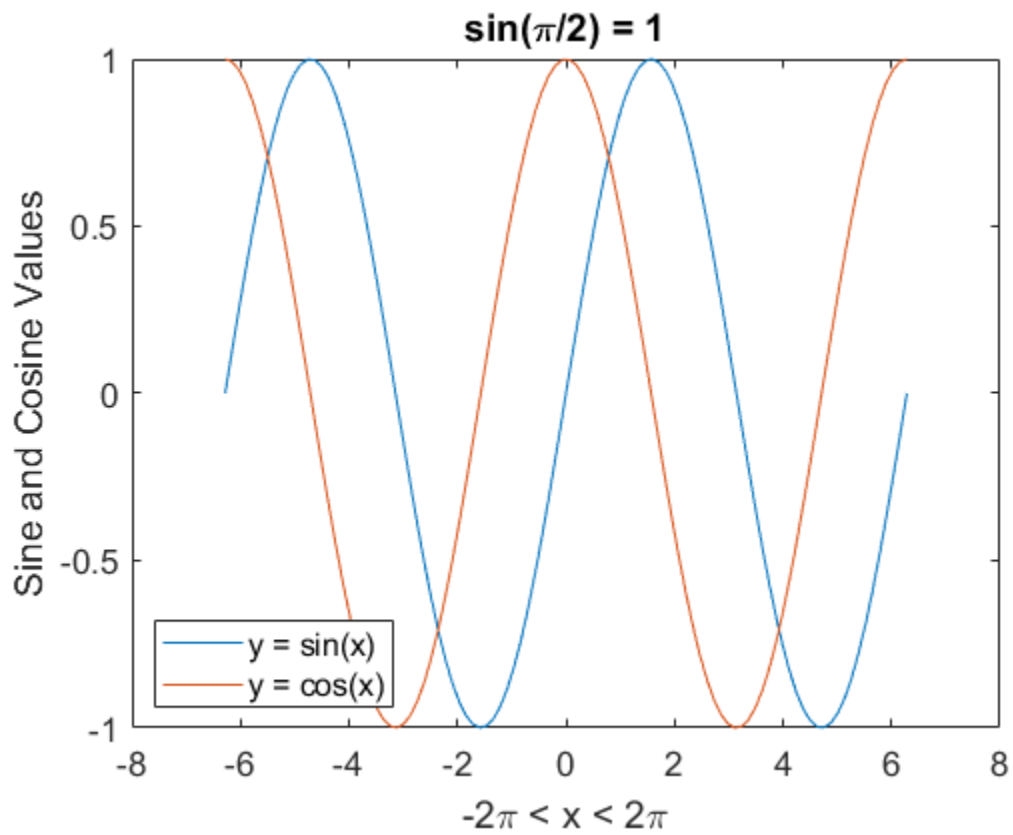
See Also
`legend` | `linspace` | `title` | `xlabel` | `ylabel`

Related Examples
- "Specify Axis Limits" on page 9-2
- "Specify Axis Tick Values and Labels" on page 9-9

Add Legend to Graph

Legends are a useful way to label data series plotted on a graph. These examples show how to create a legend and make some common modifications, such as changing the location, setting the font size, and adding a title. You also can create a legend with multiple columns or create a legend for a subset of the plotted data.

Create Simple Legend

Create a figure with a line chart and a scatter chart. Add a legend with a description for each chart. Specify the legend labels as inputs to the `legend` function.

```
figure
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1)

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled')
hold off

legend('sin(x/2)','2016')
```
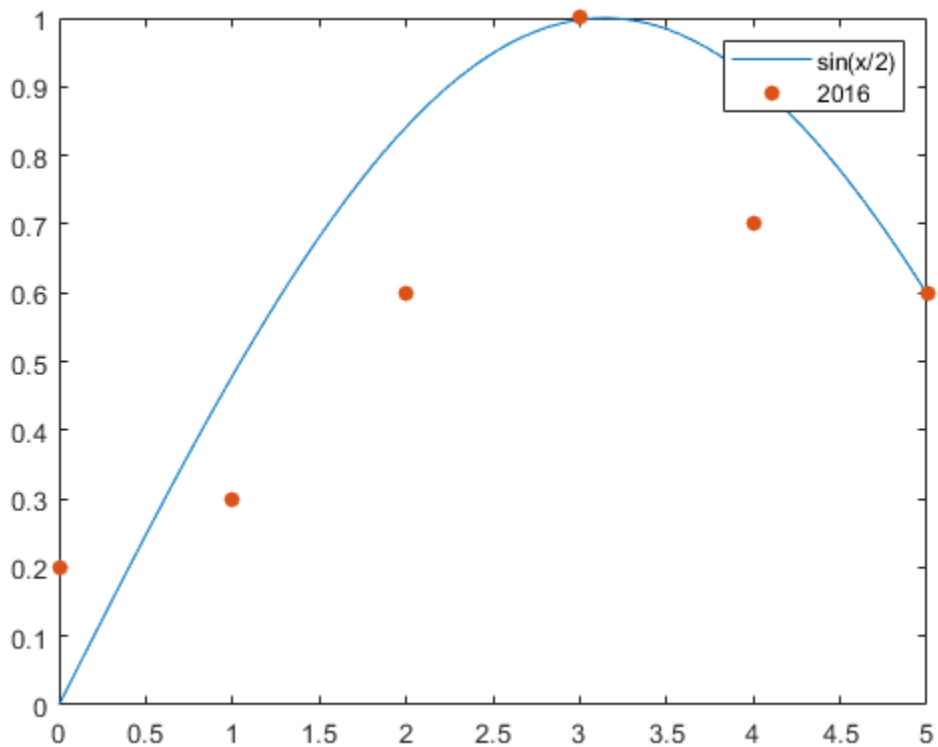
Specify Labels Using DisplayName

Alternatively, you can specify the legend labels using the `DisplayName` property. Set the `DisplayName` property as a name-value pair when calling the plotting functions. Then, call the `legend` command to create the legend.

```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1,'DisplayName','sin(x/2)')

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled','DisplayName','2016')

legend
```

Legends automatically update when you add or delete a data series. If you add more data to the axes, use the `DisplayName` property to specify the labels. If you do not set the `DisplayName` property, then the legend uses a label of the form `'dataN'`.

Add a scatter chart for 2017 data.

```
x3 = [0 1 2 3 4 5];
y3 = [0.1 0.4 0.6 0.9 0.8 0.7];
scatter(x3,y3,'filled','DisplayName','2017')
drawnow
hold off
```
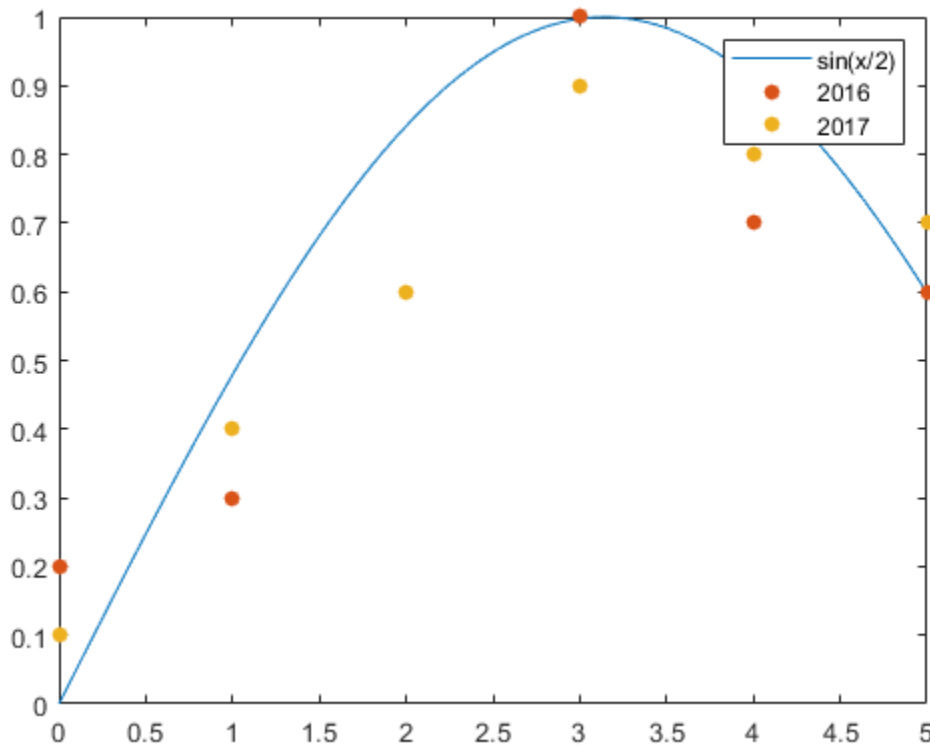
Customize Legend Appearance

The legend function creates a Legend object. Legend objects have properties that you can use to customize the appearance of the legend, such as the Location, Orientation, FontSize, and Title properties. For a full list, see Legend Properties.

You can set properties in two ways:

- Use name-value pairs in the legend command. In most cases, when you use name-value pairs, you must specify the labels in a cell array, such as legend({'label1','label2'},'FontSize',14).
- Use the Legend object. You can return the Legend object as an output argument from the legend function, such as lgd = legend. Then, use lgd with dot notation to set properties, such as lgd.FontSize = 14.

Legend Location and Orientation

Specify the legend location and orientation by setting the Location and Orientation properties as name-value pairs. Set the location to one of the eight cardinal or intercardinal directions, in this case, 'northwest'. Set the orientation to 'vertical' (the default) or 'horizontal', as in this case. Specify the labels in a cell array.

```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1)

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled')
hold off

legend({'sin(x/2)','2016'},'Location','northwest','Orientation','horizontal')
```
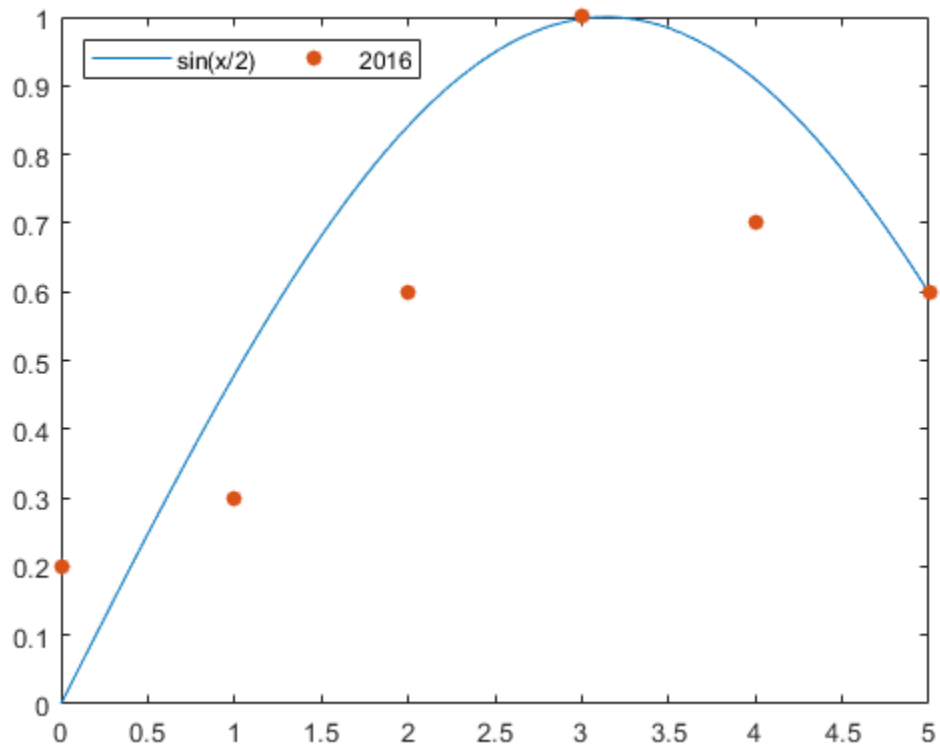
Legend Font Size and Title

Specify the legend font size and title by setting the `FontSize` and `Title` properties. Assign the `Legend` object to the variable `lgd`. Then, use `lgd` to change the properties using dot notation.

```
x1 = linspace(0,5);
y1 = sin(x1/2);
plot(x1,y1,'DisplayName','sin(x/2)')

hold on
x2 = [0 1 2 3 4 5];
y2 = [0.2 0.3 0.6 1 0.7 0.6];
scatter(x2,y2,'filled','DisplayName','2016')
hold off

lgd = legend;
lgd.FontSize = 14;
lgd.Title.String = '2016 Data';
```
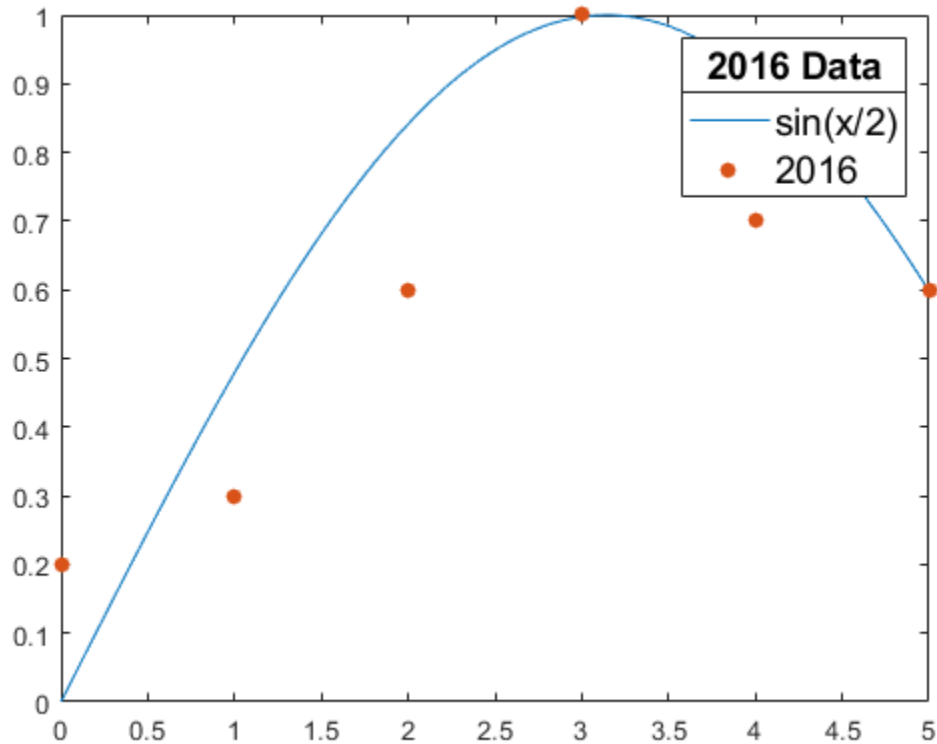
Legend with Multiple Columns

Create a chart with six line plots. Add a legend with two columns by setting the `NumColumns` property to 2.

```
x = linspace(0,10);
y1 = sin(x);
y2 = sin(0.9*x);
y3 = sin(0.8*x);
y4 = sin(0.7*x);
y5 = sin(0.6*x);
y6 = sin(0.5*x);

plot(x,y1,'DisplayName','sin(x)')
hold on
plot(x,y2,'DisplayName','sin(0.9x)')
plot(x,y3,'DisplayName','sin(0.8x)')
plot(x,y4,'DisplayName','sin(0.7x)')
plot(x,y5,'DisplayName','sin(0.6x)')
plot(x,y6,'DisplayName','sin(0.5x)')
hold off

lgd = legend;
lgd.NumColumns = 2;
```
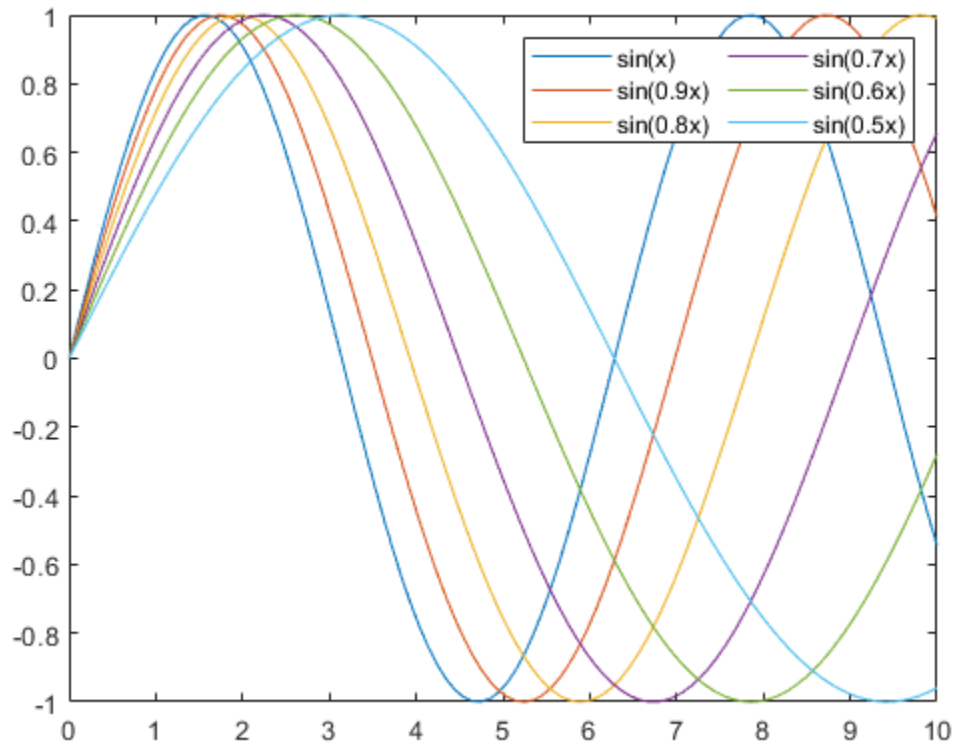
Include Subset of Charts in Legend

Combine two bar charts and a scatter chart. Create a legend that includes only the bar charts by specifying the Bar objects, b1 and b2, as the first input argument to the legend function. Specify the objects in a vector.

```
x = [1 2 3 4 5];
y1 = [.2 .4 .6 .4 .2];
b1 = bar(x,y1);

hold on
y2 = [.1 .3 .5 .3 .1];
b2 = bar(x,y2,'BarWidth',0.5);

y3 = [.2 .4 .6 .4 .2];
s = scatter(x,y3,'filled');
hold off

legend([b1 b2],'Bar Chart 1','Bar Chart 2')
```
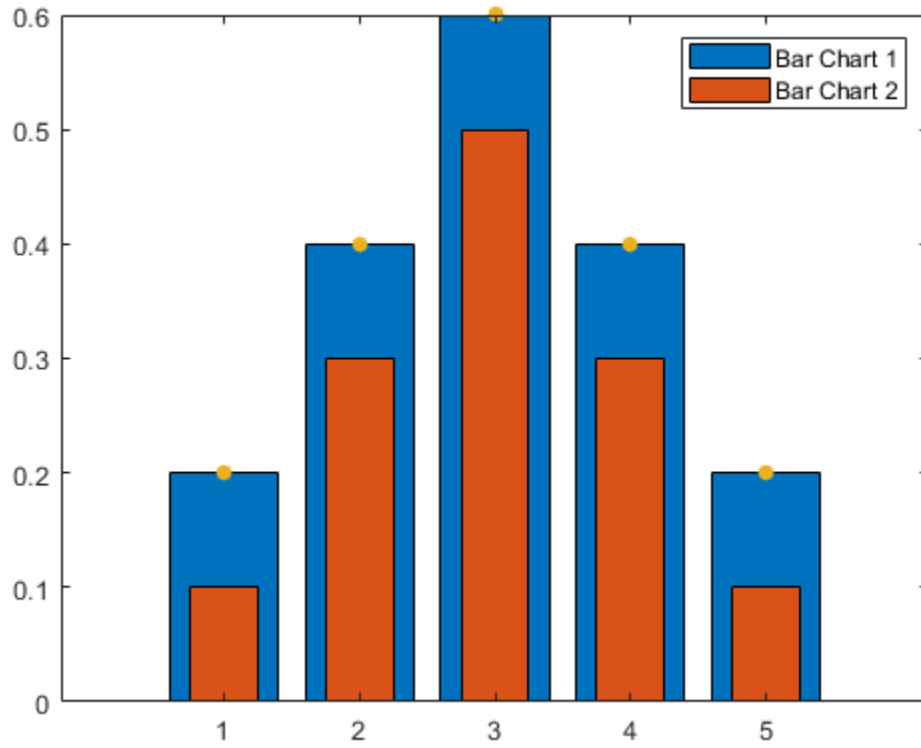
See Also
Legend Properties | `legend`

Add Text to Chart

This example shows how to add text to a chart, control the text position and size, and create multiline text.

Text Position

Add text next to a particular data point using the `text` function. In this case, add text to the point $(\pi, \sin(\pi))$. The first two input arguments to the `text` function specify the position. The third argument specifies the text.

By default, text supports a subset of TeX markup. Use the TeX markup `\pi` for the Greek letter π. Display an arrow pointing to the left by including the TeX markup `\leftarrow`. For a full list of markup, see "Greek Letters and Special Characters in Chart Text" on page 8-26.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

txt = '\leftarrow sin(\pi) = 0';
text(pi,sin(pi),txt)
```
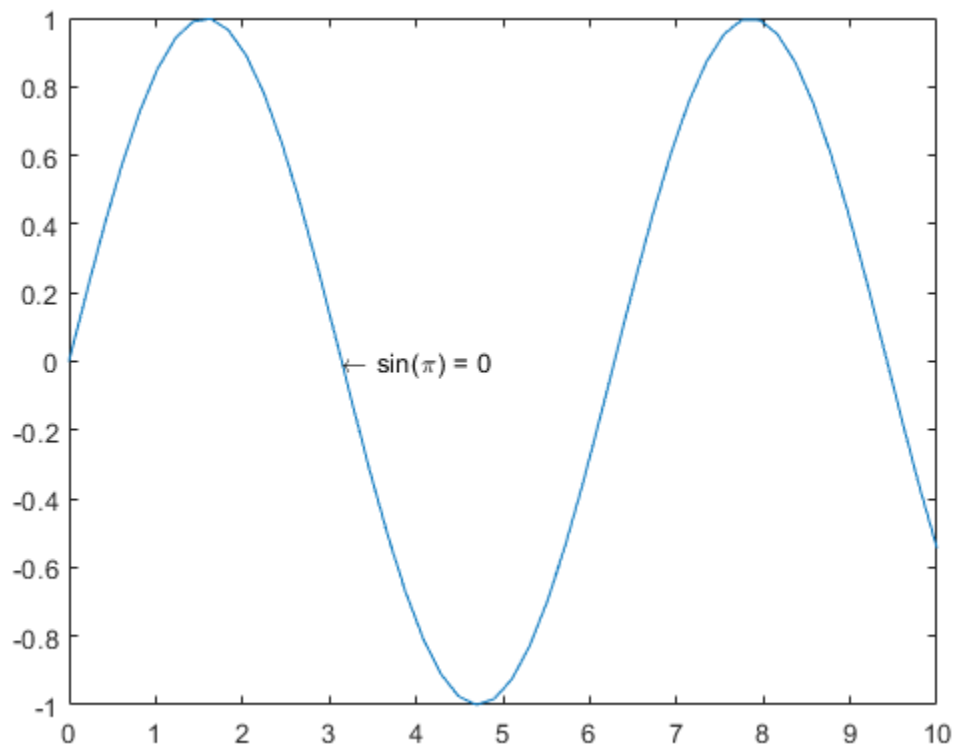

Text Alignment

By default, the specified data point is to the left of the text. Align the data point to the right of the text by specifying the `HorizontalAlignment` property as `'right'`. Use an arrow pointing to the right instead of to the left.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

txt = 'sin(\pi) = 0 \rightarrow';
text(pi,sin(pi),txt,'HorizontalAlignment','right')
```
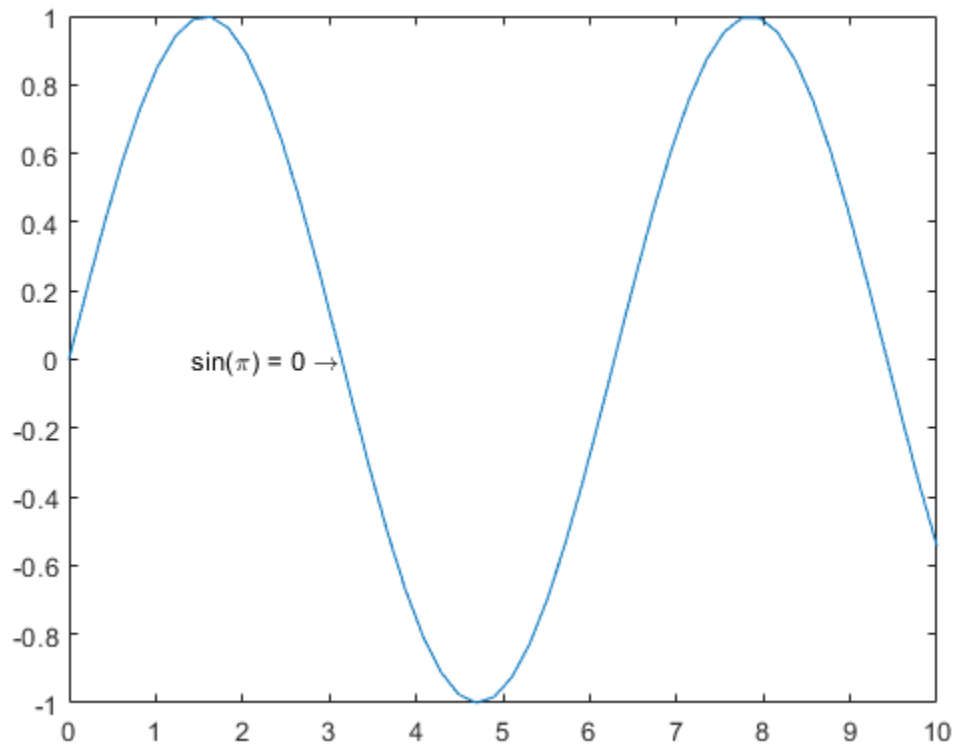

Font Size

Specify the font size for text by setting the `FontSize` property as a name-value pair argument to the `text` function. You can use a similar approach to change the font size when using the `title`, `xlabel`, `ylabel`, or `legend` functions.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

txt = '\leftarrow sin(\pi) = 0';
text(pi,sin(pi),txt,'FontSize',14)
```
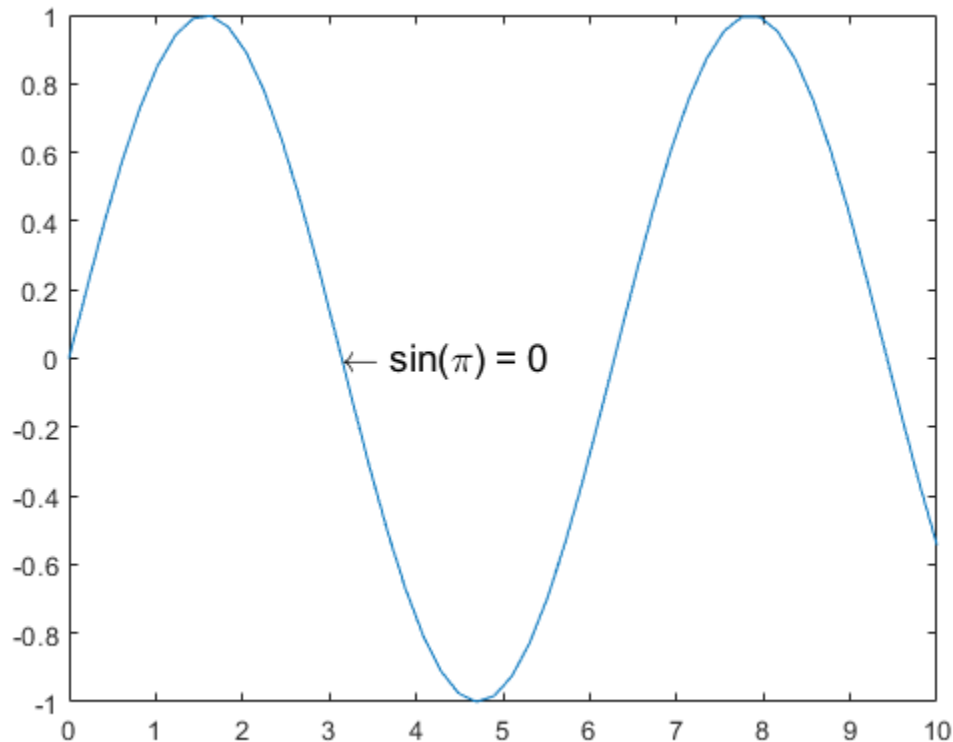
Setting Text Properties

The text function creates a Text object. `Text` objects have properties that you can use to customize the appearance of the text, such as the `HorizontalAlignment` or `FontSize`.

You can set properties in two ways:

- Use name-value pairs in the `text` command, such as `'FontSize',14`.
- Use the `Text` object. You can return the `Text` object as an output argument from the `text` function and assign it to a variable, such as `t`. Then, use dot notation to set properties, such as `t.FontSize = 14`.

For this example, change the font size using dot notation instead of a name-value pair.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

txt = '\leftarrow sin(\pi) = 0';
t = text(pi,sin(pi),txt)

t =
  Text (\leftarrow sin(\pi) = 0) with properties:

              String: '\leftarrow sin(\pi) = 0'
            FontSize: 10
          FontWeight: 'normal'
```

```
              FontName: 'Helvetica'
                 Color: [0 0 0]
   HorizontalAlignment: 'left'
              Position: [3.1416 1.2246e-16 0]
                 Units: 'data'

   Show all properties
```

```
t.FontSize = 14;
```
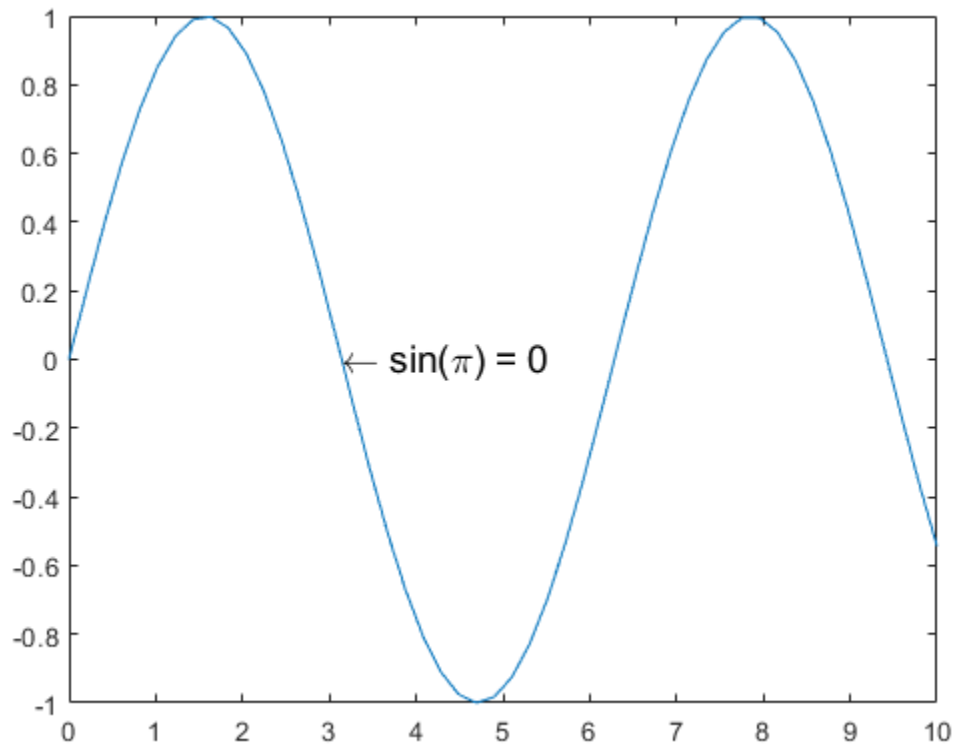

Multiline Text

Display text across multiple lines using a cell array of character vectors. Each element of the cell array is one line of text. For this example, display a title with two lines. You can use a similar approach to display multiline text with the `title`, `xlabel`, `ylabel`, or `legend` functions.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

txt = {'Plotted Data:','y = sin(x)'};
text(4,0.5,txt)
```
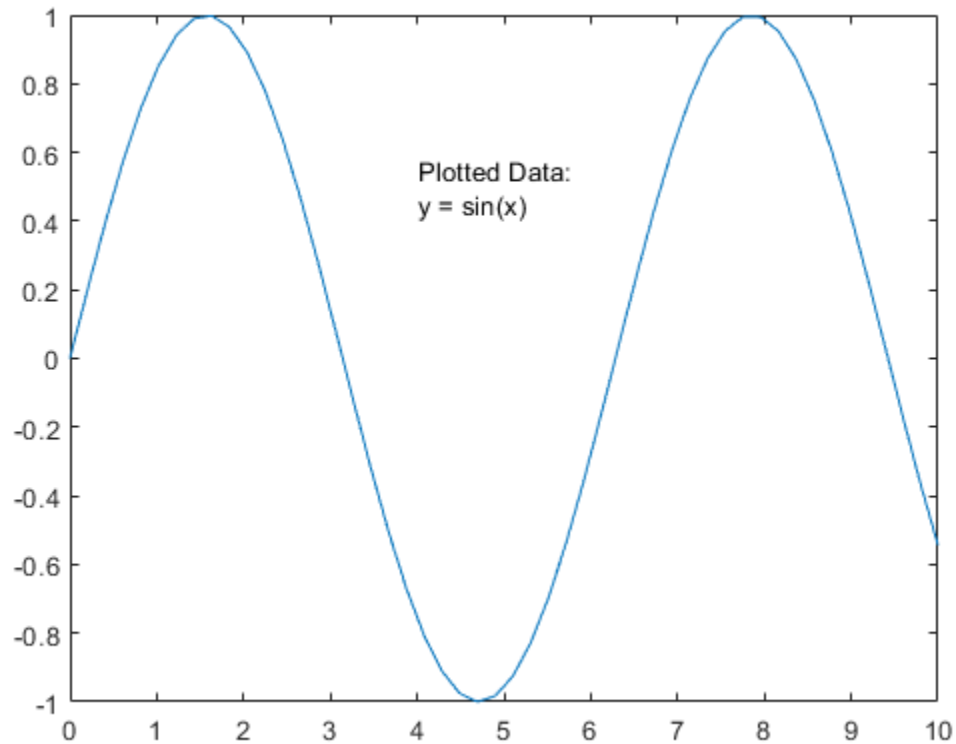
Text with Variable Value

Include a variable value in text by using the `num2str` function to convert the number to text. For this example, calculate the average *y* value and include the value in the title. You can use a similar approach to include variable values with the `title`, `xlabel`, `ylabel`, or `legend` functions.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

avg = mean(y);
txt = ['Average height: ' num2str(avg) ' units'];
text(4,0.5,txt)
```
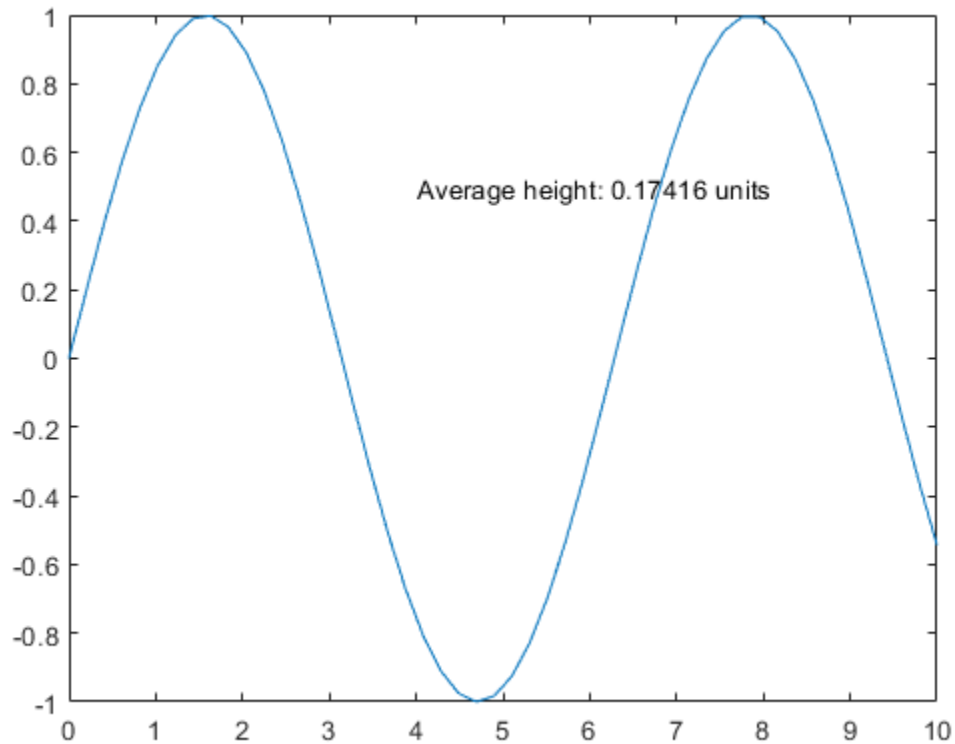
Text Outside Axes

Add text anywhere within the figure using the `annotation` function instead of the `text` function. The first input argument specifies the type of annotation. The second input argument specifies the position of the annotation in units normalized to the figure. Remove the text box border by setting the `EdgeColor` property to `'none'`. For more information on text box annotations, see the `annotation` function.

```
x = linspace(0,10,50);
y = sin(x);
plot(x,y)

annotation('textbox',[.9 .5 .1 .2],'String','Text outside the axes','EdgeColor','none')
```
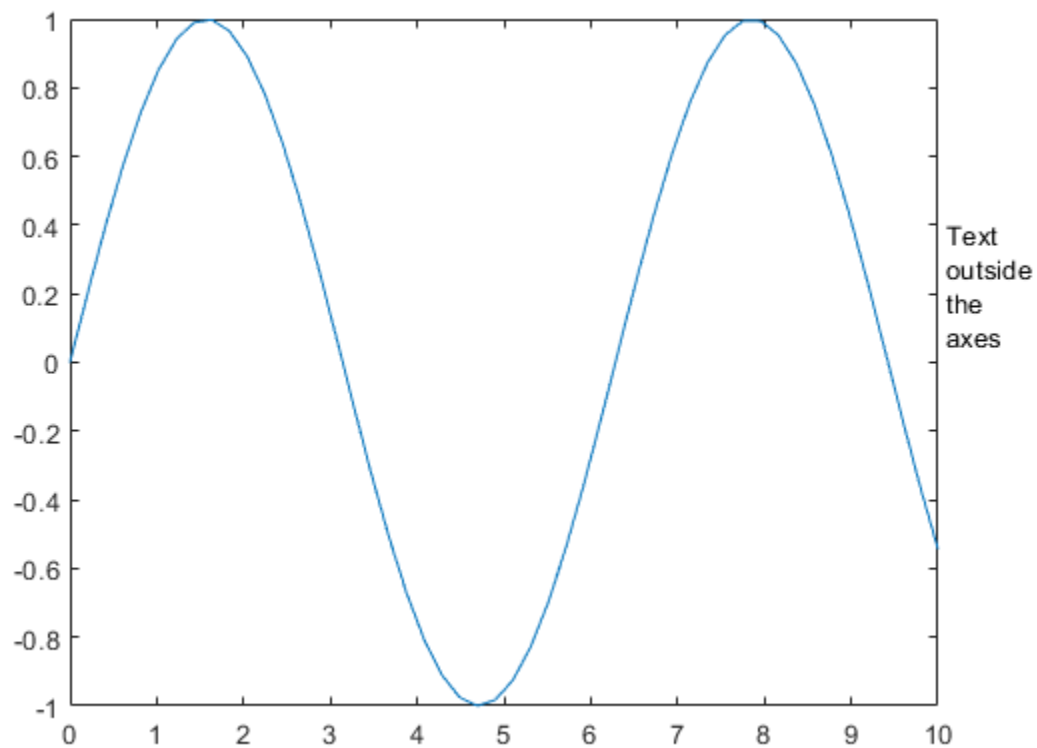
See Also
annotation | text | title | xlabel | ylabel

Related Examples
- "Greek Letters and Special Characters in Chart Text" on page 8-26

Add Annotations to Chart

Annotations are extra information added to a chart to help identify important information. This example first explains the different types of annotations, and then shows you how to add circles and text arrows to a chart.

Types of Annotations

Use the `annotation` function to add annotations to a chart. The first input to the function specifies the type of annotation you want to create.

- If you specify the type as `'line'`, `'arrow'`, `'doublearrow'`, or `'textarrow'`, then the second input is the starting and ending *x* positions of the annotation. The third input is the starting and ending *y* positions of the annotation. For example, `annotation('line',[x_begin x_end], [y_begin y_end])`.
- If you specify the type as `'rectangle'`, `'ellipse'`, or `'textbox'`, then the second argument is the location and size. For example, `annotation('rectangle',[x y w h])`.

Annotations use normalized figure units and can span multiple axes in a figure.

Create Simple Plot

Define and plot functions `f(x)` and `g(x)`.

```
x = -3.0:0.01:3.0;
f = x.^2;
g = 5*sin(x) + 5;

figure
plot(x,f)
hold on
plot(x,g)
hold off
```
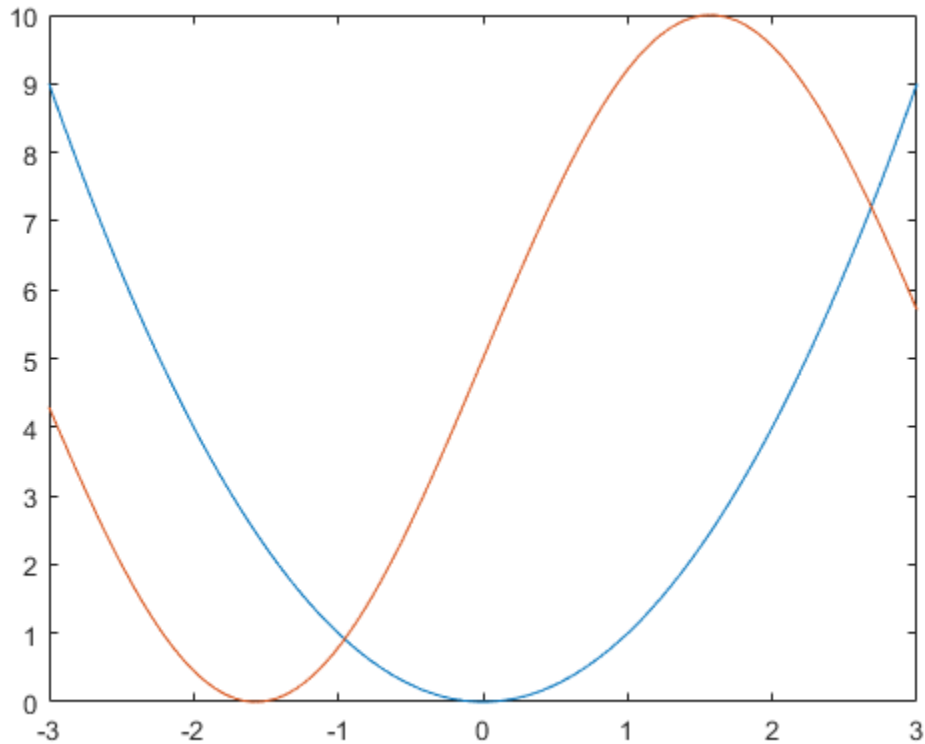
Circle Annotations

Add a circle to the chart to highlight where `f(x)` and `g(x)` are equal. To create a circle, use the `'ellipse'` option for the annotation type.

Customize the circle by setting properties of the underlying object. Return the `Ellipse` object as an output argument from the `annotation` function. Then, access properties of the object using dot notation. For example, set the `Color` property.

```
elps = annotation('ellipse',[.84 .68 .05 .05])

elps =
  Ellipse with properties:

        Color: [0 0 0]
    FaceColor: 'none'
    LineStyle: '-'
    LineWidth: 0.5000
     Position: [0.8400 0.6800 0.0500 0.0500]
        Units: 'normalized'

  Show all properties


elps.Color = [0 0.5 0.5];
```
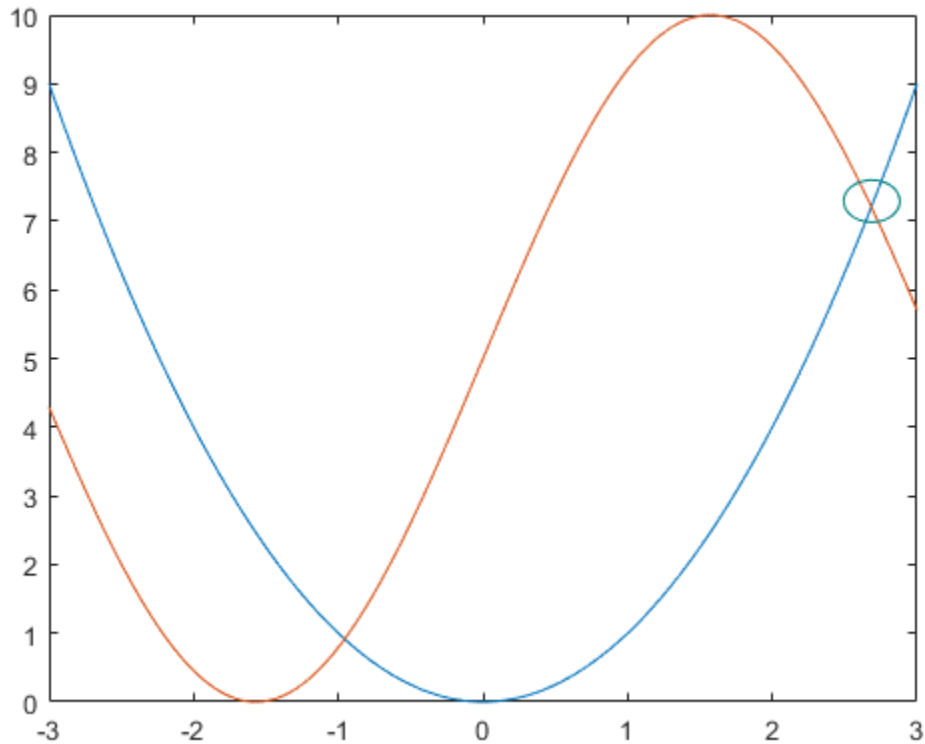
Text Arrow Annotations

Add a text arrow to the chart using the `'textarrow'` option for the annotation type.

You can customize the text arrow by setting properties of the underlying object. Return the `TextArrow` object as an output argument from the `annotation` function. Then, access properties of the object using dot notation. For example, set the `String` property to the desired text and the `Color` property to a color value.

```
ta = annotation('textarrow', [0.76 0.83], [0.71 0.71])

ta =
  TextArrow with properties:

        String: {''}
      FontName: 'Helvetica'
      FontSize: 10
         Color: [0 0 0]
     TextColor: [0 0 0]
     LineStyle: '-'
     LineWidth: 0.5000
     HeadStyle: 'vback2'
      Position: [0.7600 0.7100 0.0700 0]
         Units: 'normalized'
             X: [0.7600 0.8300]
             Y: [0.7100 0.7100]
```

Show all properties

```
ta.String = 'f(x) = g(x) ';
ta.Color = [0 0.5 0.5];
```
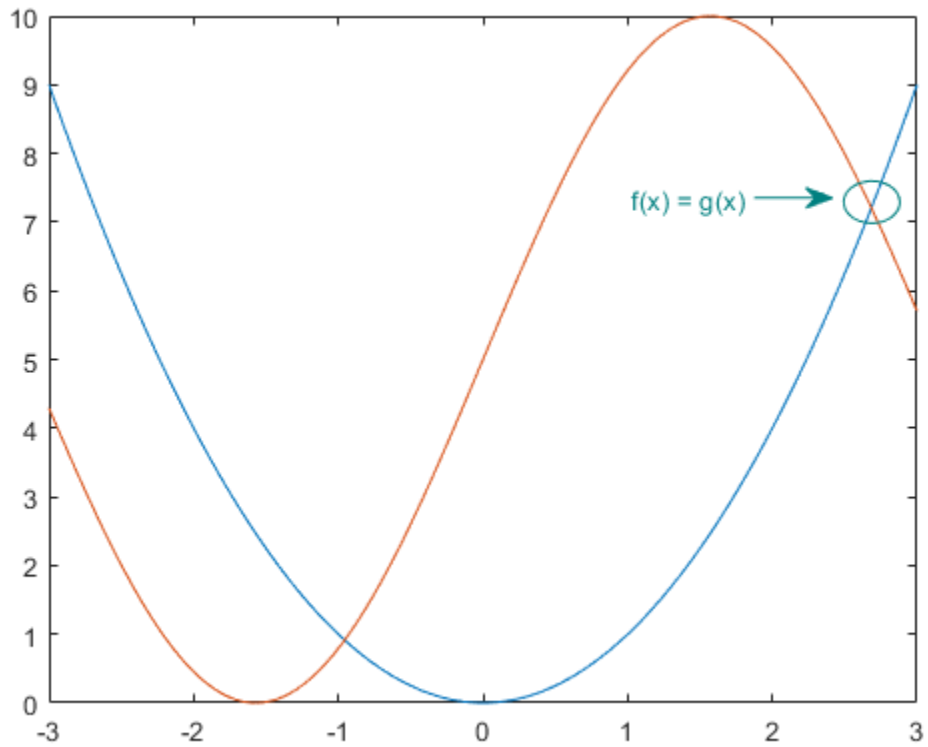

See Also
annotation | text

Related Examples
*   "Greek Letters and Special Characters in Chart Text" on page 8-26

Greek Letters and Special Characters in Chart Text

You can add text to a chart 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. By default, MATLAB supports a subset of TeX markup. To use additional special characters, such as integral and summation symbols, you can use LaTeX markup instead. This example shows how to insert Greek letters, superscripts, and annotations into chart text and explains other available TeX options.

Include Greek Letters

Create a simple line plot and add a title. 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')
```
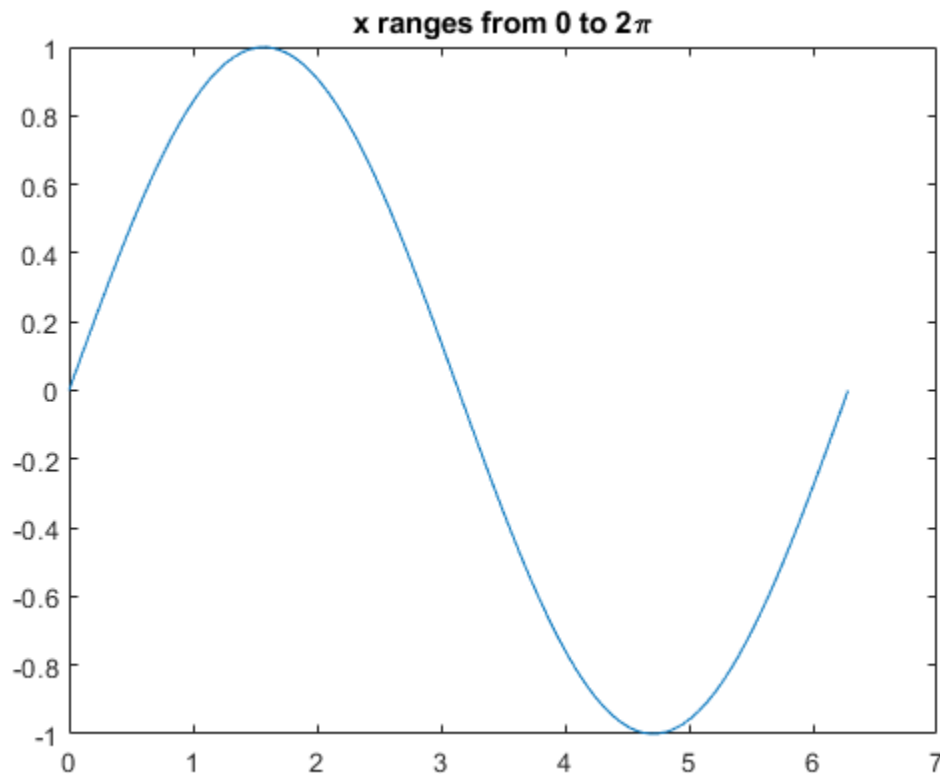

Include Superscripts and Annotations

Create a line plot and add a title and axis labels to the chart. 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')
ylabel('Amplitude')
```
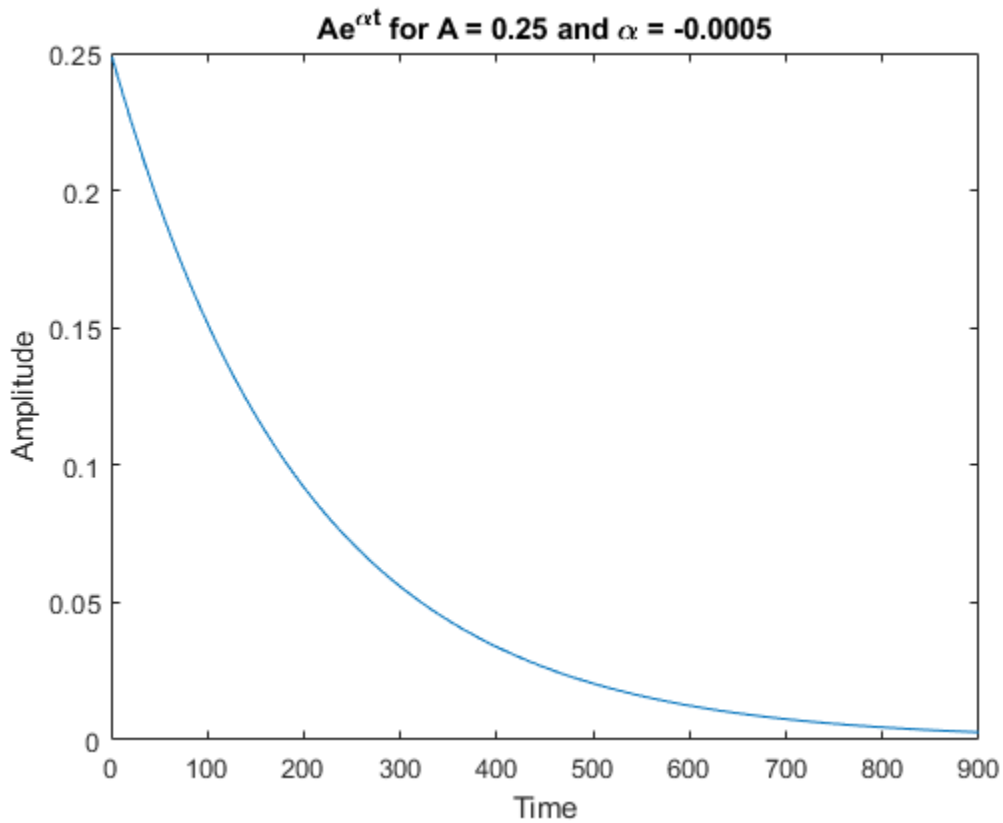

Add text at the data point where `t = 300`. Use the TeX markup `\bullet` to add a marker to the specified point and use `\leftarrow` to include an arrow pointing to the left. By default, the specified data point is to the left of the text.

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

$$Ae^{\alpha t} \text{ for A = 0.25 and } \alpha = \text{-0.0005}$$
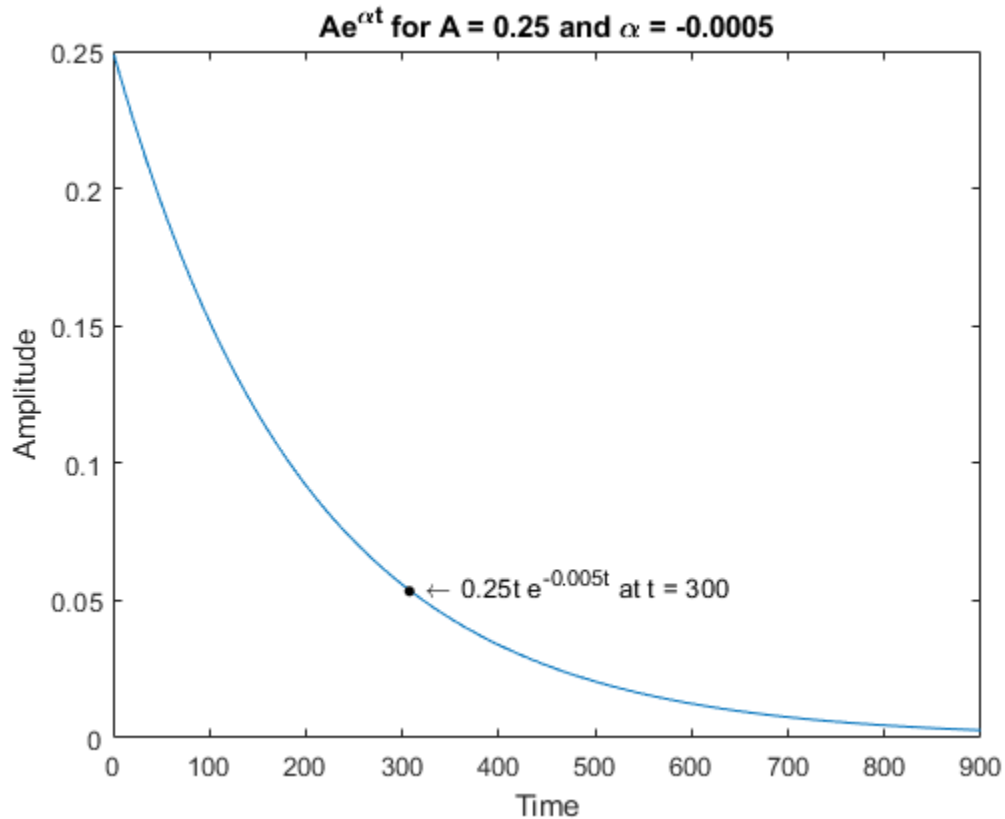
TeX Markup Options

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 TeX markup as long as the `Interpreter` property of the text object is set to `'tex'` (the default).

Modifiers remain in effect until the end of the text. Superscripts and subscripts are an exception because they modify only the next character or the characters within the curly braces. When you set the interpreter to `'tex'`, the supported modifiers are as follows.

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'`

Modifier	Description	Example
\fontname{*specifier*}	Font name — Replace *specifier* with the name of a font family. You can use this in combination with other modifiers.	'\fontname{Courier} text'
\fontsize{*specifier*}	Font size —Replace *specifier* with a numeric scalar value in point units.	'\fontsize{15} text'
\color{*specifier*}	Font color — Replace *specifier* with 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 — Replace *specifier* with a three-element RGB triplet.	'\color[rgb]{0,0.5,0.5} text'

This table lists the supported special characters for the 'tex' interpreter.

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ϵ	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\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	∝

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\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	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	⟩	\mid	|
\vee	∨	\langle	⟨	\copyright	©

Create Text with LaTeX

By default, MATLAB interprets text using TeX markup. However, for more formatting options, you can use LaTeX markup instead.

For example, plot $y = x^2\sin(x)$ and draw a vertical line at $x = 2$. Add text to the graph that contains an integral expression using LaTeX markup. To show the expression in display mode, surround the markup with double dollar signs ($$). When you call the `text` function, set the `Interpreter` property to `'latex'`.

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

str = '$$ \int_{0}^{2} x^2\sin(x) dx $$';
text(1.1,0.5,str,'Interpreter','latex')
```
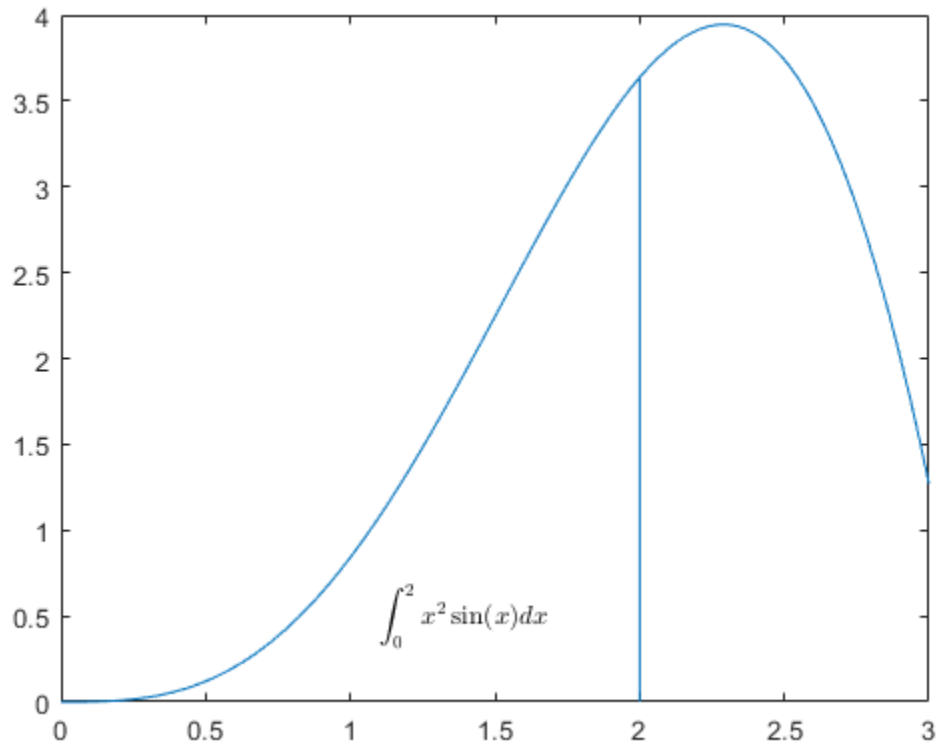
Create Plot Titles, Tick Labels, and Legends with LaTeX

You can use LaTeX markup in plot titles, tick labels, and legends. For example, create a plot of a sine wave and a cosine wave.

```
x = -10:0.1:10;
y = [sin(x); cos(x)];
plot(x,y)
```
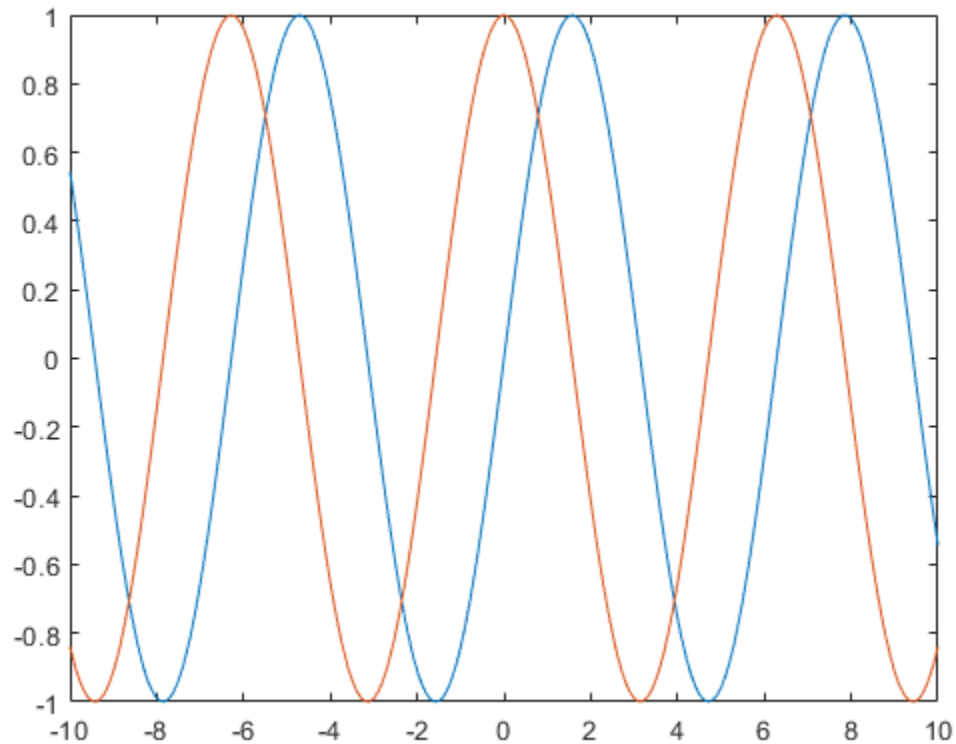
Set the *x*-axis tick values to be multiples of `pi` by calling the `xticks` function. Then, call the `gca` function to get the current axes, and set the the `TicklabelInterpreter` property to `'latex'`. Specify the tick labels using LaTeX markup. For inline expressions, surround the markup with single dollar signs ($).

```
xticks([-3*pi -2*pi -pi 0 pi 2*pi 3*pi])
ax = gca;
ax.TickLabelInterpreter = 'latex';
xticklabels({'$-3\pi$','$-2\pi$','$-\pi$','0', '$\pi$','$2\pi$','$3\pi$'});
```
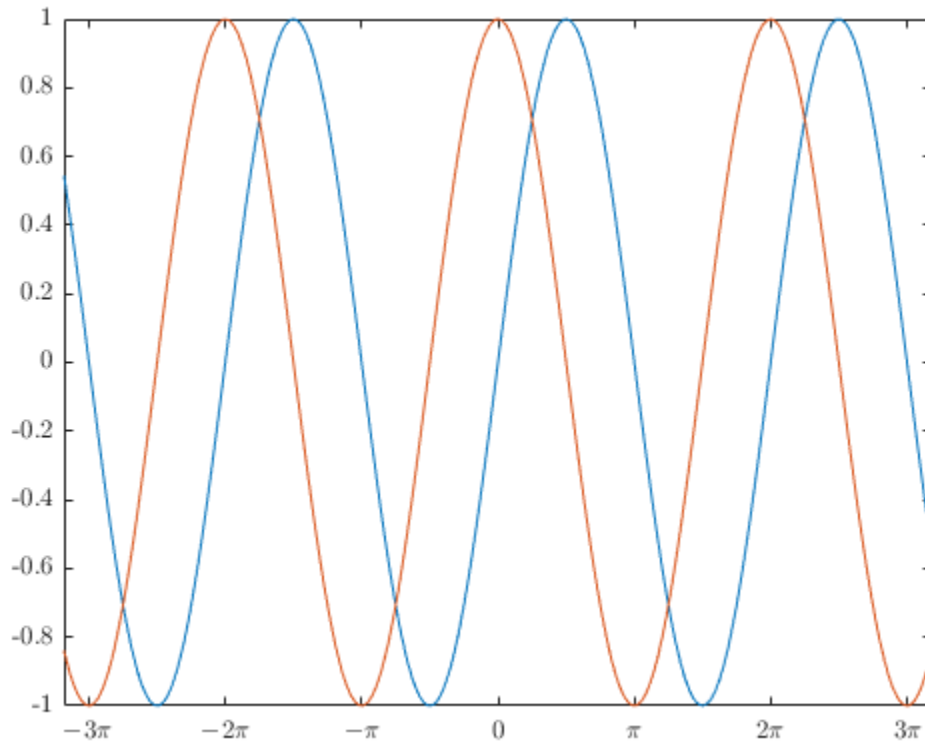
Add a title that includes LaTeX markup by calling the `title` function and setting the `Interpreter` property to `'latex'`. Similarly, create a legend with labels that include LaTeX markup.

```
% Add title
str = 'Estimates $\hat{\psi_1}$ and $\hat{\psi_2}$';
title(str,'Interpreter','latex')

% Add legend
label1 = '$\hat{\psi_1}$';
label2 = '$\hat{\psi_2}$';
legend(label1,label2,'Interpreter','latex')
```
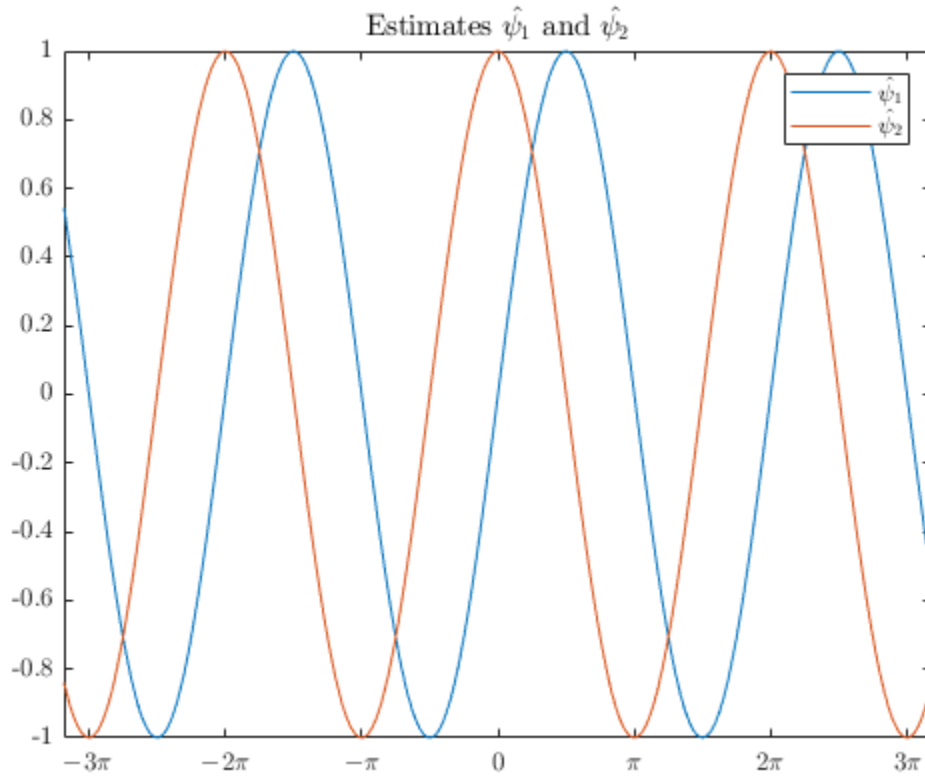
See Also
`plot` | `text` | `title` | `xlabel` | `ylabel`

More About
- "Add Title and Axis Labels to Chart" on page 8-2
- "Add Text to Chart" on page 8-15

External Websites
- https://www.latex-project.org/

Make the Graph Title Smaller

MATLAB graphics titles use a bold and slightly larger font for better visibility. As a result, some text might not fit within the extents of the figure window. For example, this code creates a graph that has a long title that does not fit within the extents of the figure window.

```
plot(1:10);
title(['This is a title that is too long and does not fit',...
    'within the extents of the figure window.'])
```
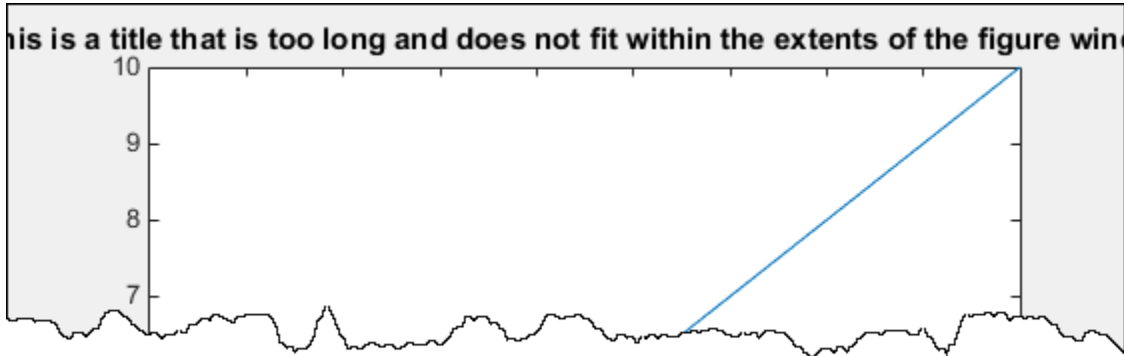

The title font size is based on the `TitleFontSizeMultiplier` and `FontSize` properties of the axes. By default the `FontSize` property is 10 points and the `TitleFontSizeMultiplier` is `1.100`, which means that the title font size is 11 points.

To change the title font size without affecting the rest of the font in the axes, set the `TitleFontSizeMultiplier` property of the axes.

```
plot(1:10);
title(['This is a title that is too long and does not fit',...
    'within the extents of the figure window.'])
ax = gca;
ax.TitleFontSizeMultiplier = 1;
```
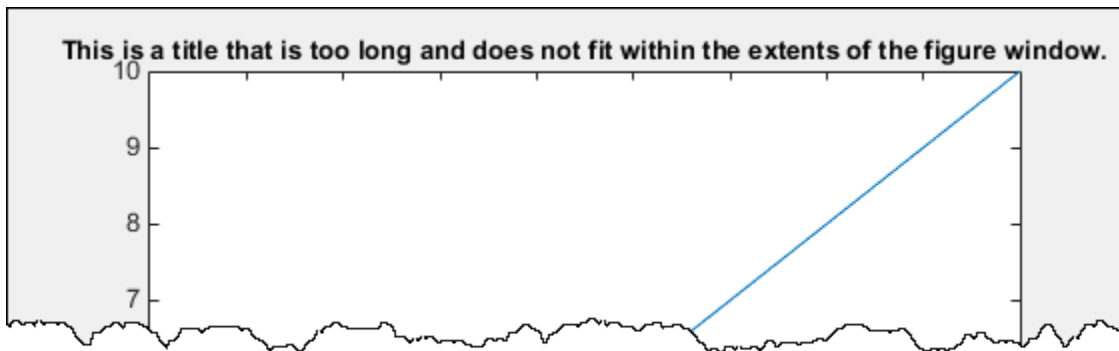

To make the font size smaller for the entire axes, set the `FontSize` property. Changing this property affects the font for the title, tick labels and axis labels, if they exist.

```
plot(1:10);
title(['This is a title that is too long and does not fit',...
    'within the extents of the figure window.'])
```

```
ax = gca;
ax.FontSize = 8;
```
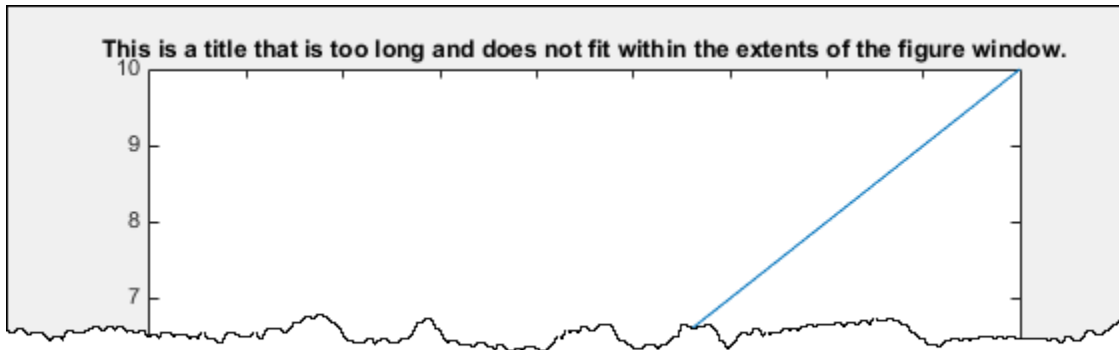

To keep the same font size and display the title across two lines, use a cell array with curly brackets {} to define a multiline title.

```
plot(1:10);
title({'This is a title that is too long and does not fit',...
    'within the extents of the figure window.'})
```


See Also

Functions
title

Properties
Axes

9

Axes Appearance

Specify Axis Limits

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])
```
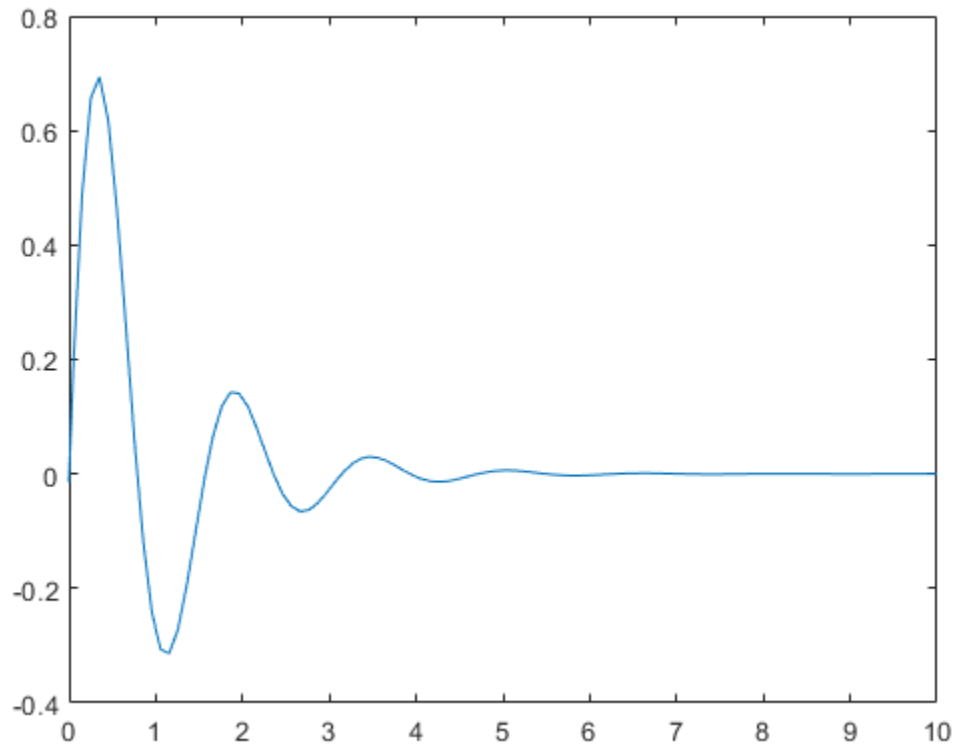

Use Semiautomatic Axis Limits

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 maximum 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])
```
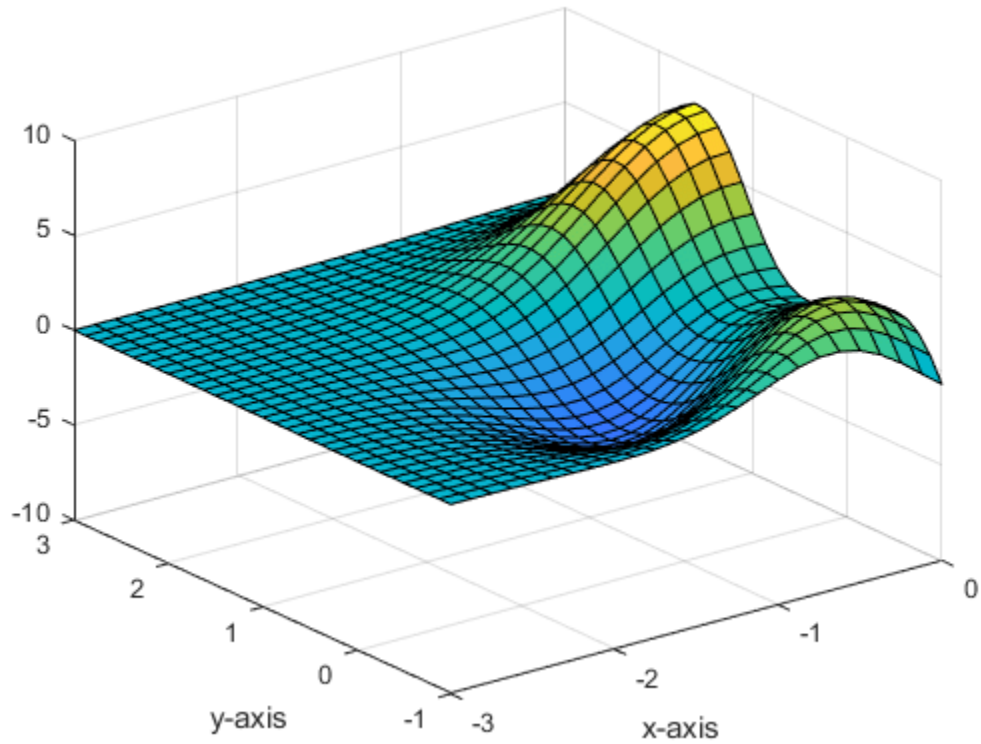

Revert Back to Default Limits

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])
```
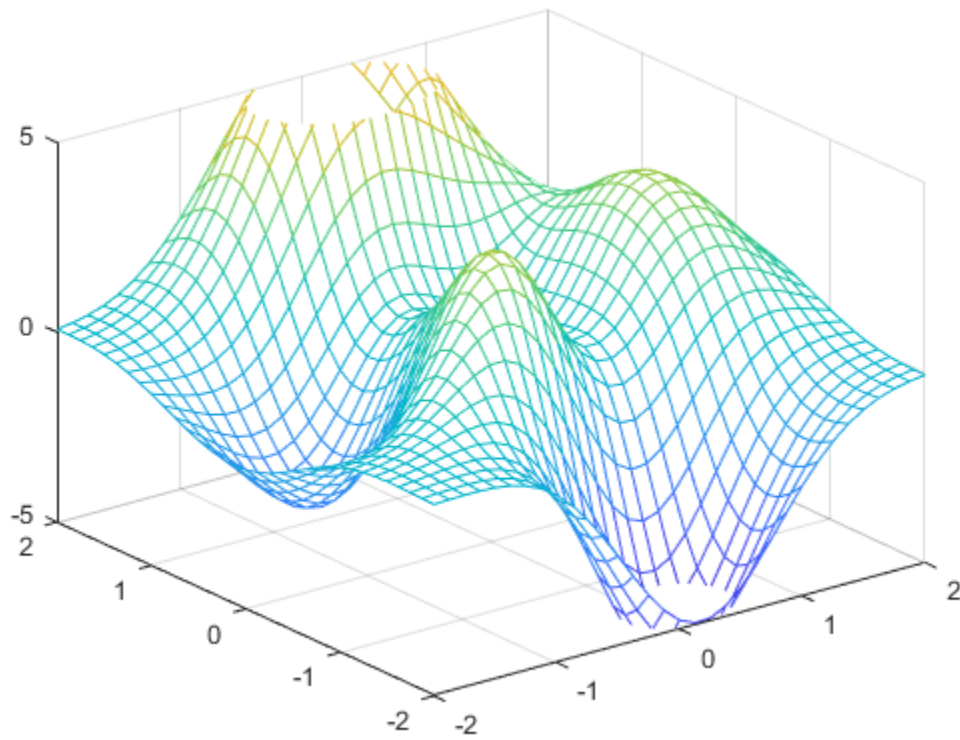
```
xlim auto
ylim auto
zlim auto
```
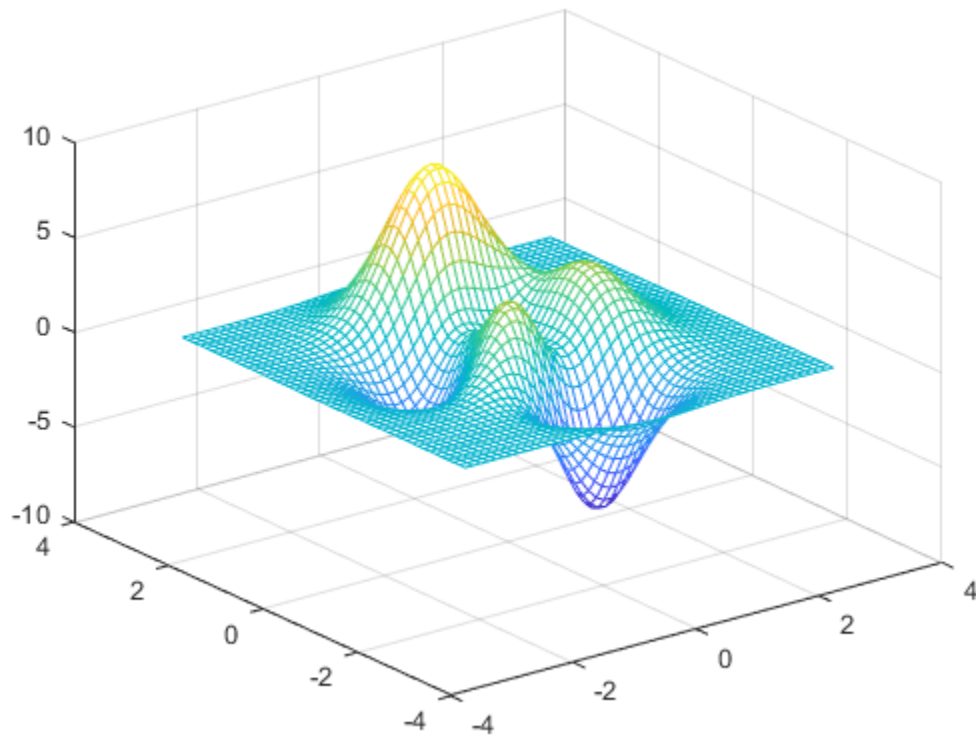
Reverse Axis Direction

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';
```
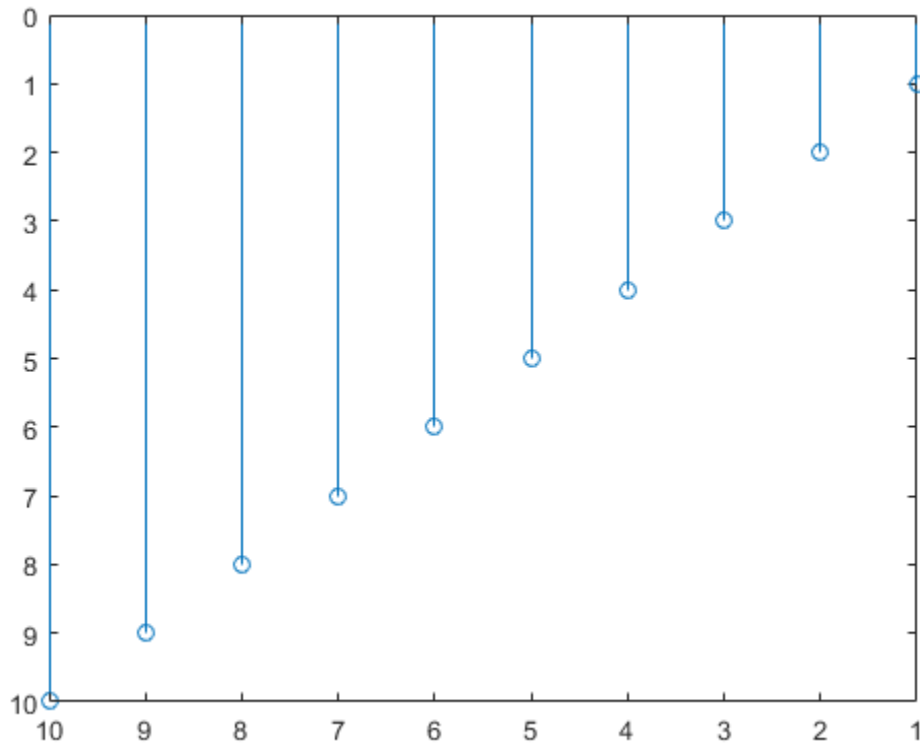
Display Axis Lines through Origin

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';
```
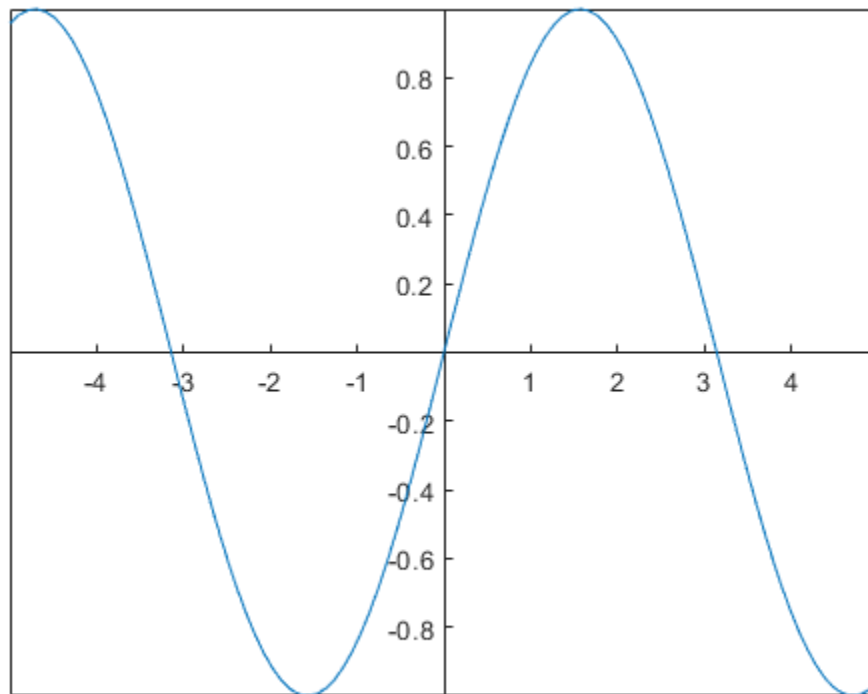
Remove the axes box outline.

```
box off
```
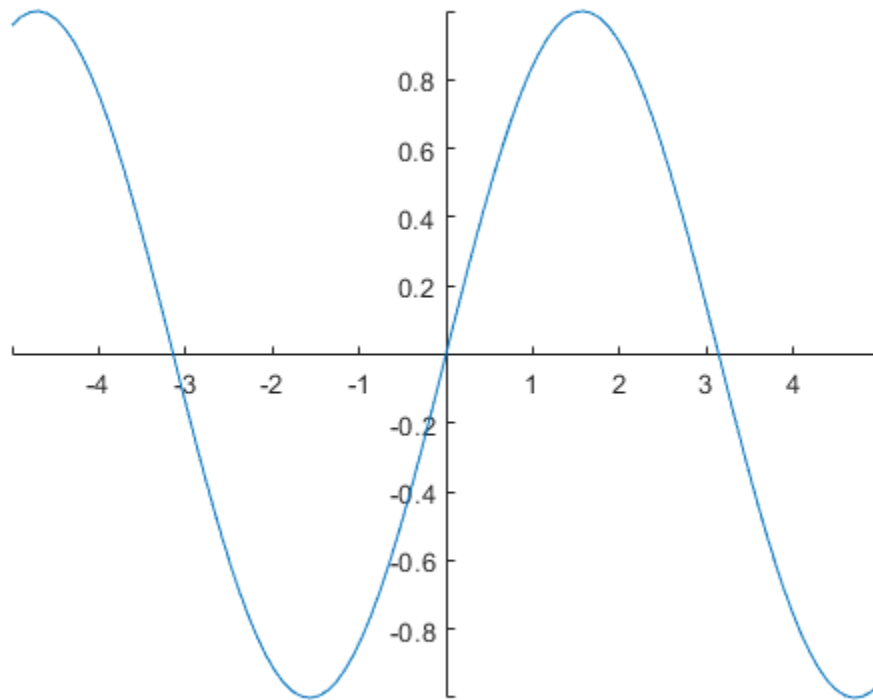
See Also

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

Properties
Axes

Related Examples

- "Specify Axis Tick Values and Labels" on page 9-9
- "Add Grid Lines and Edit Placement" on page 9-16
- "Add Title and Axis Labels to Chart" on page 8-2

Specify Axis Tick Values and Labels

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)
```
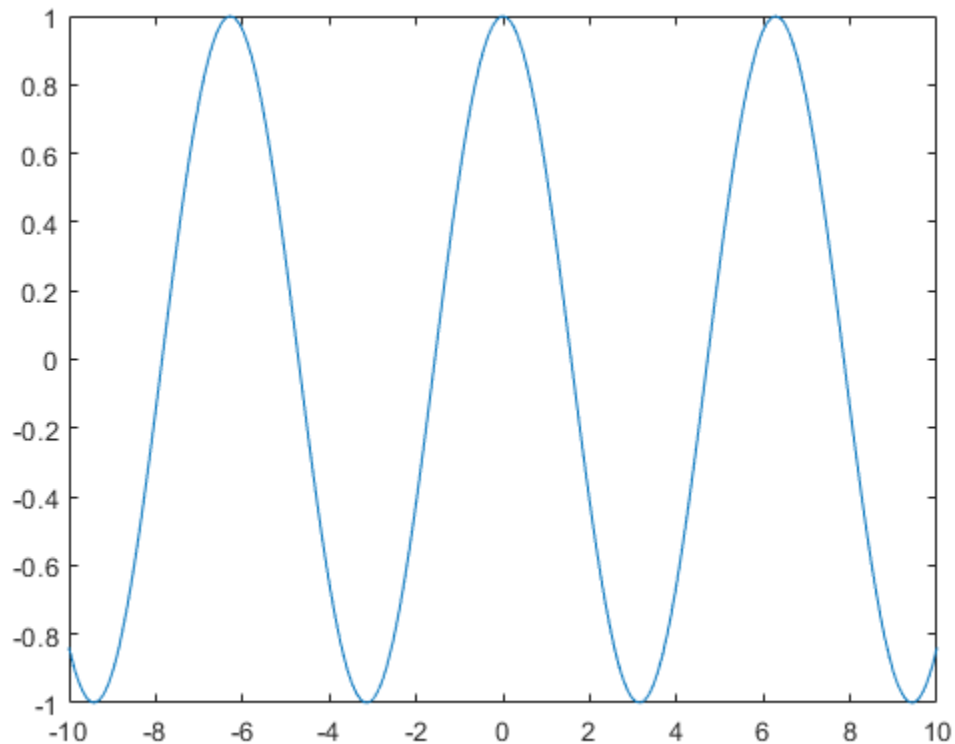

Change the tick value locations along the *x*-axis and *y*-axis. Specify the locations as a vector of increasing values. The values do not need to be evenly spaced.

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])
```
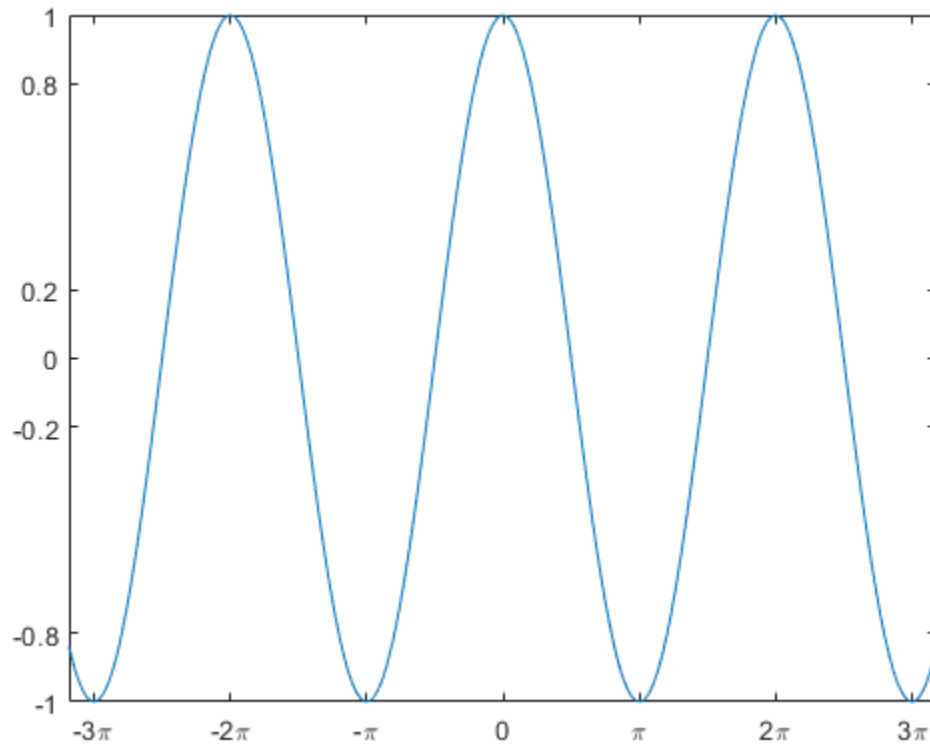
For releases prior to R2016b, instead set the tick values and labels using the `XTick`, `XTickLabel`, `YTick`, and `YTickLabel` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTick` property using dot notation, such as `ax.XTick = [-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)
```
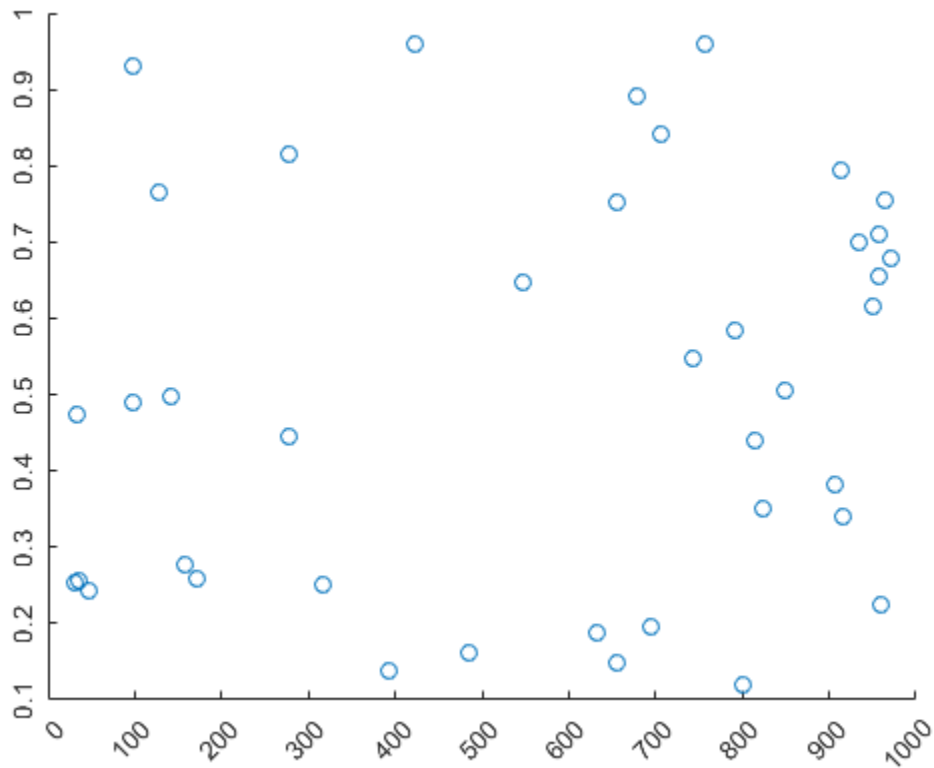
For releases prior to R2016b, specify the rotation using the `XTickLabelRotation` and `YTickLabelRotation` properties of the `Axes` object. For example, assign the `Axes` object to a variable, such as `ax = gca`. Then set the `XTickLabelRotation` property using dot notation, such as `ax.XTickLabelRotation = 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')
```
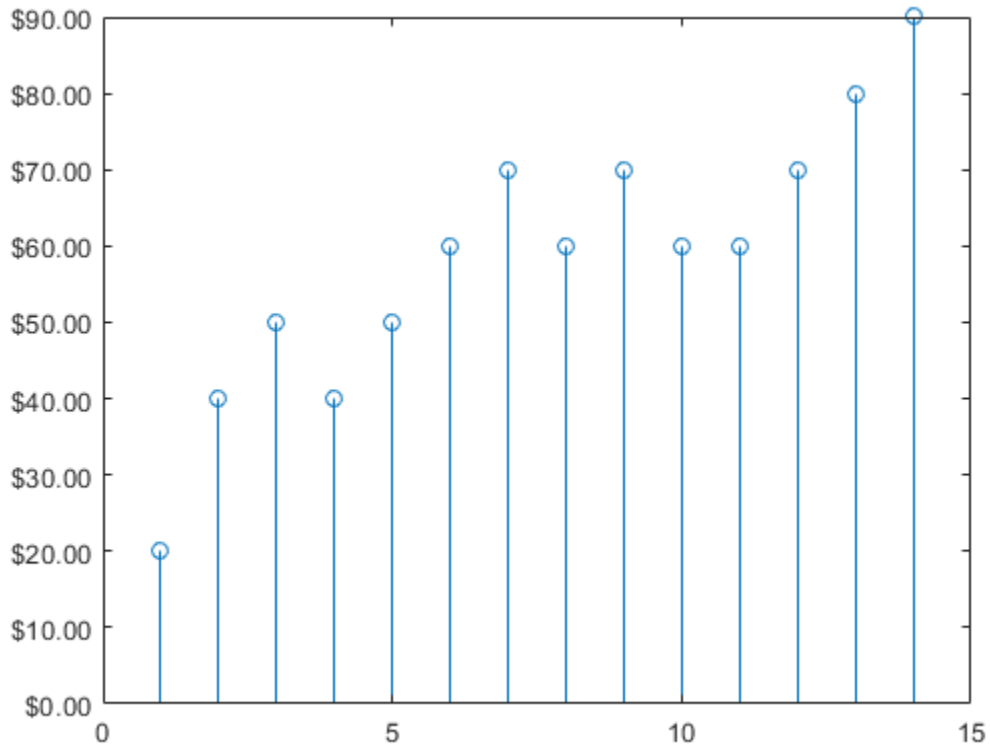
For more control over the formatting, specify a custom format. For example, show one decimal value in the x-axis tick labels using `'%.1f'`. Display the y-axis tick labels as British Pounds using `'\xA3%.2f'`. The option `\xA3` indicates the Unicode character for the Pound symbol. For more information on specifying a custom format, see the `xtickformat` function.

```
xtickformat('%.1f')
ytickformat('\xA3%.2f')
```
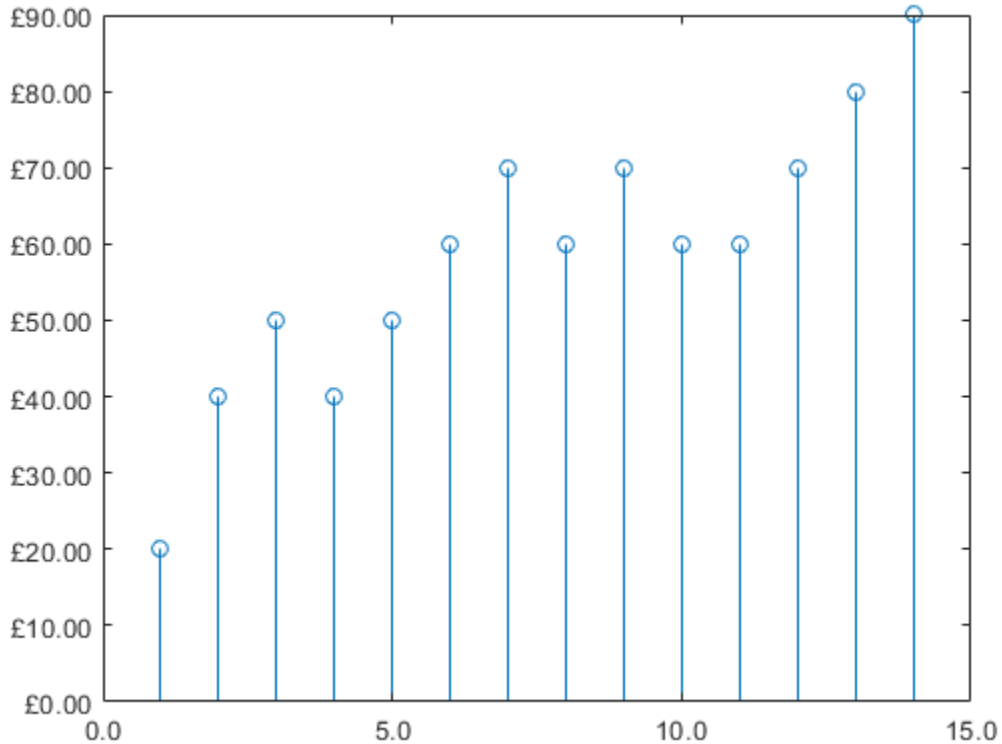
Ruler Objects for Individual Axis Control

MATLAB creates a ruler object for each axis. Like all graphics objects, ruler objects have properties that you can view and modify. Ruler objects allow for more individual control over the formatting of the *x*-axis, *y*-axis, or *z*-axis. Access the ruler object associated with a particular axis through the XAxis, YAxis, or ZAxis property of the Axes object. The type of ruler depends on the type of data along the axis. For numeric data, MATLAB creates a NumericRuler object.

```
ax = gca;
ax.XAxis

ans =
  NumericRuler with properties:

             Limits: [0 15]
              Scale: 'linear'
           Exponent: 0
         TickValues: [0 5 10 15]
    TickLabelFormat: '%.1f'

  Show all properties
```

Control Value in Exponent Label Using Ruler Objects

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. 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.

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

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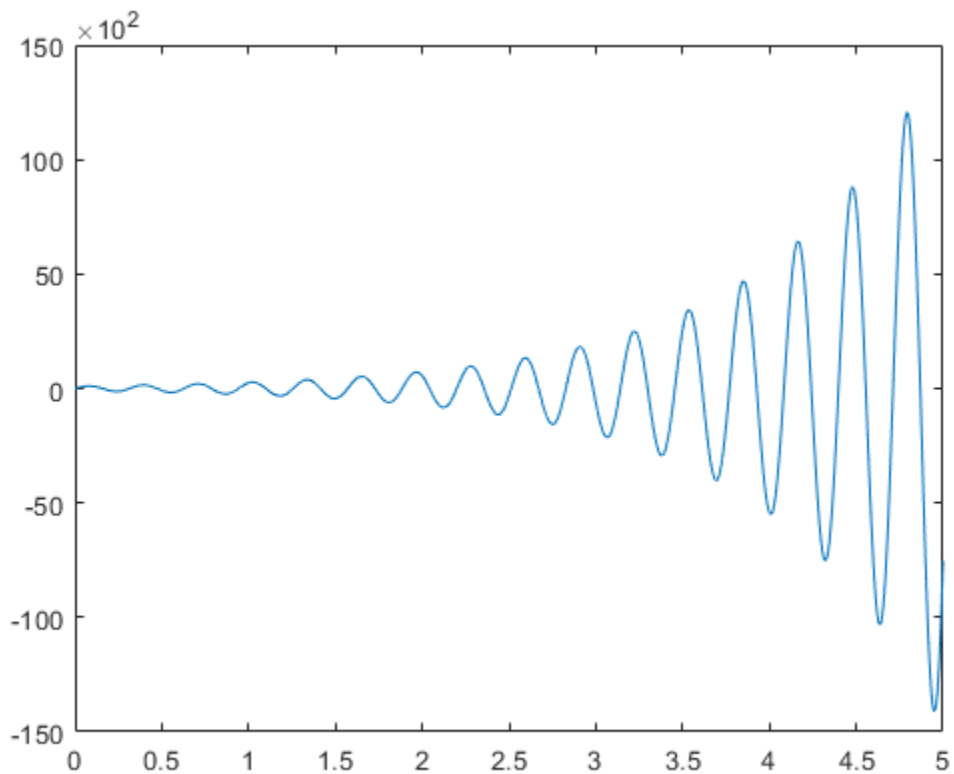

Change the exponent value to 0 so that the tick labels do not use exponential notation.

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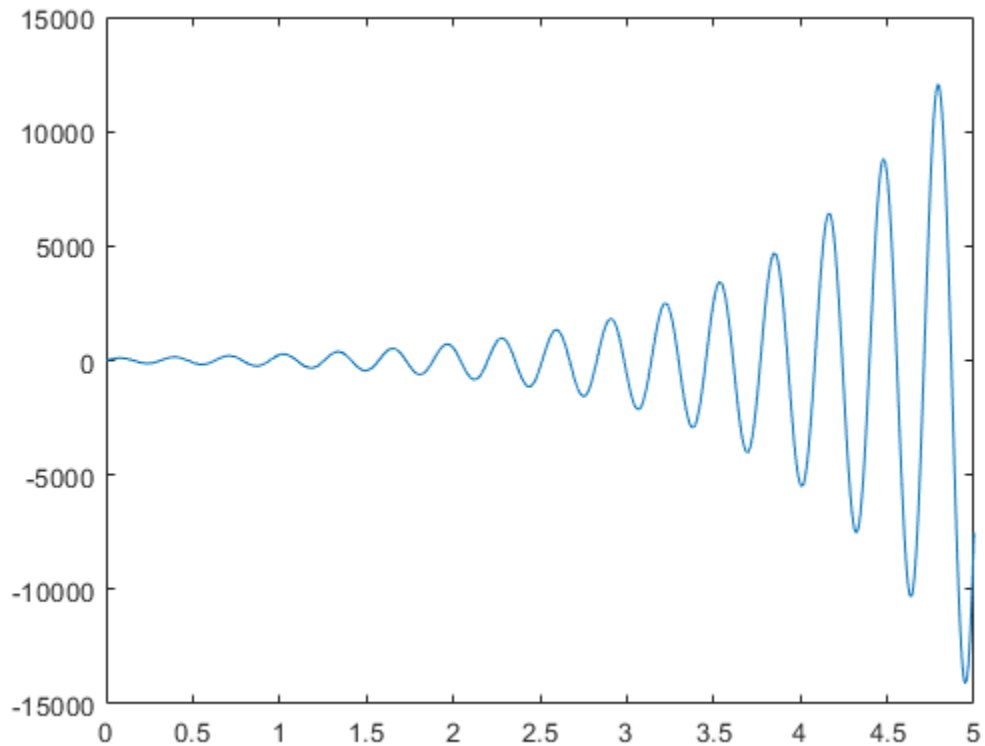
See Also

Functions
xlim | xtickangle | xtickformat | xticks | yticks | zticks

Properties
Axes | NumericRuler

Related Examples

- "Add Grid Lines and Edit Placement" on page 9-16
- "Specify Axis Limits" on page 9-2
- "Add Title and Axis Labels to Chart" on page 8-2

Add Grid Lines and Edit Placement

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
```
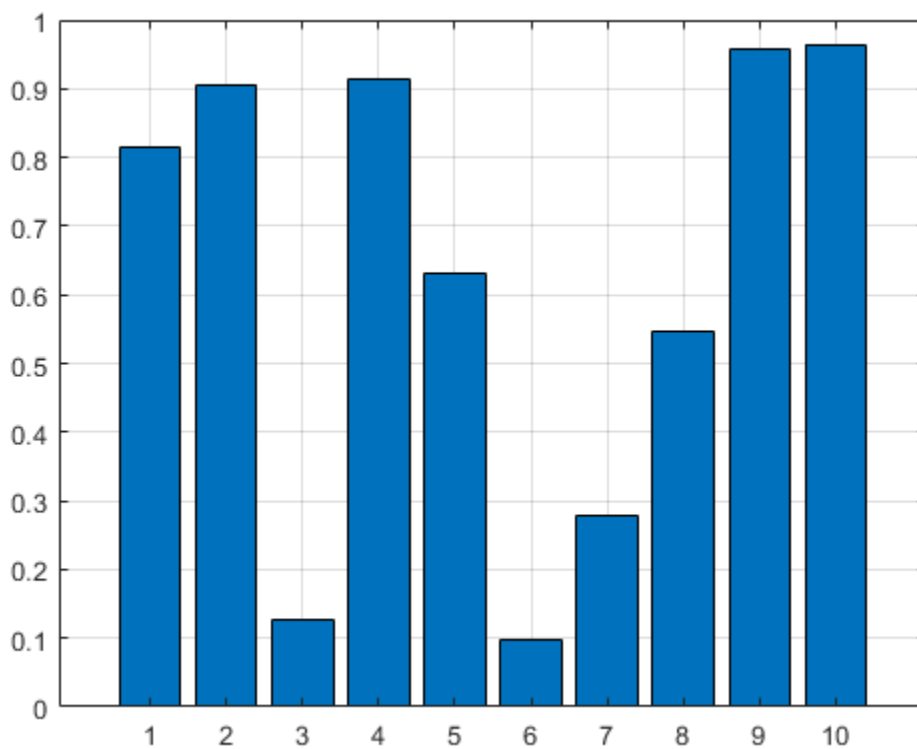

Add minor grid lines between the tick marks.

```
grid minor
```
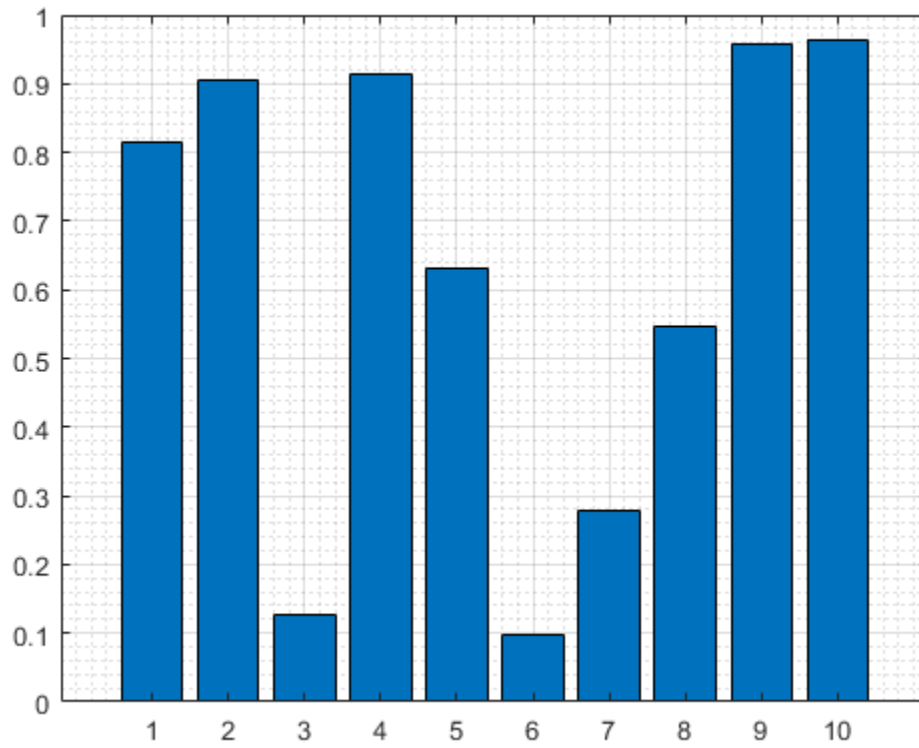
Turn off all the grid lines.

```
grid off
```
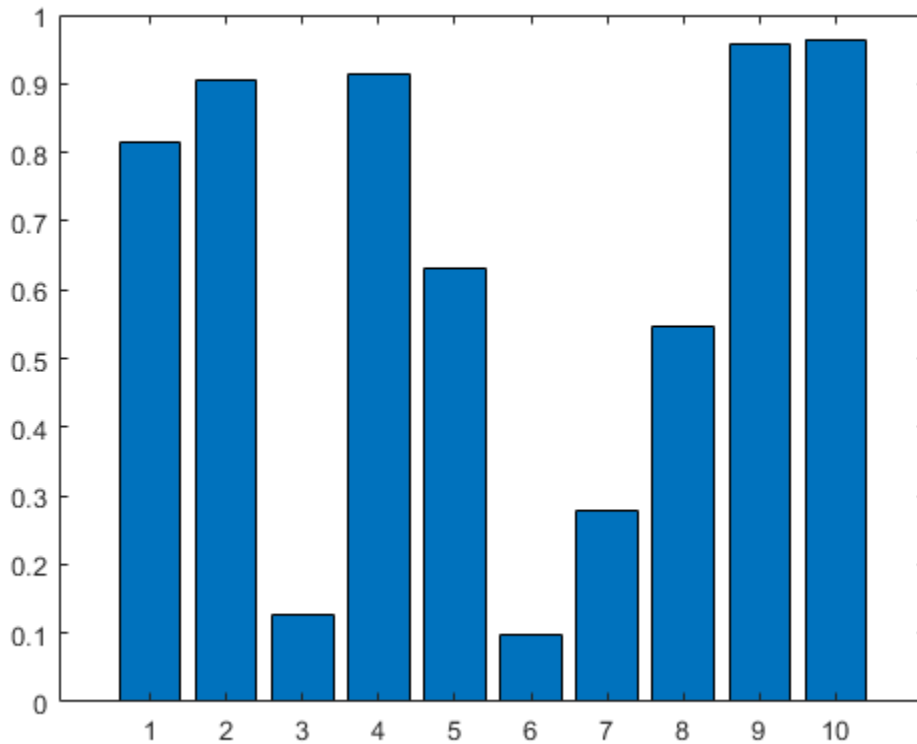
Display Grid Lines in Specific Direction

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.

```matlab
y = rand(10,1);
bar(y)
ax = gca;
ax.XGrid = 'off';
ax.YGrid = 'on';
```
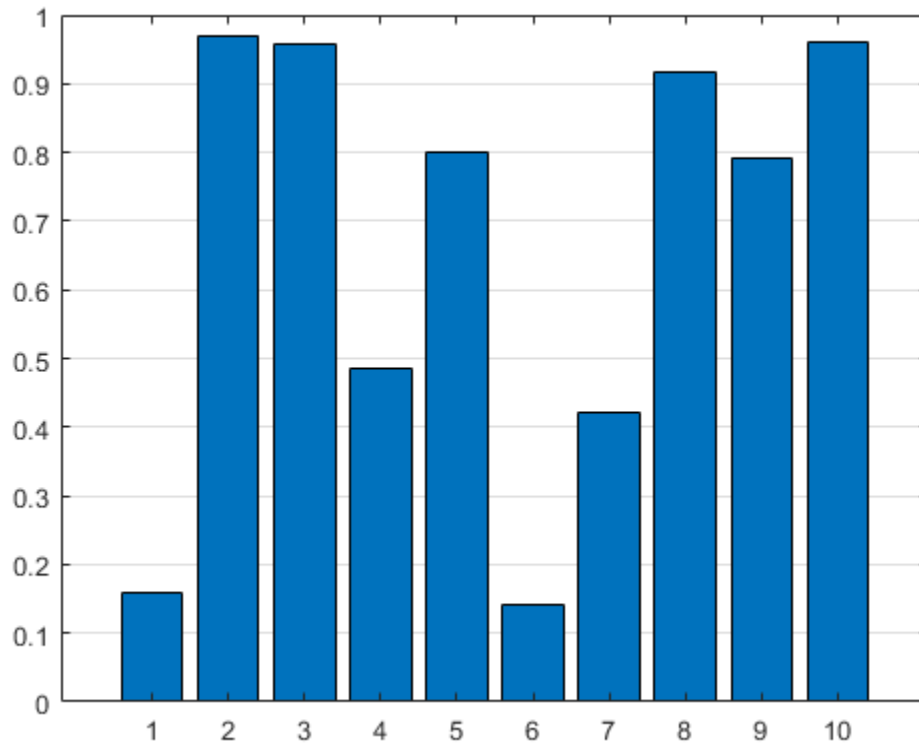
Create a 3-D plot and display the grid lines only in the z direction. Use the box on command to show the box outline around the axes.

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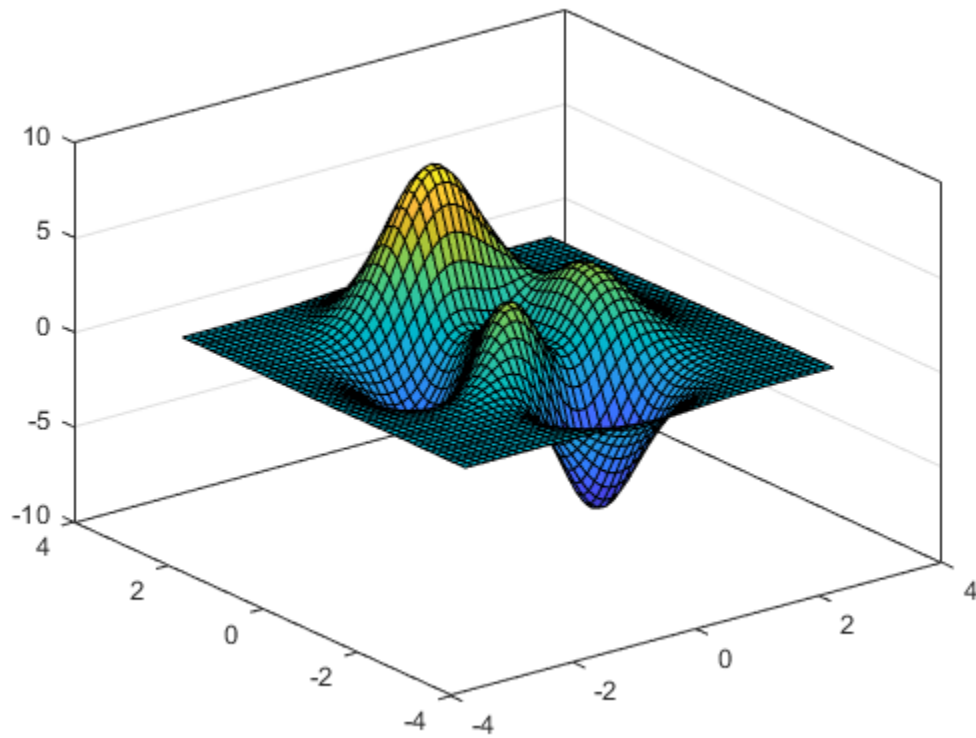
Edit Grid Line Placement

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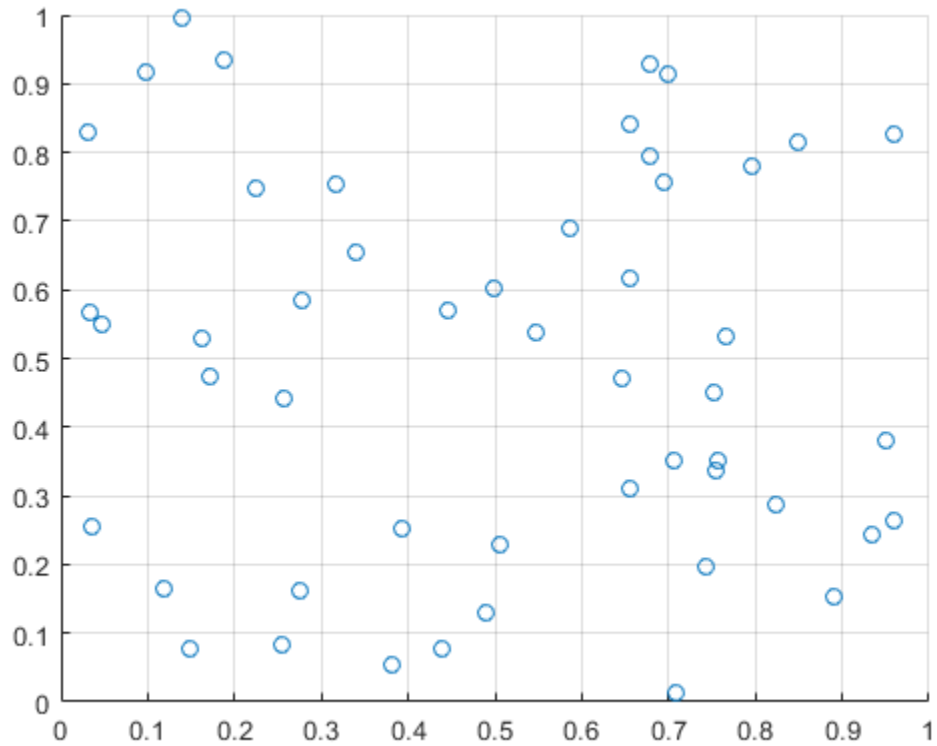
Grid lines appear at the tick mark locations. Edit the placement of the grid lines by changing the tick mark locations.

```
xticks(0:0.2:1)
yticks([0 0.5 0.8 1])
```
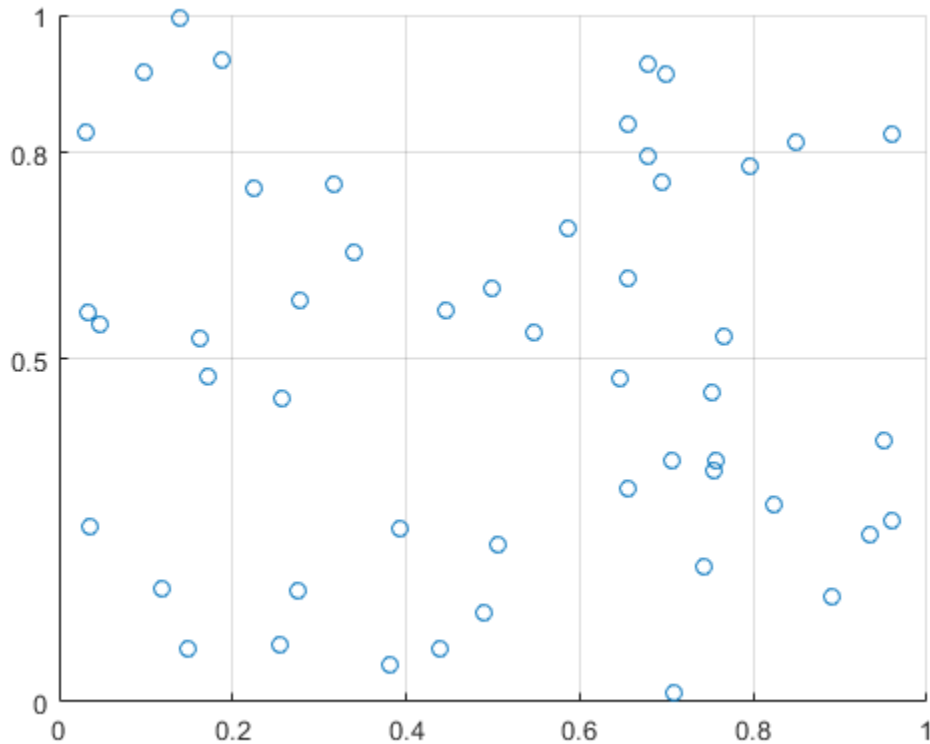
Modify Visual Appearance of Grid Lines

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';
```
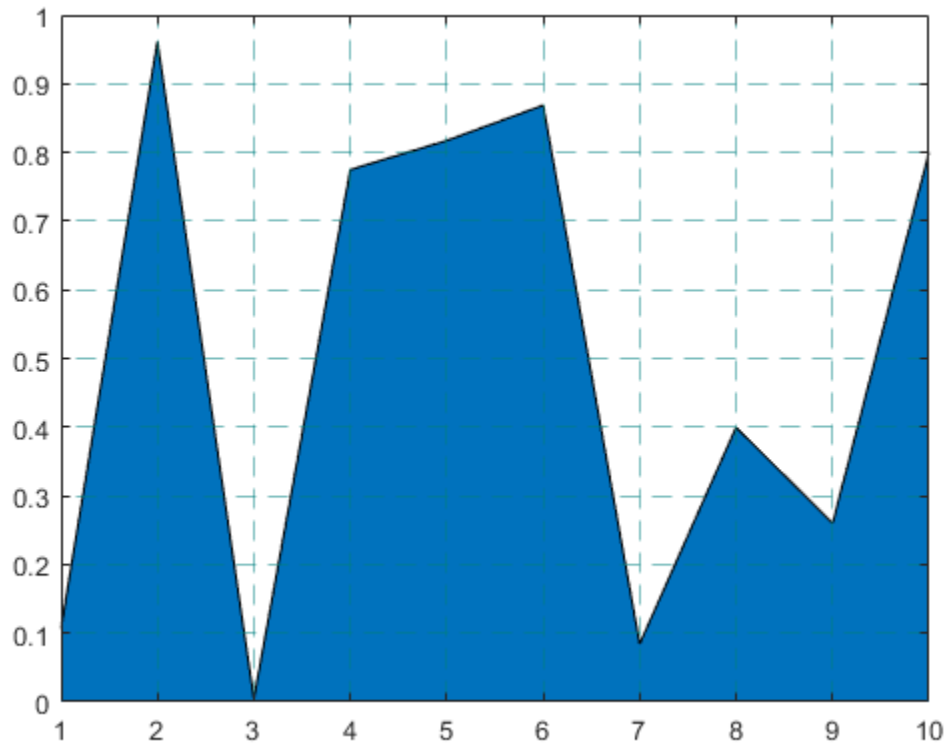
See Also

Functions
grid | xlim | xticks | yticks | zticks

Properties
Axes

Related Examples

- "Specify Axis Tick Values and Labels" on page 9-9
- "Add Title and Axis Labels to Chart" on page 8-2
- "Specify Axis Limits" on page 9-2

Combine Multiple Plots

This example shows how to combine plots in the same axes using the `hold` function, and how to create multiple axes in a figure using the `tiledlayout` function. The `tiledlayout` function is available starting in R2019b. If you are using an earlier release, use the `subplot` function instead.

Combine Plots in Same Axes

By default, new plots clear existing plots and reset axes properties, such as the title. However, you can 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);
plot(x,y1)
title('Combine Plots')

hold on

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

y3 = 2*sin(x);
scatter(x,y3)

hold off
```
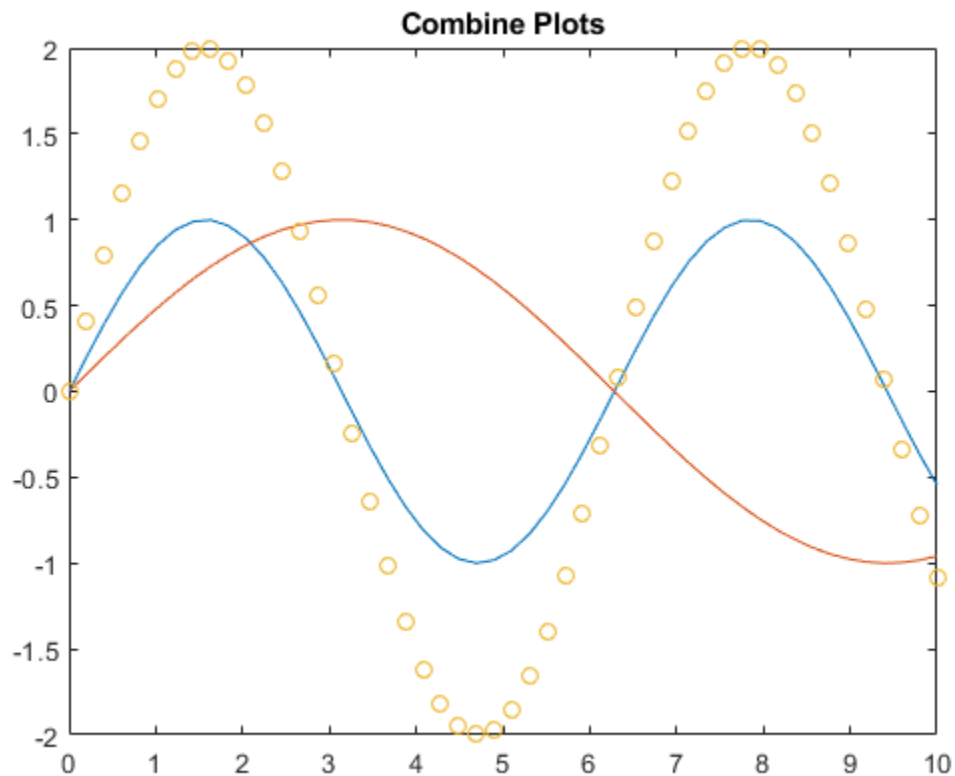
When the hold state is on, new plots do not clear existing plots or reset axes properties, such as the title or axis labels. The plots cycle through colors and line styles based on the `ColorOrder` and `LineStyleOrder` properties of the axes. The axes limits and tick values might adjust to accommodate new data.

Display Multiple Axes in a Figure

You can display multiple axes in a single figure by using the `tiledlayout` function. This function creates a tiled chart layout containing an invisible grid of tiles over the entire figure. Each tile can contain an axes for displaying a plot. After creating a layout, call the `nexttile` function to place an axes object into the layout. Then call a plotting function to plot into the axes. For example, create two plots in a 2-by-1 layout. Add a title to each plot.

Note: This code uses the `tiledlayout` function, which is available starting in R2019b. If you are using an earlier release, use the `subplot` function instead.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);
tiledlayout(2,1) % Requires R2019b or later

% Top plot
nexttile
plot(x,y1)
title('Plot 1')

% Bottom plot
nexttile
scatter(x,y2)
title('Plot 2')
```

Plot 1

Plot 2
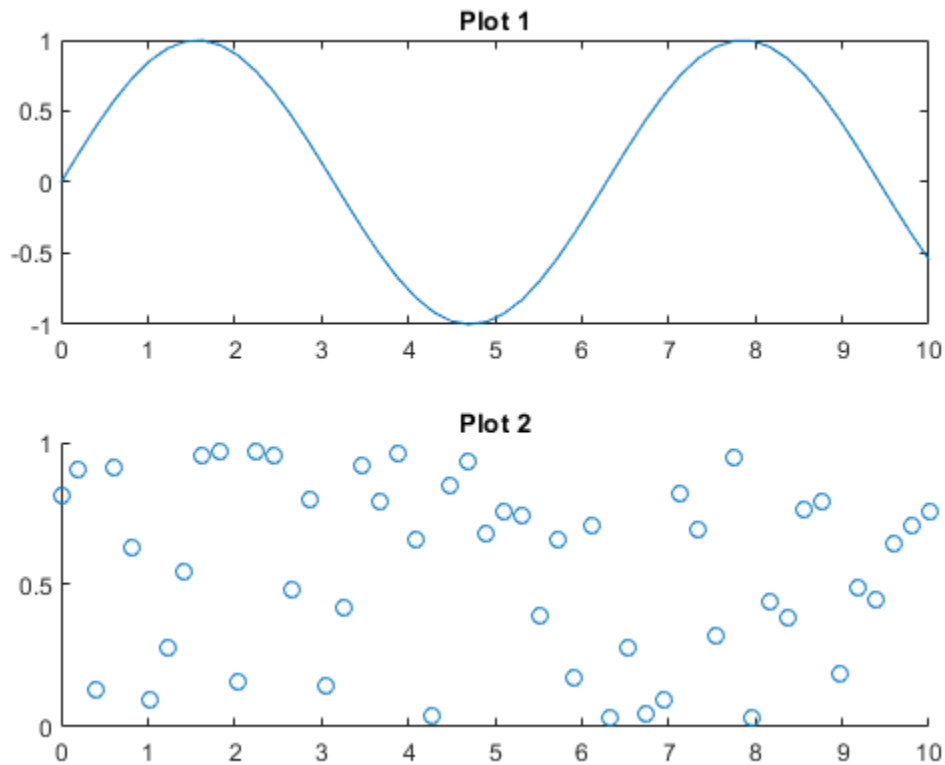
Create Plot Spanning Multiple Rows or Columns

To create a plot that spans multiple rows or columns, specify the span argument when you call nexttile. For example, create a 2-by-2 layout. Plot into the first two tiles. Then create a plot that spans one row and two columns.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);

% Top two plots
tiledlayout(2,2) % Requires R2019b or later
nexttile
plot(x,y1)
nexttile
scatter(x,y2)

% Plot that spans
nexttile([1 2])
y2 = rand(50,1);
plot(x,y2)
```
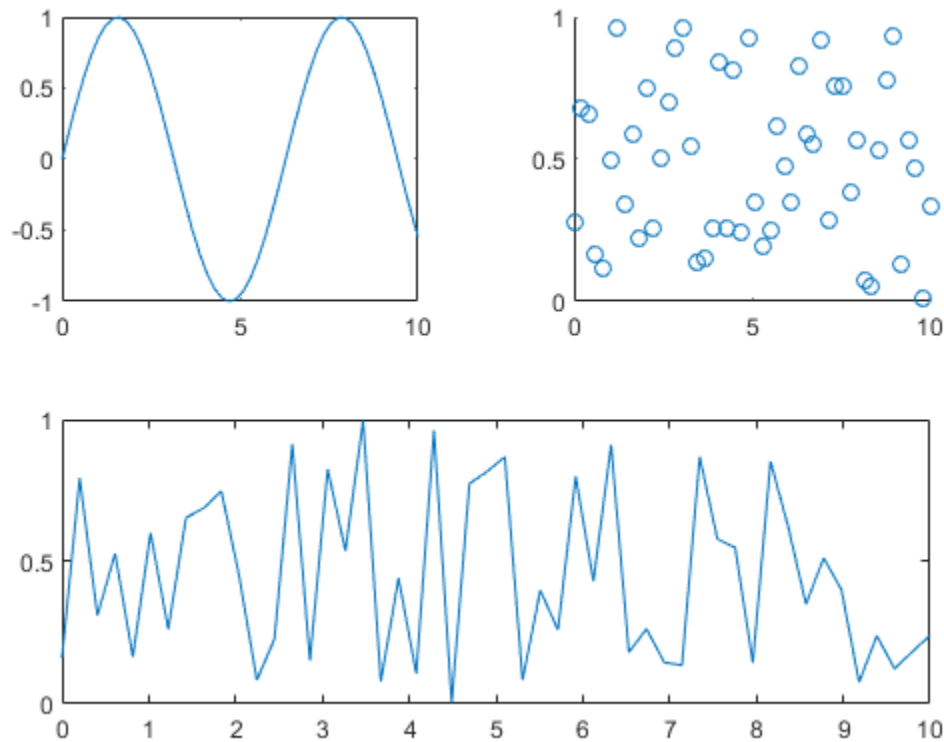
Modify Axes Appearance

Modify the axes appearance by setting properties on each of the axes objects. You can get the axes object by calling the `nexttile` function with an output argument. You also can specify the axes object as the first input argument to a graphics function to ensure that the function targets the correct axes.

For example, create two plots and assign the axes objects to the variables `ax1` and `ax2`. Change the axes font size and *x*-axis color for the first plot. Add grid lines to the second plot.

```
x = linspace(0,10,50);
y1 = sin(x);
y2 = rand(50,1);
tiledlayout(2,1) % Requires R2019b or later

% Top plot
ax1 = nexttile;
plot(ax1,x,y1)
title(ax1,'Plot 1')
ax1.FontSize = 14;
ax1.XColor = 'red';

% Bottom plot
ax2 = nexttile;
scatter(ax2,x,y2)
title(ax2,'Plot 2')
grid(ax2,'on')
```
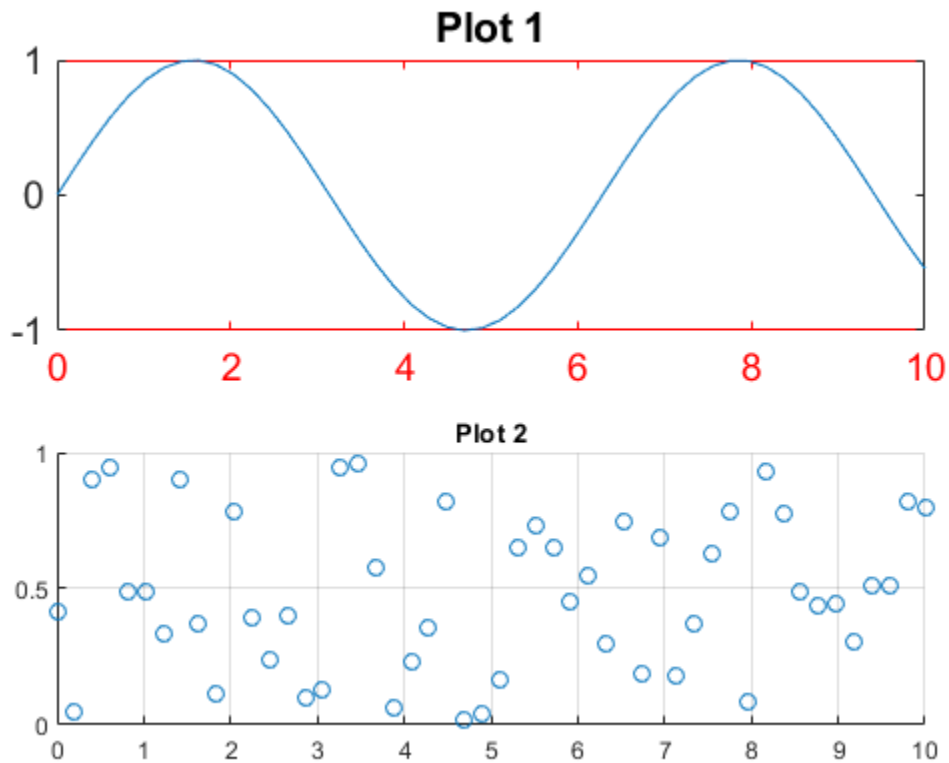
Control Spacing Around the Tiles

You can control the spacing around the tiles in a layout by specifying the `Padding` and `TileSpacing` properties. For example, display four plots in a 2-by-2 layout.

```
x = linspace(0,30);
y1 = sin(x);
y2 = sin(x/2);
y3 = sin(x/3);
y4 = sin(x/4);

% Create plots
t = tiledlayout(2,2); % Requires R2019b or later
nexttile
plot(x,y1)
nexttile
plot(x,y2)
nexttile
plot(x,y3)
nexttile
plot(x,y4)
```
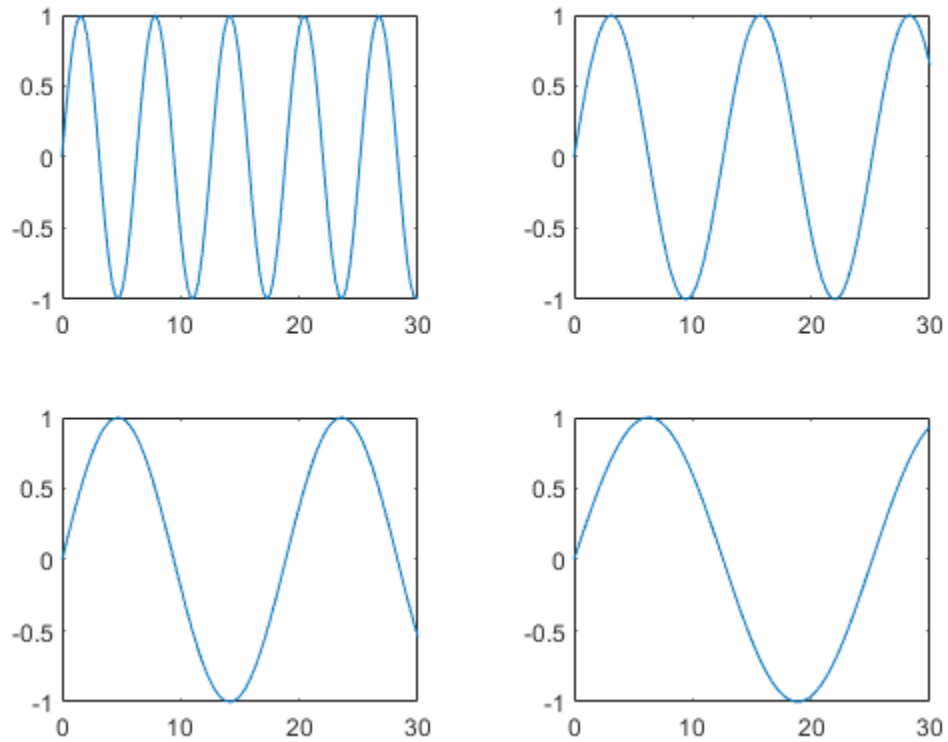
Reduce the spacing around the perimeter of the layout and around each tile by setting the `Padding` and `TileSpacing` properties to `'compact'`.

```
t.Padding = 'compact';
t.TileSpacing = 'compact';
```
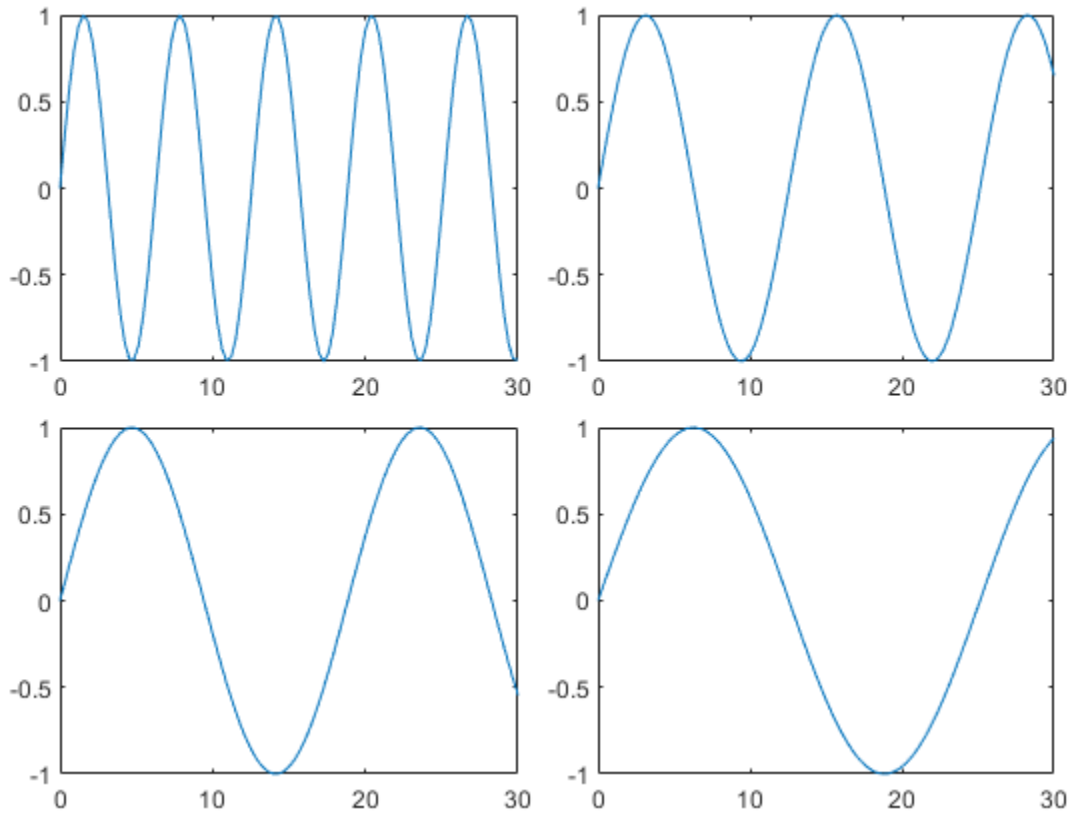
Display Shared Title and Axis Labels

You can display a shared title and shared axis labels in a layout. Create a 2-by-1 layout t. Then display a line plot and a stem plot. Synchronize the *x*-axis limits by calling the linkaxes function.

```
x1 = linspace(0,20,100);
y1 = sin(x1);
x2 = 3:17;
y2 = rand(1,15);

% Create plots.
t = tiledlayout(2,1); % Requires R2019b or later
ax1 = nexttile;
plot(ax1,x1,y1)
ax2 = nexttile;
stem(ax2,x2,y2)

% Link the axes
linkaxes([ax1,ax2],'x');
```
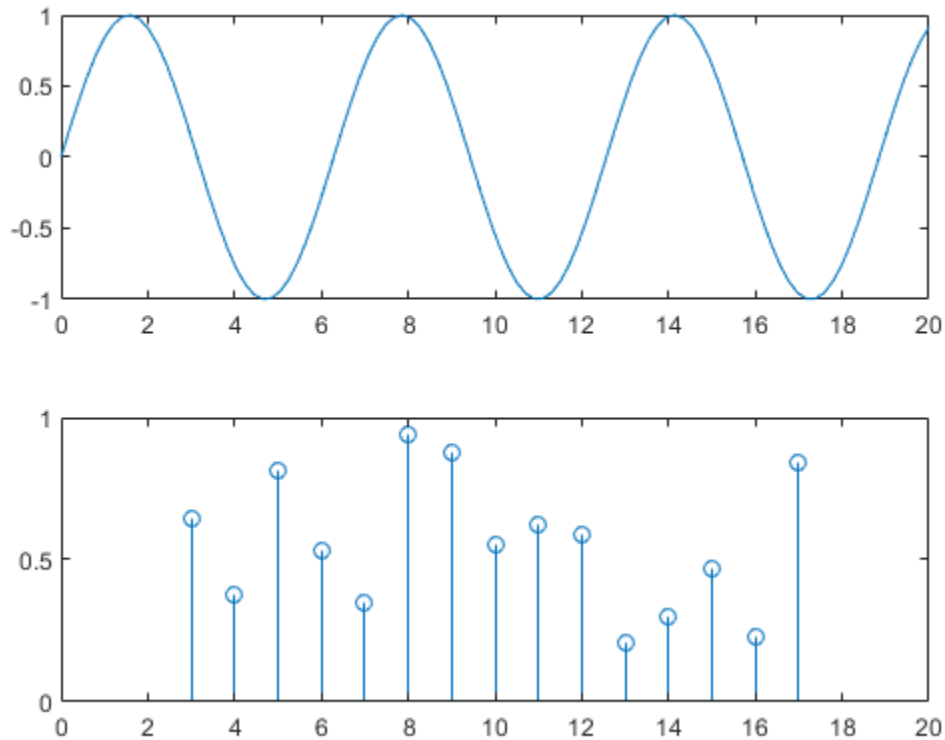
Add a shared title and shared axis labels by passing `t` to the `title`, `xlabel`, and `ylabel` functions. Move the plots closer together by removing the *x*-axis tick labels from the top plot and setting the `TileSpacing` property of `t` to `'compact'`.

```
% Add shared title and axis labels
title(t,'My Title')
xlabel(t,'x-values')
ylabel(t,'y-values')

% Move plots closer together
xticklabels(ax1,{})
t.TileSpacing = 'compact';
```

My Title
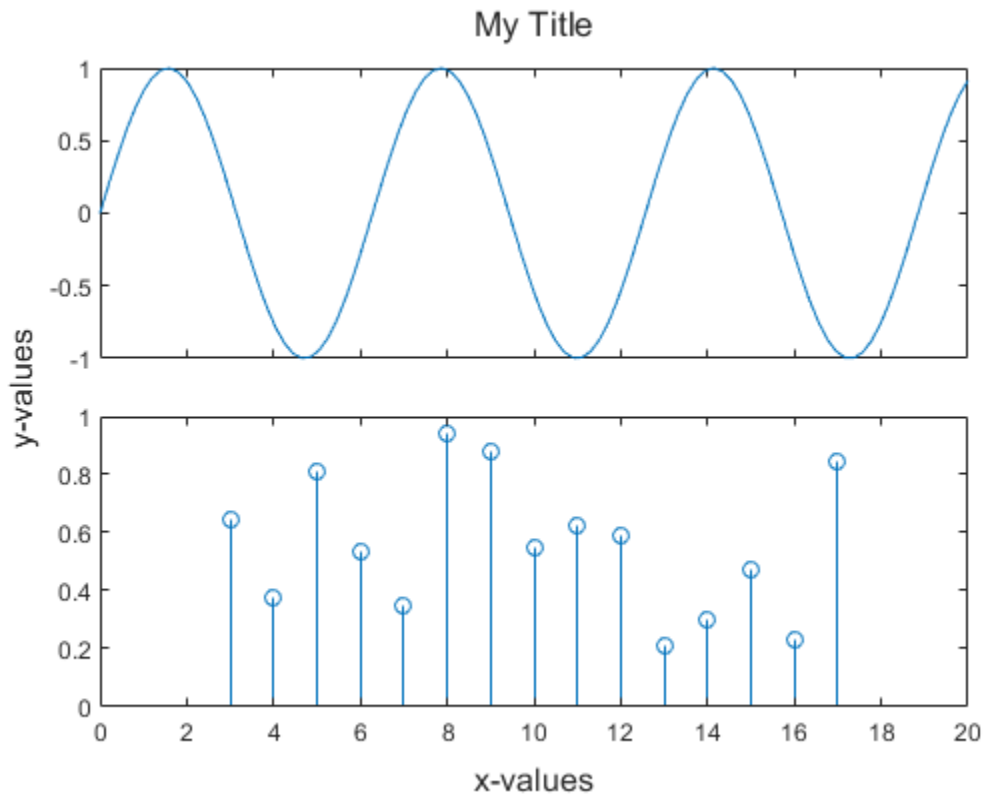
See Also

Functions
hold | nexttile | tiledlayout | title

Related Examples

- "Create Chart with Two y-Axes" on page 9-42
- "Specify Axis Tick Values and Labels" on page 9-9

Customized Presentations and Special Effects with Tiled Chart Layouts

A tiled chart layout lets you control the placement of elements in a visualization and create special effects. This topic covers four examples:

- "Stacked Colorbar and Plot with Shared Title" on page 9-33
- "Irregular Grid of Plots" on page 9-34
- "Main Plot with Adjacent Smaller Plots" on page 9-36
- "Region-of-Interest Plot" on page 9-38

Stacked Colorbar and Plot with Shared Title

Tiled chart layouts have several features for presenting multiple elements together under a shared title. In this case, stack a colorbar on top of a plot under a shared title.

Create a 1-by-1 tiled chart layout. Then display a contour plot of the `peaks` data set.

```
figure
t = tiledlayout(1,1);
nexttile
contourf(peaks)
```
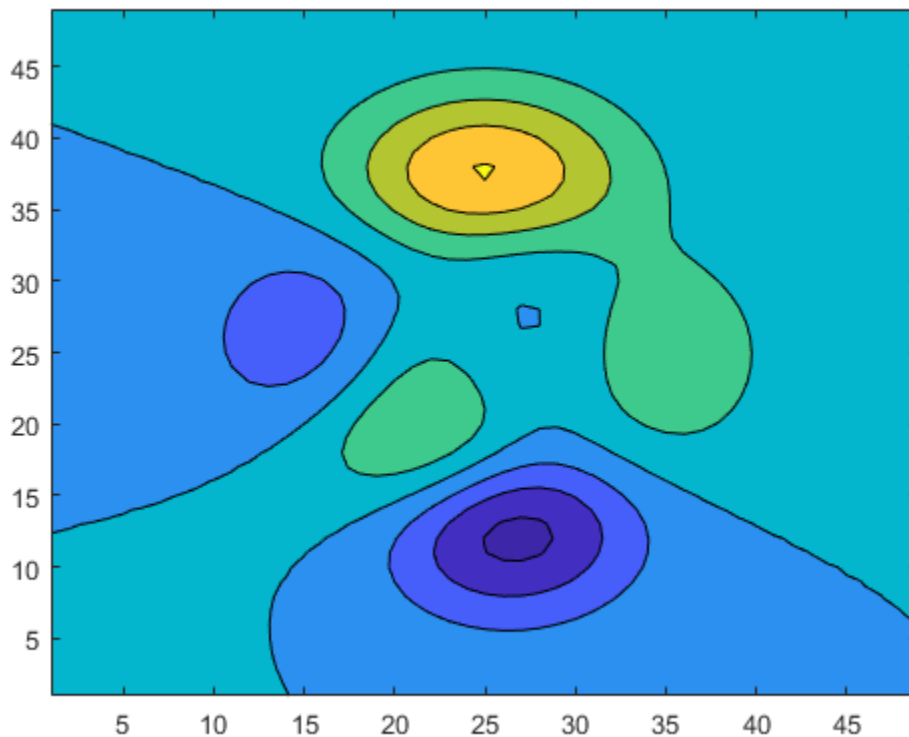
Create a colorbar, and specify the `'northoutside'` location. Then add a shared title by passing the layout object, `t`, to the `title` function.

```
cb = colorbar('Location','northoutside');
title(t,'Contours of Peaks')
```
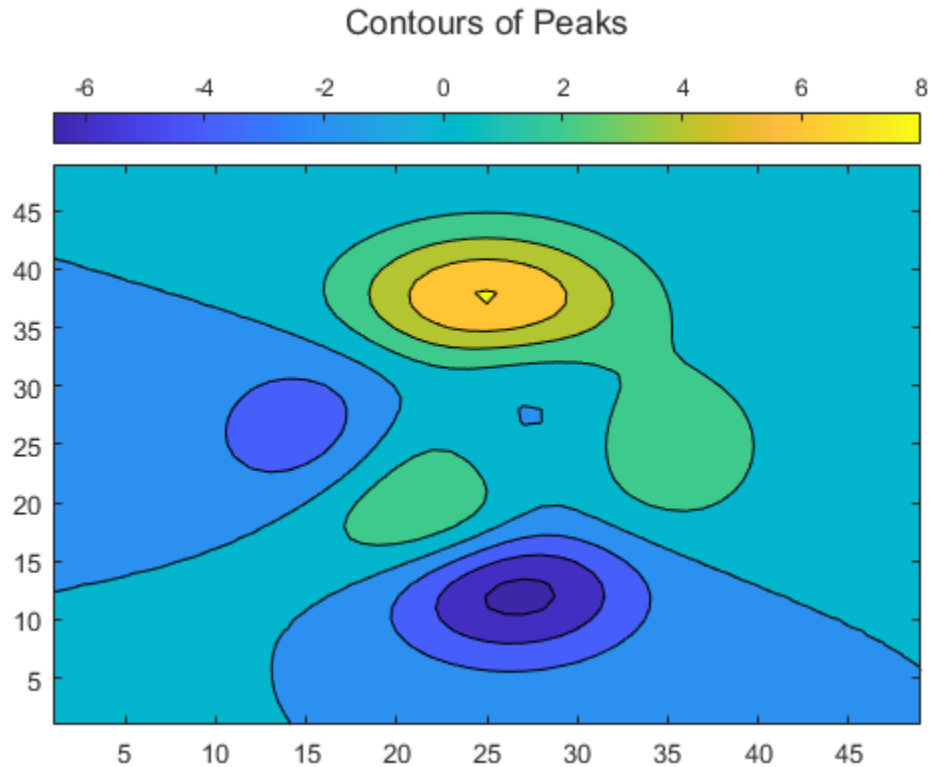

Irregular Grid of Plots

Tiled chart layouts can be nested. This is useful for varying the arrangement of the tiles in a layout. In this case, create two columns that each have a different number of tiles.

Create a 1-by-2 tiled layout called `t1`. Then create two nested layouts, `t2` and `t3`, where `t2` is in the first tile of `t1`, and `t3` is in the second tile. Then display two contour plots in `t2`, and add a title above the plots.

```
figure
t1 = tiledlayout(1,2,'TileSpacing','Compact');
t2 = tiledlayout(t1,'flow','TileSpacing','Compact');
t3 = tiledlayout(t1,'flow','TileSpacing','Compact');
t3.Layout.Tile = 2;

% Add two contour plots to left side
[X,Y,Z] = peaks;
nexttile(t2);
contourf(X,Y,Z)
nexttile(t2)
```

```
contourf(X,Y,Z,10)
title(t2,"Contours of Peaks")
```


Contours of Peaks
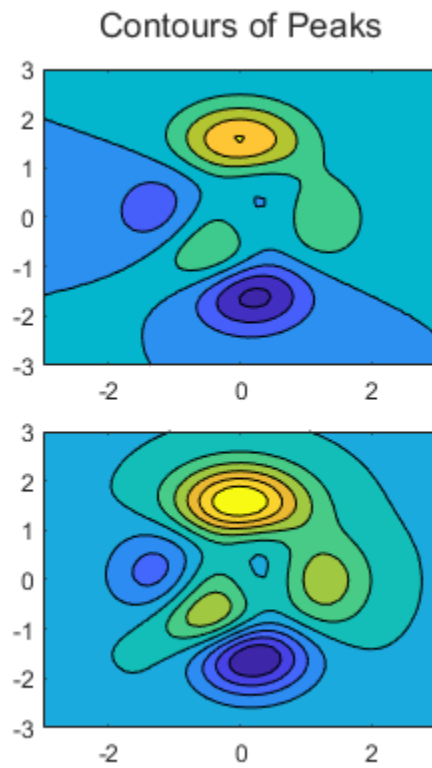
Display three line plots within t3, and add a title.

```
nexttile(t3)
plot(Z(15,:))
nexttile(t3)
plot(Z(25,:))
nexttile(t3)
plot(Z(35,:))
title(t3,"Y = -1.25, 0, and 1.25")
```
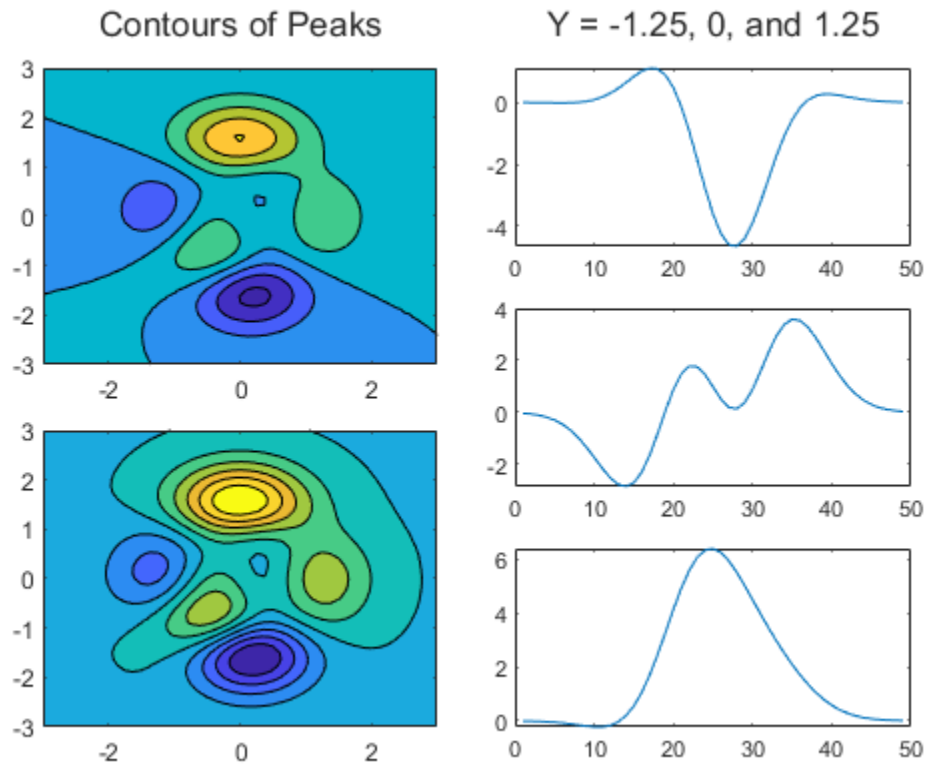
Main Plot with Adjacent Smaller Plots

A tiled chart layout consists of a grid of tiles surrounded by four outer tiles. You can place legends, colorbars, additional axes, or even a nested layout into any of these tiles. In this case, create a main plot in the center, and use one of the outer tiles to display a set of supplemental plots.

First, create a matrix of sine waves and plot them together in a 1-by-1 tiled chart layout.

```
x = (0:0.1:10)';
y = sin([x x+1 x+2 x+3 x+4 x+5]);
figure
t = tiledlayout(1,1);
nexttile
plot(x,y)
```
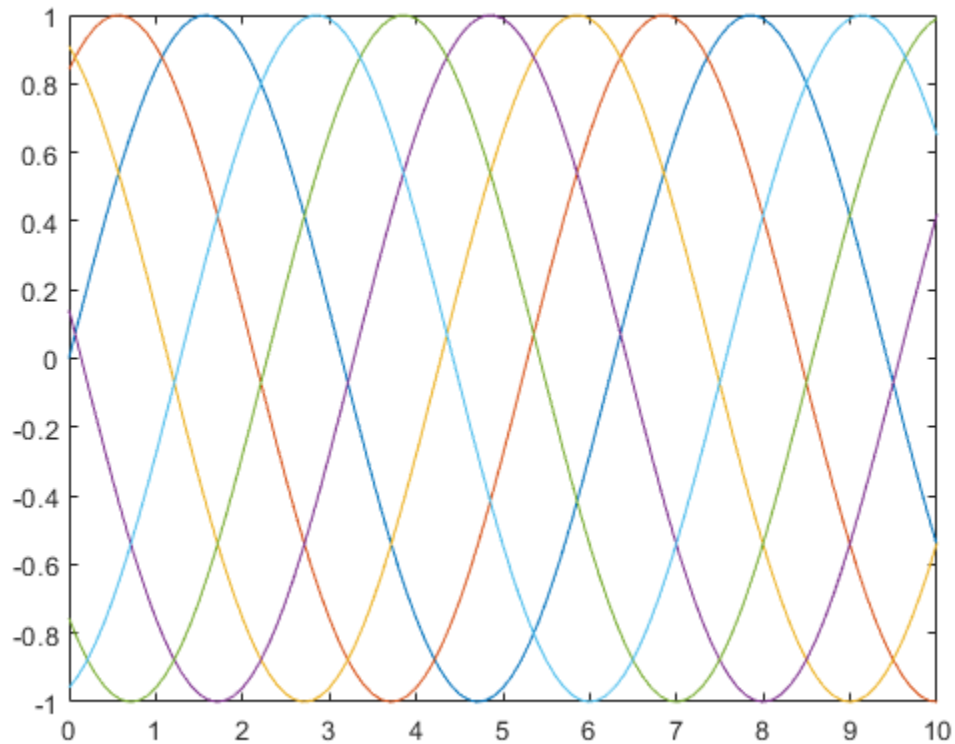
Plot the individual sine waves below the main plot. To do this, create a new layout called `ts` in the south tile of layout `t`. When you create `ts`, specify the `'flow'` tile arrangement so that the plots fill the entire south tile at each iteration of the `for` loop.

```
ts = tiledlayout(t,'flow');
ts.Layout.Tile = 'south';
for i=1:5
    nexttile(ts);
    plot(x,y(:,i))
end
```
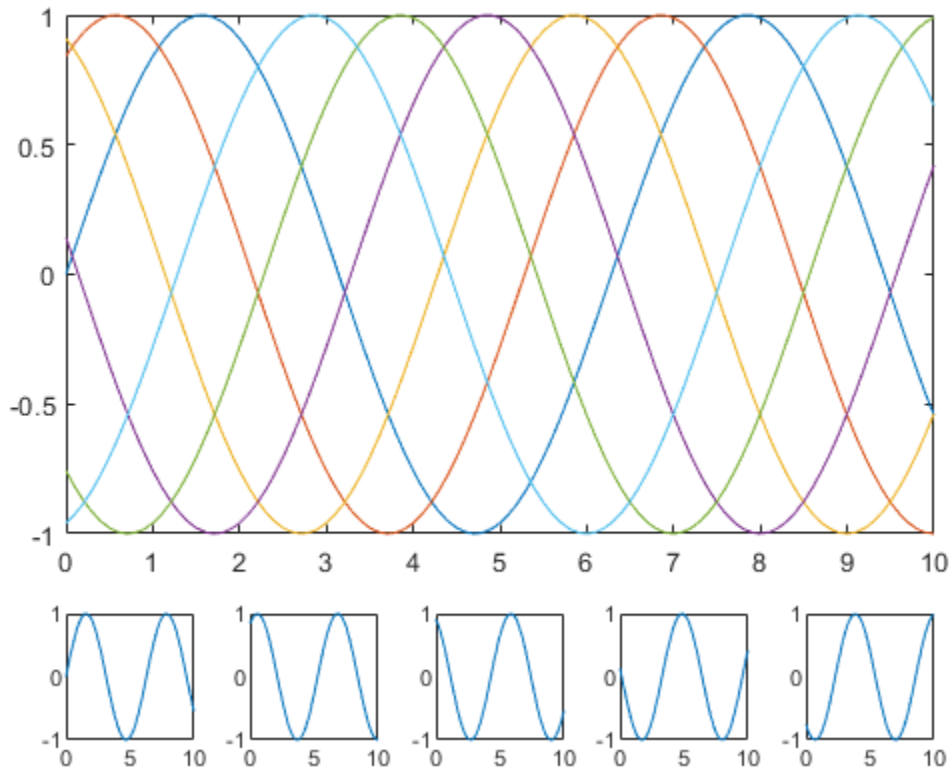
Region-of-Interest Plot

A plot can span multiple tiles of a layout. You can use this feature to display a main plot with an accompanying inset plot. In this case, the inset plot shows a zoomed-in view of a region of interest.

Create a 3-by-3 tiled chart layout, and create an axes that spans all of the tiles. Then display a scatter plot in the axes.

```
figure
t = tiledlayout(3,3,'Padding','compact');
ax1 = nexttile(1,[3 3]);
x = randn(2000,1);
y = randn(2000,1);
scatter(ax1,x,y,1,'filled');
```
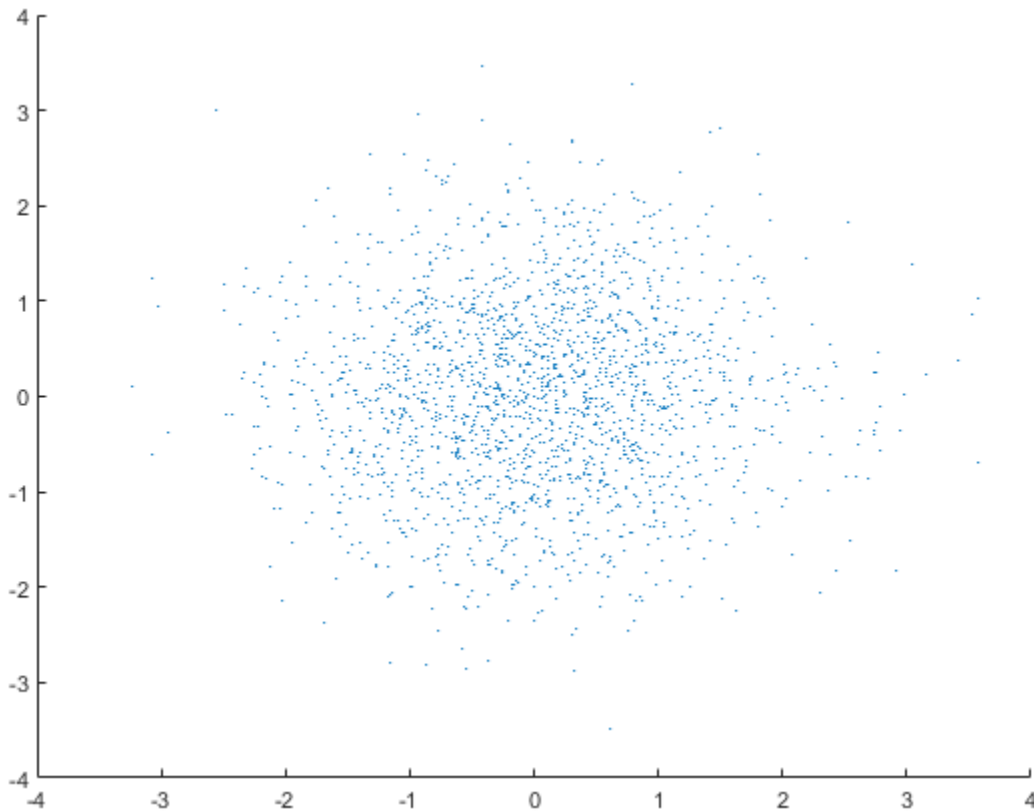
Next, create a rectangle that defines the region of interest in the scatter plot. Set properties on the axes last, to ensure that those properties persist.

- Define the bounds of the rectangle. Define `left` and `bottom` as the left and bottom edges of the rectangle. Specify `left` in *x*-axis units, and specify `bottom` in *y*-axis units. Similarly, define `width` and `height` as the width and height of the rectangle, also in axis units.
- Display the rectangle. Call the `hold` function to preserve the contents of the axes, and then call the `rectangle` function.
- Set properties on the axes. Set the font size to `10` points, set the *x*- and *y*-axis limits, and turn the grid on.

```
% Define bounds of the rectangle
left = -0.5;
bottom = -0.7;
width = 0.4;
height = 0.4;

% Display the rectangle
hold(ax1,'on');
r = rectangle('Position',[left bottom width height], ...
    'EdgeColor','red','LineWidth',1.5);

% Set properties on the axes
ax1.FontSize = 10;
ax1.XLim = [-4.5 4.5];
```

```
ax1.YLim = [-4.5 4.5];
grid(ax1,'on')
```
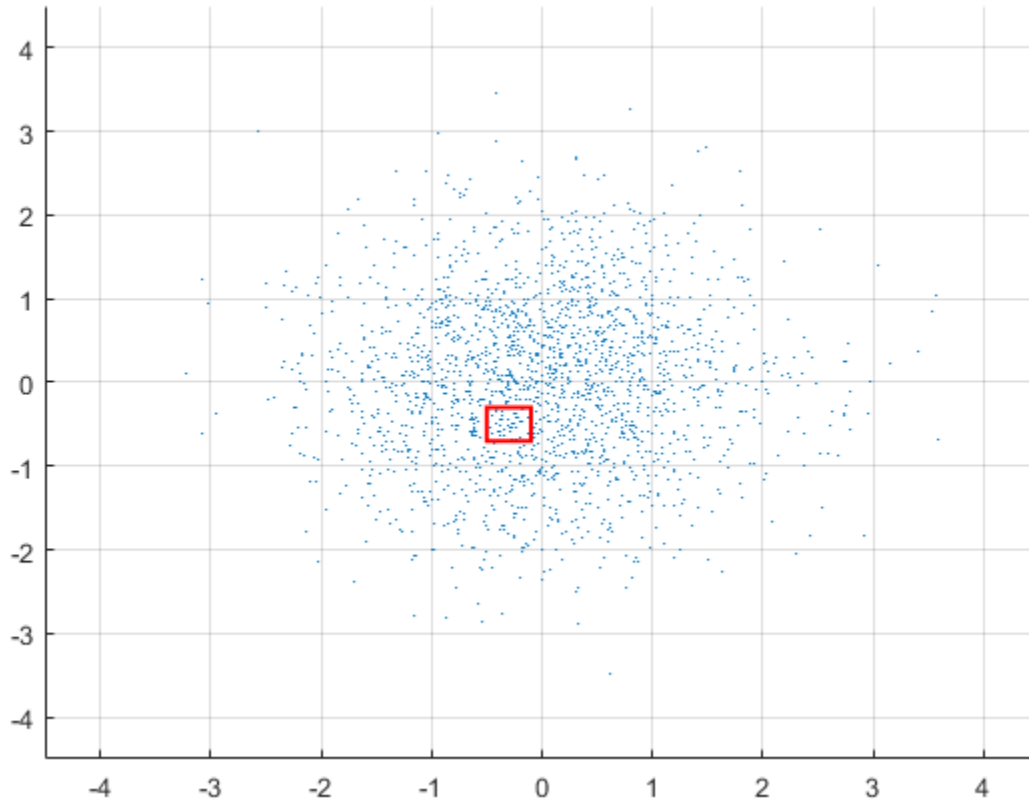

Display a smaller, zoomed-in plot with *x*- and *y*-axis limits that match the region of interest.

- Create the axes for the zoomed-in view. Call the `axes` function to create the axes. Move the axes to the third tile by setting the `Layout.Tile` property. Then plot the entire set of *x* and *y* data.
- Adjust the axis limits to match the region of interest. Set the `XLim` and `YLim` properties of `ax2` to match the region of interest. Then remove the ticks from the plot box.
- Set other properties on the axes. Turn the axes box on, set the *x*- and *y*- axis colors to red, and display a title.

```
% Create axes for zoomed-in view
ax2 = axes(t);
ax2.Layout.Tile = 3;
scatter(ax2,x,y,10,'filled');

% Adjust axis limits and remove ticks
ax2.XLim = [left left+width];
ax2.YLim = [bottom bottom+height];
ax2.XTick = [];
ax2.YTick = [];

% Set other properties on the axes
ax2.Box = 'on';
ax2.XAxis.Color = 'red';
```

```
ax2.YAxis.Color = 'red';
title(ax2,'100x Magnification','Color','red');
```
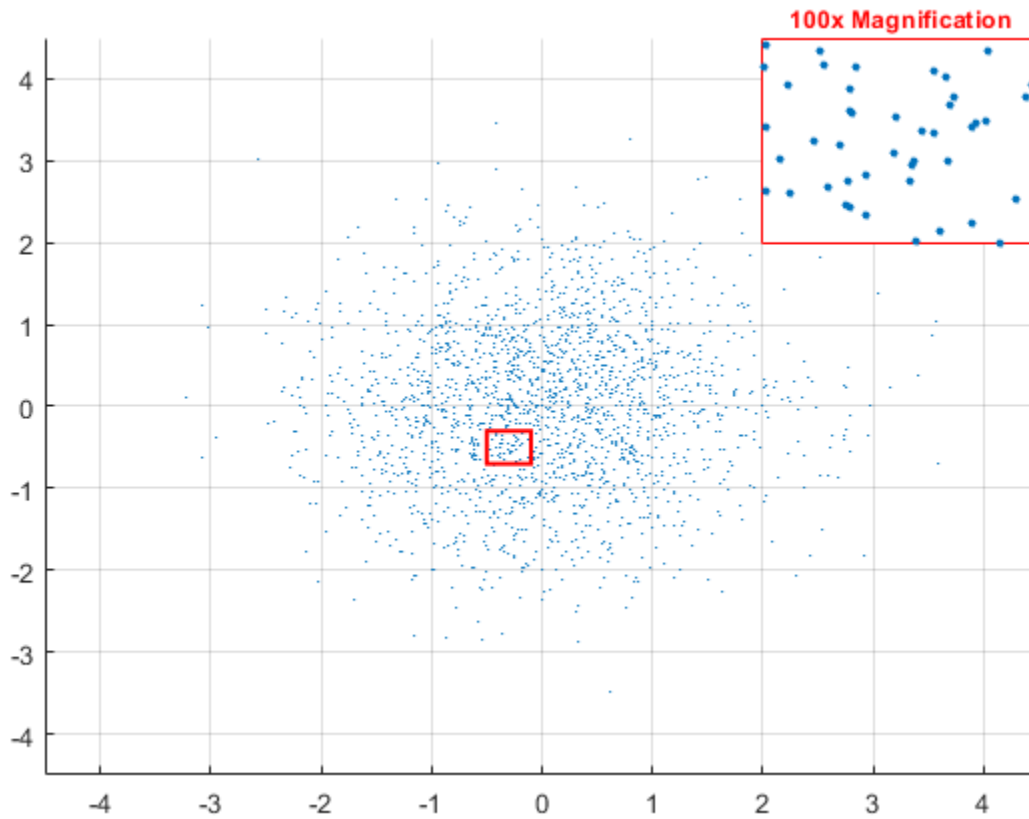

See Also

Functions
axes | colorbar | tiledlayout

Properties
Axes Properties

Related Examples

- "Combine Multiple Plots" on page 9-24

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);
```
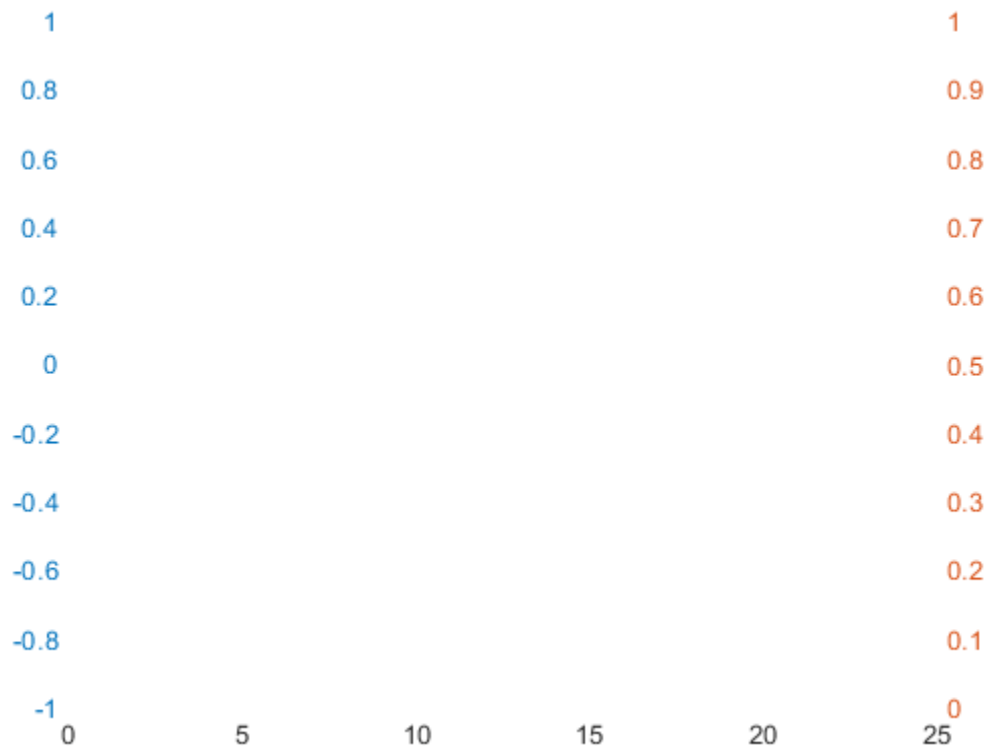

Plot Data Against Right *y*-Axis

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);
```
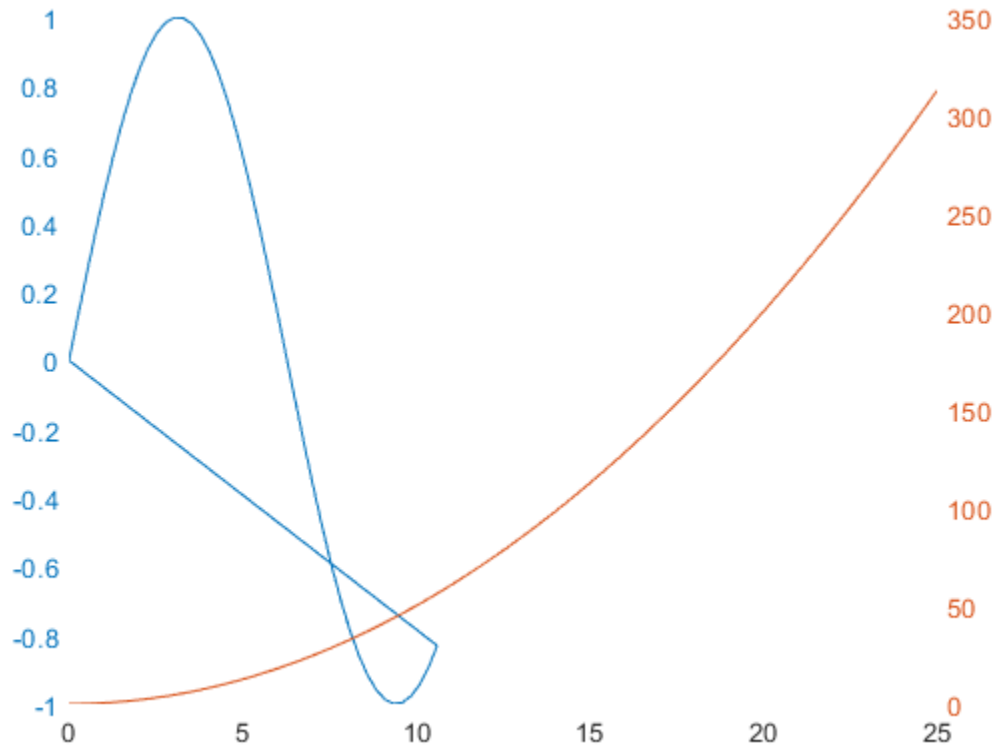
Add Title and Axis Labels

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')
```

Plots with Different y-Scales
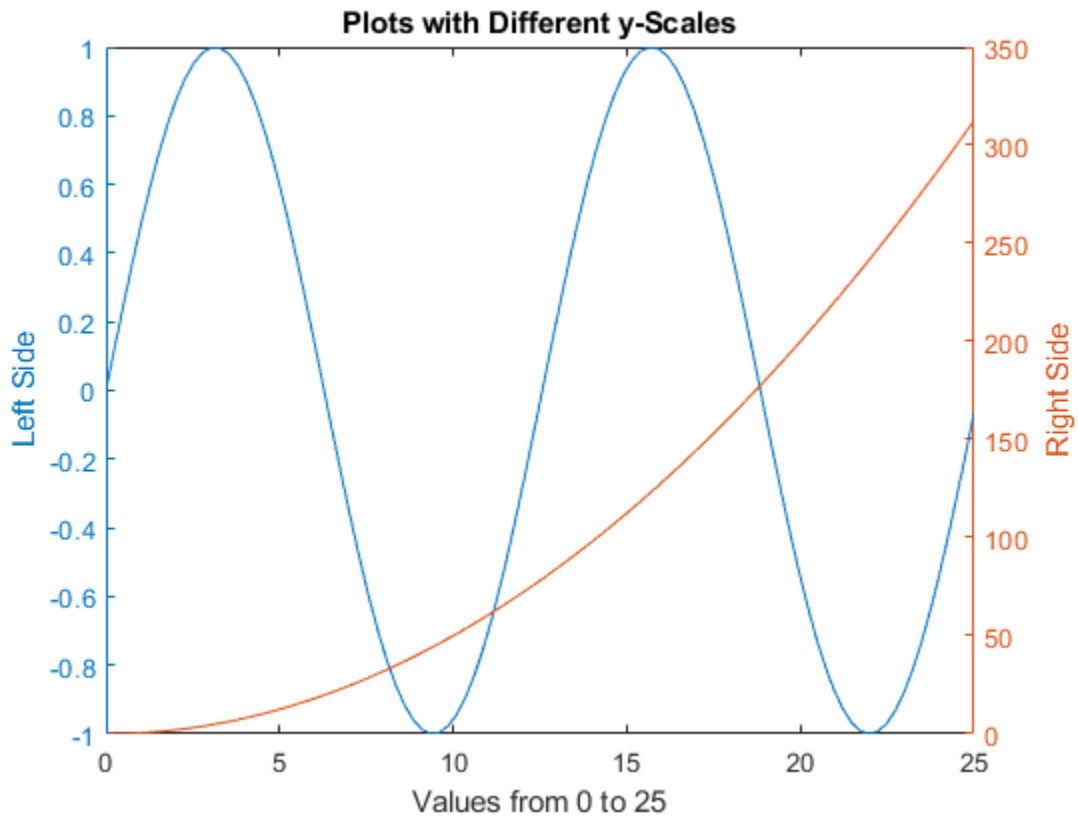
Plot Additional Data Against Each Side

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
```
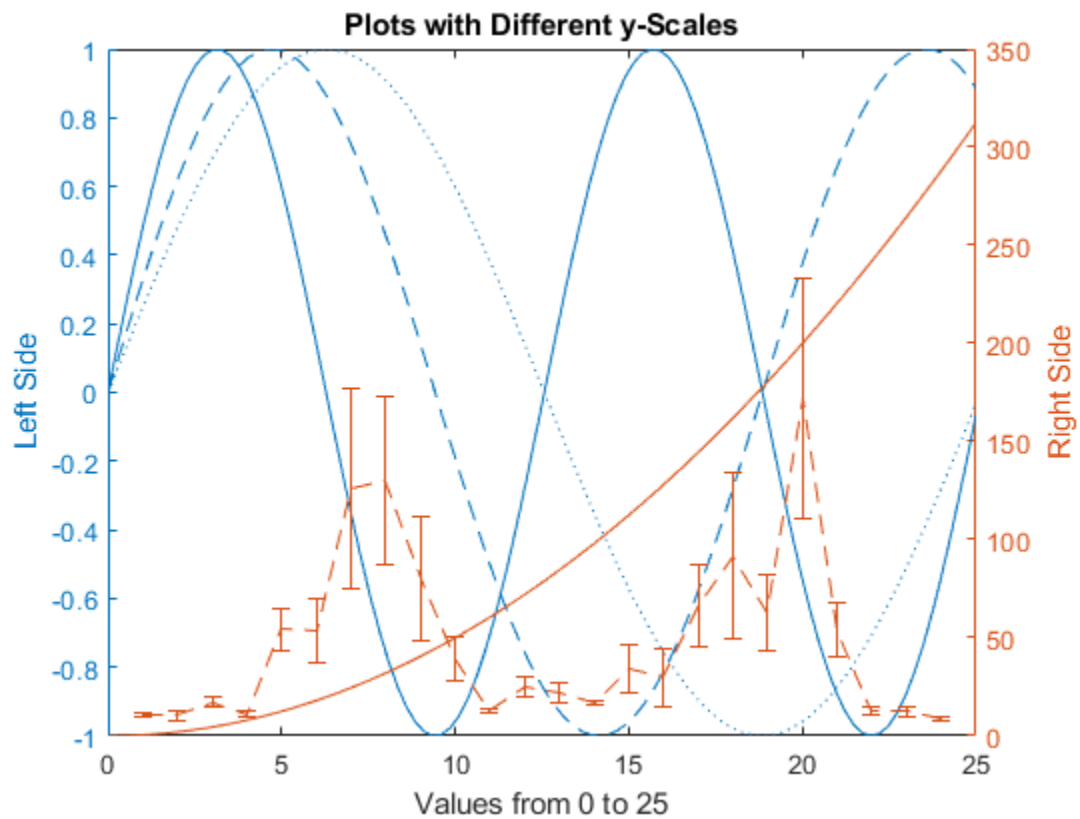
Clear One Side of Axes

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

```
yyaxis right
cla
```

Plots with Different y-Scales
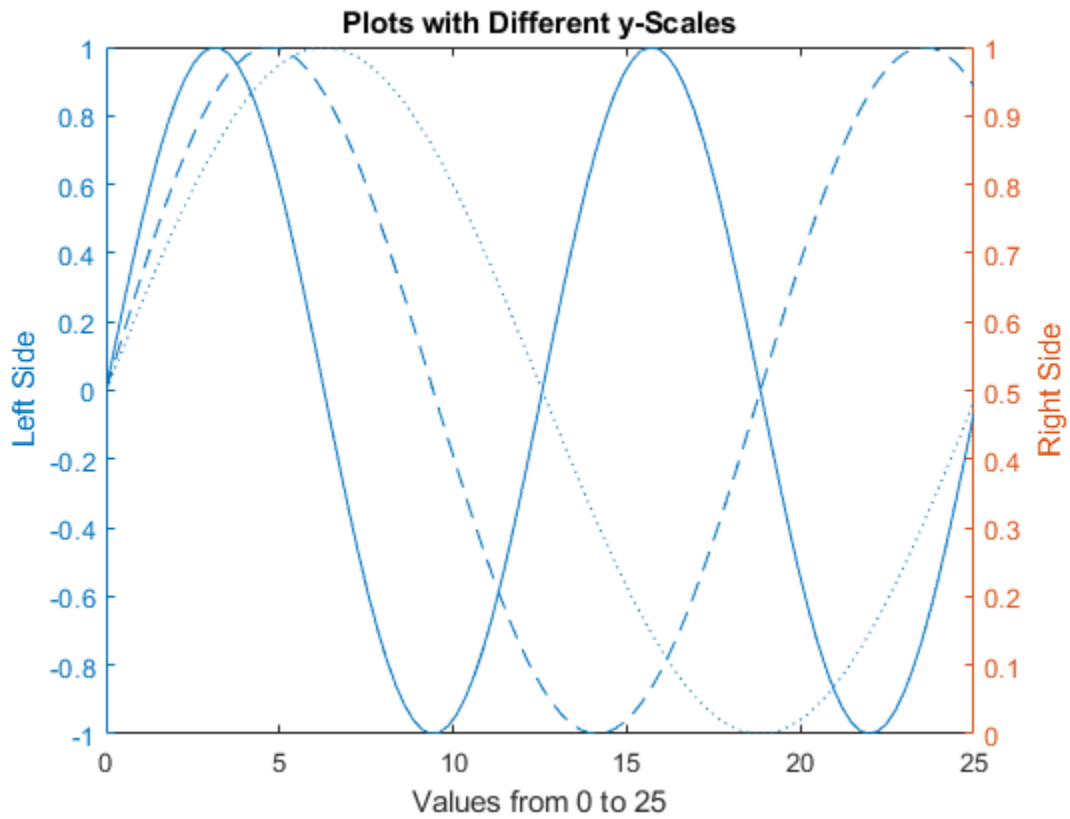
Clear Axes and Remove Right *y*-Axis

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

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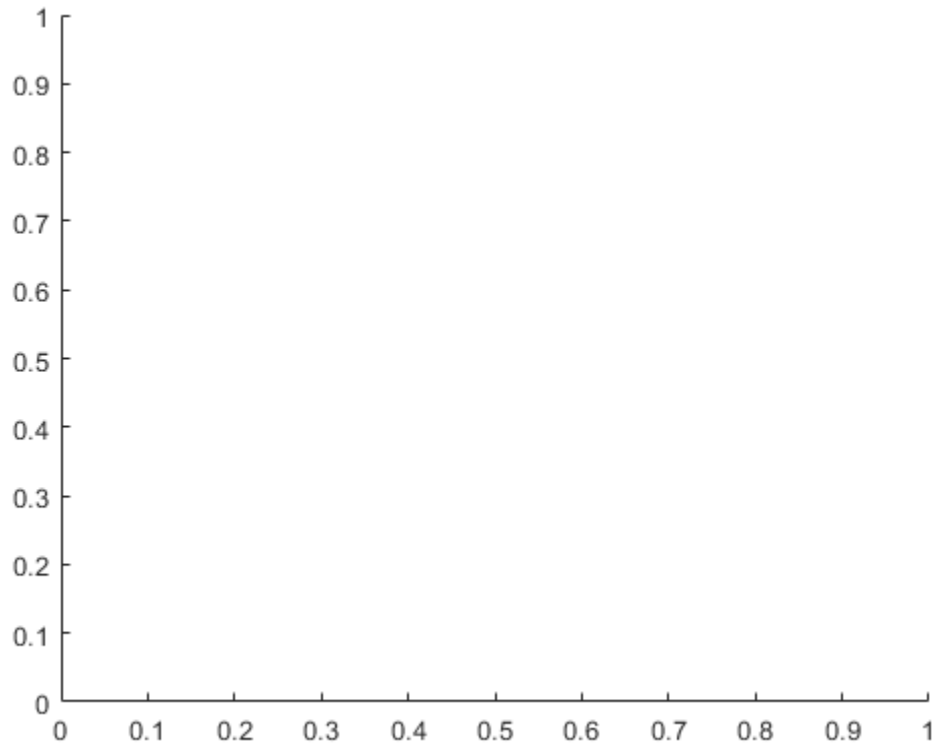
Now when you create a plot, it only has one *y*-axis. For example, plot three lines against the single y-axis.

```
xx = linspace(0,25);
yy1 = sin(xx/4);
yy2 = sin(xx/5);
yy3 = sin(xx/6);
plot(xx,yy1,xx,yy2,xx,yy3)
```
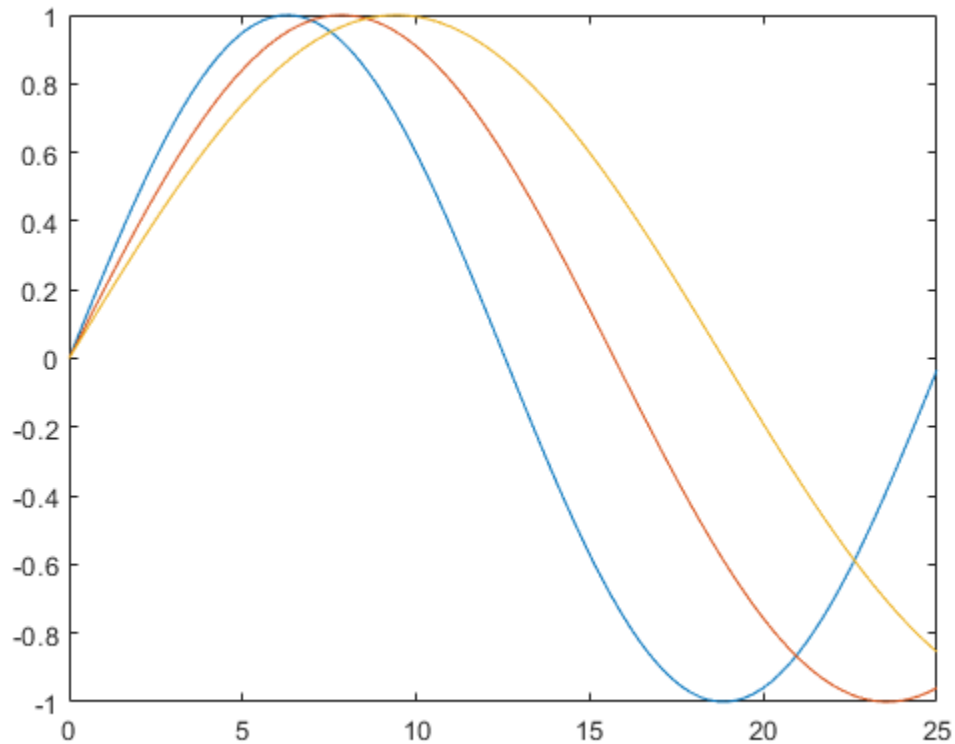
Add Second *y*-Axis to Existing Chart

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)
```
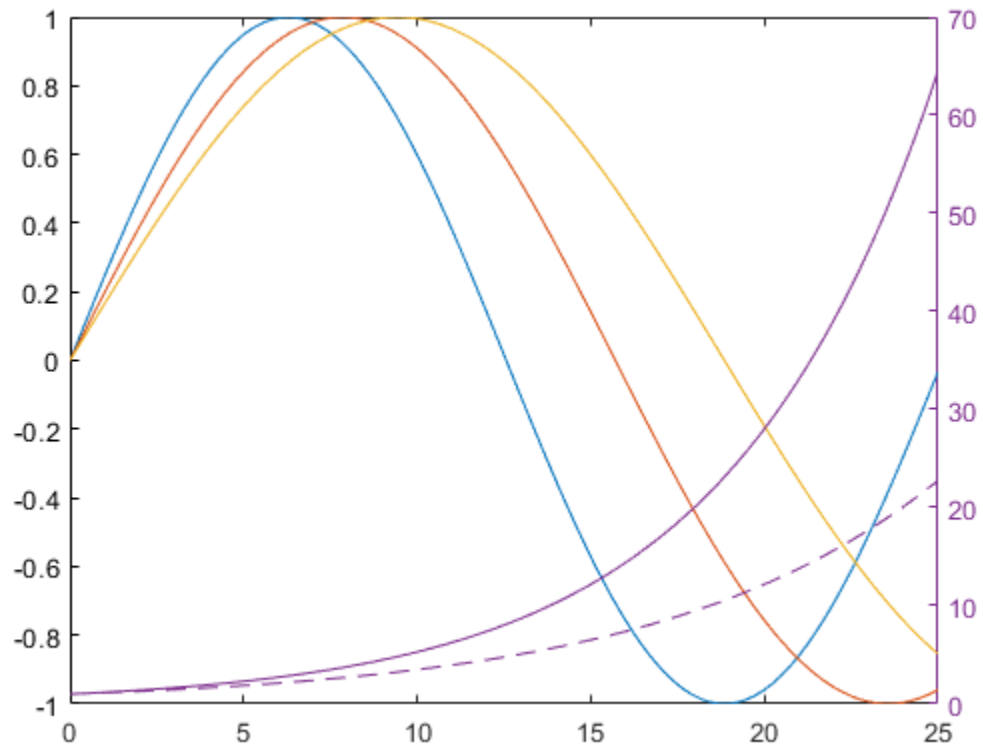
See Also

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

Related Examples

- "Modify Properties of Charts with Two y-Axes" on page 9-50
- "Combine Multiple Plots" on page 9-24

Modify Properties of Charts with Two y-Axes

In this section...
"Change Axes Properties" on page 9-50
"Change Ruler Properties" on page 9-52
"Specify Colors Using Default Color Order" on page 9-54

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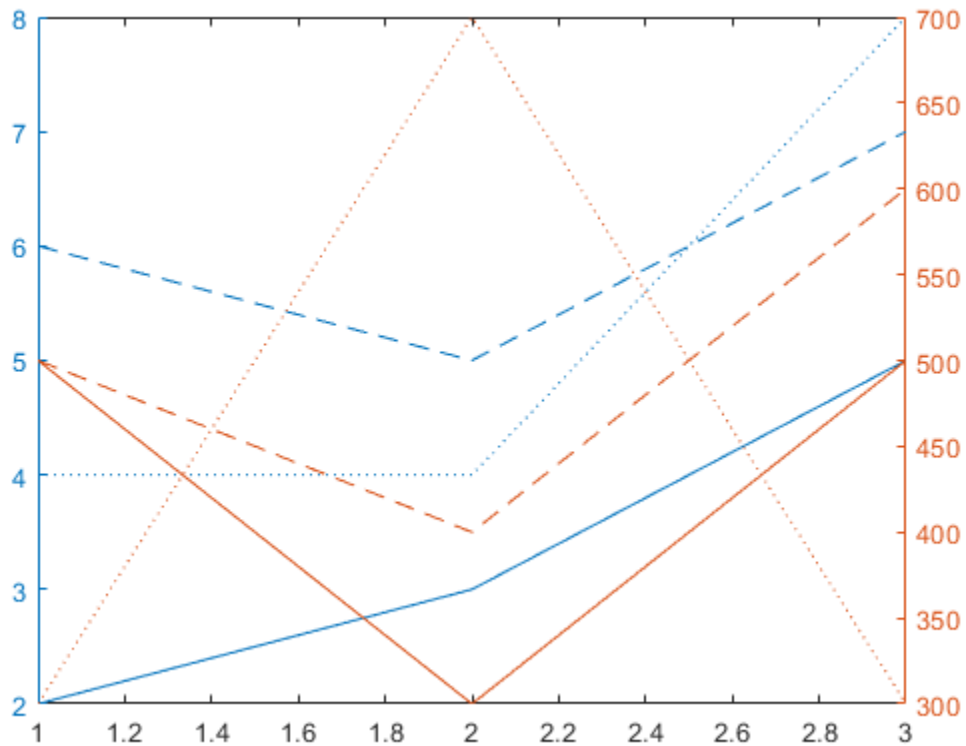
Reverse the direction of increasing values along each y-axis. Use `yyaxis left` to activate the left side and set the direction for the left y-axis. Similarly, use `yyaxis right` to activate the right side. Then, set the direction for the right y-axis.

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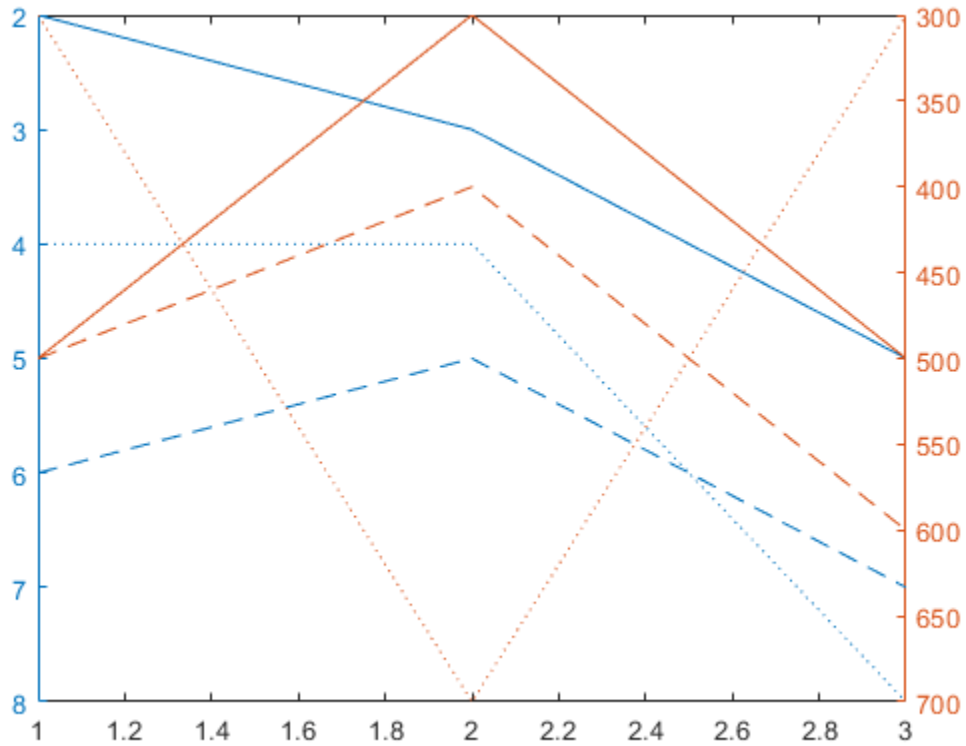
Change Ruler Properties

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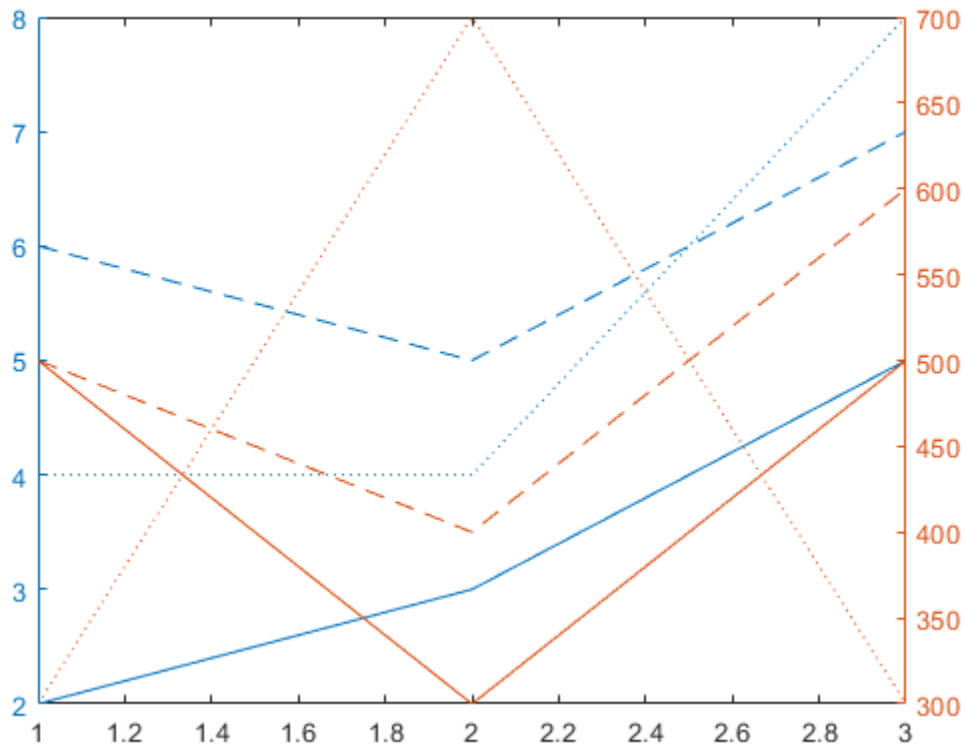
Reverse the direction of increasing values along each *y*-axis by setting properties of the ruler object associated with each axis. Use `ax.YAxis(1)` to refer to the ruler for the left side and `ax.YAxis(2)` to refer to the ruler for the right side.

```
ax = gca;
ax.YAxis(1).Direction = 'reverse';
ax.YAxis(2).Direction = 'reverse';
```
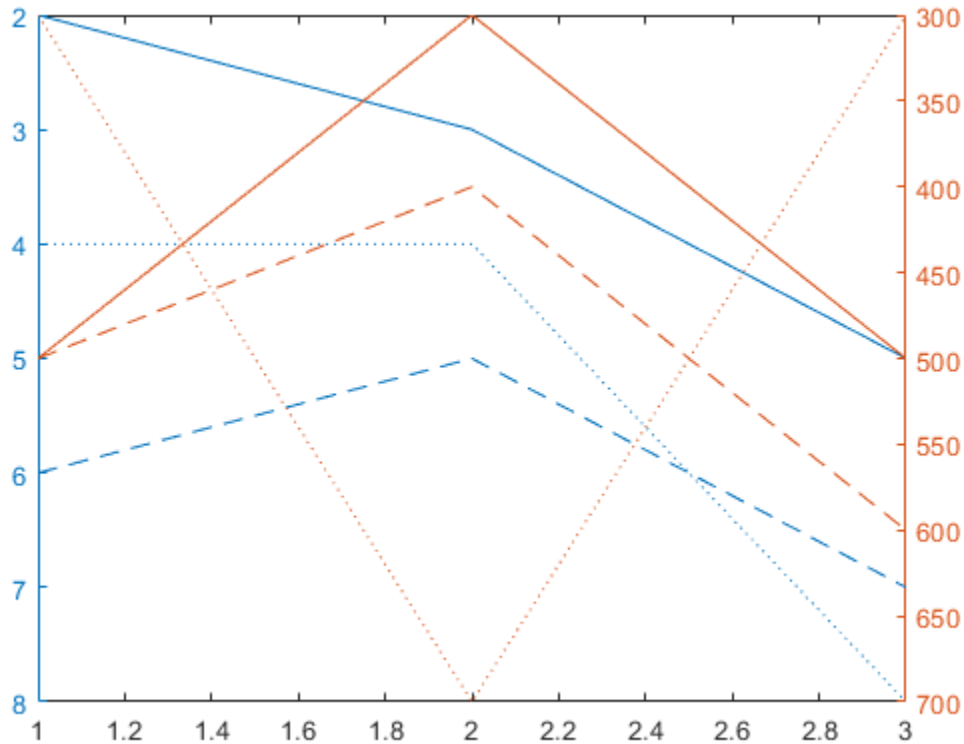
Specify Colors Using Default Color Order

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)
```
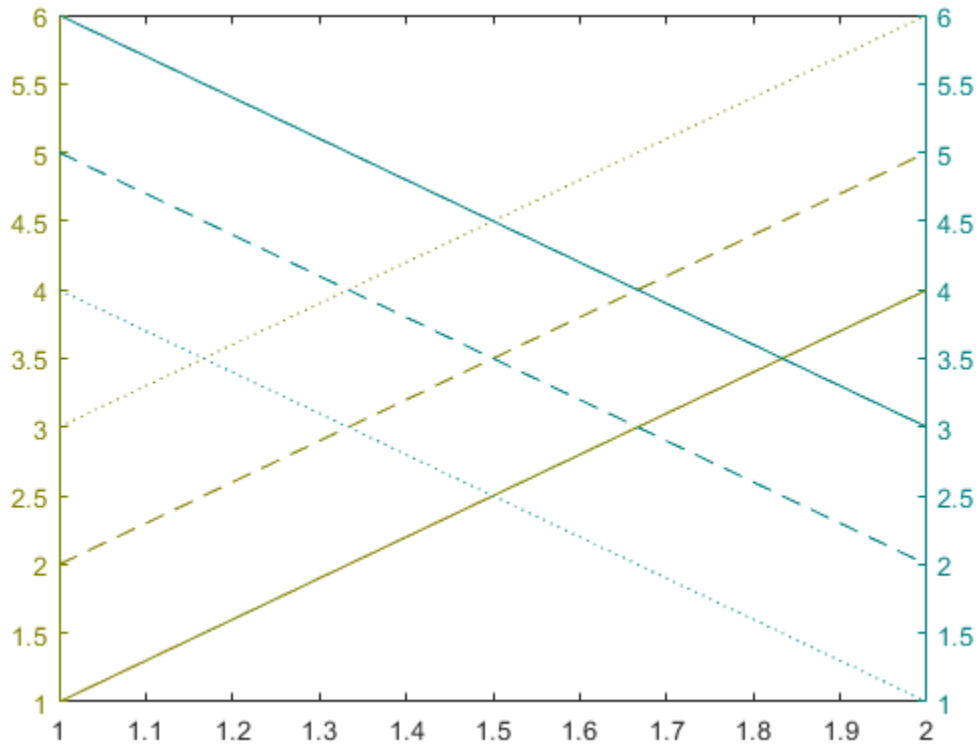
See Also

Functions
```
plot | yyaxis
```

Properties
Axes | Numeric Ruler

Related Examples

- "Create Chart with Two y-Axes" on page 9-42
- "Default Property Values" on page 17-23

Display Data with Multiple Scales and Axes Limits

You can use a variety of techniques to visualize data with multiple scales and axis limits. For example, you can use the `yyaxis` function create plots with two *y*-axes. To create plots with multiple *x*- and *y*-axes, multiple colorbars, or to create a plot with a discontinuous axis that is broken into intervals, use the `tiledlayout` function.

Display Data with Two y-Axes

Use the `yyaxis` function to create a plot with two *y*-axes. For example, you can use two *y*-axes to plot two lines on different scales.

Create an axes object, and activate the left *y*-axis by calling `yyaxis left`. Then plot a sine wave.

```
figure
yyaxis left
x = linspace(0,10);
y = sin(3*x);
plot(x,y)
```
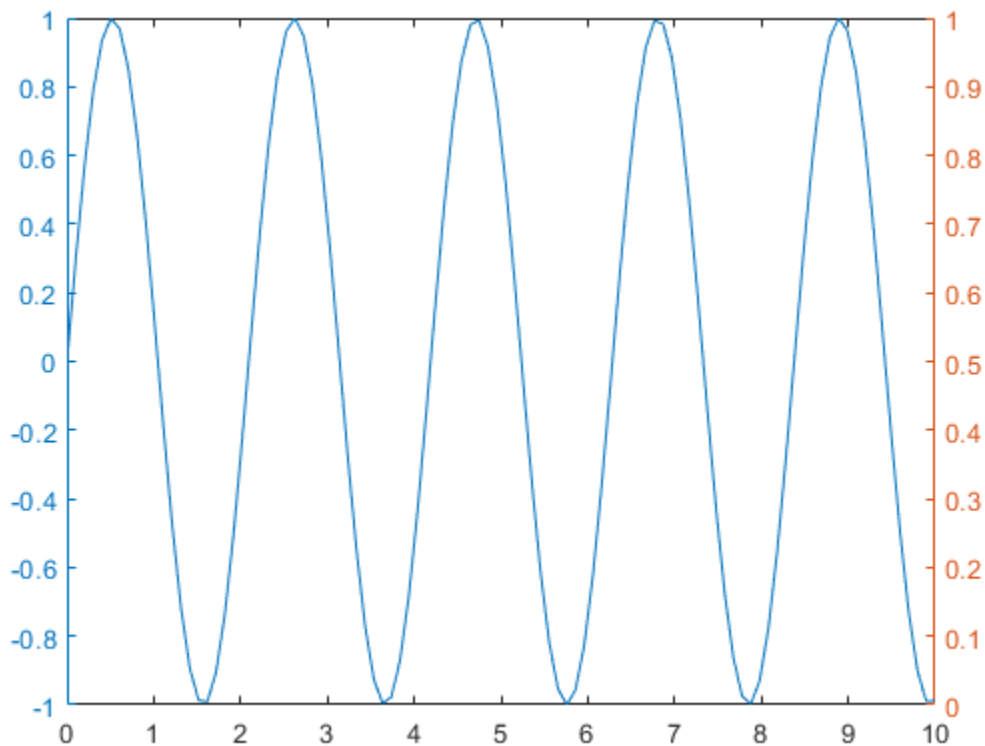

Activate the right *y*-axis by calling `yyaxis right`. Then plot an amplified sine wave.

```
yyaxis right
y2 = sin(3*x).*exp(0.5*x);
plot(x,y2)
```
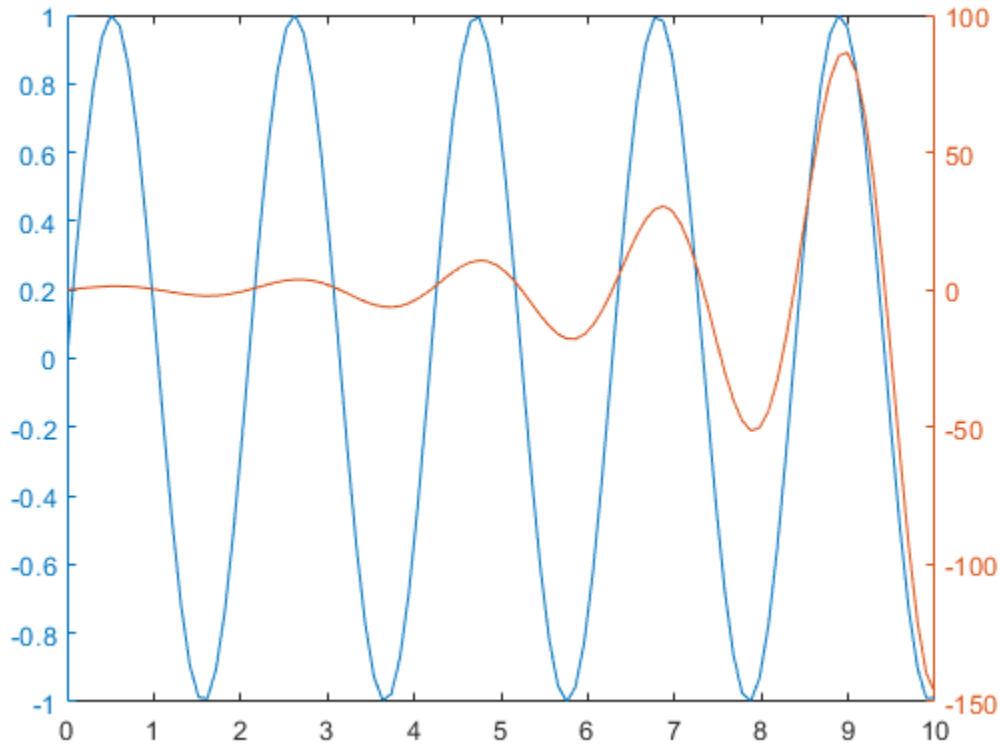
Display Data with Multiple x-Axes and y-Axes

Since R2019b

To plot two sets of data with separate *x*- and *y*-axes, create two separate axes objects in a tiled chart layout. Within one of the axes objects, move the *x*-axis to the top of the plot box, and move the *y*-axis to the right side of the plot box.

For example, you can create two plots that have different *x*- and *y*-axis limits.

First, create two sets of *x*- and *y*-coordinates.

```
x1 = 0:0.1:40;
y1 = 4.*cos(x1)./(x1+2);
x2 = 1:0.2:20;
y2 = x2.^2./x2.^3;
```

Create a tiled chart layout and an axes object. Then plot into the axes:

- Create a 1-by-1 tiled chart layout t.
- Create an axes object ax1 by calling the axes function and specifying t as the parent object.
- Plot x1 and y1 as a red line, and specify ax1 as the target axes.
- Change the color of the *x*-axis and *y*-axis to match the plotted line. Setting properties on the axes after plotting ensures that the settings persist.

```
t = tiledlayout(1,1);
ax1 = axes(t);
plot(ax1,x1,y1,'-r')
ax1.XColor = 'r';
ax1.YColor = 'r';
```
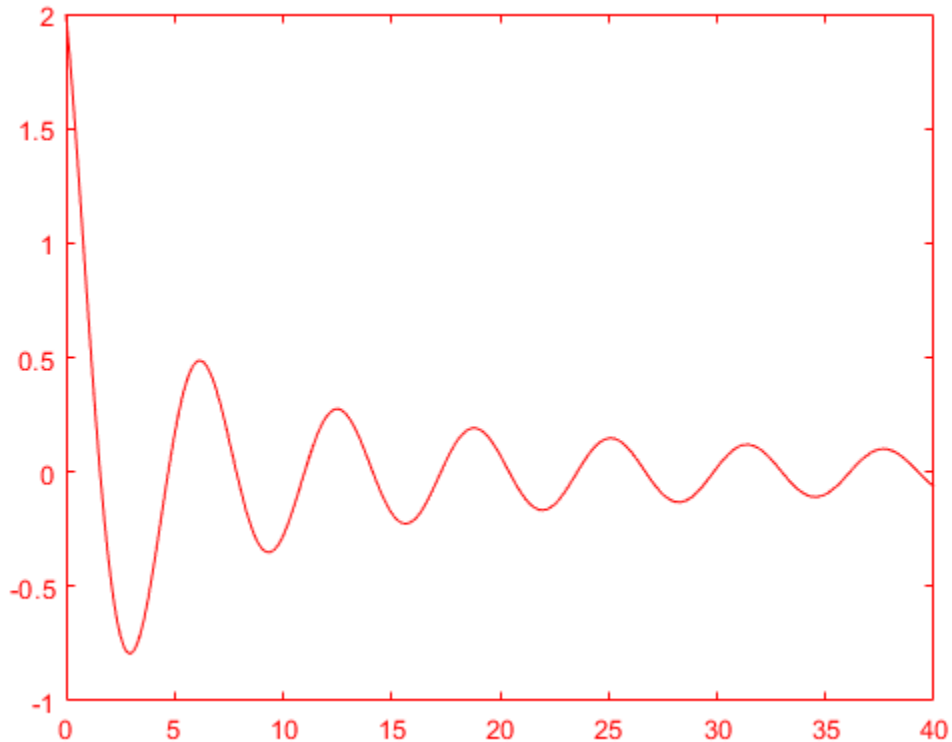

Create a second axes object and plot the second set of data in black rather than red. Then, set properties on the second axes object to move the *x*-axis and *y*-axis, and to ensure that neither plot obscures the other.

- Create an axes object `ax2` by calling the `axes` function and specifying `t` as the parent object.
- Plot `x2` and `y2` as a black line, and specify `ax2` as the target axes.
- Move the *x*-axis to the top, and move the *y*-axis to the right.
- Set the color of the axes object to `'none'` so that the underlying plot is visible.
- Turn off the plot boxes to prevent the box edges from obscuring the *x*- and *y*-axes.

```
ax2 = axes(t);
plot(ax2,x2,y2,'-k')
ax2.XAxisLocation = 'top';
ax2.YAxisLocation = 'right';
ax2.Color = 'none';
ax1.Box = 'off';
ax2.Box = 'off';
```
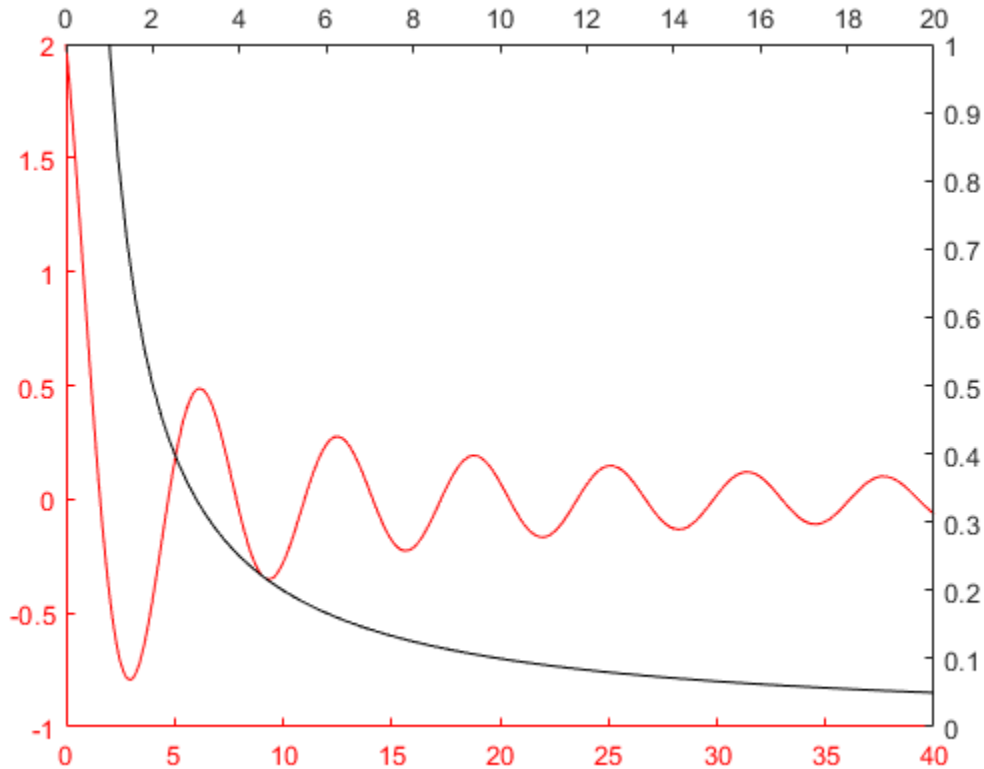
Plot Data on Discontinuous x-Axis

Since R2019b

You can use a tiled chart layout to give the appearance of a plot that is broken into intervals along one axis. For example, you might want to exclude one section of the *x*-axis to focus on other regions of interest.

Create coordinate vectors x and y.

```
x = 0:0.1:60;
y = 4.*cos(x)./(x+2);
```

Create a tiled chart layout containing two tiles, and place an axes object across both tiles. In the final presentation, this axes object will appear in the background, behind two other axes objects. A section of its *x*-axis will be visible to give the appearance of one long *x*-axis.

- Create a 1-by-2 tiled chart layout t, and specify compact tile spacing. Setting the tile spacing allows you to control the size of the gap between the *x*-axis intervals.
- Create the background axes bgAx by calling the axes function and specifying t as the parent object. Specify name-value arguments to remove all the ticks and turn off the plot box.
- Span the background axes across both tiles by setting the Layout.TileSpan property of bgAx to [1 2].

```
figure
t = tiledlayout(1,2,'TileSpacing','compact');
bgAx = axes(t,'XTick',[],'YTick',[],'Box','off');
bgAx.Layout.TileSpan = [1 2];
```


Create an axes object in front of `bgAx` in the first tile. Plot *x* and *y*, and set the *x*-axis limits to the first interval:

- Create `ax1` by calling the `axes` function and specifying `t` as the parent object. By default, the axes goes into the first tile.

- Plot x and y into `ax1`.

- Call the `xline` function to display a dotted vertical line at the upper limit of the first interval.

- Set the *x*-axis limits to the first interval, `[0 15]`.

- Add an axis label to identify the first interval.

```
ax1 = axes(t);
plot(ax1,x,y)
xline(ax1,15,':');
ax1.Box = 'off';
xlim(ax1,[0 15])
xlabel(ax1, 'First Interval')
```
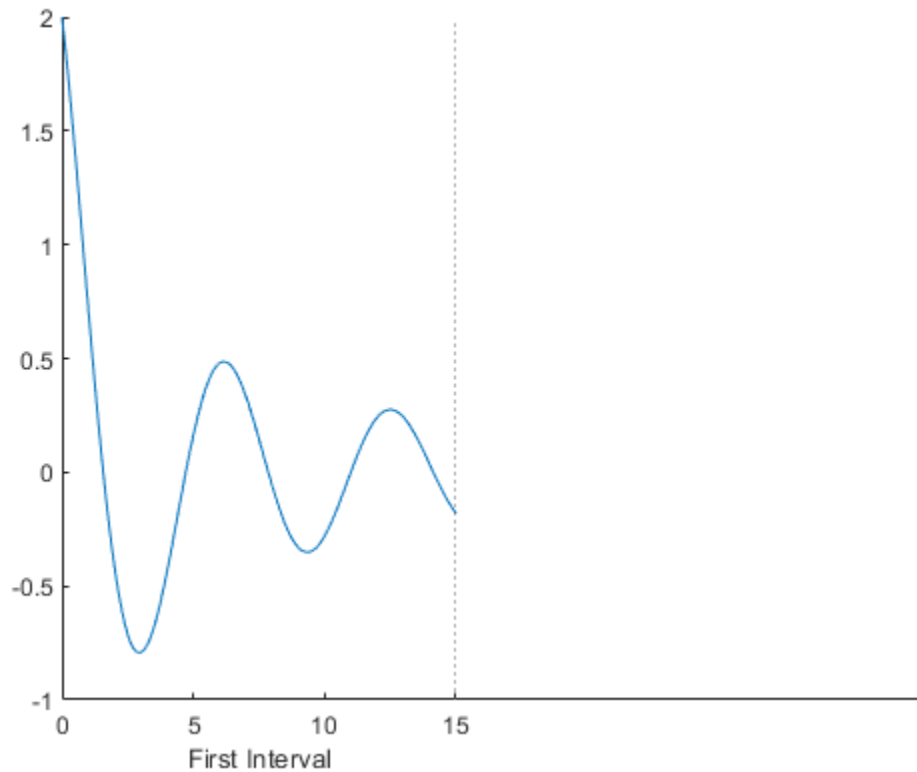
Repeat the process to create another axes object and plot for the second interval. The axes appears in the first tile by default. Move it to the second tile by setting the `Layout.Tile` property of the axes to 2. Then, link the axes so that the limits of both *y*-axes match.

```
% Create second plot
ax2 = axes(t);
ax2.Layout.Tile = 2;
plot(ax2,x,y)
xline(ax2,45,':');
ax2.YAxis.Visible = 'off';
ax2.Box = 'off';
xlim(ax2,[45 60])
xlabel(ax2,'Second Interval')

% Link the axes
linkaxes([ax1 ax2], 'y')
```
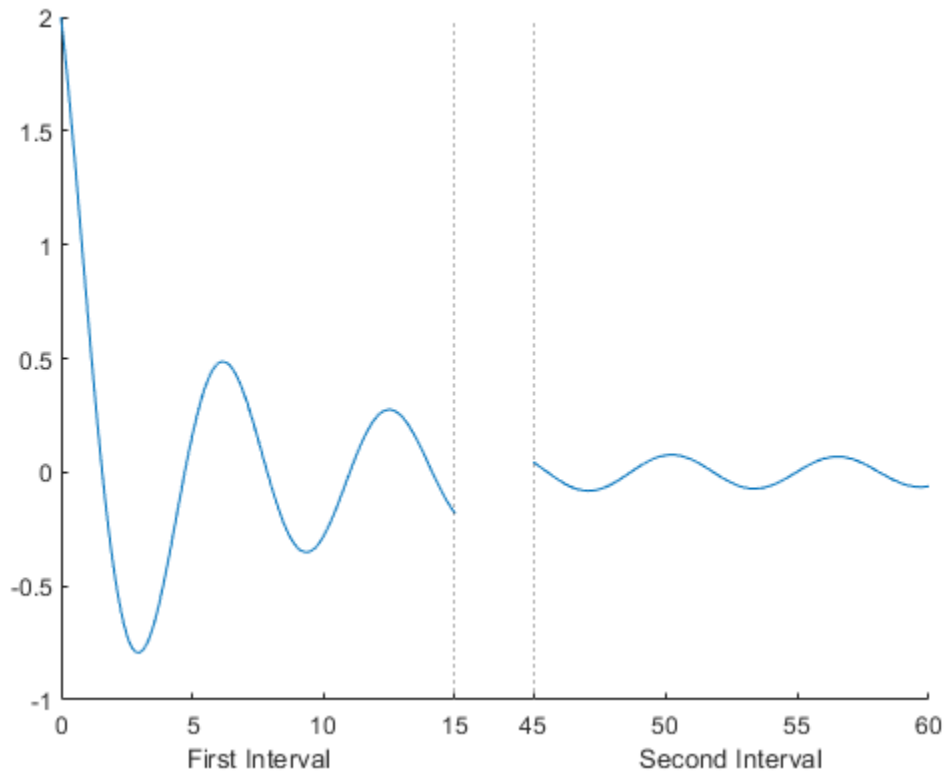
To add a title, pass the tiled chart layout to the `title` function.

```
title(t,'Attenuated Cosine Function')
```

Attenuated Cosine Function
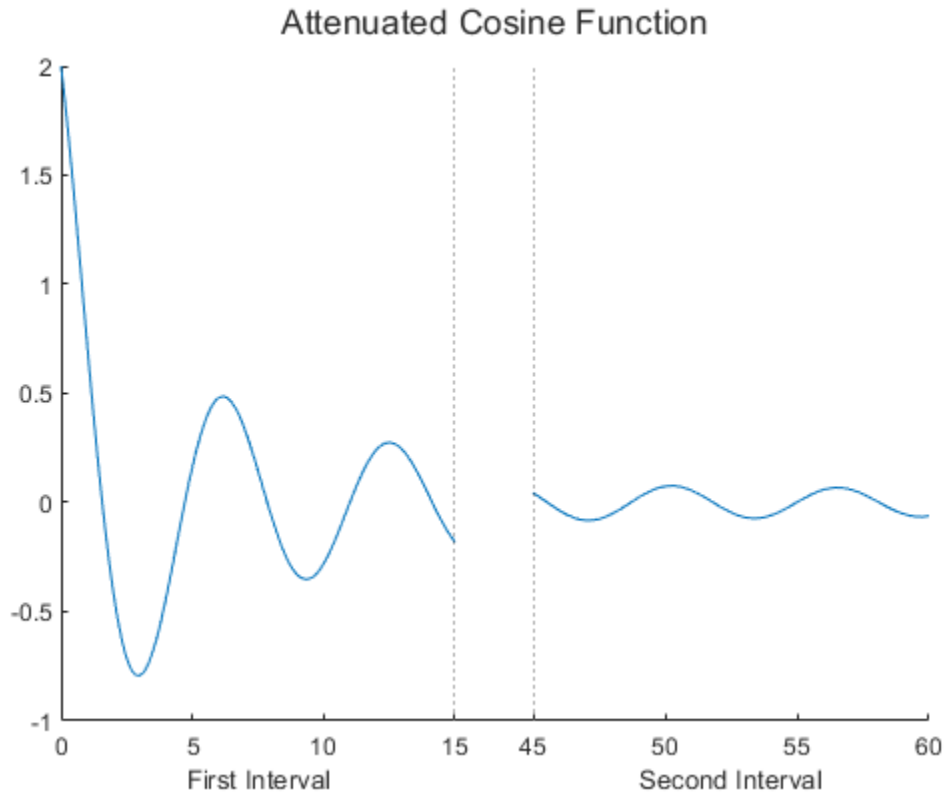
Display Two Sets of Data with Separate Colorbars

Since R2020b

An axes object can accommodate only one colorbar. To create a visualization with multiple colorbars, stack multiple axes objects in a tiled chart layout. Make only one of the axes visible, but display a colorbar next to each of them in an outer tile of the layout.

Create the coordinate vectors, size data, and color data for two bubble charts.

```
x = 1:15;
n = 70 * randn(1,15) + 50;
y1 = n + x.^2;
y2 = n - linspace(1,225,15);
sz1 = rand(1,15);
sz2 = rand(1,15);
c = linspace(1,10,15);
```

Stack two axes objects, each containing a bubble chart, in a tiled chart layout.

- Create a 1-by-1 tiled chart layout t.
- Create axes object ax1 and a bubble chart with the winter colormap.
- Create axes object ax2 and a bubble chart with the autumn colormap. Make this axes object invisible by setting the Visible property to 'off'.

- Link the axes objects to keep them in sync. In this case, you can pass the children of `t` to the `linkaxes` function. Alternatively, you can pass a vector of individual axes objects to the function.

```
% create first bubble chart with winter colormap
t = tiledlayout(1,1);
ax1 = axes(t);
bubblechart(ax1,x,y1,sz1,c)
colormap(ax1,'winter')

% create second bubble chart with autumn colormap
ax2 = axes(t);
bubblechart(ax2,x,y2,sz2,c)
colormap(ax2,'autumn')
ax2.Visible = 'off';

% link the limits of axes
linkaxes(t.Children)
```
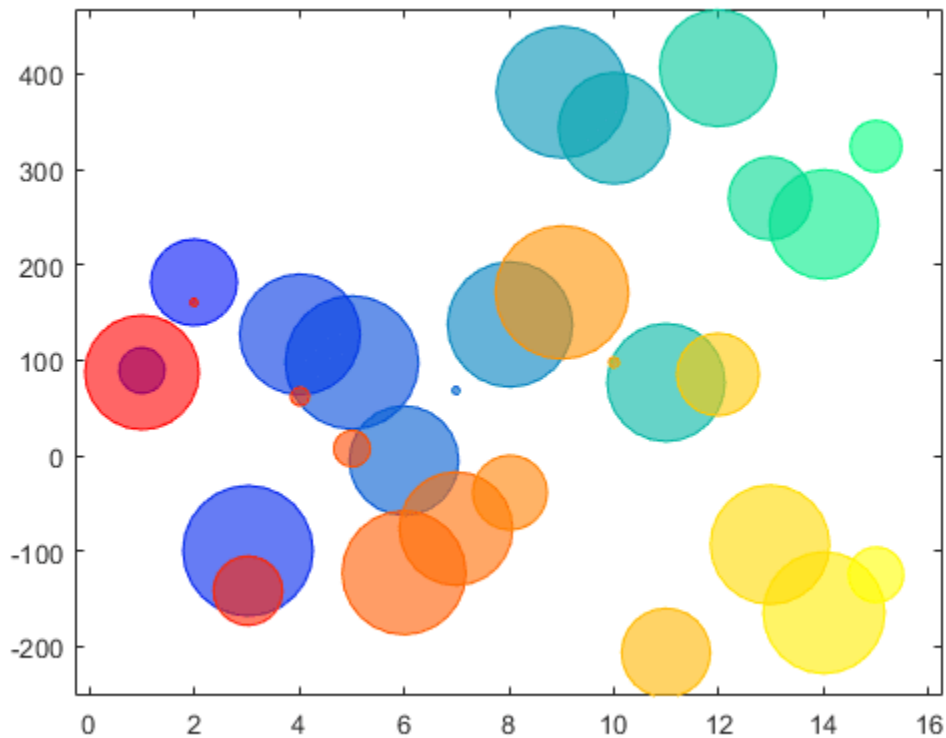

Display a colorbar with a label for each axes object in the east tile of the layout. The layout arranges the colorbars and keeps them aligned.

```
cb1 = colorbar(ax1);
cb1.Layout.Tile = 'east';
cb1.Label.String = 'Time (s)';

cb2 = colorbar(ax2);
cb2.Layout.Tile = 'east';
cb2.Label.String = 'Concentration (M)';
```
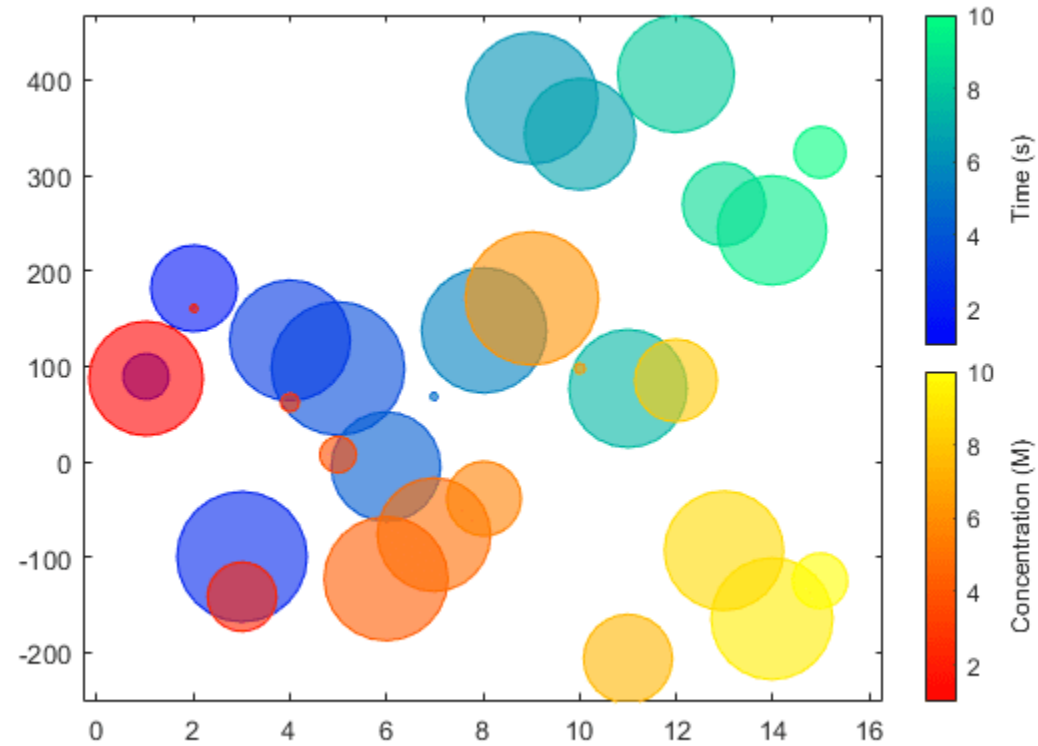
See Also

Functions
axes | tiledlayout | yyaxis

Related Examples

- "Combine Multiple Plots" on page 9-24
- "Creating Colorbars" on page 10-2

Control Ratio of Axis Lengths and Data Unit Lengths

In this section...
"Plot Box Aspect Ratio" on page 9-66
"Data Aspect Ratio" on page 9-68
"Revert Back to Default Ratios" on page 9-71

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])
```
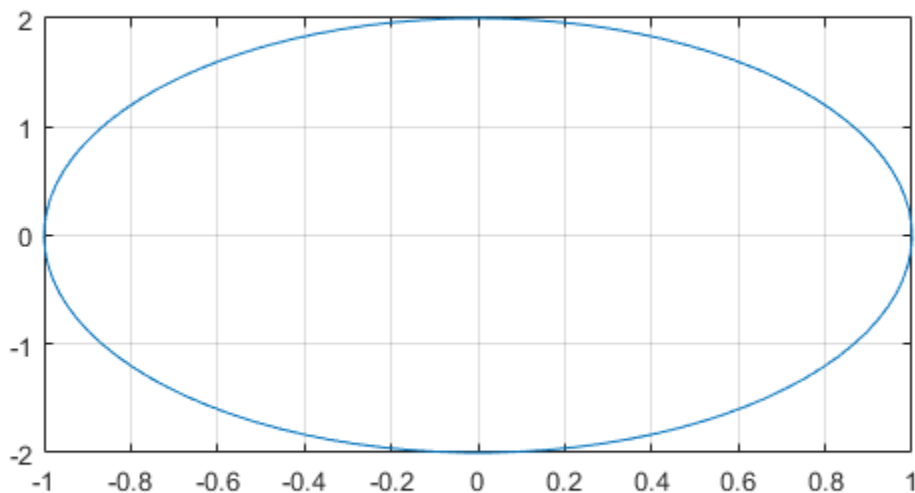
Show the axes in a 3-D view to see the *z*-axis.

```
view(3)
```
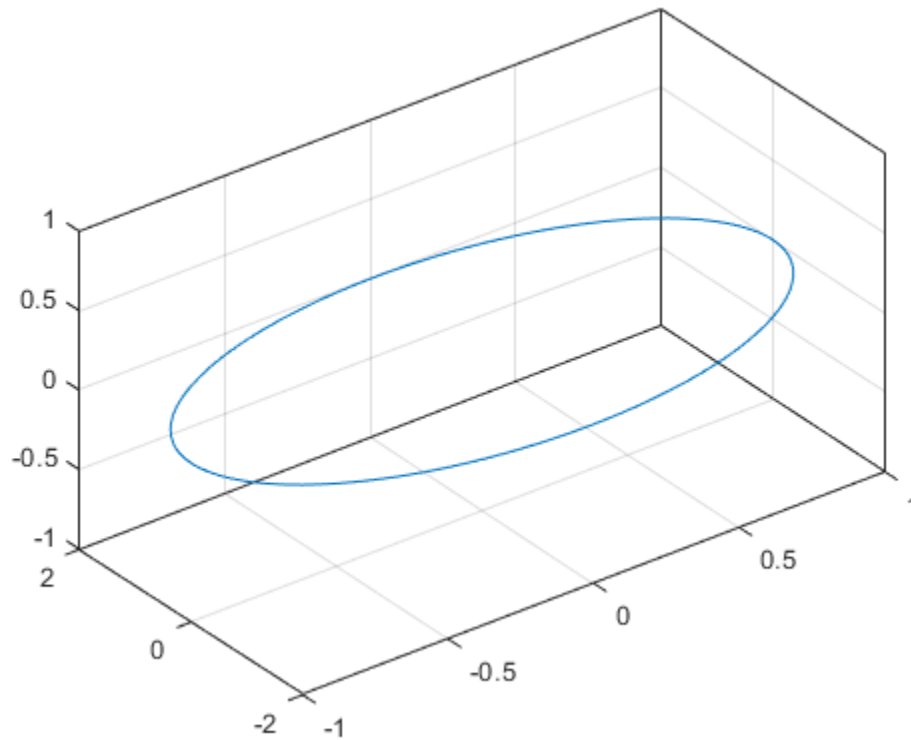

For square axes, use [1 1 1]. This value is similar to using the `axis square` command.

```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
pbaspect([1 1 1])
```
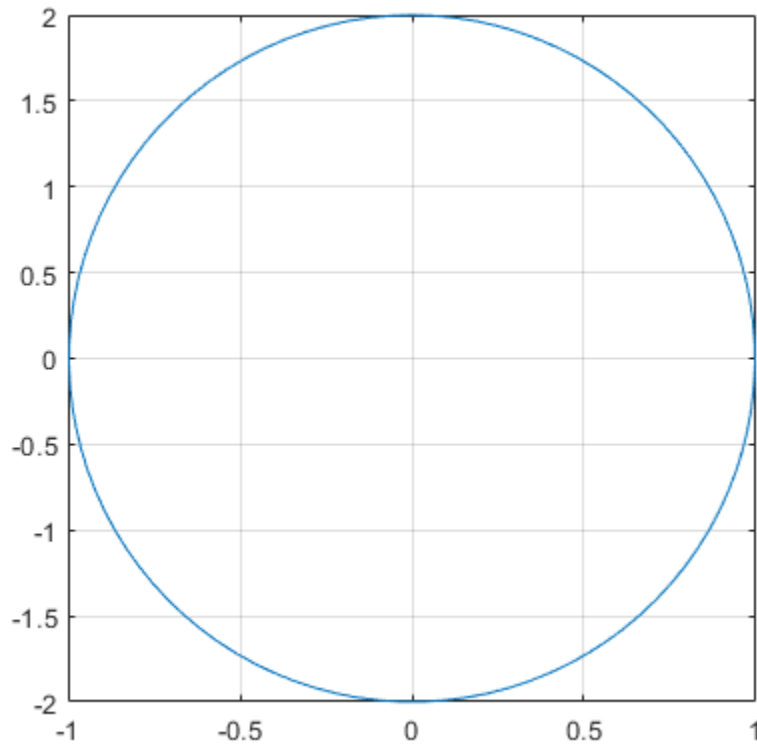
Data Aspect Ratio

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])
```
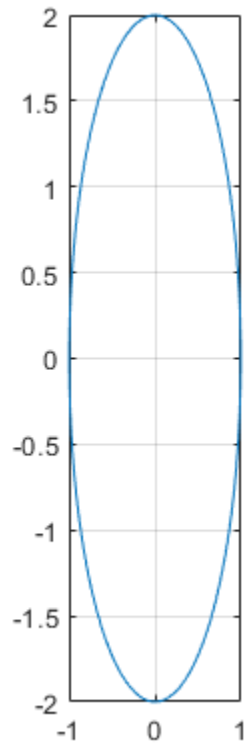
Show the axes in a 3-D view to see the *z*-axis.

```
view(3)
```
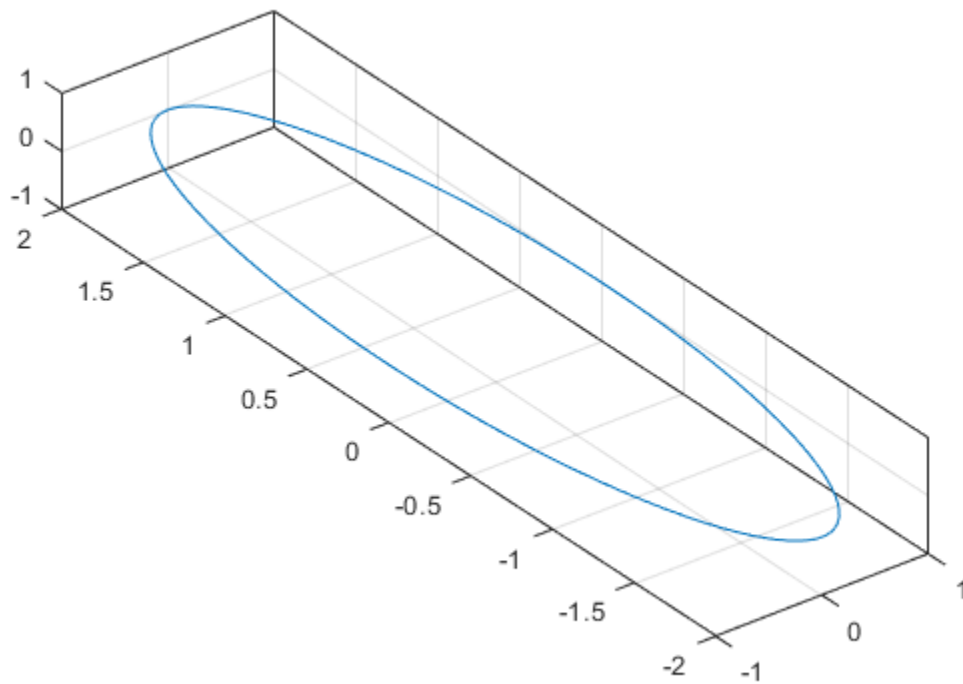
For equal data units in all directions, use [1 1 1]. This value is similar to using the `axis equal` command. One data unit in the x direction is the same length as one data unit in the y and z directions.

```
t = linspace(0,2*pi);
plot(sin(t),2*cos(t))
grid on
daspect([1 1 1])
```
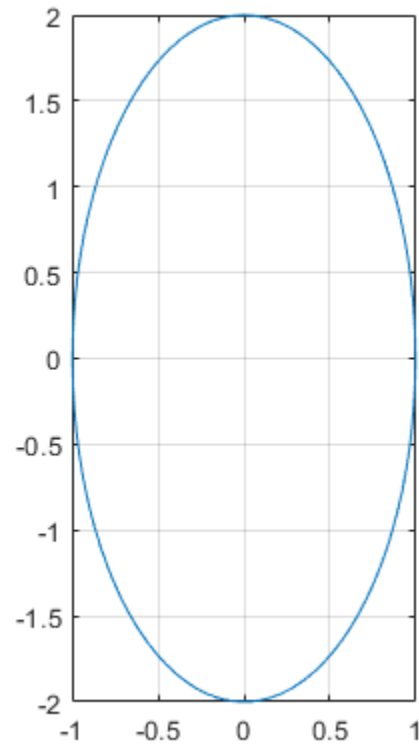
Revert Back to Default Ratios

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
```
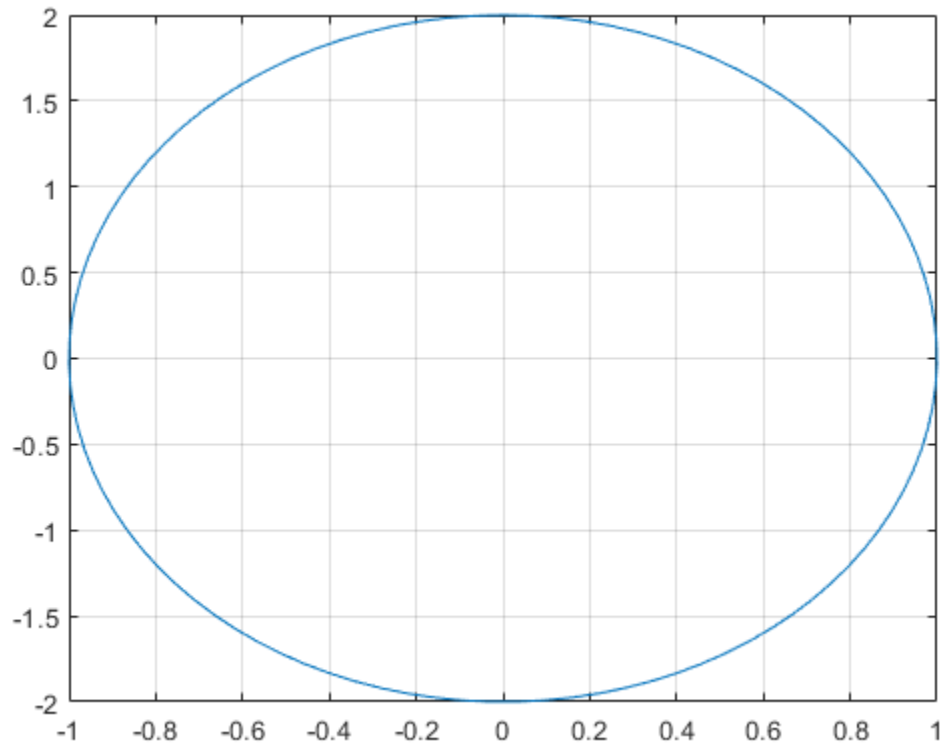
See Also

Functions
axis | daspect | pbaspect

Related Examples

- "Specify Axis Limits" on page 9-2
- "Control Axes Layout" on page 9-73

Control Axes Layout

In this section...

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.

- `PositionConstraint` — Position property preserved when the `Axes` object changes size, specified as either `'outerposition'` or `'innerposition'`.

- `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).
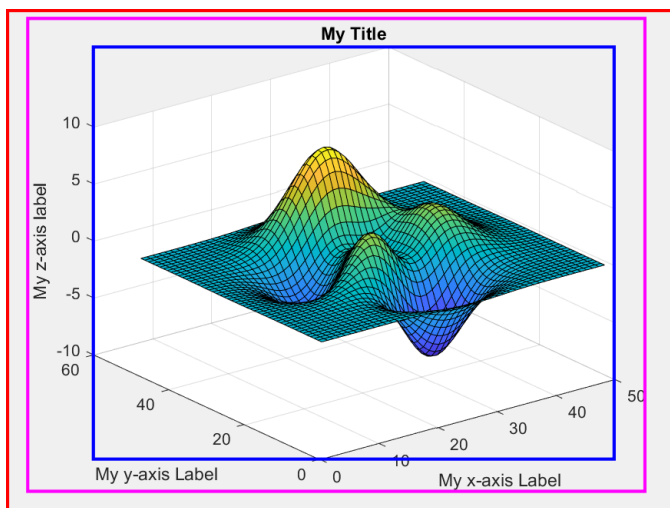
Controlling Automatic Resize Behavior

Some scenarios 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 `PositionConstraint` property indicates which values to preserve. Specify the `PositionConstraint` property 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.

- `'innerposition'` — Preserve the `InnerPosition` 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 `PositionConstraint` property 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 axes and specify the same width and height for each axes position. Set the `PositionConstraint` property to `'outerposition'` for the upper axes and to `'innerposition'` for the lower axes. Notice that in the upper axes, the inner area shrinks to accommodate the text, but the text does not run outside the figure. In the lower axes, the size of the inner area is preserved, but some of the text is cut off.

Note The following code uses the `PositionConstraint` property, which is new starting in R2020a. If you are using an earlier release, set the `ActivePositionProperty` to either `'outerposition'` or `'position'`.

```
figure;
ax1 = axes('Position',[0.13 0.58 0.77 0.34]);
ax1.PositionConstraint = 'outerposition';
% R2019b and earlier: 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 = axes('Position',[0.13 0.11 0.77 0.34]);
ax2.PositionConstraint = 'innerposition';
% R2019b and earlier: 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'})
```
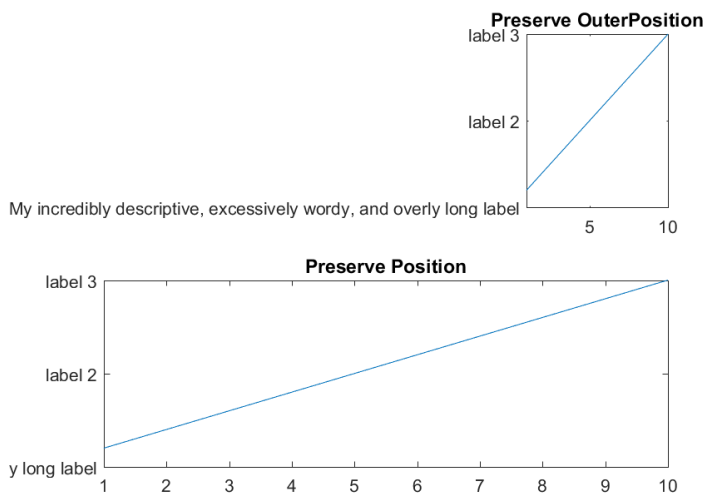
Stretch-to-Fill Behavior

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.
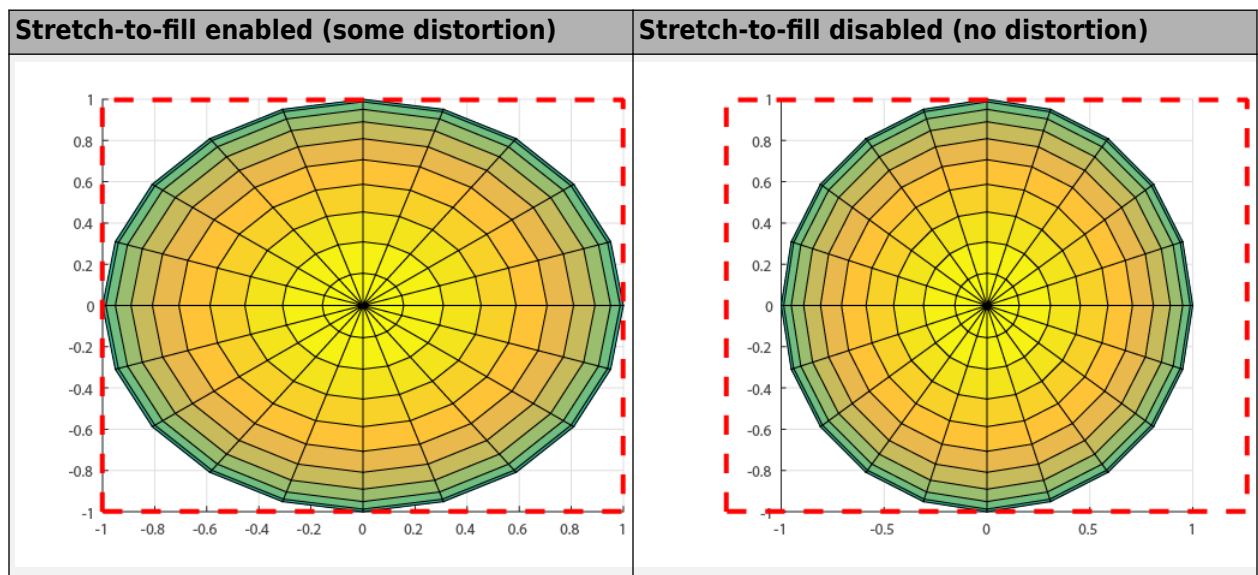
See Also

Functions
axes | daspect | pbaspect | tiledlayout | title

Properties
Axes

Related Examples

*   "Saving and Copying Plots with Minimal White Space" on page 16-25

Manipulating Axes Aspect Ratio

In this section...
"Axes Aspect Ratio Properties" on page 9-77
"Default Aspect Ratio Selection" on page 9-78
"Maintaining the Axes Proportions with Figure Resize" on page 9-80
"Aspect Ratio Properties" on page 9-82
"Displaying Real Objects" on page 9-86

Axes Aspect Ratio Properties

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

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

When the mode properties are set to `auto`, MATLAB automatically determines values for all of these properties and then stretches the axes to fit the figure shape. You can override any property's automatic operation by specifying a value for the property or setting its mode property to manual.

The value you select for a particular property depends primarily on what type of data you want to display. Much of the data visualized with MATLAB is either

• Numerical data displayed as line, mesh plots, or other specialized plot

• Representations of real-world objects (e.g., a motor vehicle or a section of the earth's topography)

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

The MATLAB default property values are designed to

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

Default Aspect Ratio Selection

The axes `Position` property specifies the location and dimensions of the axes within the figure. The third and fourth elements of the `Position` vector (width and height) define a rectangle in which MATLAB draws the axes. MATLAB fits the axes to this rectangle.

The default value for the axes `Units` property is normalized to the parent figure dimensions. This means the shape of the figure window determines the shape of the position rectangle. As you change the size of the figure window, MATLAB reshapes the position rectangle to fit it.

```
sphere
set(gcf,'Color',[0.90 0.90 0.90])
set(gca,'BoxStyle','full','Box','on')
```
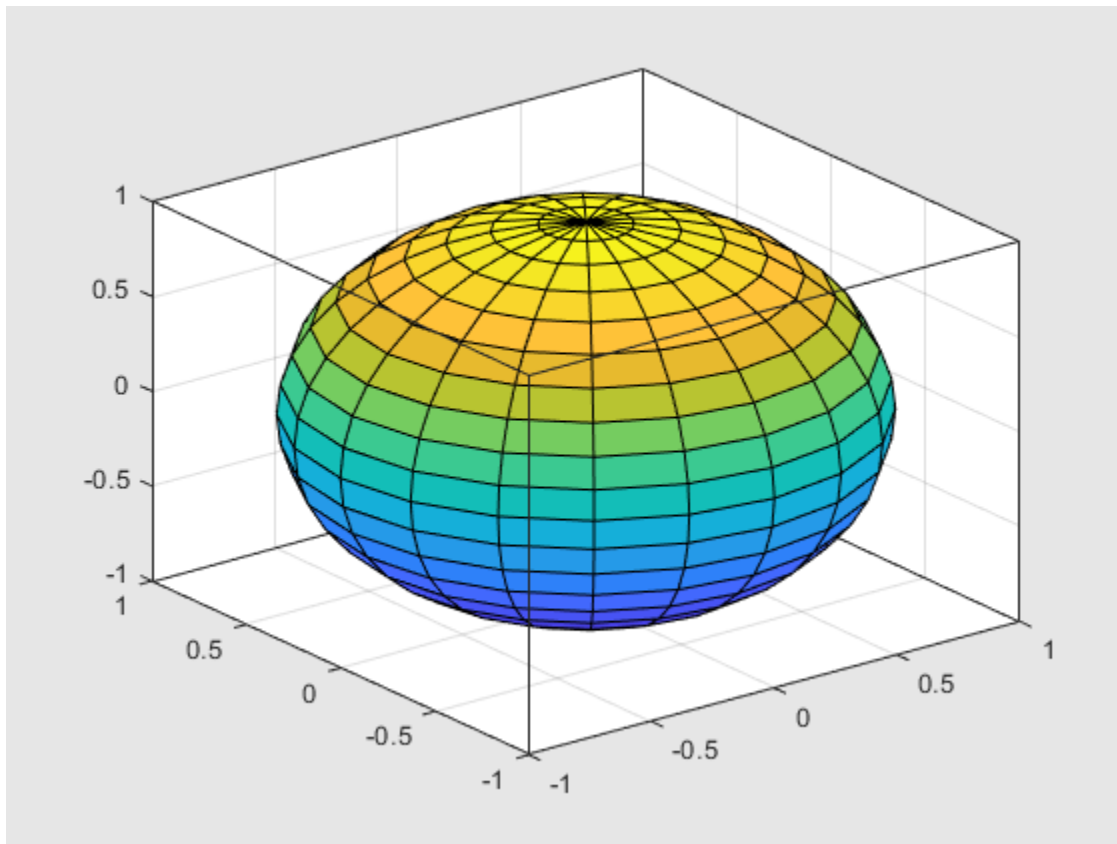
Changing the size and shape of the figure causes a change in the size and shape of the axes. The axes might select new axis tick mark locations as well.

```
f = gcf;
f.Position(3) = f.Position(3) * 0.67;
```
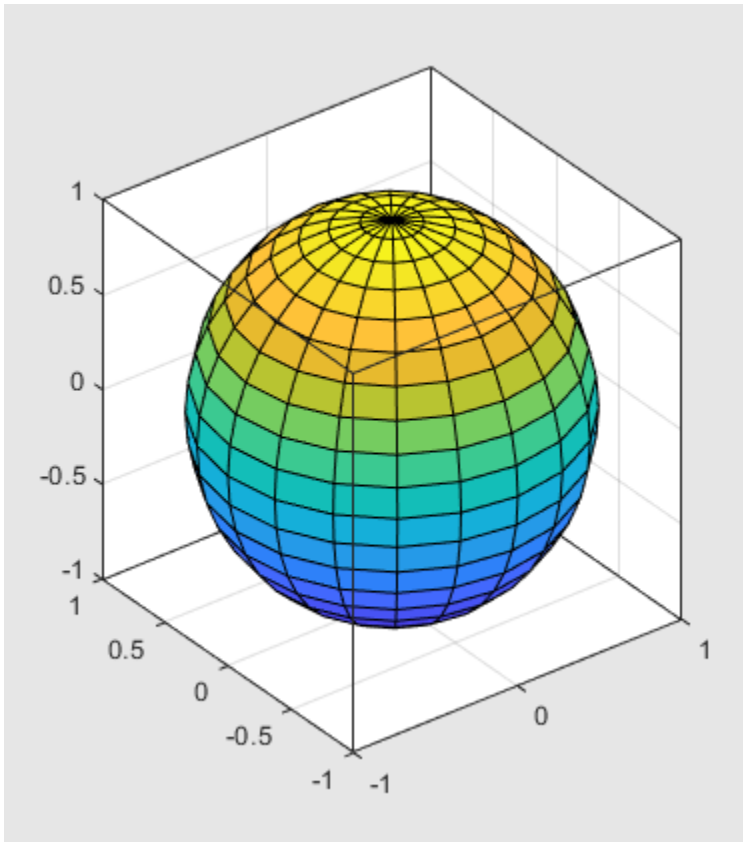
Reshaping the axes to fit into the figure window can change the aspect ratio of the graph. MATLAB fits the axes to fill the position rectangle and in the process can distort the shape. This is generally desirable for graphs of numeric data, but not for displaying objects realistically.

Maintaining the Axes Proportions with Figure Resize

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

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

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

The first two sets of properties affect the aspect ratio directly. Setting either of the mode properties to manual simply disables stretch-to-fill while maintaining all current property values. In this case, MATLAB enlarges the axes until one dimension of the position rectangle constrains it. For example, set the `DataAspectRatio` to `[1 1 1]`. Also set the figure's color to see the relationship between the figure and the axes.

```
sphere
daspect([1 1 1])
set(gca,'BoxStyle','full','Box','on')
set(gcf,'Color',[0.90 0.90 0.90])
```
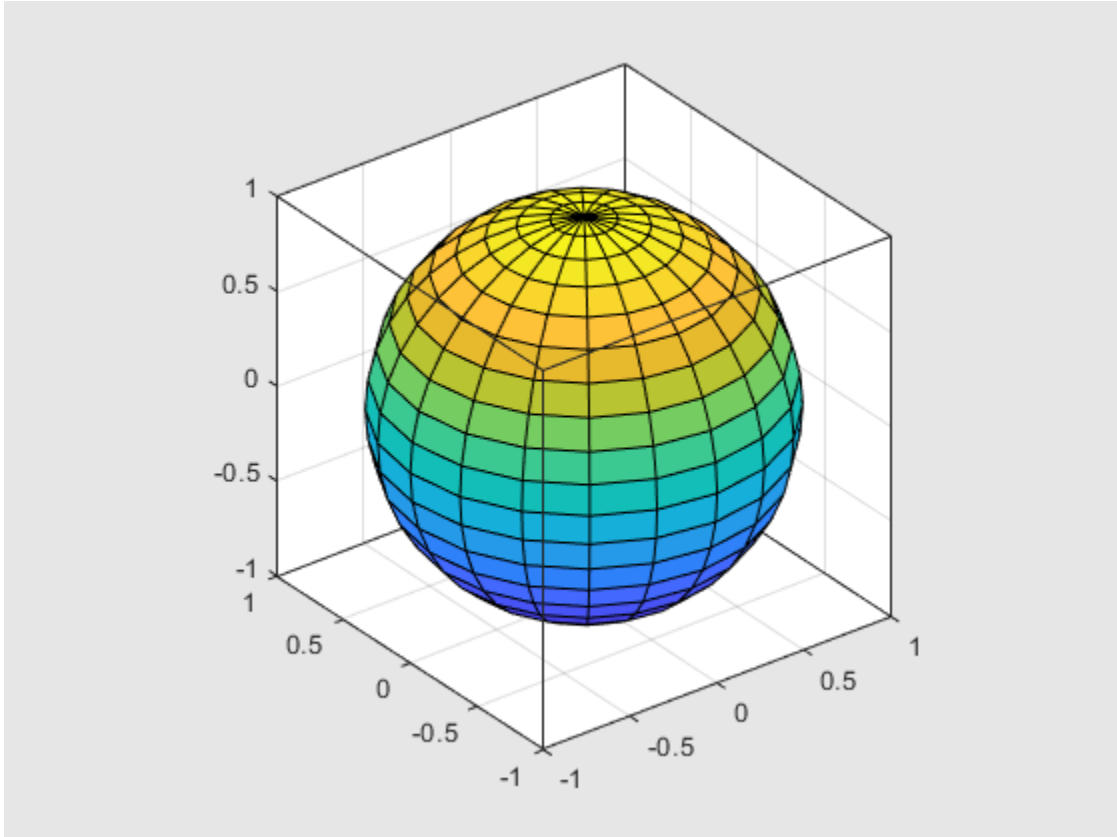

Changing the size and shape of the figure does not change the aspect ratio of the axes.

```
f = gcf;
f.Position(3) = f.Position(3) * 0.67;
```
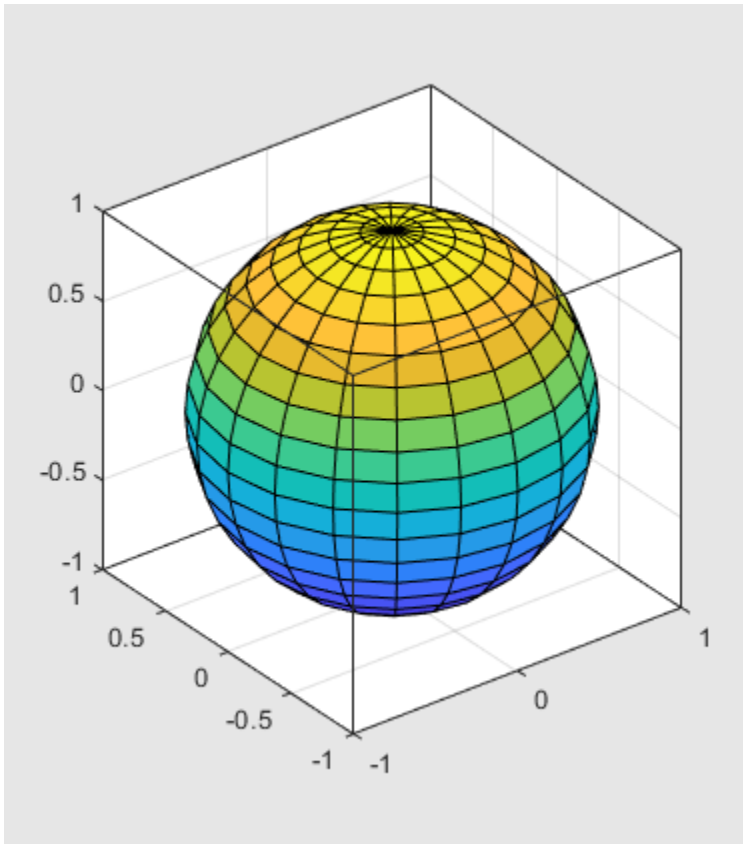
Setting the `CameraViewAngle` property disables stretch-to-fill, and also prevents MATLAB from readjusting the size of the axes if you change the view.

Aspect Ratio Properties

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

Data Aspect Ratio

The `DataAspectRatio` property controls the ratio of the axis scales. For example, to display a surface plot of a mathematical expression MATLAB selects a data aspect ratio that emphasizes the function's values:

```
[X,Y] = meshgrid((-2:.15:2),(-4:.3:4));
Z = X.*exp(-X.^2 - Y.^2);
surf(X,Y,Z)
set(gca,'BoxStyle','full','Box','on')
```
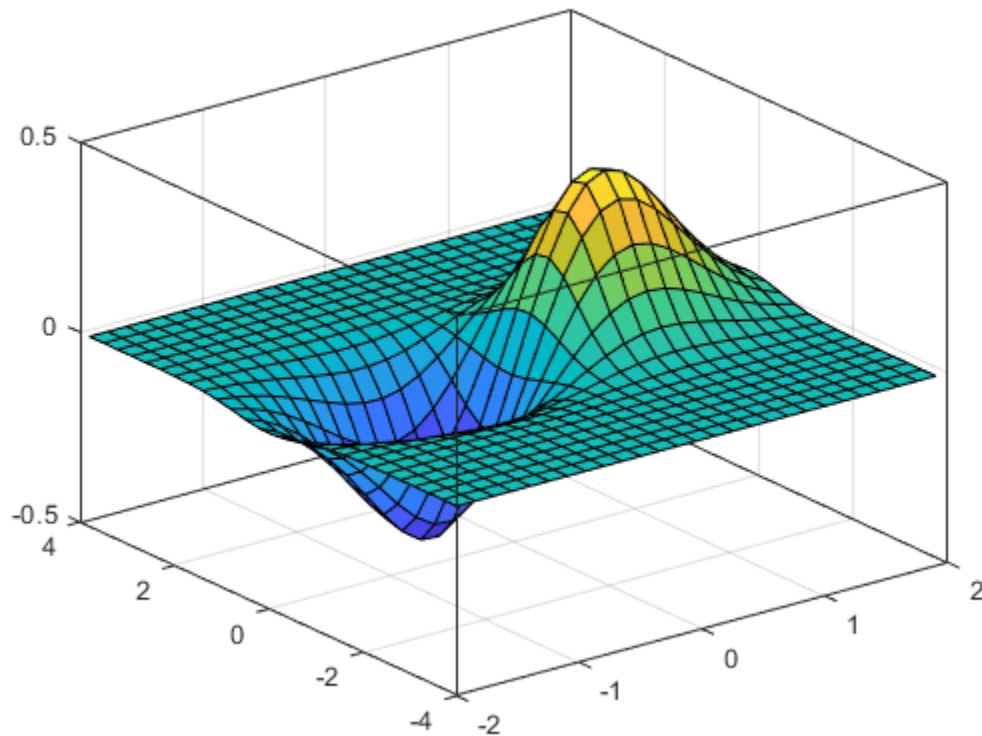
The `daspect` function returns the actual value of the `DataAspectRatio` property.

```
daspect
```

ans = *1×3*

```
    4     8     1
```

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

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

```
daspect([1 1 1])
```
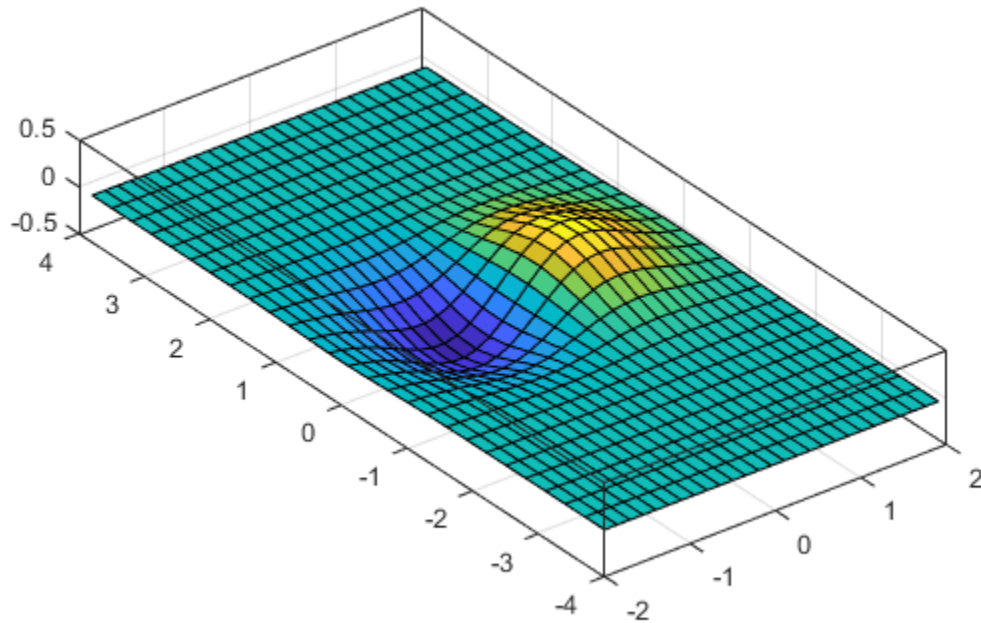
Setting the value of the `DataAspectRatio` property also sets the `DataAspectRatioMode` to `manual` and overrides stretch-to-fill so the specified aspect ratio is achieved.

Plot Box Aspect Ratio

Looking at the value of the `PlotBoxAspectRatio` for the graph in the previous section shows that it has now taken on the former value of the `DataAspectRatio`. The `pbaspect` function to returns the value of the `PlotBoxAspectRatio`:

```
pbaspect
```

```
ans = 1×3

    4    8    1
```

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

The `PlotBoxAspectRatio` property controls the shape of the axes plot box. By default, MATLAB sets this property to `[1 1 1]` and adjusts the `DataAspectRatio` property so that graphs fill the plot box or until reaching a constraint.

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

If you specify both the `DataAspectRatio` and the `PlotBoxAspectRatio`, MATLAB is forced to change the axis limits to obey the two constraints you have already defined.

Continuing with the mesh example, if you set both properties, MATLAB changes the axis limits to satisfy the two constraints placed on the axes.

```
daspect([1 1 1])
pbaspect([1 1 1])
```
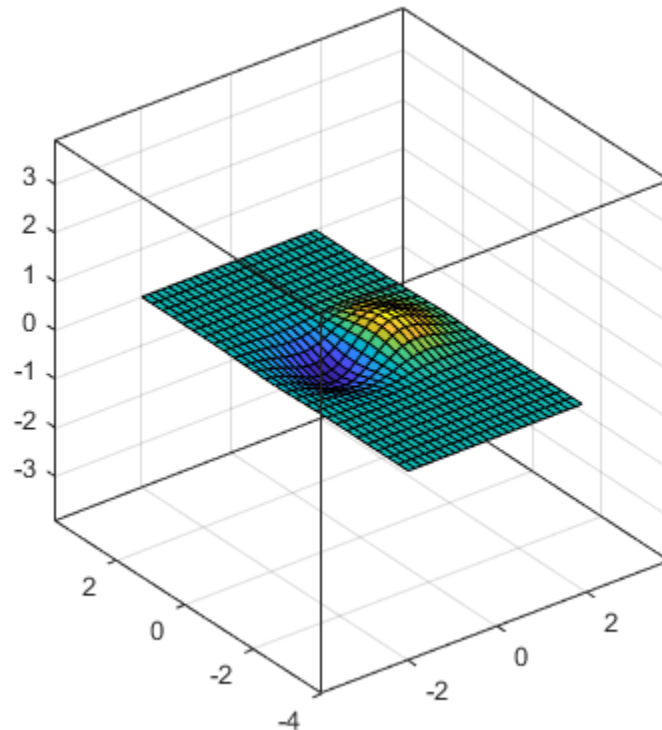

Adjusting Axis Limits

The axes also has properties for setting the *x*-, *y*-, and *z*-axis limits. However, specifying the axis limits with the `PlotBoxAspectRatio` and `DataAspectRatio` properties overconstrains the axes. For example, this command specifies axis limits that conflict with the `PlotBoxAspectRatio` value.

```
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],...
        'XLim',[-4 4],...
        'YLim',[-4 4],...
        'ZLim',[-1 1])
```
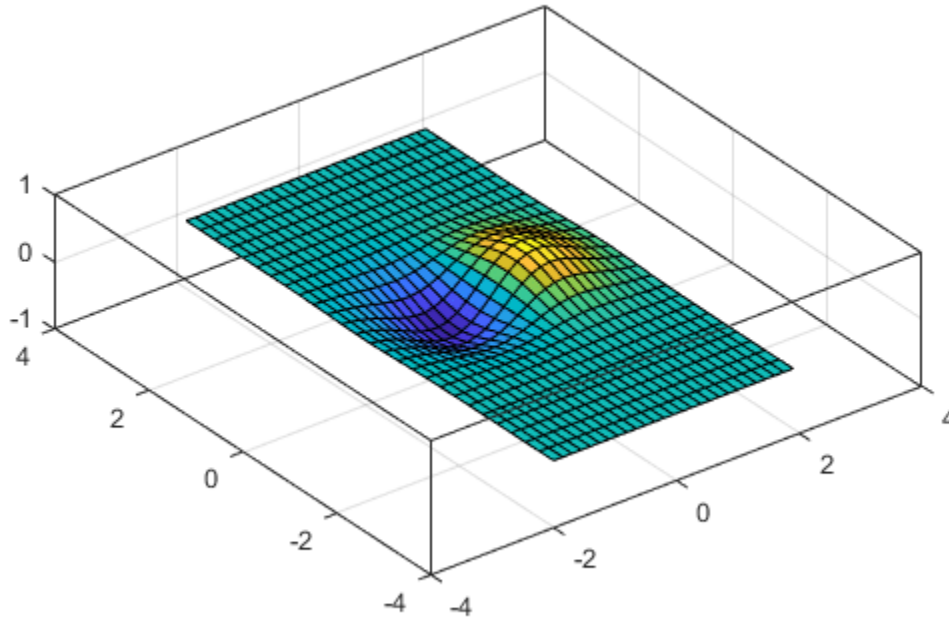
If you query the plot box aspect ratio, you can see that the `PlotBoxAspectRatio` value changed to accommodate the axis limits.

```
pbaspect
```

ans = *1×3*

     4   4   1

Displaying Real Objects

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

```
vert = [0 0 0; 0 1 0; 1 1 0; 1 0 0; 0 0 1; 0 1 1; 1 1 4; 1 0 4];
fac = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
patch('Vertices',vert,'Faces',fac,...
    'FaceColor',[0.7 0.7 0.7],'EdgeColor','k')
view(3)
```
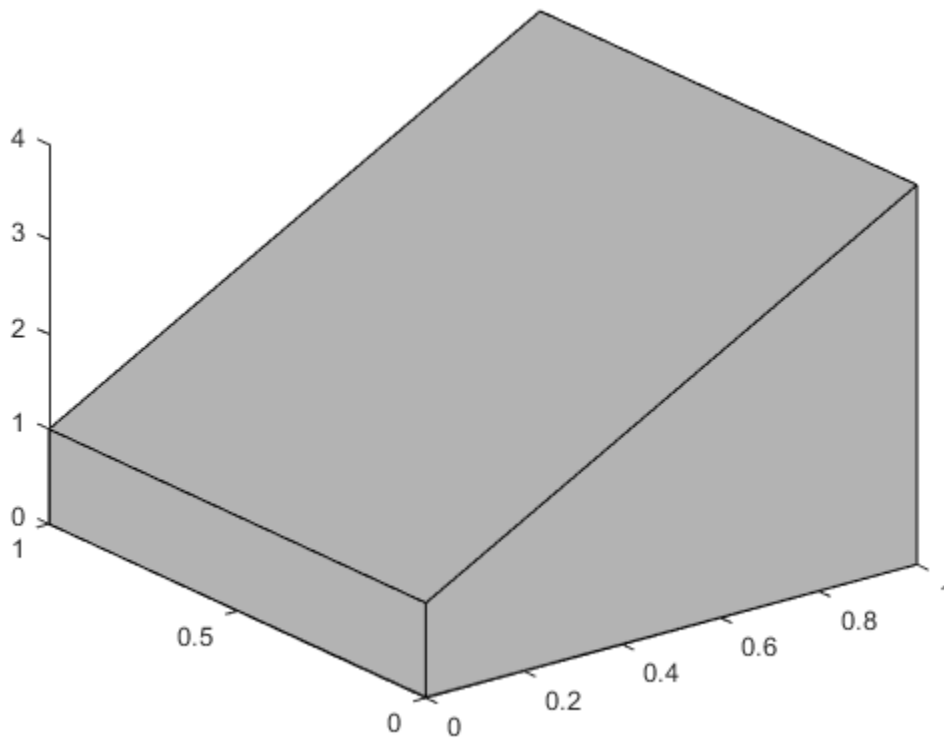
However, this axes distorts the actual shape of the solid object defined by the data. To display it in correct proportions, set the `DataAspectRatio`. Setting this property makes the units equal in the x-, y-, and z-directions and prevents the axes from being stretched to fill the position rectangle, revealing the true shape of the object.

```
set(gca,'DataAspectRatio',[1 1 1])
```
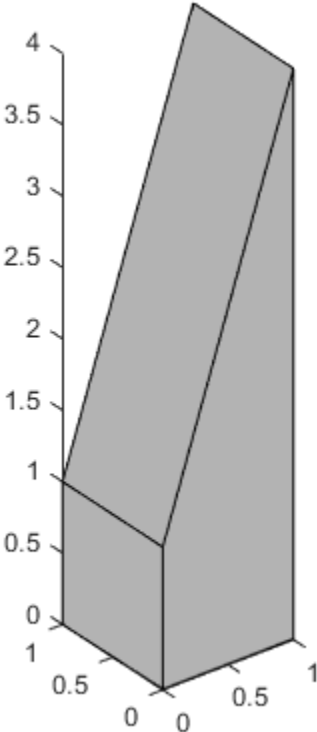
Control Colors, Line Styles, and Markers in Plots

In this section...

When you plot multiple data sets together in the same axes, MATLAB automatically assigns different colors (and possibly line styles and markers) to the plot objects. You can customize the colors, line styles, and markers when you call plotting functions.

For example, this code plots a solid red line and a dashed green line with circular markers.

```
plot([0 1 2],'-r')
hold on
plot([2 1 0],'--og')
hold off
```

You can also change the color, line style, and marker by setting properties on the object after creating it. For example, this code creates a line and then changes it to a green dashed line with circular markers.

```
p = plot([0 1 2]);
p.Color = 'g';
p.LineStyle = '--';
p.Marker = 'o';
```

These techniques are useful for customizing just a few lines. However, they are less flexible in other situations, such as plotting data in a loop, or passing matrix data to plotting functions. In such cases, you can change the properties that control how MATLAB automatically assigns colors, line styles, and markers.

Note Some of the functionality in the following examples is available starting in R2019b, and some of the functionality is available starting in R2020a. To modify plot colors and line styles in an earlier release, see Why Are Plot Lines Different Colors? and Line Styles Used for Plotting — LineStyleOrder.

How Automatic Assignment Works

MATLAB assigns colors to plot objects (such as `Line`, `Scatter`, and `Bar` objects) by cycling through the colors listed in the `ColorOrder` property of the axes. The `ColorOrder` property contains an array of RGB triplets, where each RGB triplet defines a color. The default `ColorOrder` array contains seven colors. If you create more objects than there are colors, the colors repeat.

If the plot objects support line styles and markers, MATLAB also cycles through the list in the `LineStyleOrder` property of the axes. The `LineStyleOrder` property contains a cell array of character sequences, where each character sequence corresponds to a line style (or a line style combined with a marker). The default `LineStyleOrder` array contains only the solid line style, (`'-'`). All of the colors in the `ColorOrder` array are used with one character sequence in the `LineStyleOrder` array before the next sequence is used. The cycle continues for each new plot object. If there are more objects than combinations of colors and character sequences, then the cycle repeats.

For a given pair of `ColorOrder` and `LineStyleOrder` arrays, the colors, line styles, and markers for a particular plot object are determined by the value of the object's `SeriesIndex`, which is a new property starting in R2020a. By default, the `SeriesIndex` property is a number that corresponds to the object's order of creation, starting at `1`. MATLAB uses the number to calculate indices into the `ColorOrder` and `LineStyleOrder` arrays.

For example, create an axes object with two colors in its `ColorOrder` array (red and blue) and two line styles in its `LineStyleOrder` array (solid and dashed). Then plot five lines.

```
ax = axes;
ax.ColorOrder = [1 0 0; 0 0 1];
ax.LineStyleOrder = {'-','--'};

hold on
for i = 1:5
    plot([i i+2])
end
hold off
```
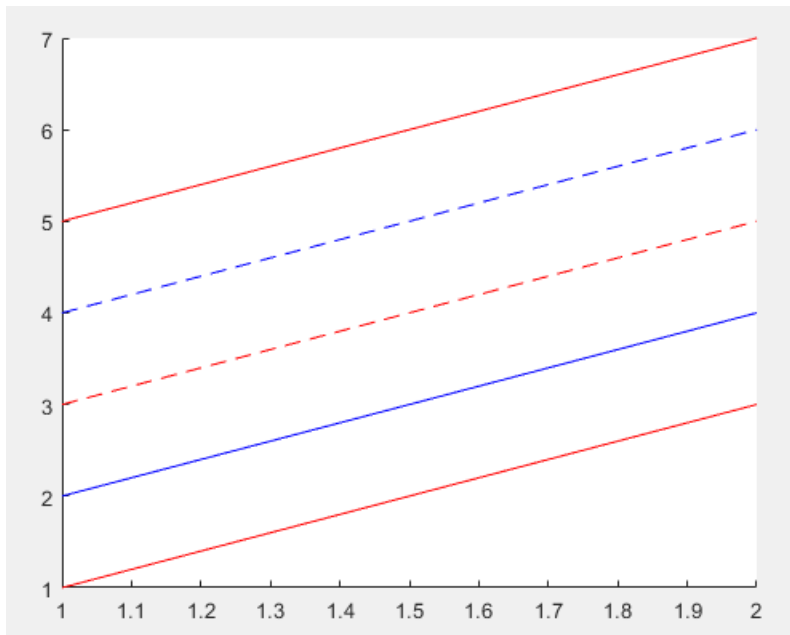

This table lists the `SeriesIndex`, the index into the `ColorOrder` array, and the index into the `LineStyleOrder` array for each line in the preceding plot.

	SeriesIndex	Index into ColorOrder Array	Index into LineStyleOrder Array	Line Appearance
First Line	1	1	1	Red solid line
Second Line	2	2	1	Blue solid line
Third Line	3	1	2	Red dashed line
Fourth Line	4	2	2	Blue dashed line
Fifth Line	5	1	1	Red solid line

You can change the colors, line styles, and markers of plot objects by modifying the `ColorOrder` or `LineStyleOrder` properties of the axes, or by changing the `SeriesIndex` properties of the plot objects.

Changing Color Schemes and Line Styles

Changing the `ColorOrder` property of the axes changes the color scheme of your plot. Changing the `LineStyleOrder` property of the axes changes the line styles (and possibly markers) used in your plot. For example, plot eight lines in a loop using the default colors and line style.

```
ax = axes;
hold on
for i = 0:7
    plot([i i+2])
end
hold off
```
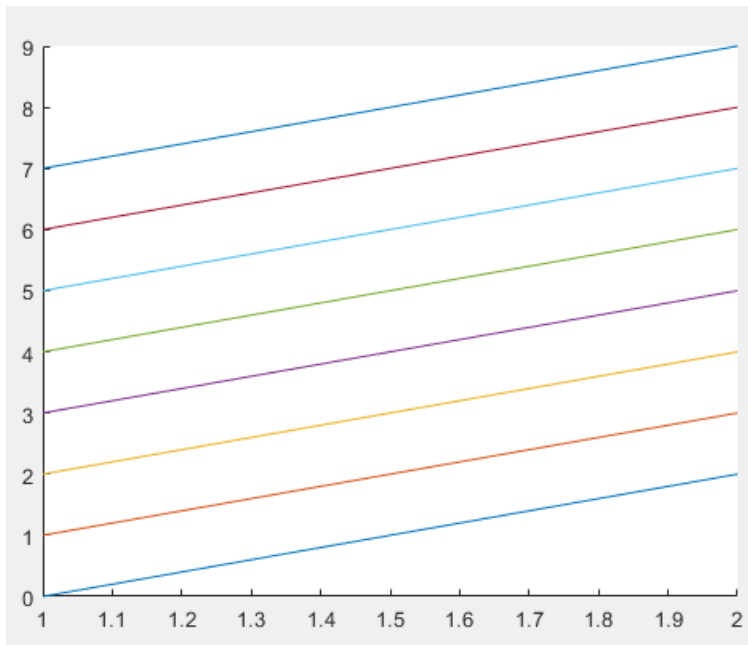

Replace the `ColorOrder` array with a new array that contains four colors (you can also replace this array using the `colororder` function). Then replace the `LineStyleOrder` array with a new cell array that contains two line styles. The lines automatically use the new colors and line styles.

```
% Updates existing plots in R2019b or later
ax.ColorOrder = [1 0 0; 0 1 0; 0 0 1; 0 0 0];
ax.LineStyleOrder = {'-','--'};
```
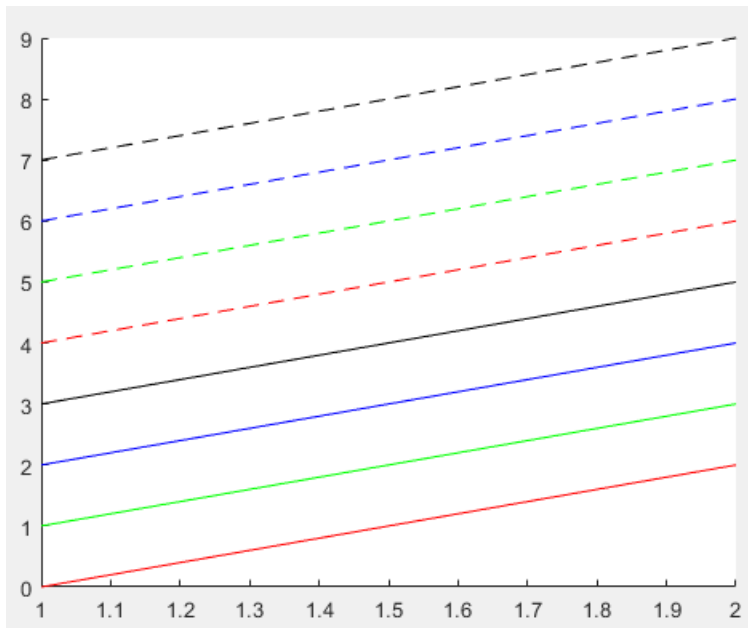
Changing Indices into the ColorOrder and LineStyleOrder Arrays

Changing the `SeriesIndex` property on a plot object changes the indices into the `ColorOrder` and `LineStyleOrder` arrays. Changing the indices is useful when you want the color, line style, and marker of an object to match another object.

For example, plot four sine waves in a loop, varying the wavelength and phase. For each sine wave, set the `SeriesIndex` property according to the wavelength. In the resulting plot, the sine waves that have the same wavelength also have the same color.

```
x = linspace(0,10,200);
ax = axes;
hold on
for phi = 0:3:3
    for t = 1:2
        plot(x,sin(x/t + phi),'SeriesIndex',t) % Requires R2020a or later
    end
end
hold off
```
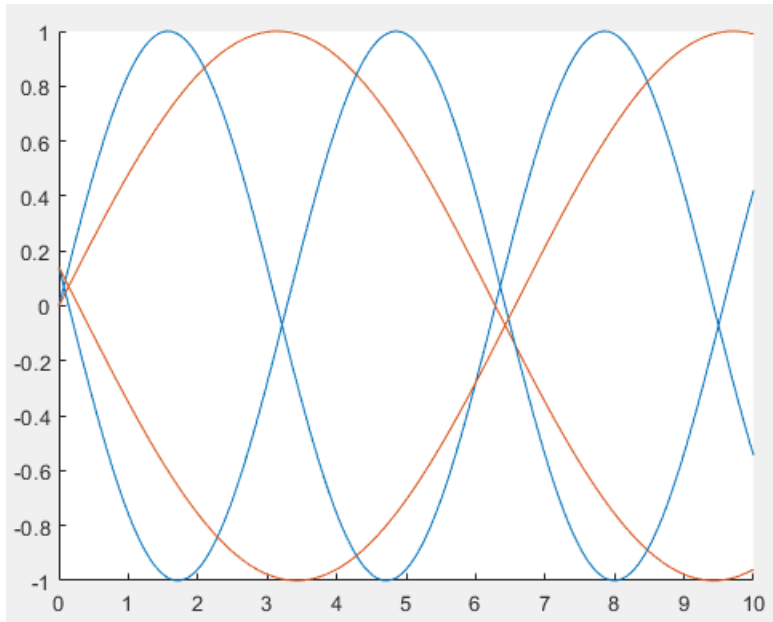
To make one pair of sine waves more prominent, change the color order to different set of colors.

```
ax.ColorOrder = [0.8 0.8 0.9; 0.2 0.2 0.8];
```
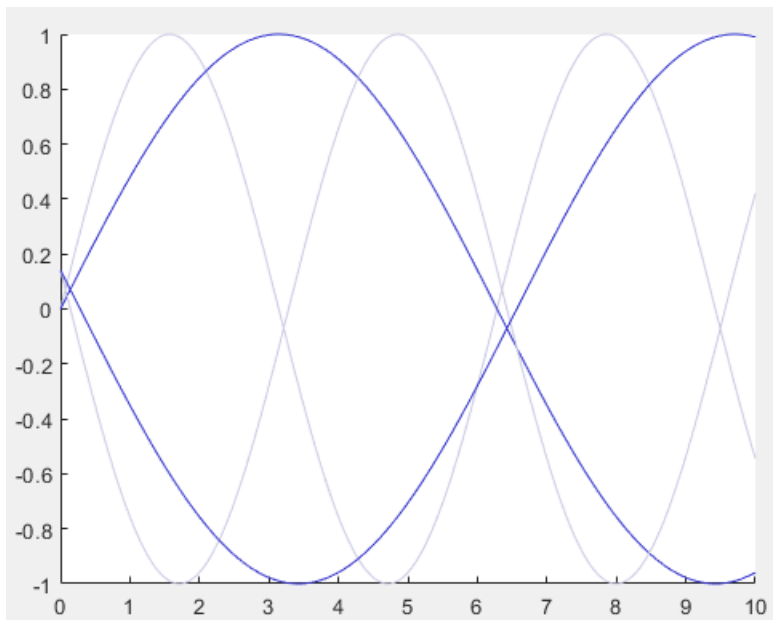

See Also

Functions
colororder | gca | plot

Properties
Axes

Clipping in Plots and Graphs

This example shows how MATLAB® uses clipping in plots and how to control clipping.

What is Clipping?

Clipping occurs when part of a plot occurs outside the boundaries of an axes. In MATLAB®, the part of the plot that is clipped does not appear on the screen or in printed output. The axis limits of the plot determine the boundaries.

Turn Clipping Off

By default, MATLAB clips plots that extend outside of the axes limits.

```
figure
surf(peaks)
zlim([-4 4])
```
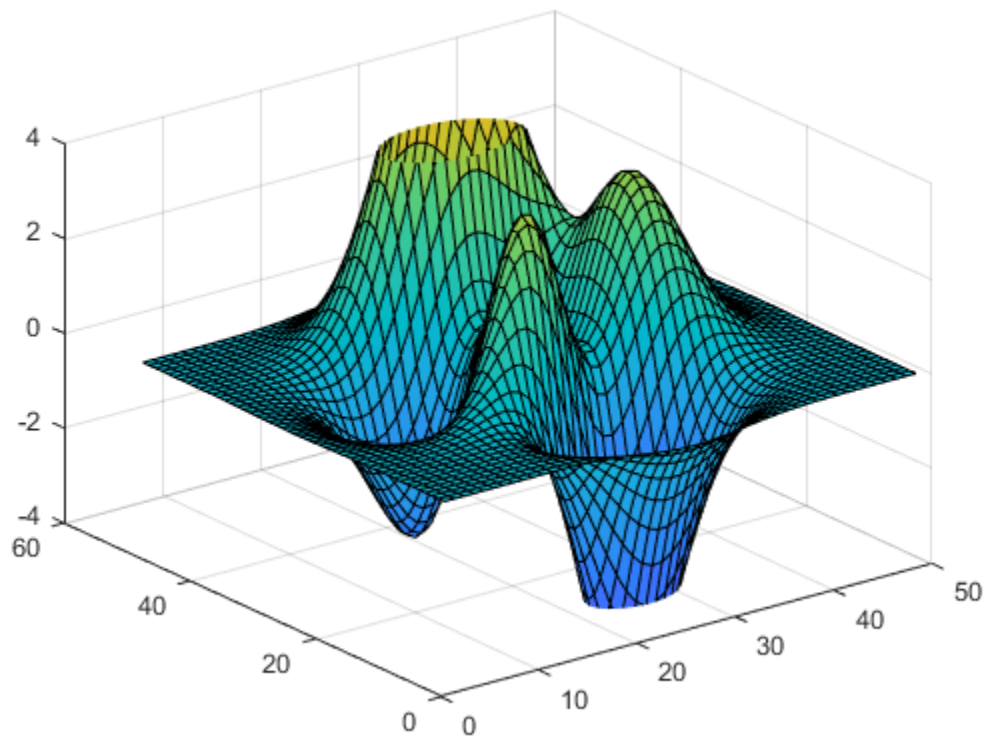

Use the axes `Clipping` property to control the clipping behavior.

```
ax = gca;              % get the current axis
ax.Clipping = 'off';   % turn clipping off
```
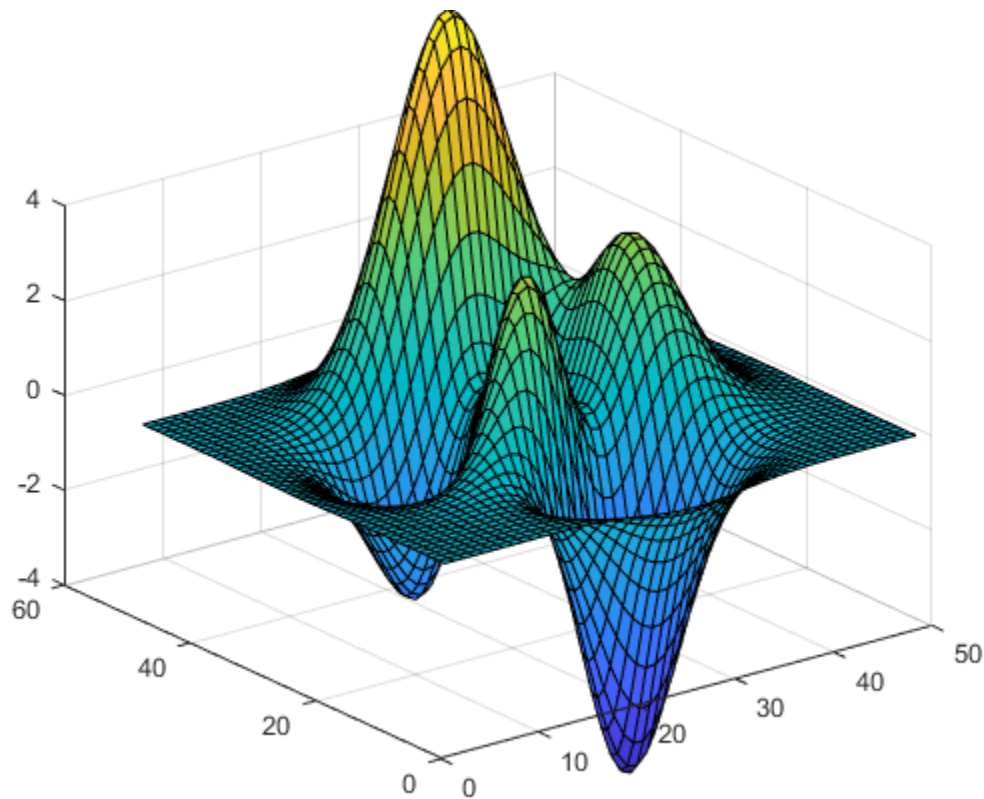
Control the Clipping Style

Use the `ClippingStyle` property to control the way clipping works. If the `ClippingStyle` is set to `'3dbox'`, then MATLAB clips the plots to the volume defined by the limits of the x, y, and z axes. If the `ClippingStyle` is set to `'rectangle'`, then MATLAB clips the plots to an imaginary rectangle drawn around the outside of the x, y, and z axes. The plots below show the difference between the two clipping styles.
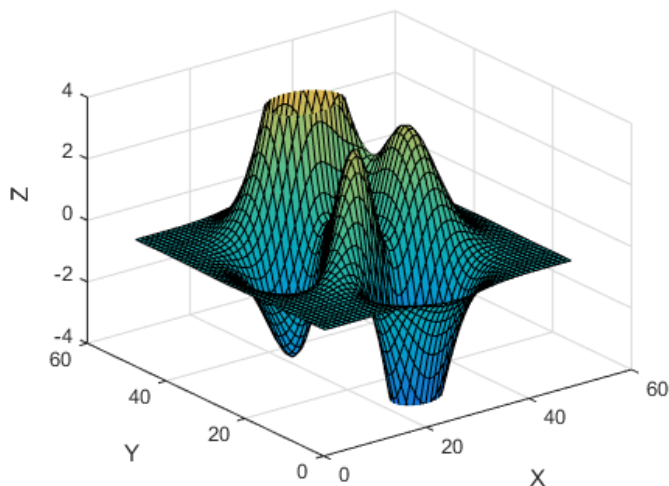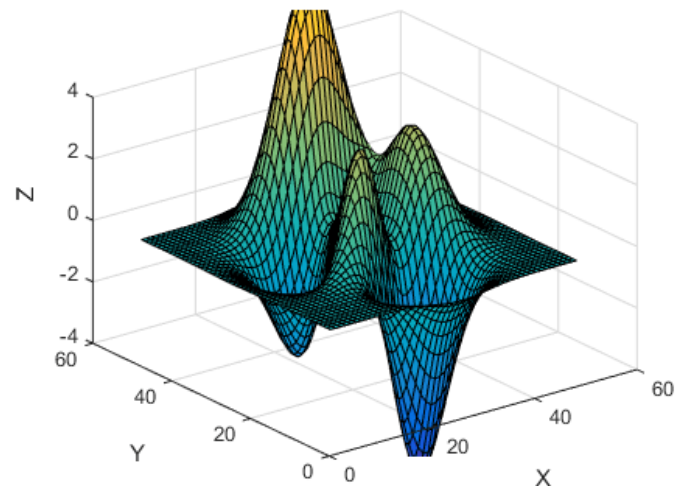
Clipping in 2D plots

Clipping is also used in 2D plots. For example, MATLAB clips the sine wave in the plot below.

```
x = -pi:pi/20:pi;
y = sin(-pi:pi/20:pi);
plot(x,y)
ylim([-0.9 0.9])
```
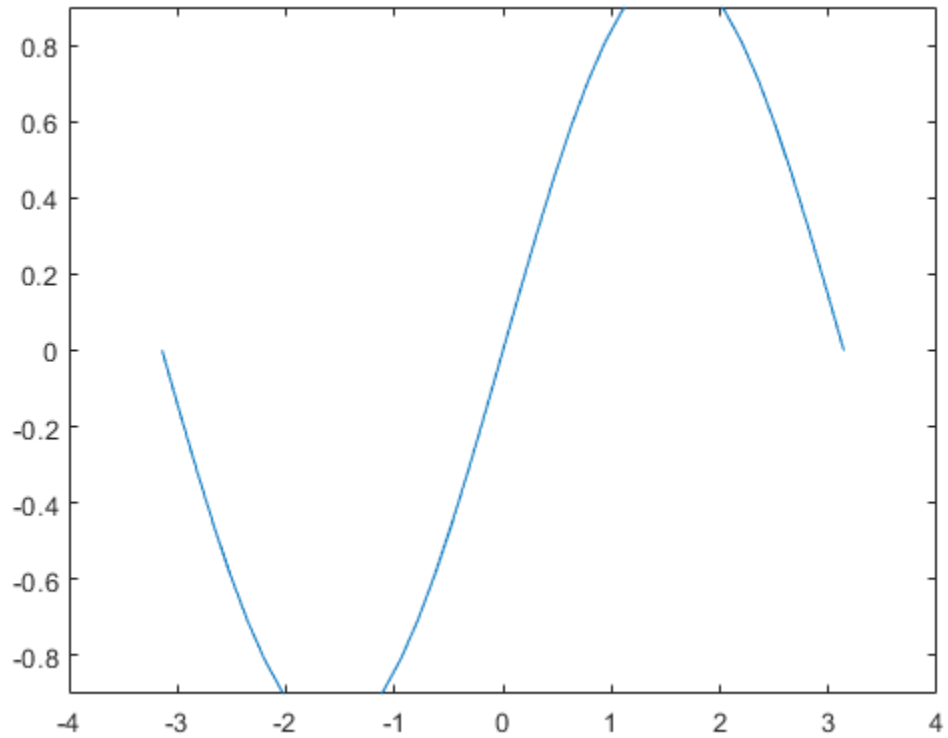

If clipping is turned off, then MATLAB displays the entire sine wave.

```
ax = gca;
ax.Clipping = 'off';
```
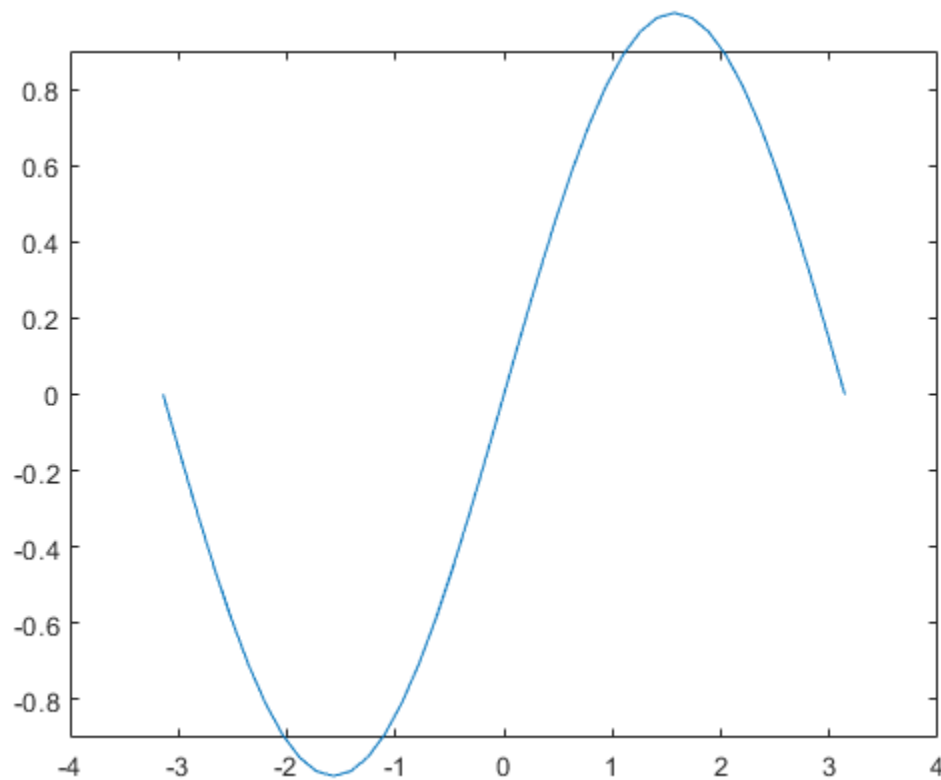
Clipping and Markers

Clipping does not affect markers drawn at each data point as long as the data point itself is inside the x and y axis limits of the plot. MATLAB displays the entire marker even if it extends slightly outside the boundaries of the axes.

```
p = plot(1:10,'*');
p.MarkerSize = 10;
axis([1 10 1 10])
```
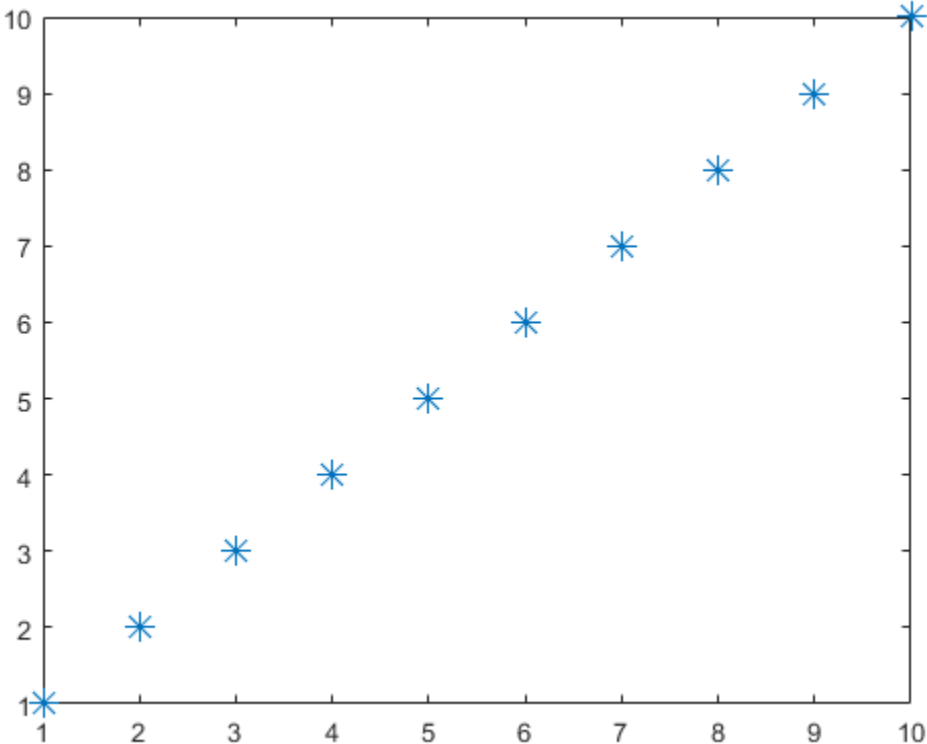
Using Graphics Smoothing

This example shows how to use graphics and font smoothing in MATLAB plots.

What is Graphics Smoothing?

Graphics smoothing improves the appearance of graphics in plots. Smoothing removes jagged edges that result from using pixels or dots to represent continuous objects. Techniques used for graphics smoothing include multi-sampling and anti-aliasing.

Graphics Smoothing in a Figure

Graphics smoothing is controlled in a figure by using the `GraphicsSmoothing` property. By default, the `GraphicsSmoothing` property is set to 'on'.

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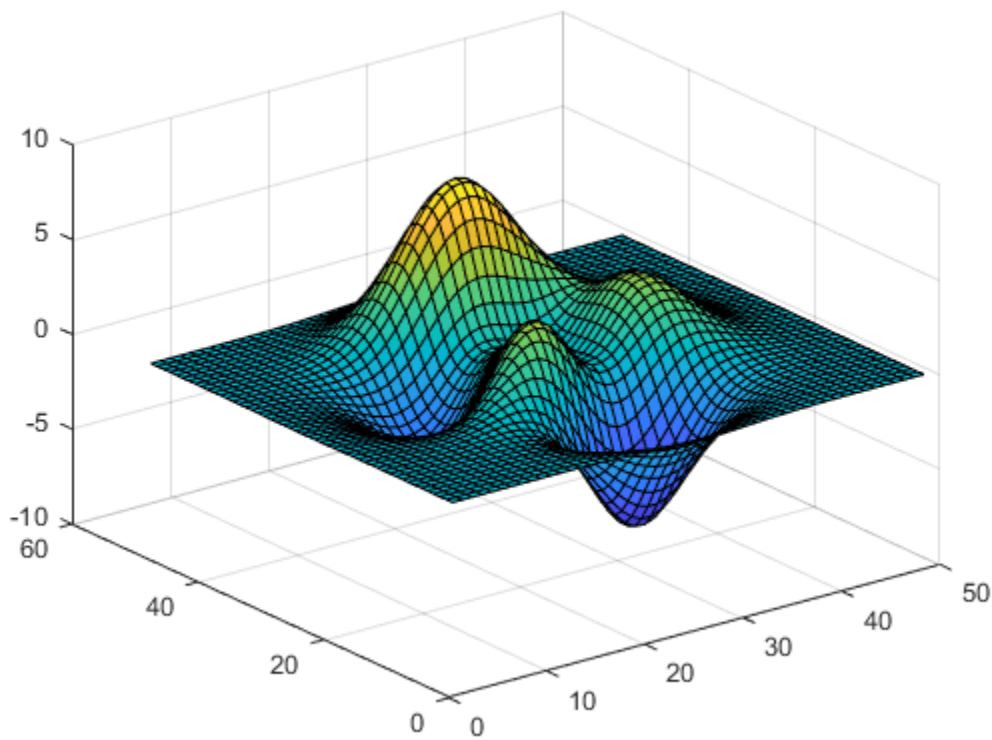

You can turn off graphics smoothing by setting the `GraphicsSmoothing` property to 'off'.

```
f.GraphicsSmoothing = 'off';
```
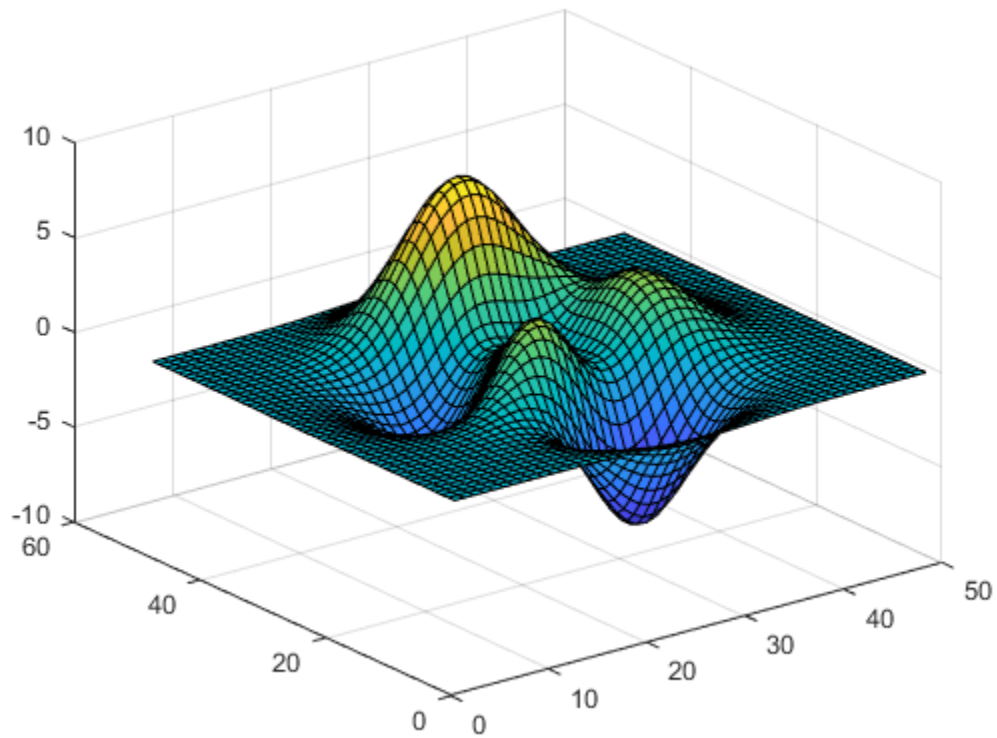
Font Smoothing for Text and Axes Objects

The `FontSmoothing` property of a text or an axes object controls how text is rendered. When `FontSmoothing` is set to 'on', text will be drawn with smoothed edges. Font smoothing is 'on' by default.

```
t = text(14,27,-8.5, 'Minimum of Peaks');
```

Minimum of Peaks
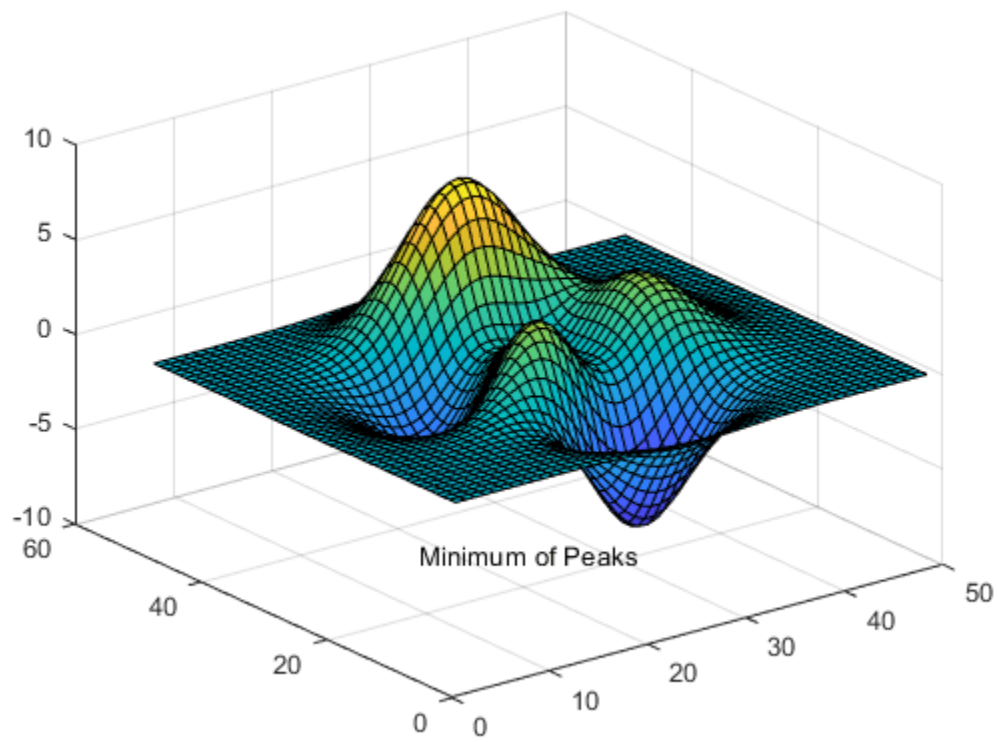
```matlab
t.FontSmoothing = 'off';
```
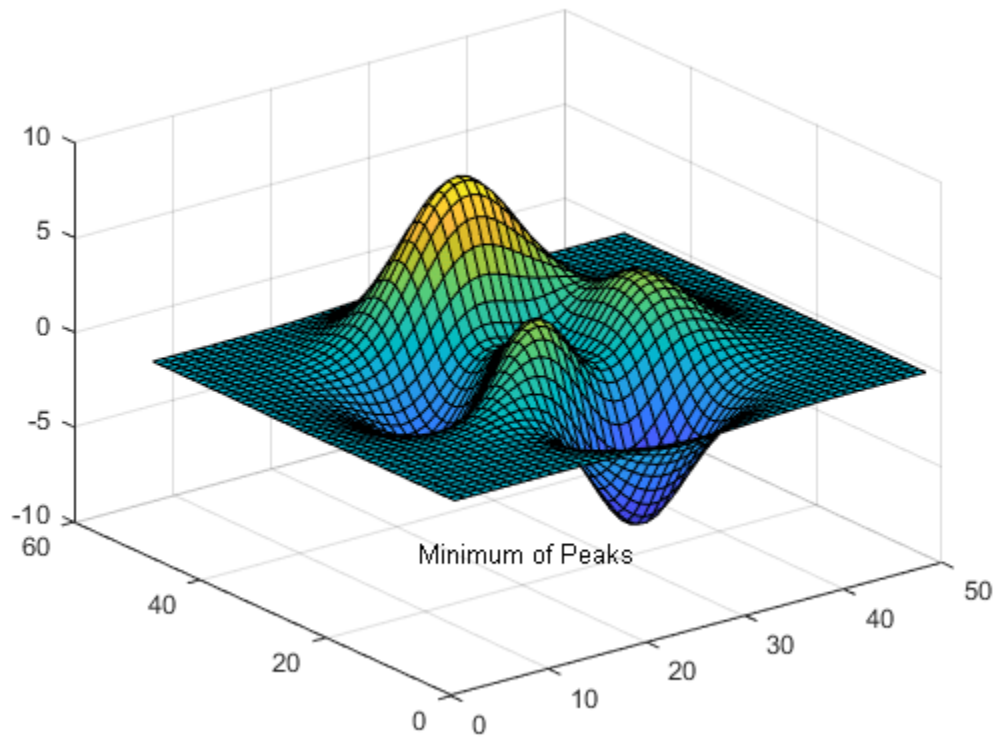
Why Turn Graphics Smoothing Off?

Without graphics smoothing, horizontal and vertical lines will appear sharper. Certain chart types may look better when graphics smoothing is turned off. Similarly, turning off font smoothing may make text using small fonts appear clearer.

```
pcolor(rand(6))
```
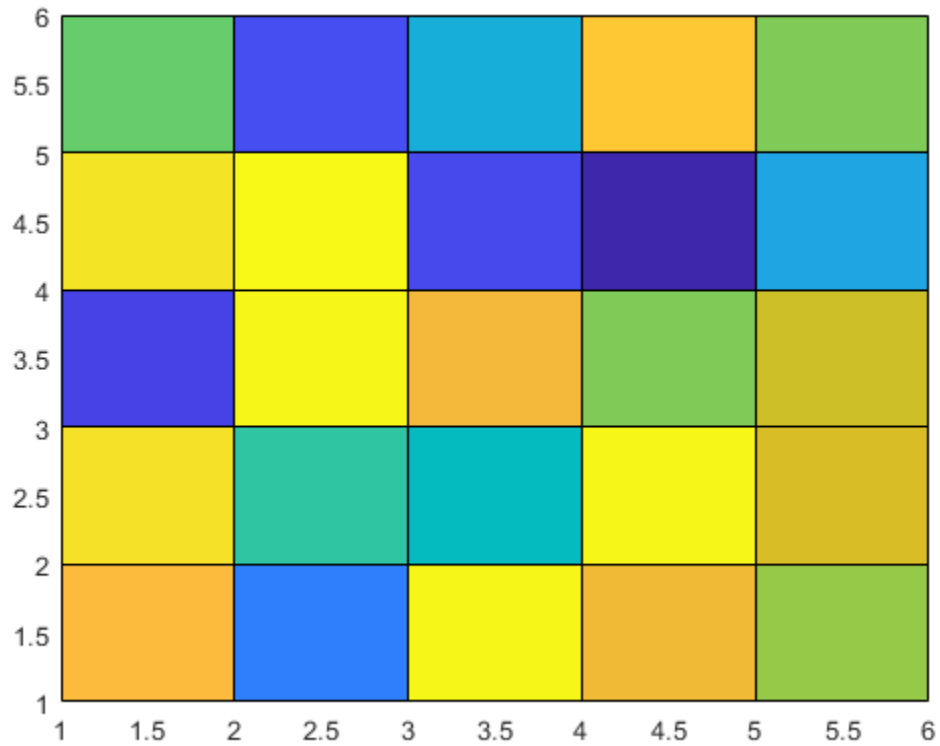
```
ax = gca;                        % get current axes
ax.FontSmoothing = 'off';        % turn off axes font smoothing
```
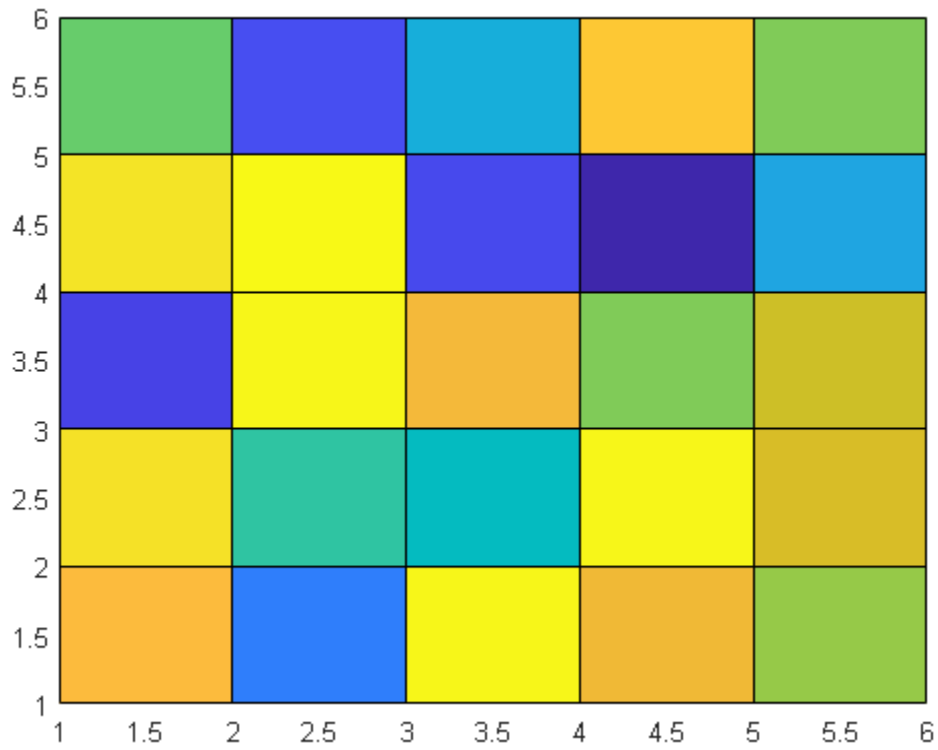
```
f.GraphicsSmoothing = 'off';    % turn off figure graphics smoothing
```

Coloring Graphs

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;
```
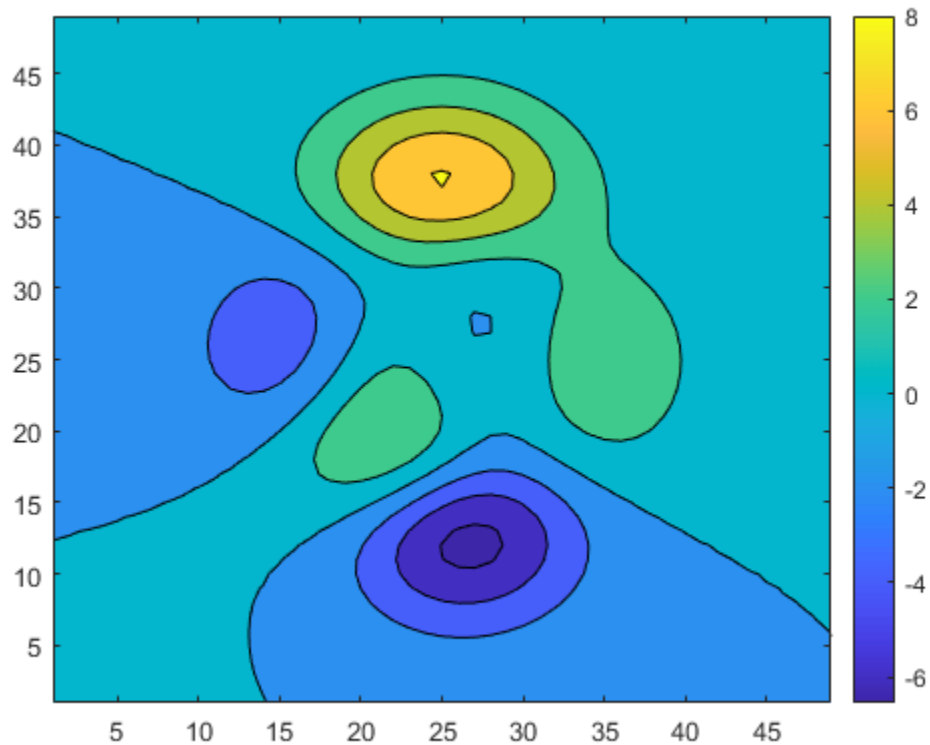

The default location of the colorbar is on the right side of the axes. However, you can move the colorbar to a different location by setting the `Location` property. In this case, the `'southoutside'` option places the colorbar below the axes.

```
c.Location = 'southoutside';
```
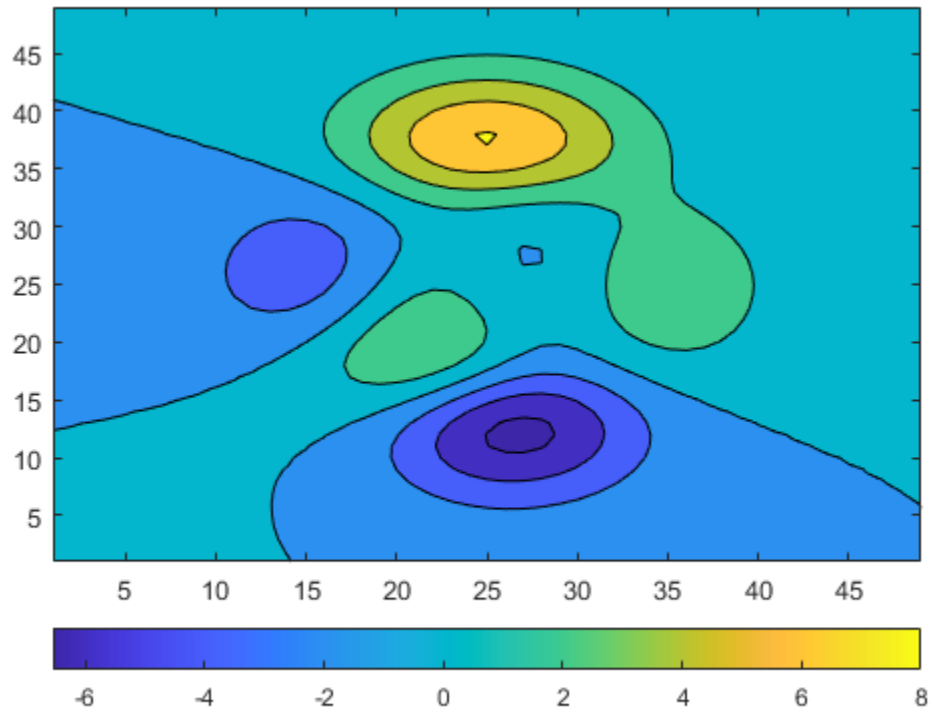
You can also change the thickness of the colorbar. The "Position" property controls the location and size of most graphics objects, including axes and colorbars. Because this colorbar is horizontal, the fourth value in `c.Position` (which corresponds to height) controls its thickness. Here, the colorbar 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;
```
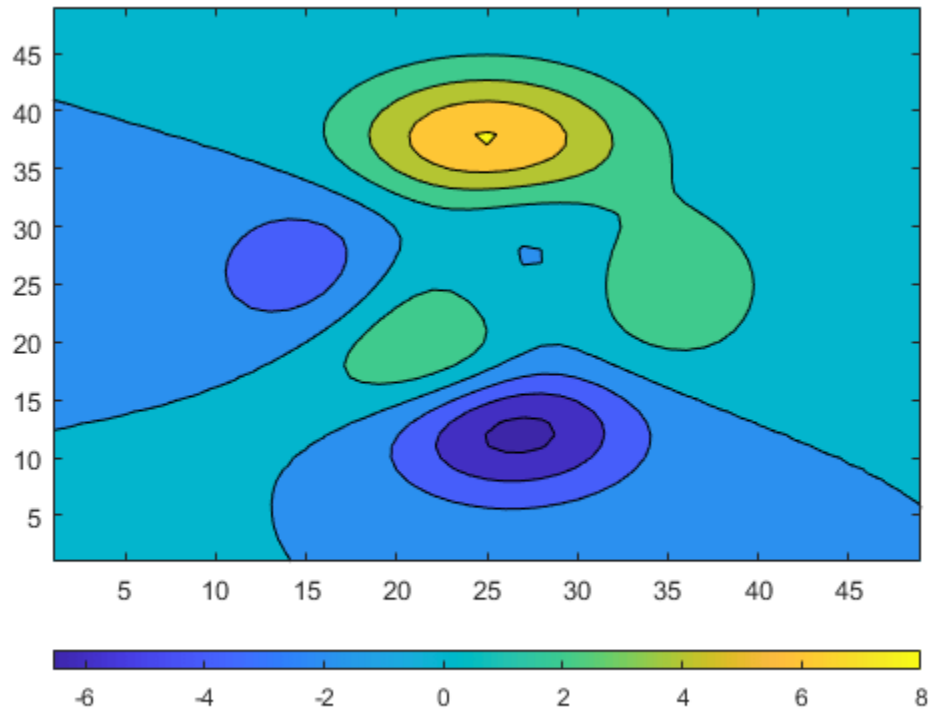
`Colorbar` objects have several properties for modifying the tick spacing and labels. For example, you can specify that the ticks occur in only three places: `-6.5`, `0`, and `8`.

```
c.Ticks = [-6.5 0 8];
```
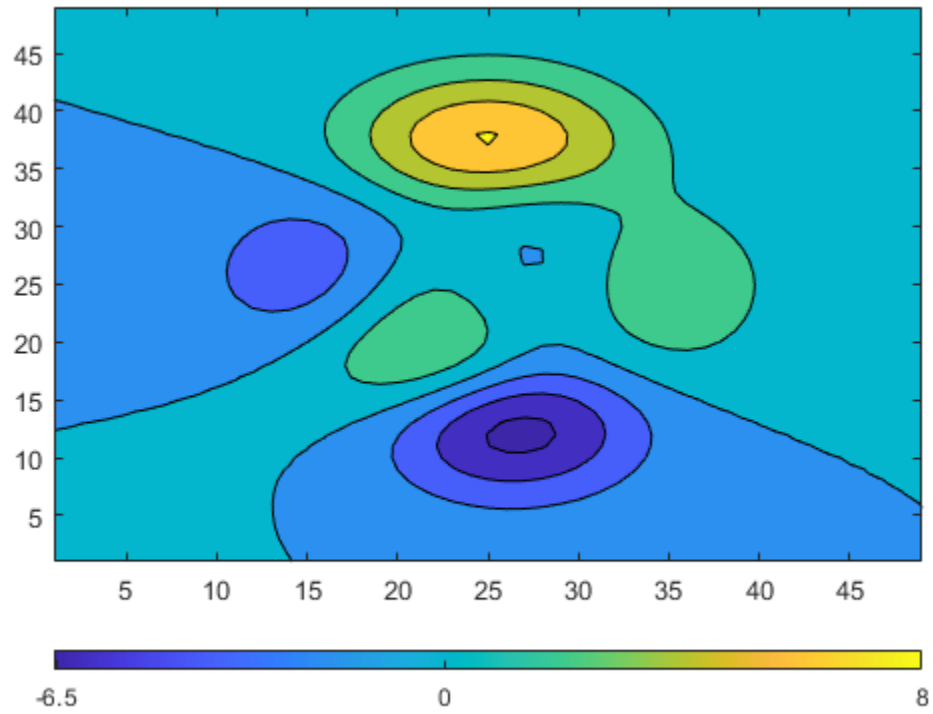
You can change the tick labels to any values. Use a cell array to specify the tick labels.

```
c.TickLabels = {'Frigid','Freezing','Cold'};
```
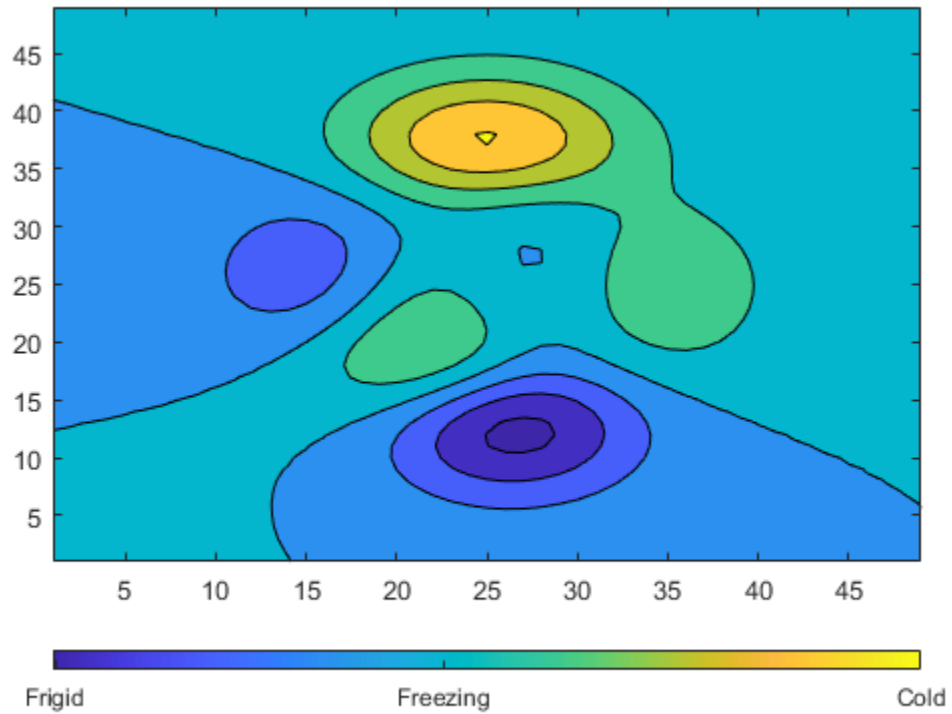
You can also use TeX or LaTeX markup. Use the `TickLabelInterpreter` property to set the interpreter when you use TeX or LaTeX.

```
c.TickLabelInterpreter = 'tex';
c.TickLabels = {'-6.5\circ','0\circ','8\circ'};
```
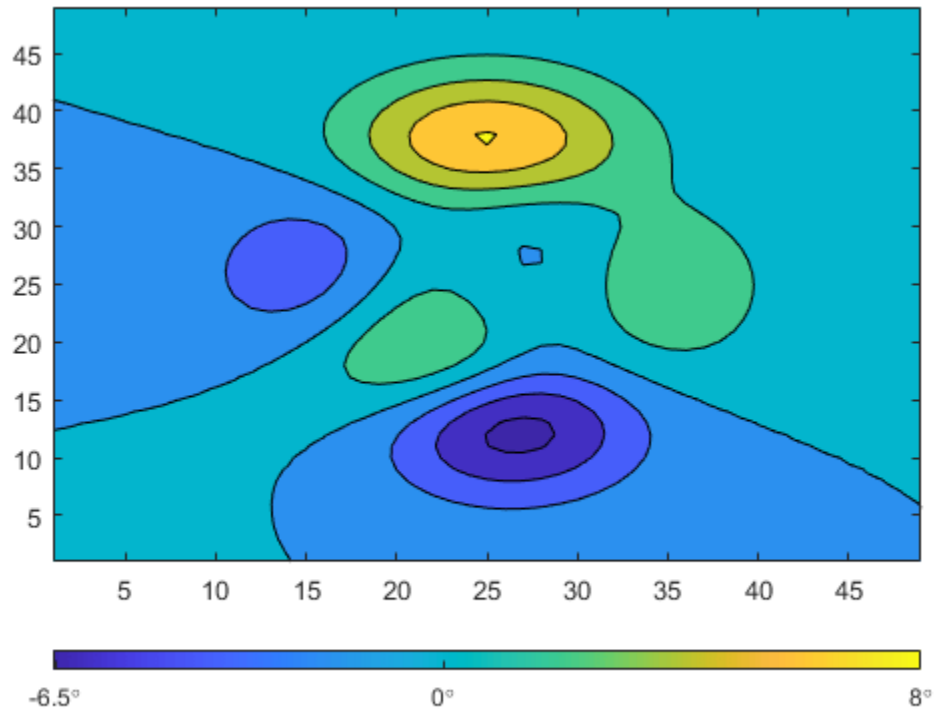
You can change the limits of the colorbar to focus on a specific region of color. For example, you can narrow the limits and adjust the tick labels to reflect the new limits. The resulting colorbar excludes the dark blue shades that used to be on the left and the yellow shades that used to be on the right.

```
c.Limits = [-4 4];
c.Ticks = [-4 0 4];
c.TickLabels = {'-4\circ','0\circ','4\circ'};
```
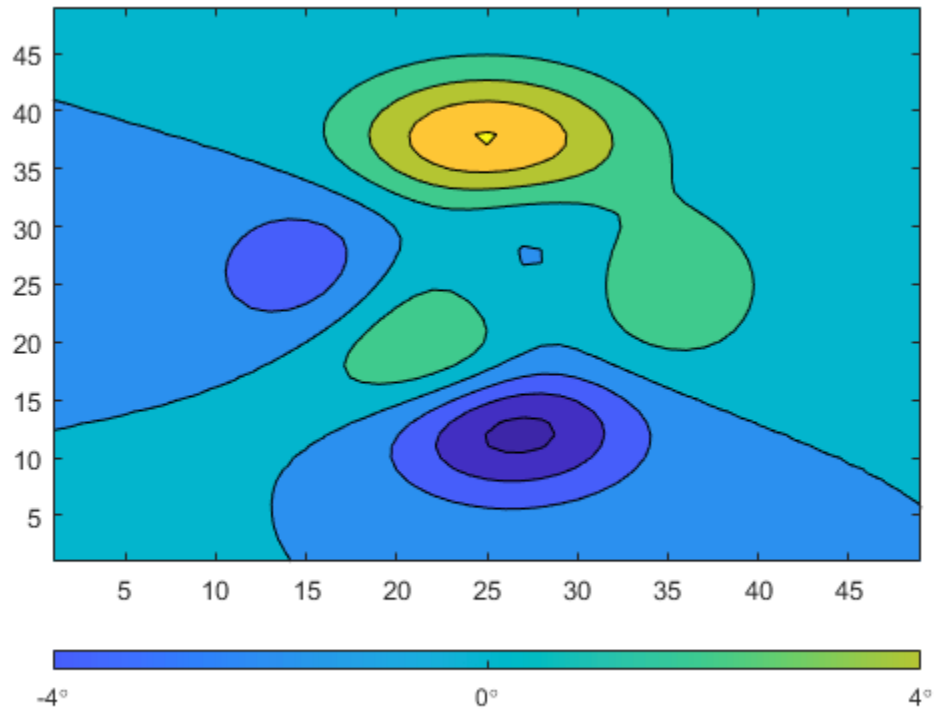
Add a descriptive label to the colorbar using the `Label` property. Because the `Label` property must be specified as a `Text` object, you must set the `String` property of the `Text` object first. Then you can assign that `Text` object to the `Label` property. The following command accomplishes both tasks in one step.

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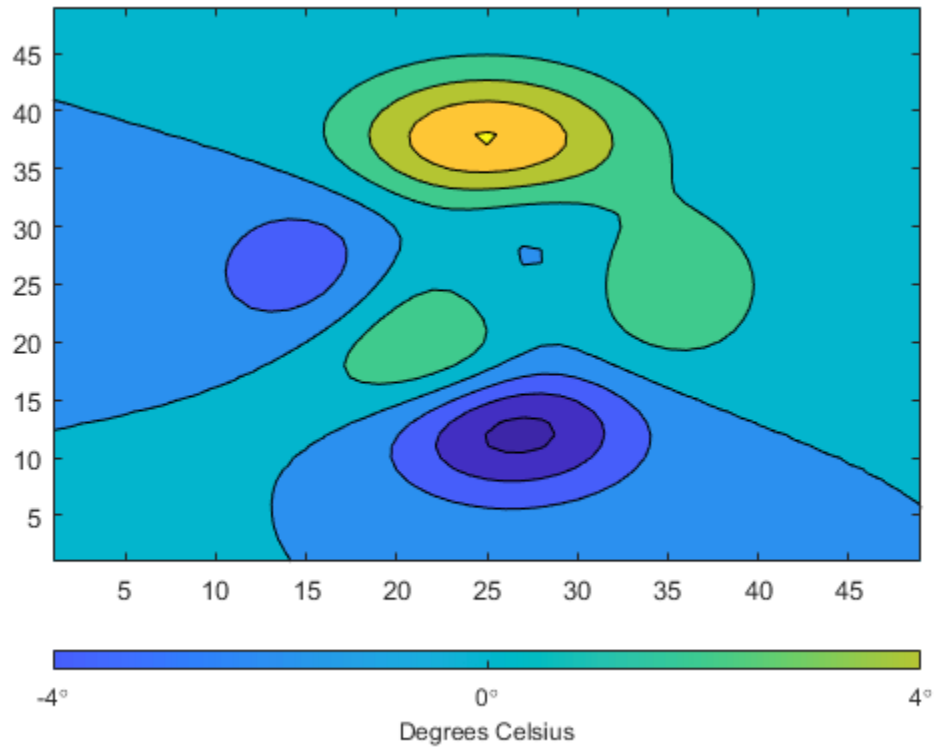
See Also

Functions
`colorbar` | `pcolor`

Properties
Colorbar

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. Colormaps are three-column arrays containing RGB triplets in which each row defines a distinct color.

For example, here is a surface plot with the default color scheme.

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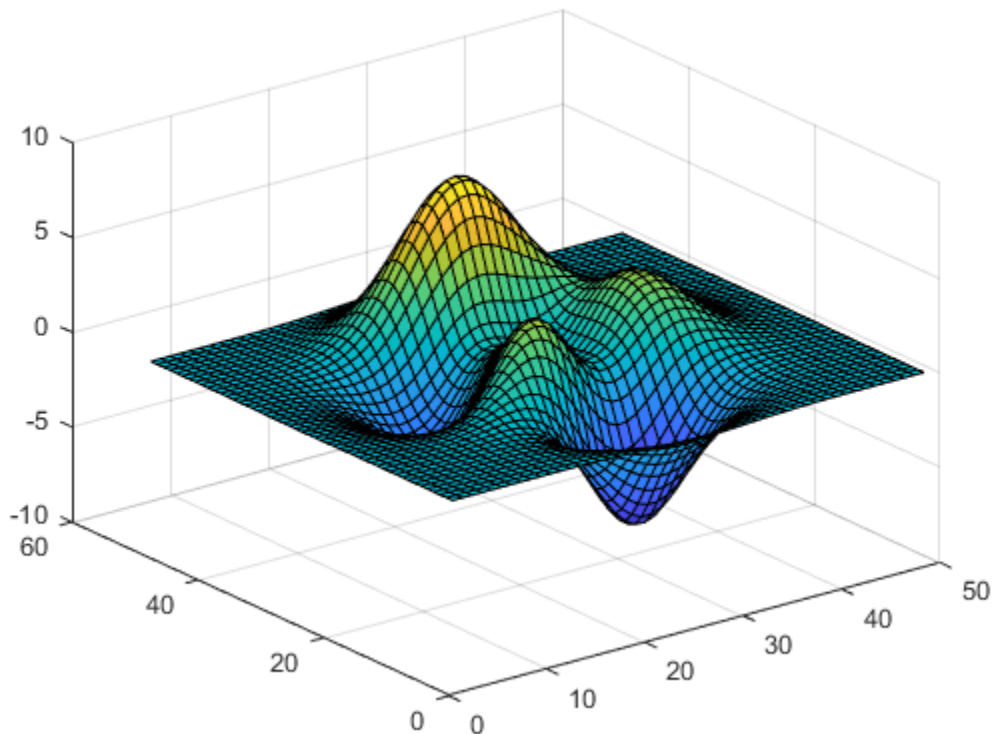

The following command changes the colormap of the current figure to `winter`, one of several predefined colormaps (see "Colormaps" for a full list).

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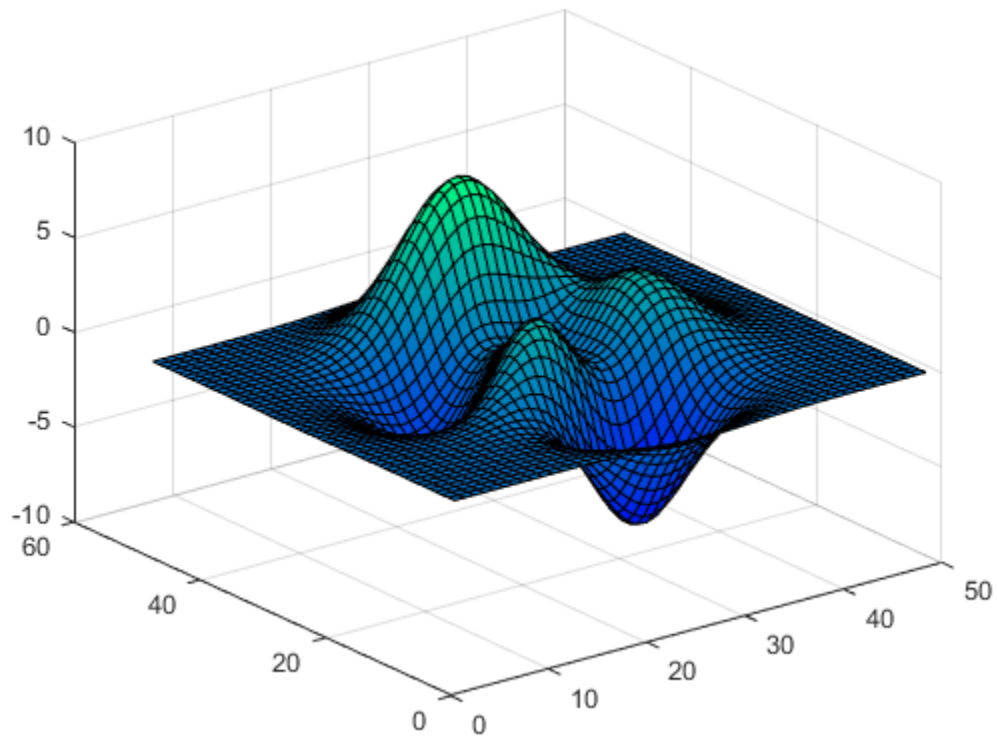
If you have multiple figures open, pass the `Figure` object as the first argument to the `colormap` function.

```
colormap(f,hot);
```
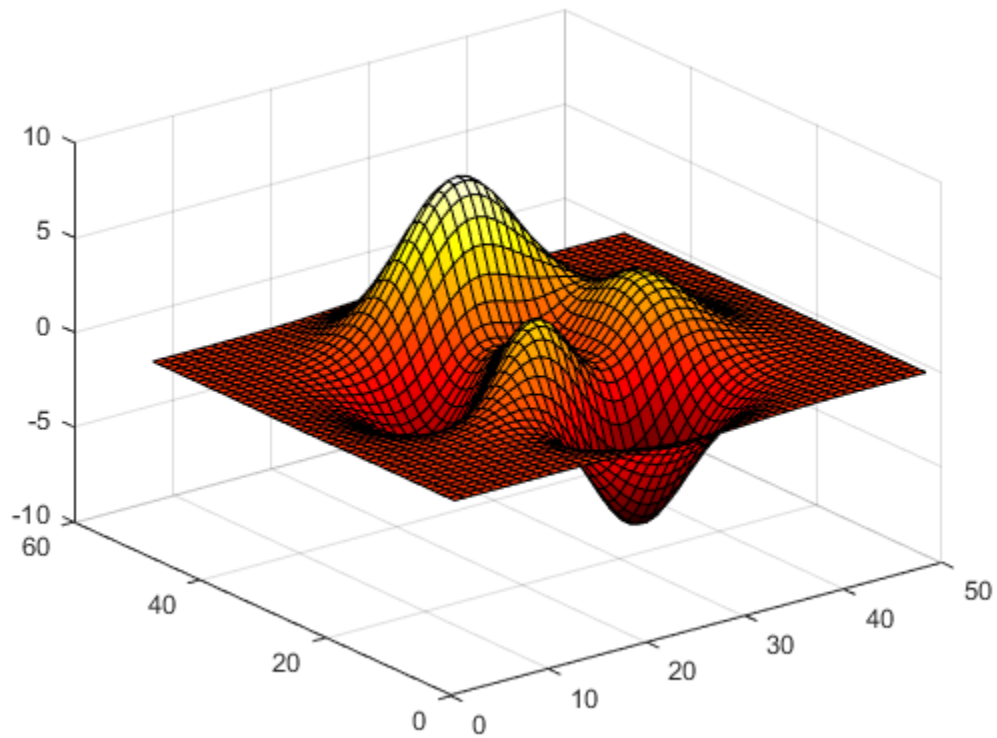
Each predefined colormap provides a palette of 256 colors by default. However, you can specify any number of colors by passing a whole number to the predefined colormap function. For example, here is the hot colormap with ten entries.

```
c = hot(10);
colormap(c);
```
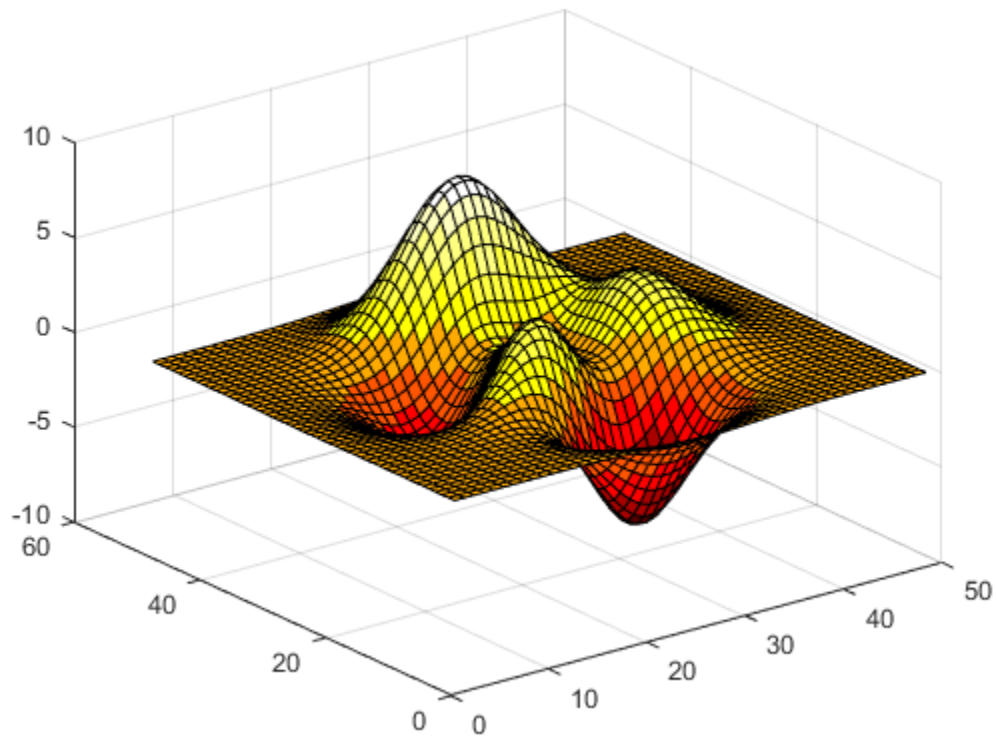
You can also create your own colormap as an m-by-3 array. Each row in the array contains the red, green, and blue intensities of a different color. The intensities are in the range [0,1]. Here is a simple colormap that contains three entries.

```
mycolors = [1 0 0; 1 1 0; 0 0 1];
colormap(mycolors);
```
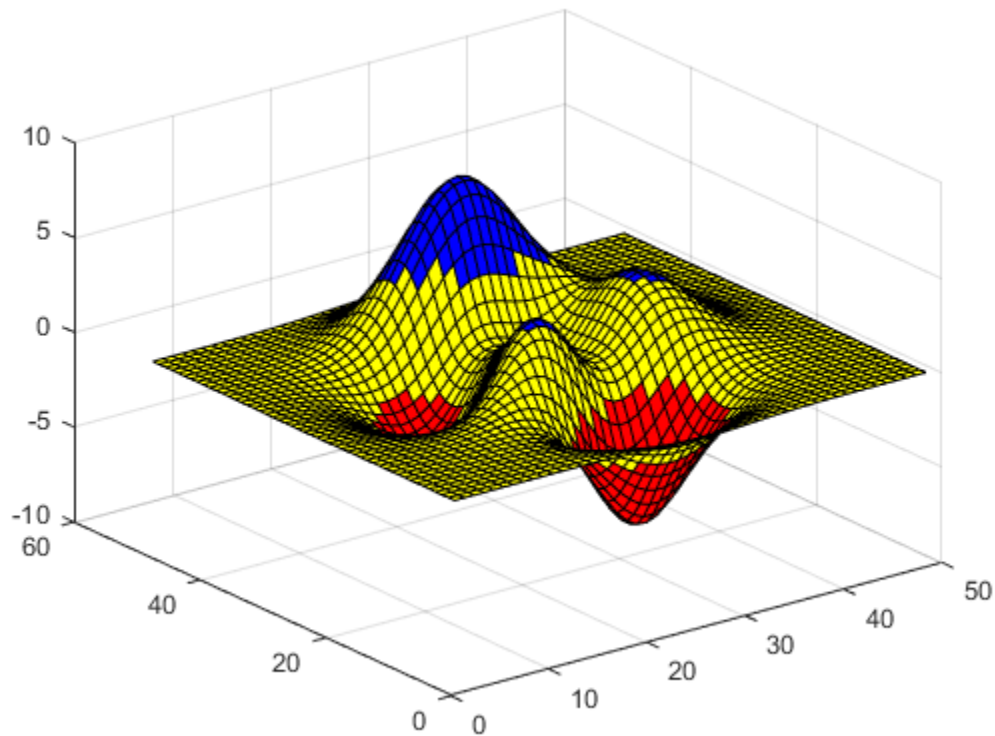
If you are working with multiple axes, you can assign a different colormap to each axes by passing the axes object to the `colormap` function.

```
tiledlayout(1,2)
ax1 = nexttile;
surf(peaks);
shading interp;
colormap(ax1,parula(10));

ax2 = nexttile;
surf(peaks);
shading interp;
colormap(ax2,cool(10));
```
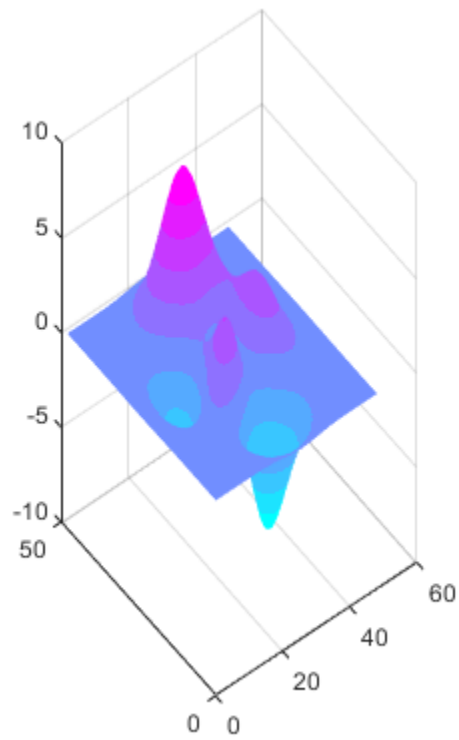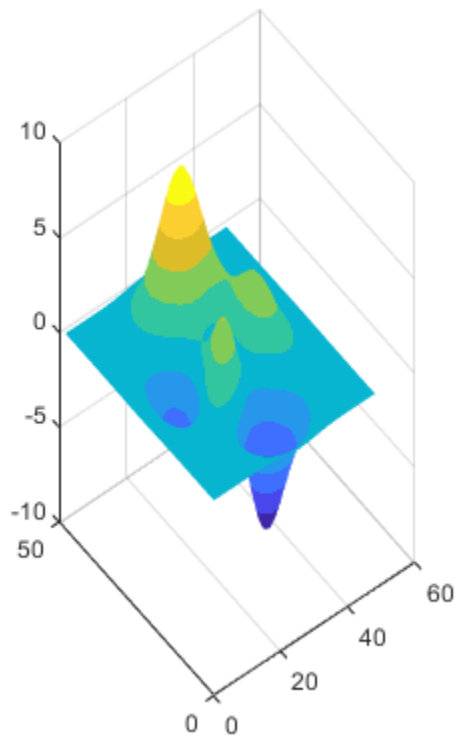
See Also

Related Examples

- "How Surface Plot Data Relates to a Colormap" on page 10-16

How Surface Plot Data Relates to a Colormap

When you create surface plots using functions such as `surf` or `mesh`, you can customize the color scheme by calling the `colormap` function. If you want further control over the appearance, you can change the direction or pattern of the colors across the surface. This customization requires changing values in an array that controls the relationship between the surface and the colormap.

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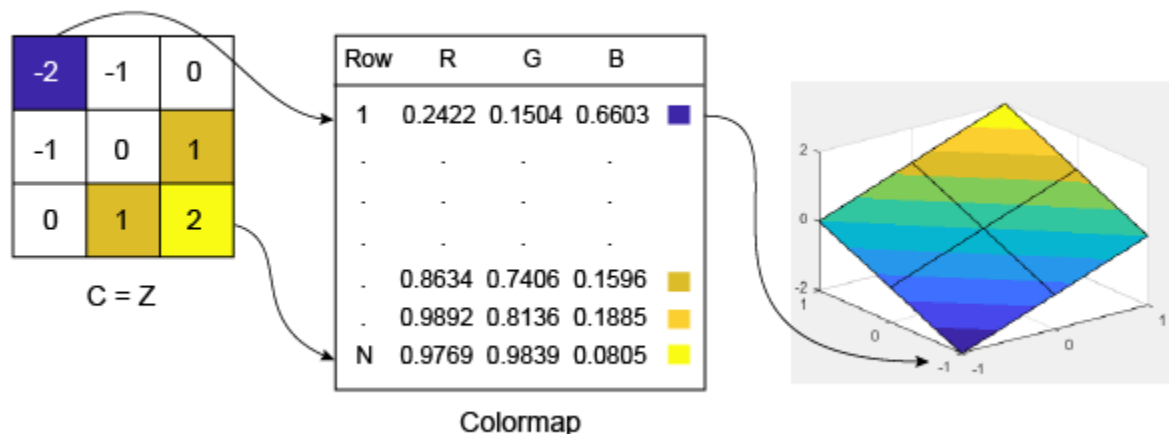
Notice that the smallest value (`-2`) maps to the first row in the colormap. The largest value (`2`) maps to the last row in the colormap. The intermediate values in `C` map linearly to the intermediate rows in the colormap.

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');
```
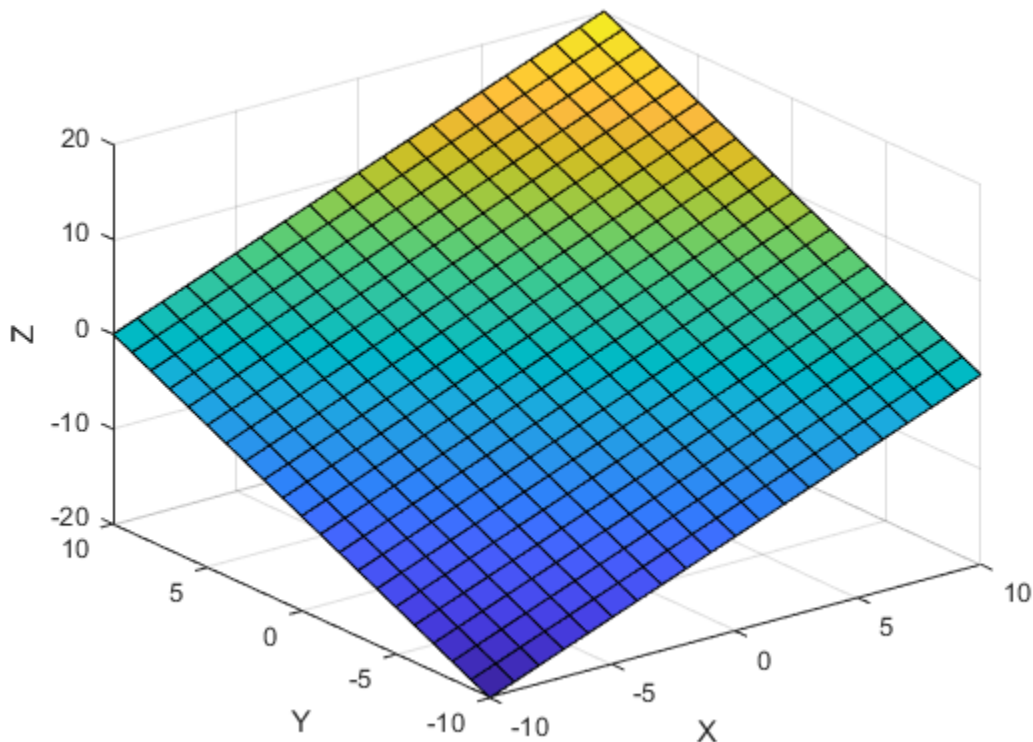

You can change this behavior by specifying C when you create the surface. For example, the colors on this surface vary with X.

```
C = X;
s = surf(X,Y,Z,C);
xlabel('X');
ylabel('Y');
zlabel('Z');
```
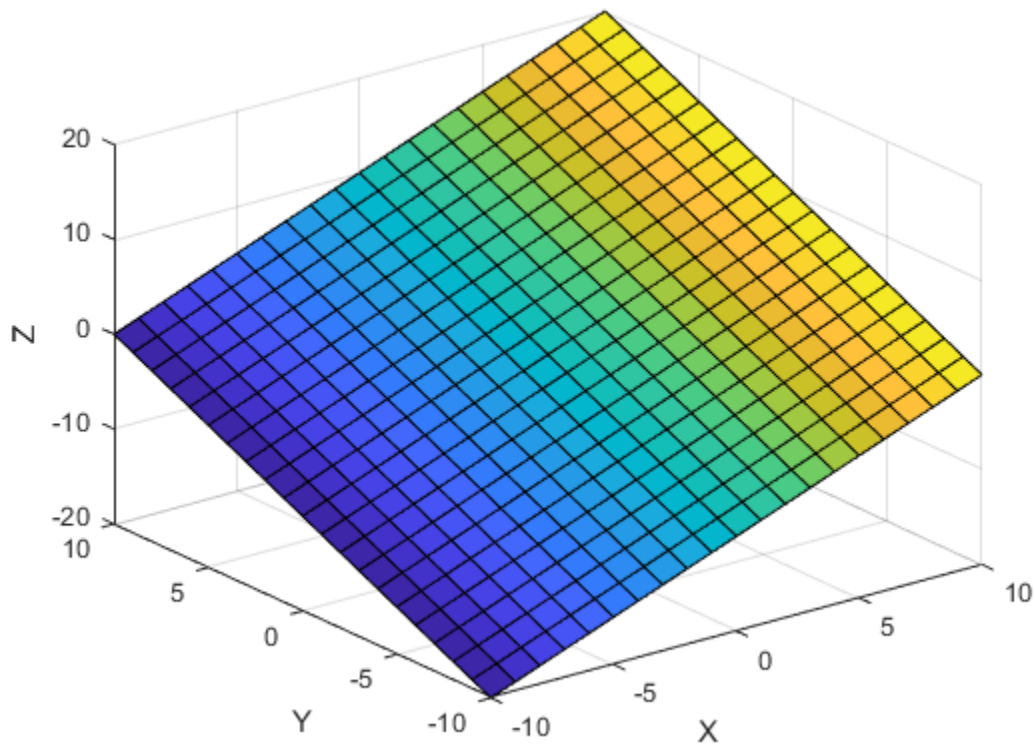
Alternatively, you can set the `CData` property directly. This command makes the colors vary with Y.

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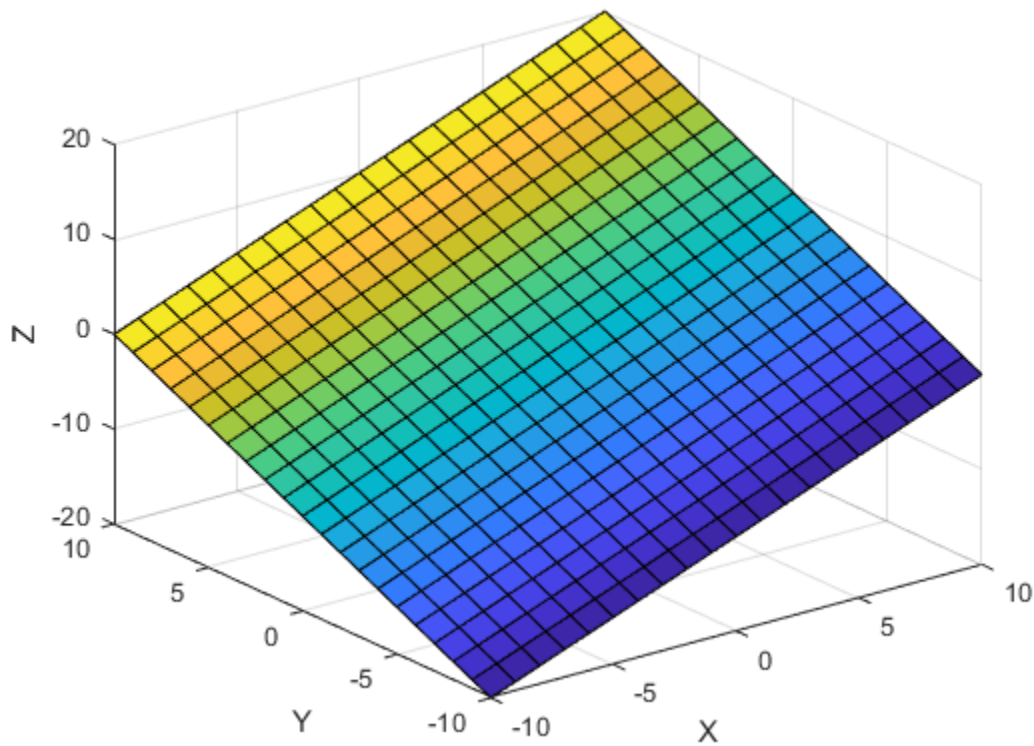
The colors do not need to follow changes in a single dimension. In fact, CData can be *any* array that is the same size as Z. For example, the colors on this plane follow the shape of a sinc function.

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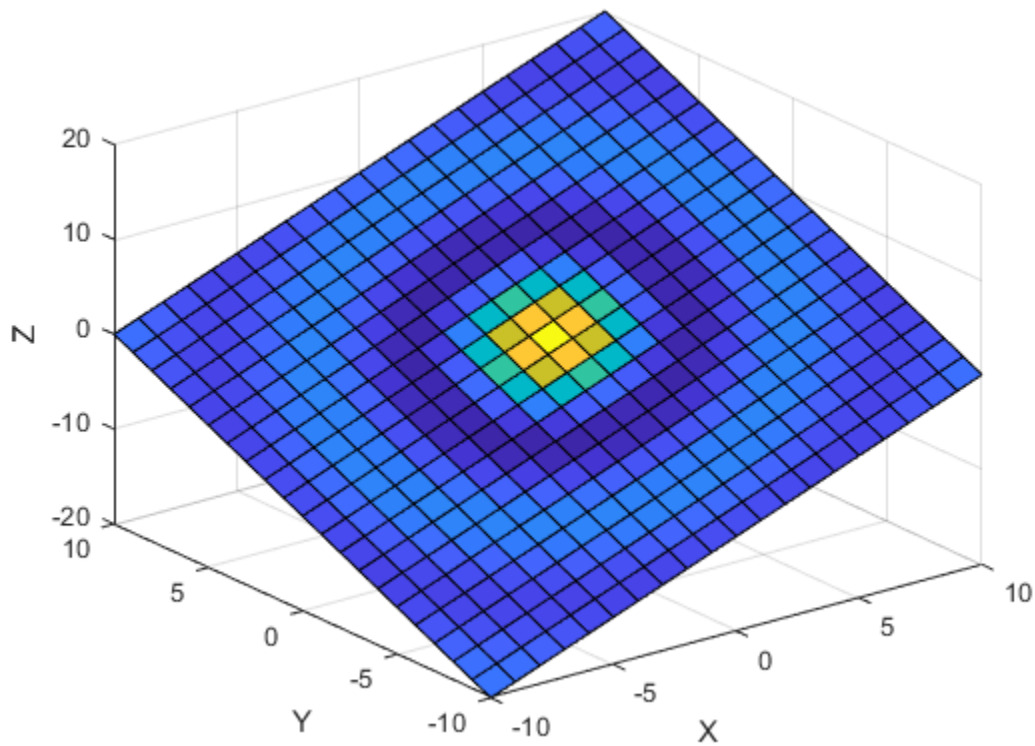
See Also

Functions
mesh | surf

Properties
Chart Surface

Related Examples

- "Change Color Scheme Using a Colormap" on page 10-10
- "Differences Between Colormaps and Truecolor" on page 10-38

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)

A = 5×5

    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


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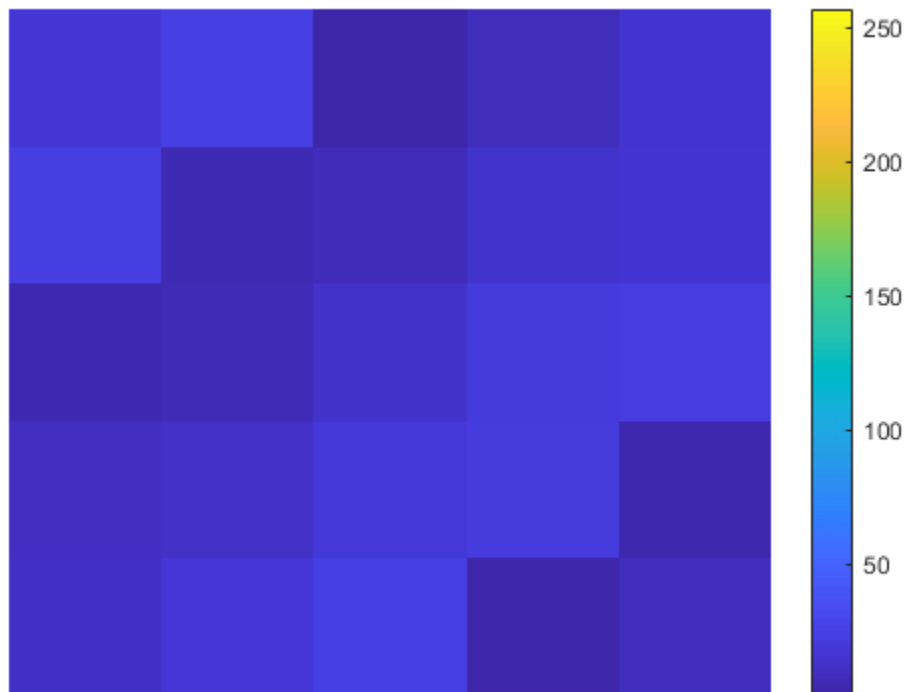

A contains values between 1 and 25. MATLAB treats those values as indices into the colormap, which has 64 entries. Thus, all the pixels in the preceding image map to the first 25 entries in the colormap (roughly the blue region of the colorbar).

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';
```
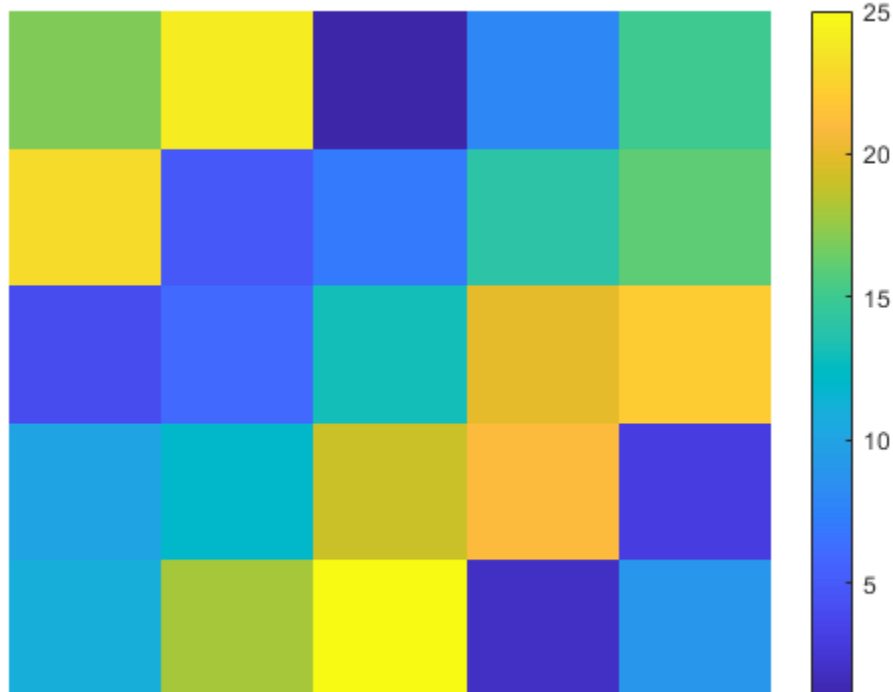
The `'scaled'` option maps the smallest value in A to the first entry in the colormap, and maps largest value in A maps to the last entry in the colormap. All intermediate values of A are linearly scaled to the colormap.
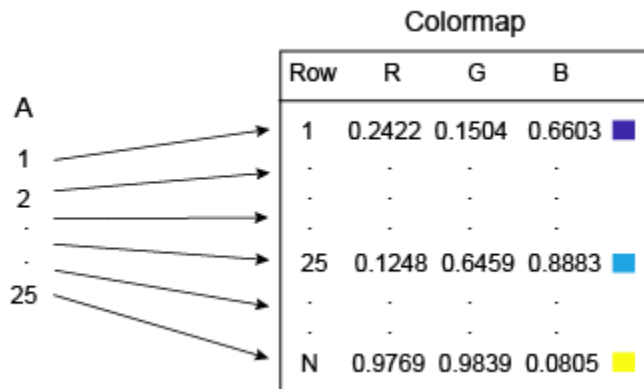
As an alternative to setting the `CDataMapping` property to `'scaled'`, you can call the `imagesc` function to get the same effect.

```
imagesc(A);
axis off
colorbar
```
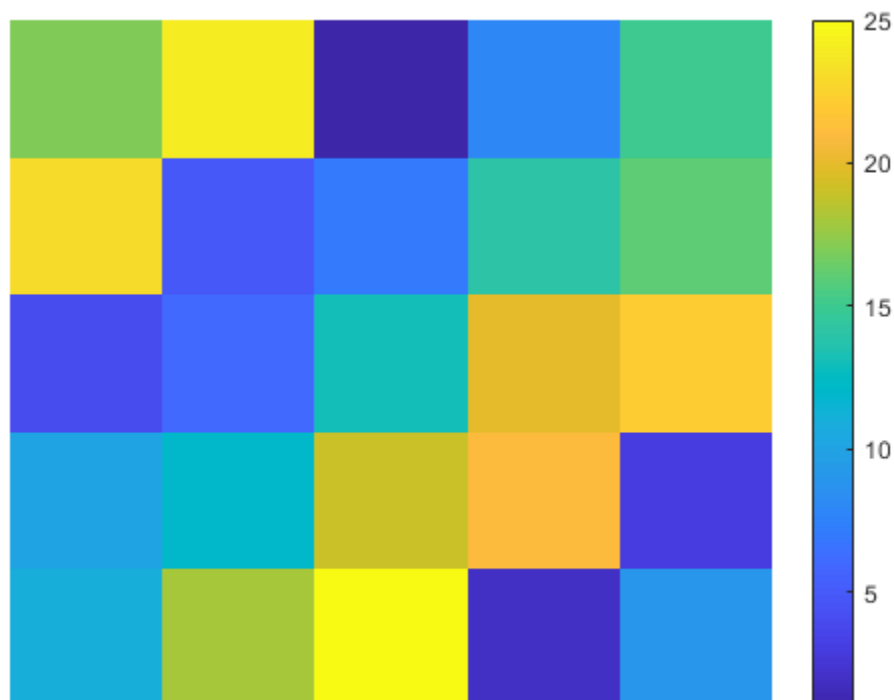
If you change the colormap, the values in A are scaled to the new colormap.

```
colormap(gray);
```
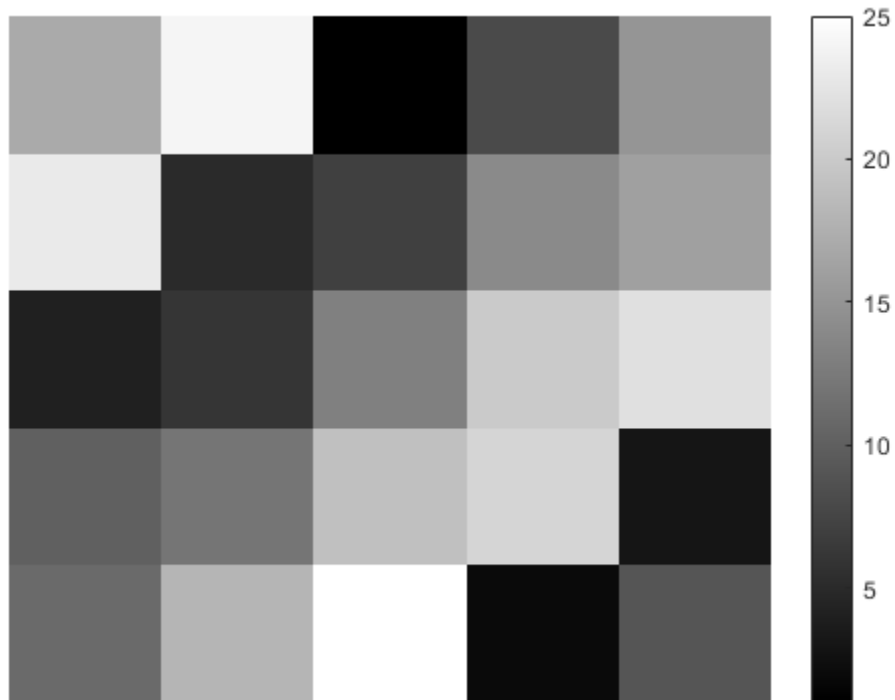

Scaled mapping is also useful for displaying pictorial images that have no colormap, or when you want to change the colormap for a pictorial image. The following commands display an image using the gray colormap, which is different than the original colormap that is stored with this image.

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

See Also

Functions
image | imagesc

Properties
Image

Related Examples

- "Image Types" on page 15-4
- "Differences Between Colormaps and Truecolor" on page 10-38

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,:)`).
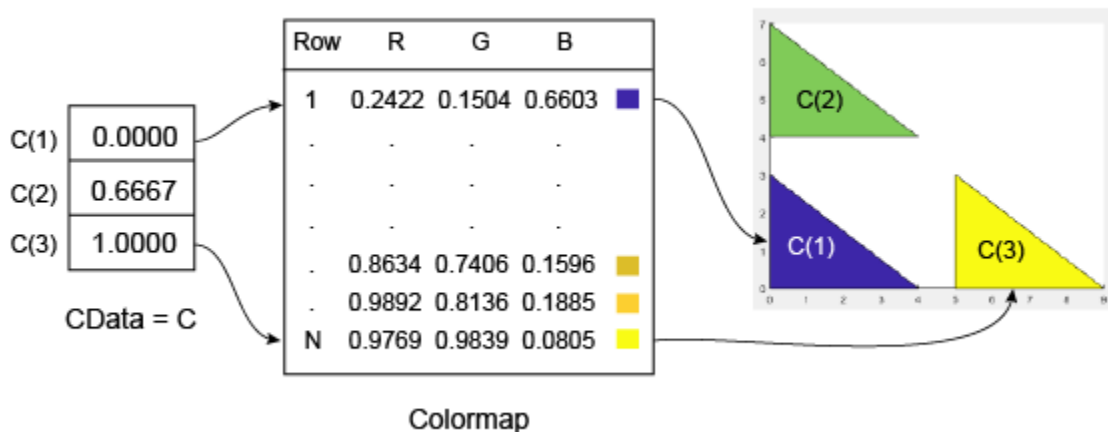
Colormap

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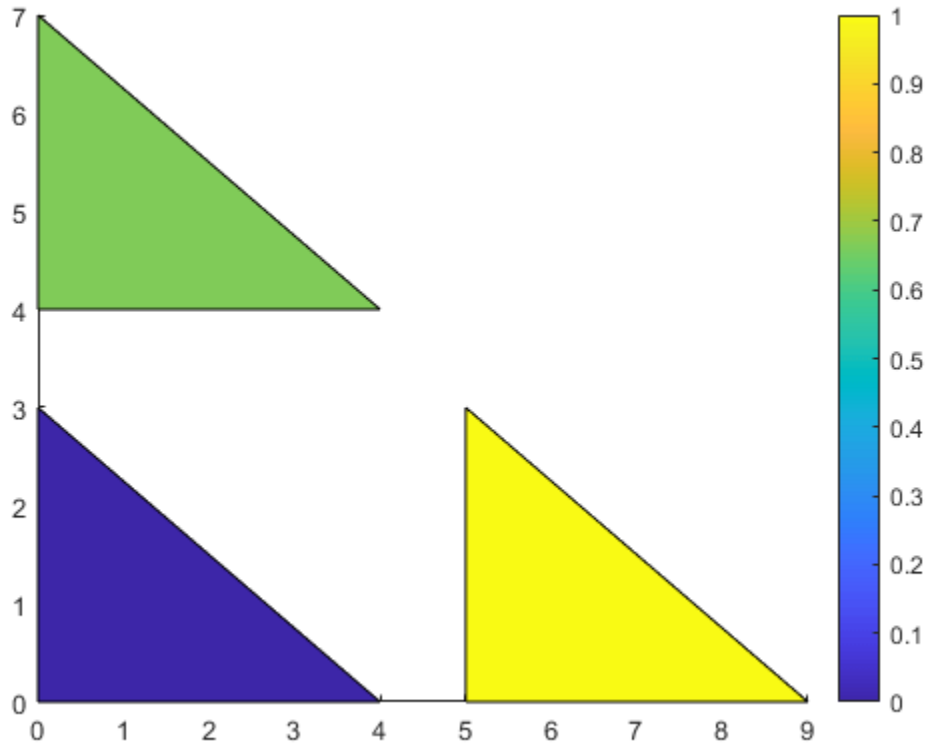
To assign colors to the vertices, specify C as an array with these characteristics:

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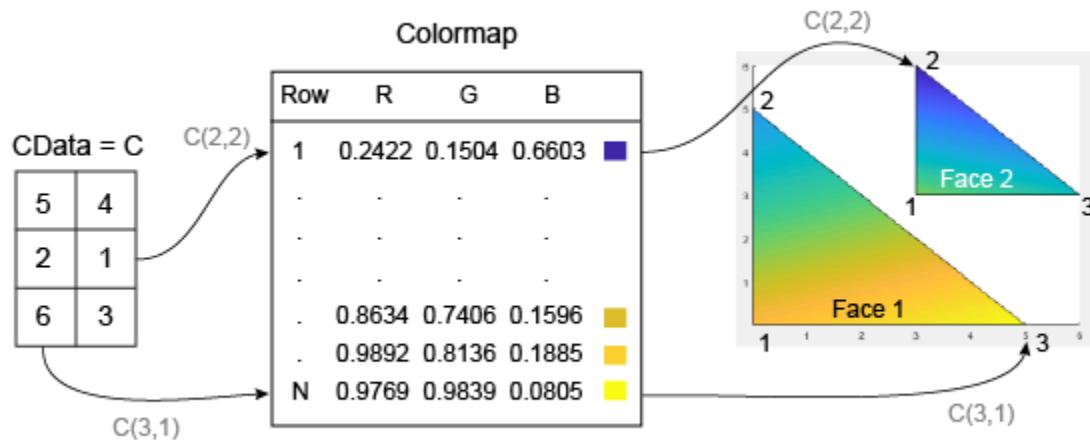
As with patch faces, MATLAB scales the values in C to the number of rows in the colormap. In this case, the smallest value is C(2,2)=1, and it maps to the first row in the colormap. The largest value is C(3,1)=6, and it maps to the last row in the colormap.

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
```
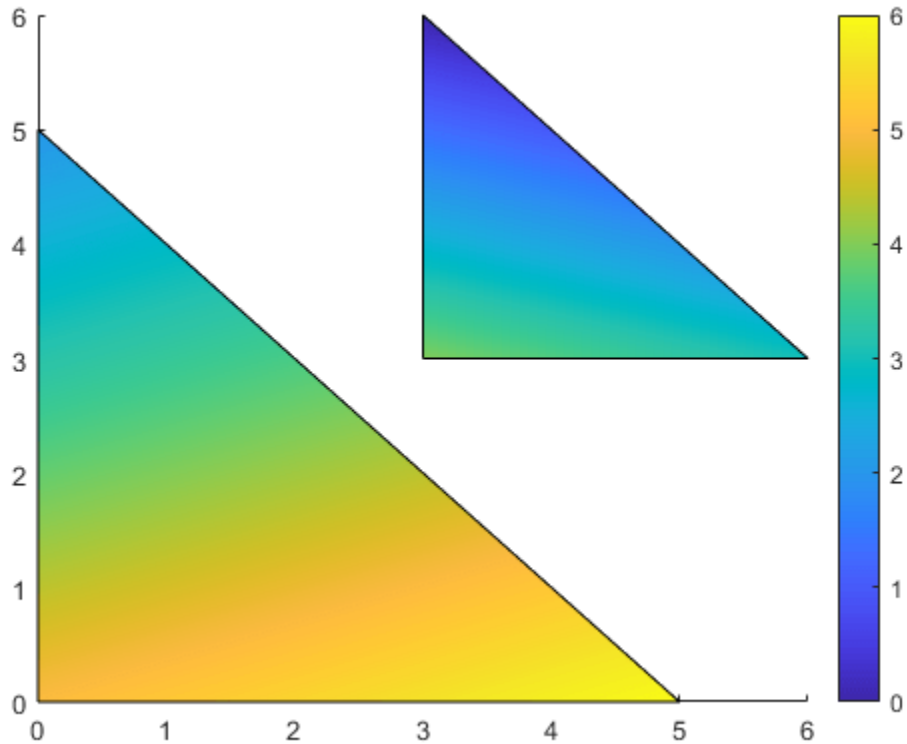
Relationship of the Colormap to Face-Vertex Data

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.
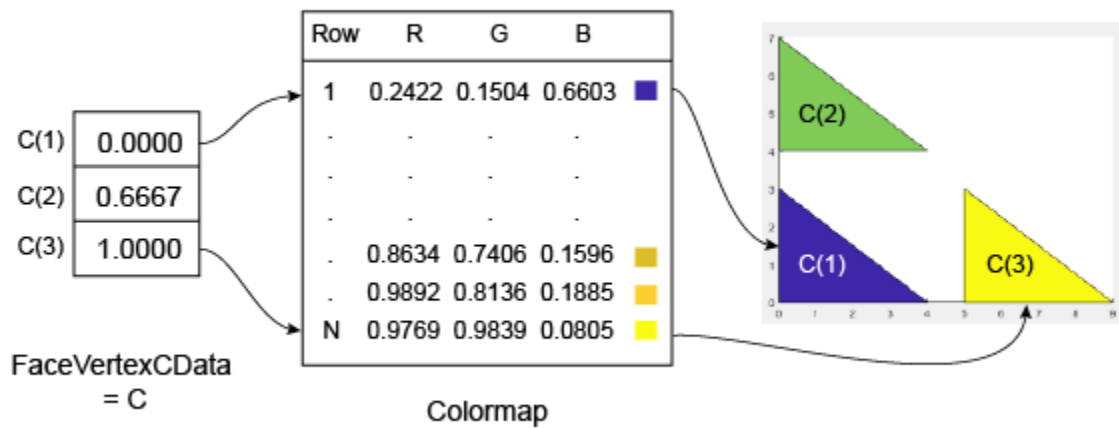
The smallest value in C is 0, and it maps to the first row in the colormap. The largest value in C is 1, and it maps to the last value in the colormap. Intermediate values of C map linearly to the intermediate rows in the colormap. In this case, C(2) maps to the color located about two-thirds from the 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
```
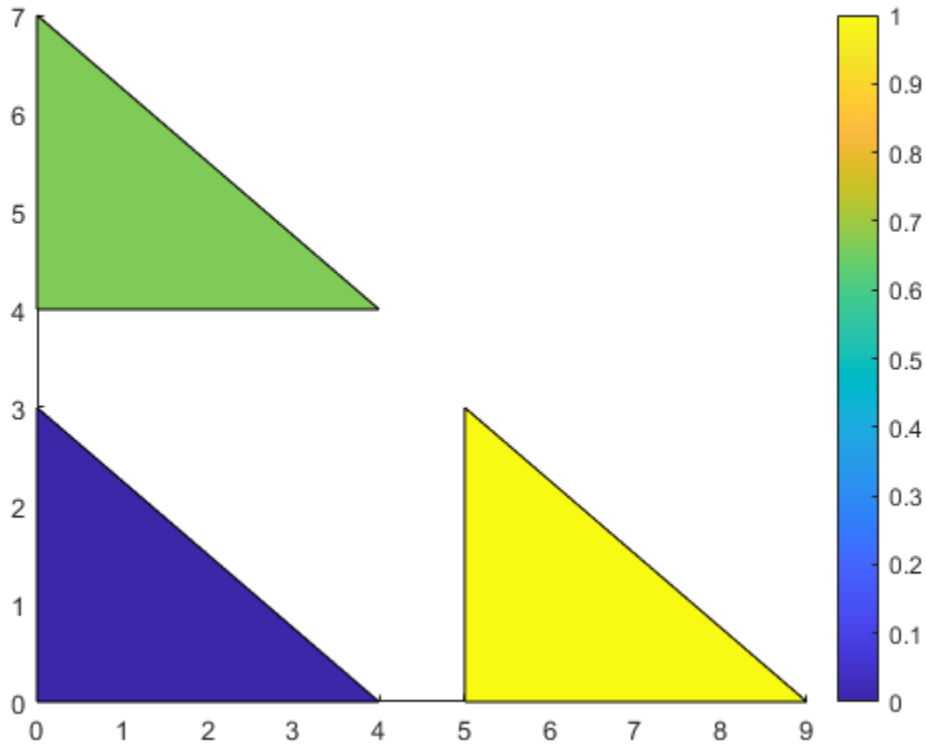
To assign colors to the vertices, specify the `FaceVertexCData` property of the `Patch` object as array C with these characteristics:

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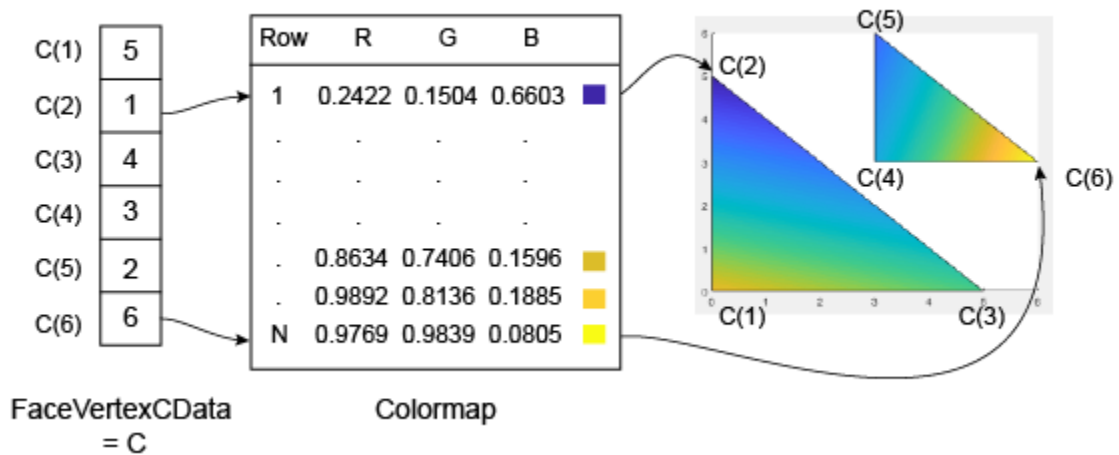
As with patch faces, MATLAB scales the values in `C` to the number of rows in the colormap. In this case, the smallest value is `C(2)=1`, and it maps to the first row in the colormap. The largest value is `C(6)=6`, and it maps to the last row in the colormap.

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
```
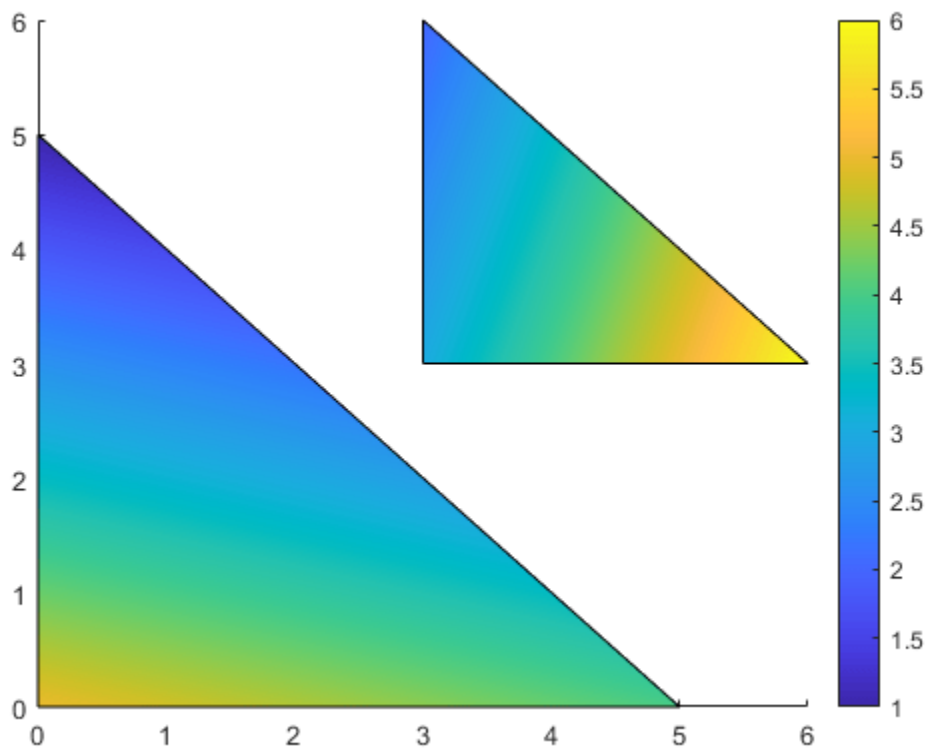

See Also

Functions
`patch`

Properties
Patch

Related Examples

- "Change Color Scheme Using a Colormap" on page 10-10
- "Differences Between Colormaps and Truecolor" on page 10-38

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
```
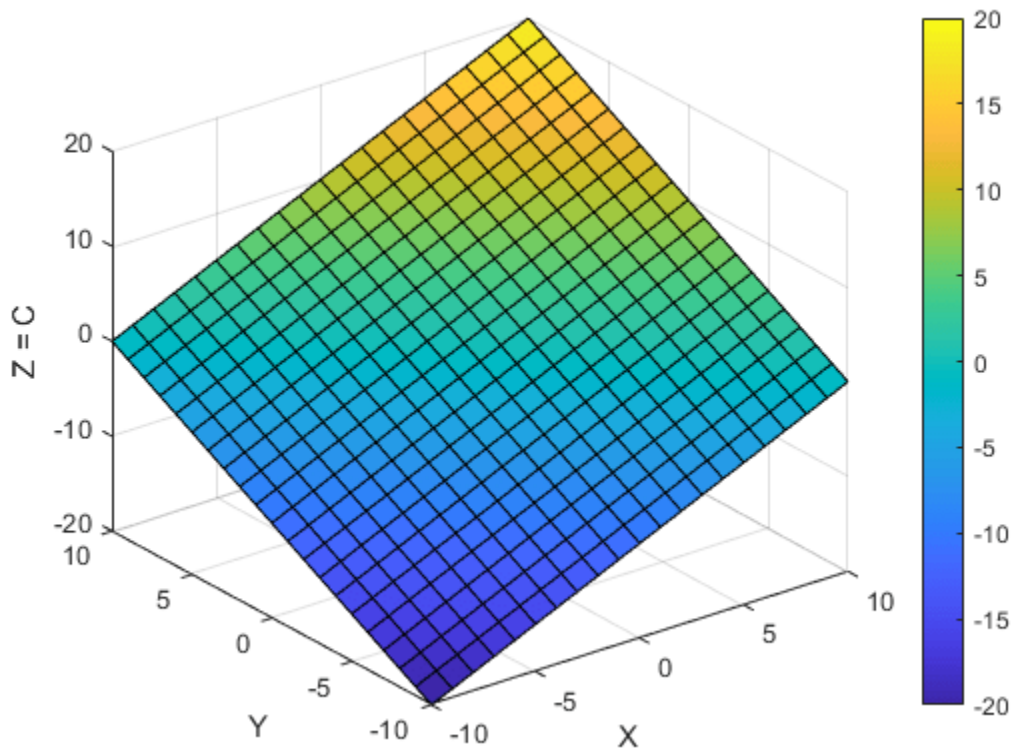

"How Surface Plot Data Relates to a Colormap" on page 10-16 describes the properties that control the color in this presentation. Essentially, the `CData` property of the `Surface` object contains an

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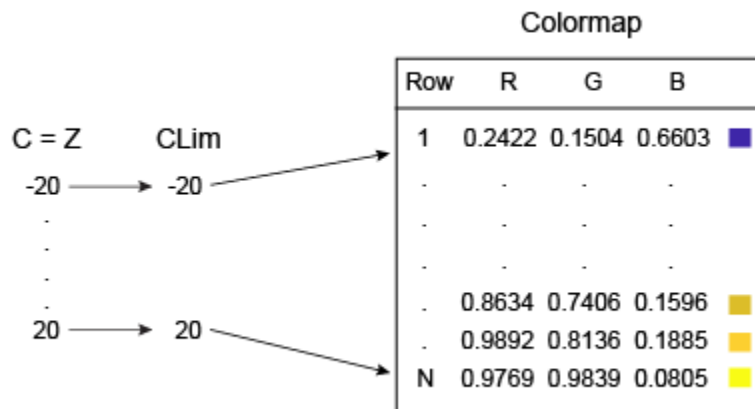
Changing CLim to [0 20] clips all the values at or below 0 to the first color in the colormap.
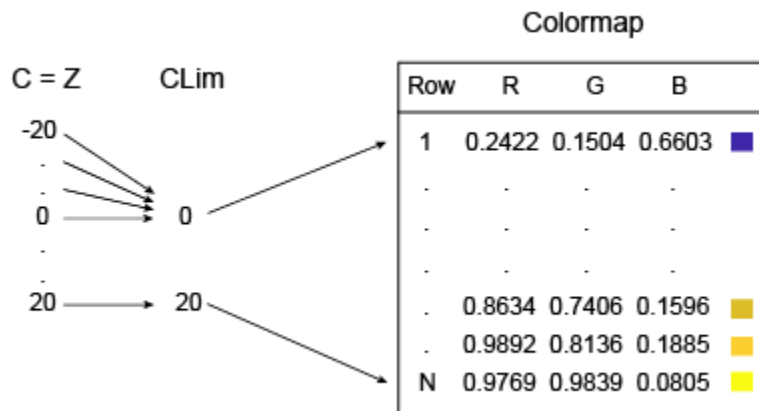
This command changes the CLim property to [0 20]. Notice that the lower half of the surface maps to the first color in the colormap (dark blue). This clipping occurs because C (which is equal to Z) is less than or equal to zero at those points.

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

10-35
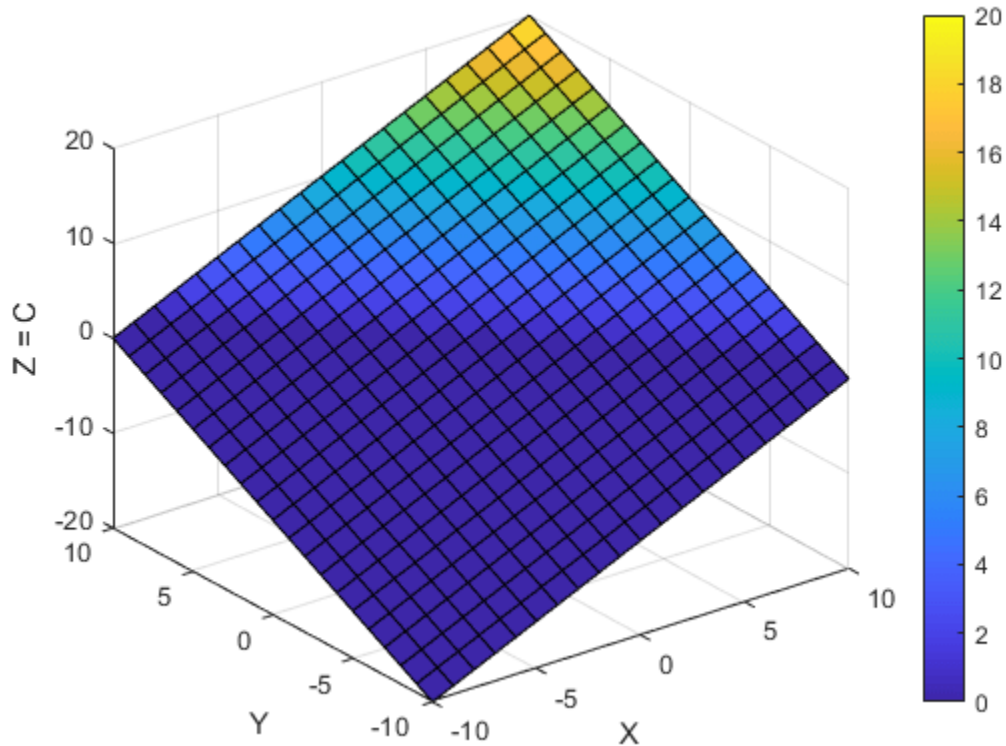
You can also widen the mapping range to see where your data lies within that range. For example, changing the range to [-60 20] results in a surface that only uses half of the colors. The lower half of the colormap corresponds to values that are outside the range of C, so those colors are not represented on the surface.

```
caxis([-60 20]);
```
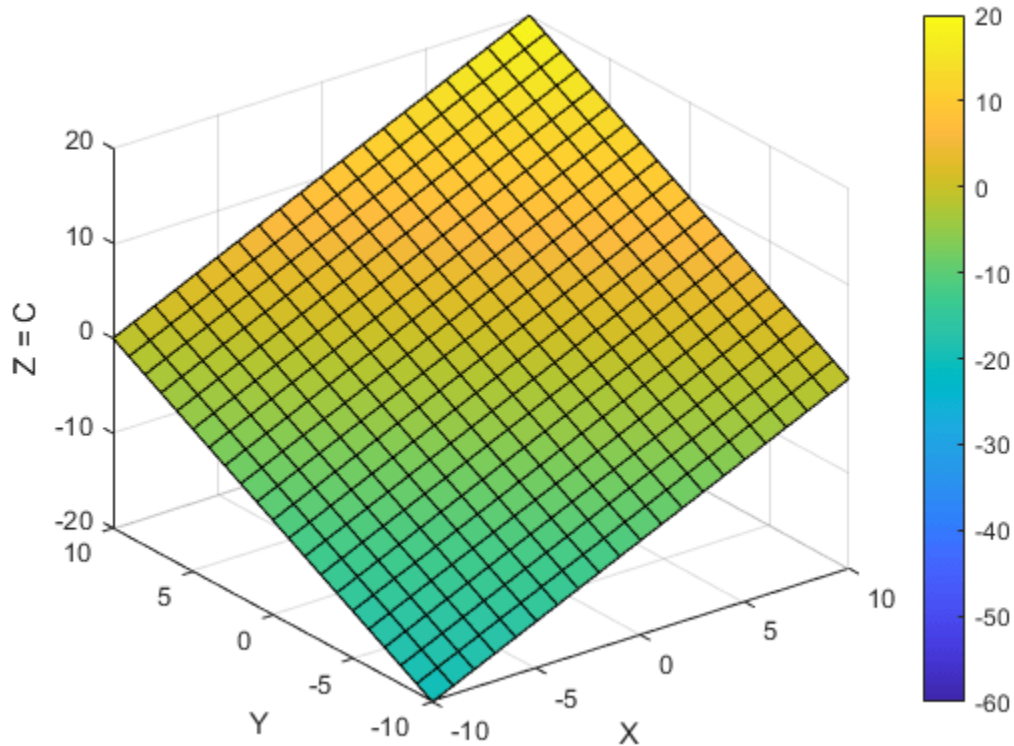
Note You can set the `CLim` property for surface plots, patches, images, or any graphics object that uses a colormap. However, this property only affects graphics objects that have the `CDataMapping` property set to `'scaled'`. If the `CDataMapping` property is set to `'direct'`, then all values of C 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 10-10
- "How Surface Plot Data Relates to a Colormap" on page 10-16
- "Creating Colorbars" on page 10-2

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. `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)`.

Type of Graphics Object	Property that Contains Color Array C	Shape of C for Indexed Color	Shape of C for Truecolor
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).
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 $256 \times 256 \times 256 \approx 1.68$ 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.
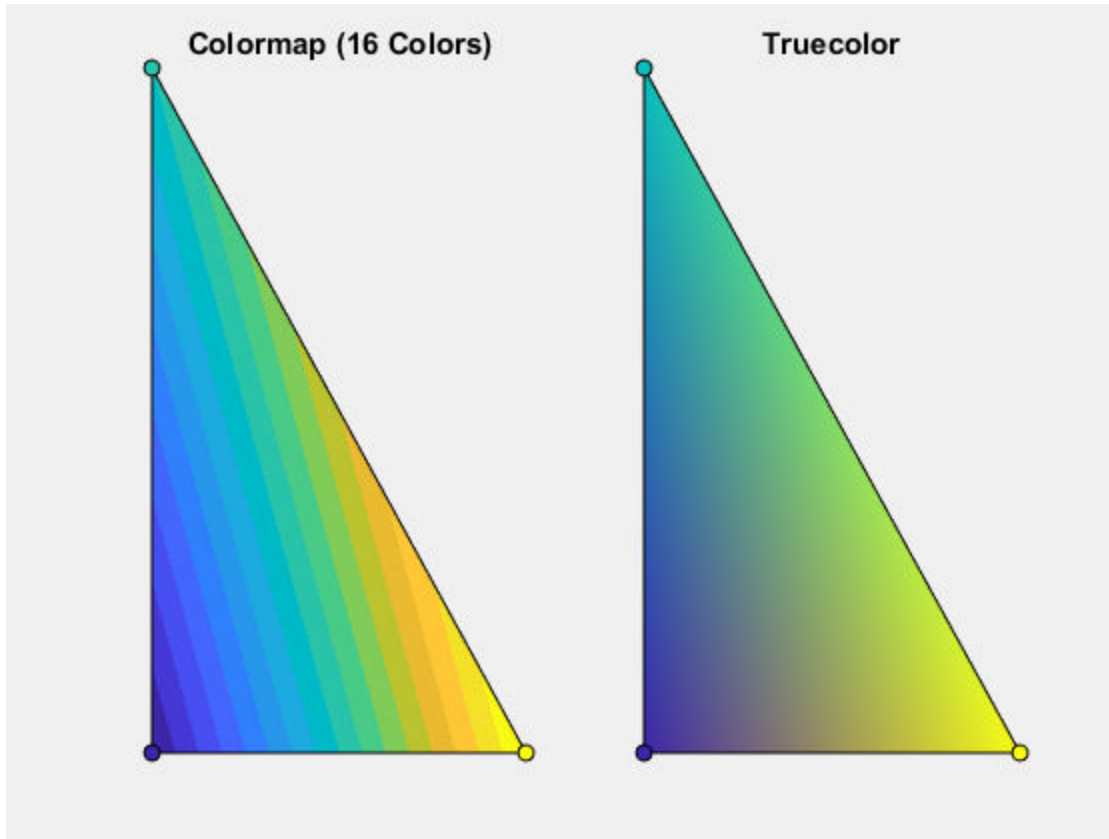
If you are concerned about the limited palette of a colormap, you can add more colors to it. "Change Color Scheme Using a Colormap" on page 10-10 shows how to use a colormap with a specific number of colors.

See Also

Related Examples

- "Image Types" on page 15-4
- "Change Color Scheme Using a Colormap" on page 10-10
- "How Surface Plot Data Relates to a Colormap" on page 10-16
- "How Image Data Relates to a Colormap" on page 10-21
- "How Patch Data Relates to a Colormap" on page 10-26

Lighting

- "Lighting Overview" on page 11-2
- "Reflectance Characteristics of Graphics Objects" on page 11-7

Lighting Overview

In this section...
"Lighting Commands" on page 11-2
"Light Objects" on page 11-2
"Properties That Affect Lighting" on page 11-3
"Examples of Lighting Control" on page 11-4

Lighting Commands

The MATLAB graphics environment provides commands that enable you to position light sources and adjust the characteristics of the objects that are reflecting the lights. These commands include the following.

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

You can set light and lit-object properties to achieve specific results. In addition to the material in this topic area, you can explore the lighting examples as an introduction to lighting for visualization.

Light Objects

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

- `Color` — Color of the light cast by the light object
- `Style` — Either infinitely far away (the default) or local
- `Position` — Direction (for infinite light sources) or the location (for local light sources)

The `Color` property determines the color of the directional light from the light source. The color of an object in a scene is determined by the color of the object and the light source.

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

The `Position` property specifies the location of the light source in axes data units. In the case of a light source at infinity, `Position` specifies the direction to the light source.

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

Properties That Affect Lighting

You cannot see light objects themselves, but you can see their effects on any patch and surface objects present in the axes containing the light. A number of functions create these objects, including surf, mesh, pcolor, fill, and fill3 as well as the surface and patch functions.

You control lighting effects by setting various axes, light, patch, and surface object properties. All properties have default values that generally produce desirable results. However, you can achieve the specific effect you want by adjusting the values of these properties.

Property	Effect
AmbientLightColor	An axes property that specifies the color of the background light in the scene, which has no direction and affects all objects uniformly. Ambient light effects occur only when there is a visible light object in the axes.
AmbientStrength	A patch and surface property that determines the intensity of the ambient component of the light reflected from the object.
DiffuseStrength	A patch and surface property that determines the intensity of the diffuse component of the light reflected from the object.
SpecularStrength	A patch and surface property that determines the intensity of the specular component of the light reflected from the object.
SpecularExponent	A patch and surface property that determines the size of the specular highlight.
SpecularColorReflectance	A patch and surface property that determines the degree to which the specularly reflected light is colored by the object color or the light source color.
FaceLighting	A patch and surface property that determines the method used to calculate the effect of the light on the faces of the object. Choices are either no lighting, flat, or Gouraud, lighting algorithm.
EdgeLighting	A patch and surface property that determines the method used to calculate the effect of the light on the edges of the object. Choices are either no lighting, flat, or Gouraud lighting algorithm.
BackFaceLighting	A patch and surface property that determines how faces are lit when their vertex normals point away from the camera. This property is useful for discriminating between the internal and external surfaces of an object.
FaceColor	A patch and surface property that specifies the color of the object faces.
EdgeColor	A patch and surface property that specifies the color of the object edges.
VertexNormals	A patch and surface property that contains normal vectors for each vertex of the object. MATLAB uses vertex normal vectors to perform lighting calculations. While MATLAB automatically generates this data, you can also specify your own vertex normals.
NormalMode	A patch and surface property that determines whether MATLAB recalculates vertex normals if you change object data (auto) or uses the current values of the VertexNormals property (manual). If you specify values for VertexNormals, MATLAB sets this property to manual.

For more information, see Axes, Chart Surface, and Patch.

Examples of Lighting Control

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

Example — Adding Lights to a Scene

This example displays the membrane surface and illuminates it with a light source emanating from a location to the right of the camera position.

```
membrane
camlight
```

Creating a light activates a number of lighting-related properties controlling characteristics such as the ambient light and reflectance properties of objects.
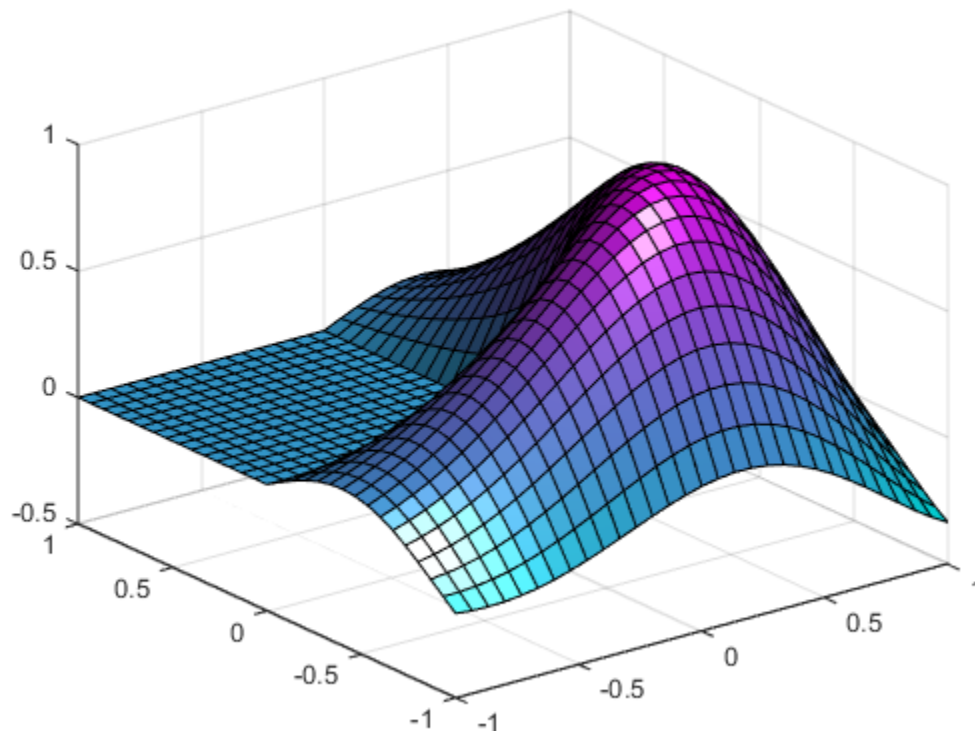
Example — Illuminating Mathematical Functions

Lighting can enhance surface graphs of mathematical functions. For example, use the `ezsurf` command to evaluate the expression

$$\sin\left(\sqrt{x^2 + y^2}\right) \div \left(\sqrt{x^2 + y^2}\right)$$

over the region -6π to 6π.

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


$$sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)$$
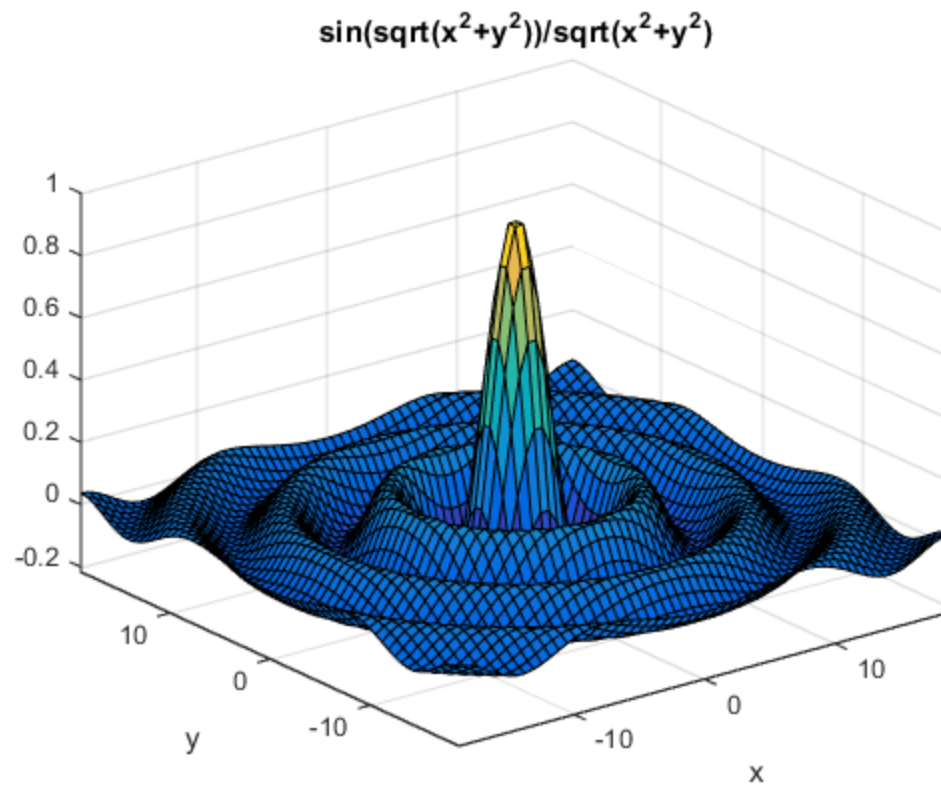
Now add lighting using the `lightangle` function, which accepts the light position in terms of azimuth and elevation.

```
view(0,75)
shading interp
lightangle(-45,30)
h.FaceLighting = 'gouraud';
h.AmbientStrength = 0.3;
h.DiffuseStrength = 0.8;
h.SpecularStrength = 0.9;
h.SpecularExponent = 25;
h.BackFaceLighting = 'unlit';
```

sin(sqrt(x²+y²))/sqrt(x²+y²)
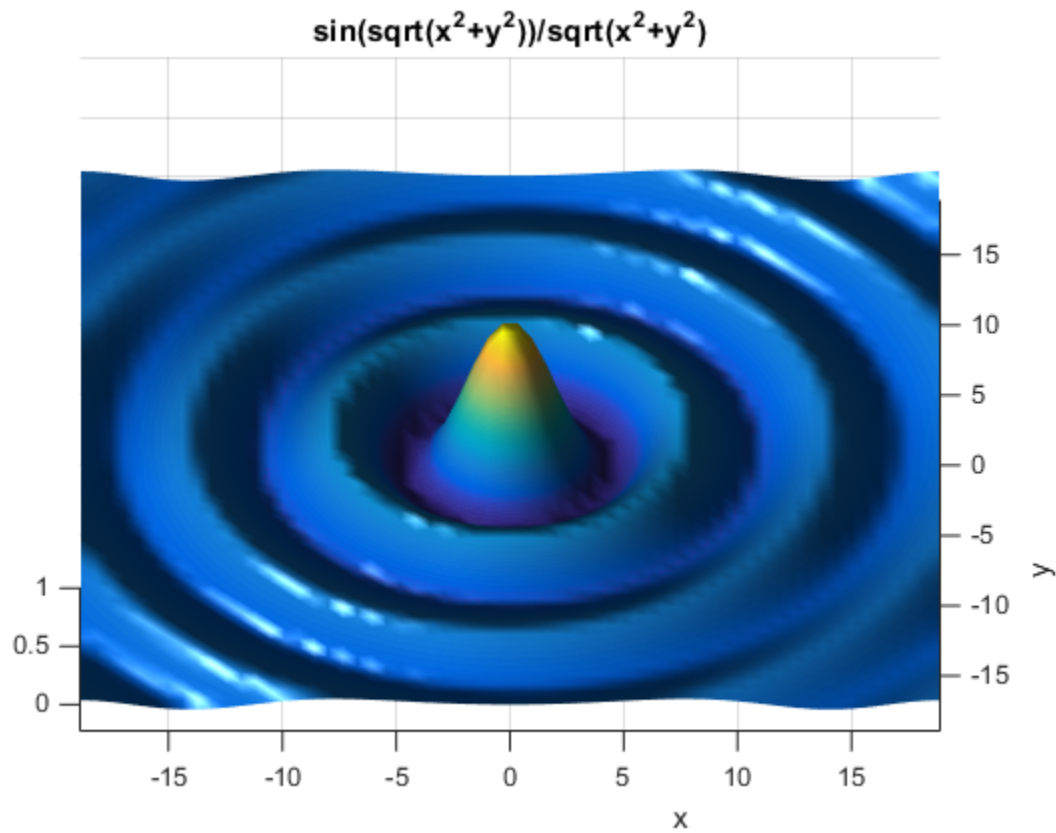
After obtaining the surface object's handle using `findobj`, you can set properties that affect how the light reflects from the surface. See "Properties That Affect Lighting" on page 11-3 for more detailed descriptions of these properties.

Reflectance Characteristics of Graphics Objects

Specular and Diffuse Reflection

You can specify the reflectance characteristics of patch and surface objects and thereby affect the way they look when lights are applied to the scene. It is likely you will adjust these characteristics in combination to produce particular results.

Also see the `material` command for a convenient way to produce certain lighting effects.

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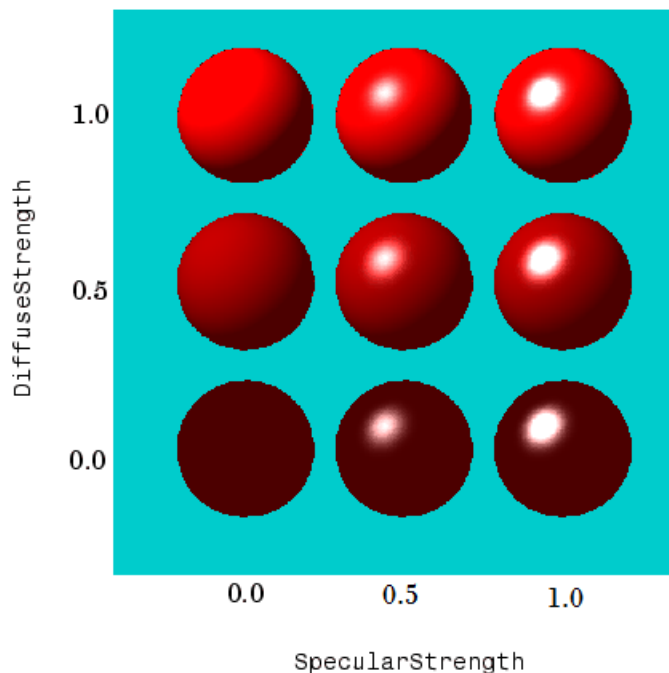
Ambient Light

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

property of patch and surface objects that determines the intensity of the ambient light on the particular object.

This illustration shows three different ambient light colors at various intensities. The sphere is red and there is a white light object present.
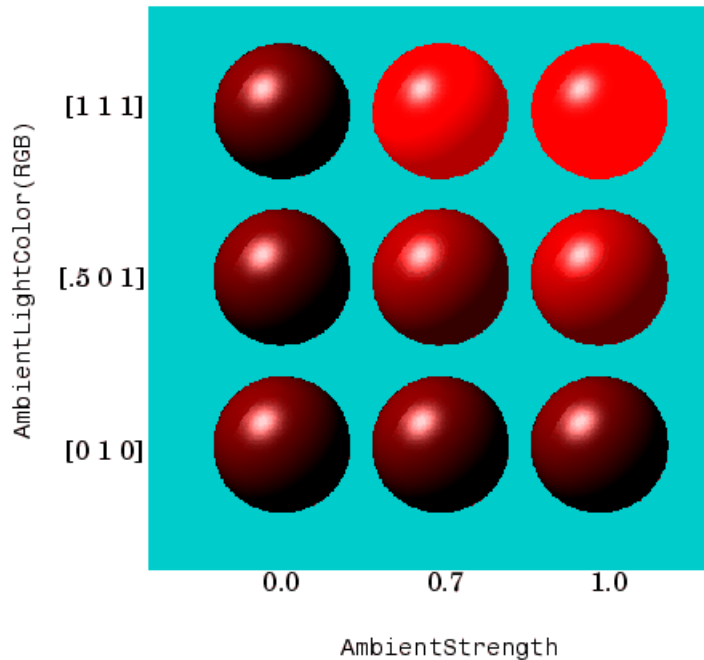
The green [0 1 0] ambient light does not affect the scene because there is no red component in green light. However, the color defined by the RGB values [.5 0 1] does have a red component, so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

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

This illustration shows a red sphere illuminated by a white light with three different values for the `SpecularExponent` property.
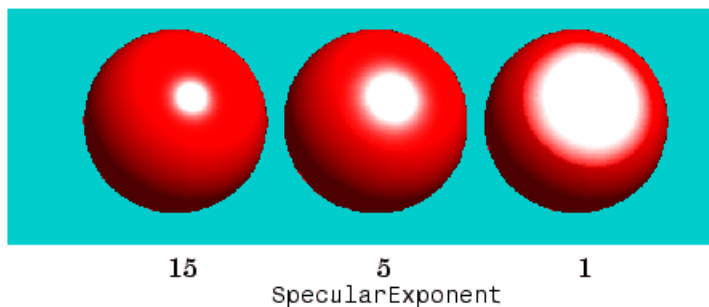
Specular Color Reflectance

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

SpecularColorReflectance
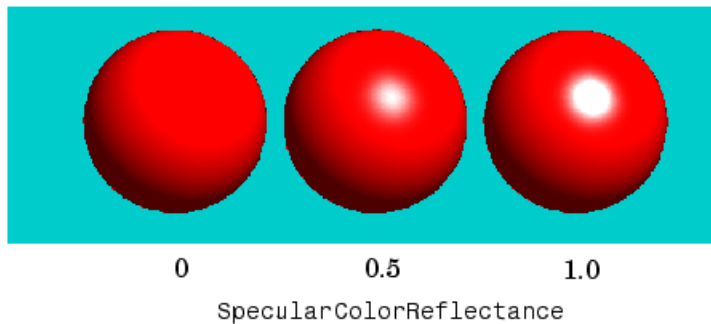
Back Face Lighting

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

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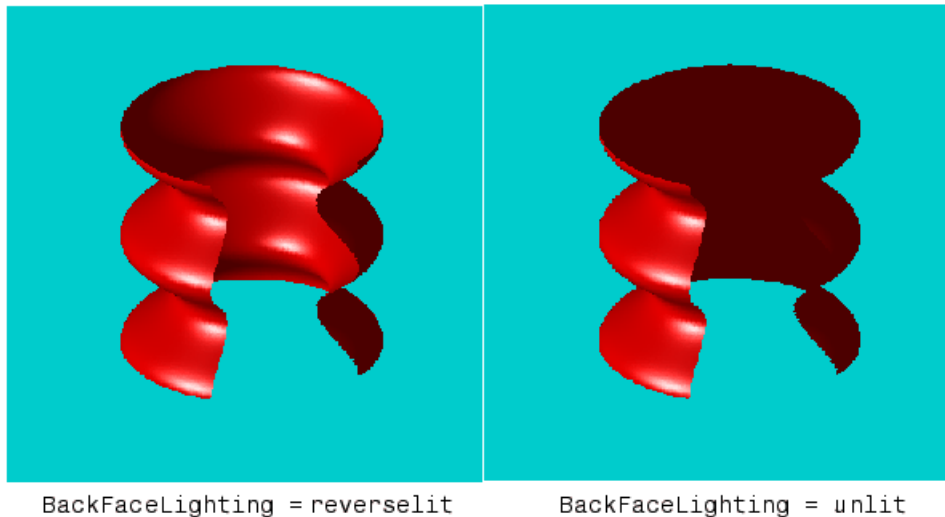
The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point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

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

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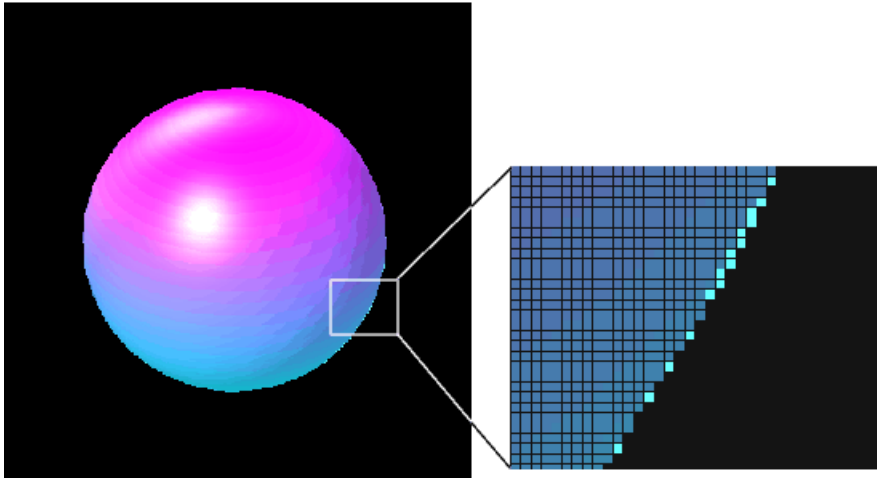
BackFaceLighting = reverselit
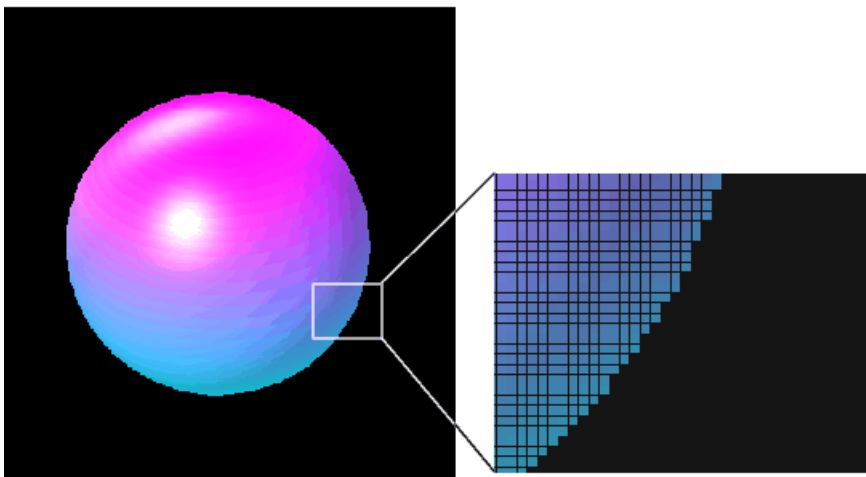
BackFaceLighting = lit

Positioning Lights in Data Space

This example creates a sphere and a cube and illuminates then with two light sources. The light objects are located at infinity, but in the directions specified by their position vectors.

```
% Create a sphere
sphere(36);
axis([-3 3 -3 3 -3 3])
hold on
% Create a cube
```

```
fac = [1 2 3 4;2 6 7 3;4 3 7 8;1 5 8 4;1 2 6 5;5 6 7 8];
vert = [1 1 1;1 2 1;2 2 1;2 1 1;1 1 2;1 2 2;2 2 2;2 1 2];
patch('faces',fac,'vertices',vert,'FaceColor','y');
% Add lights
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
hold off
```

The `light` functions define two light objects located at infinity in the direction specified by the `Position` vectors. These vectors are defined in axes coordinates [x, y, z].
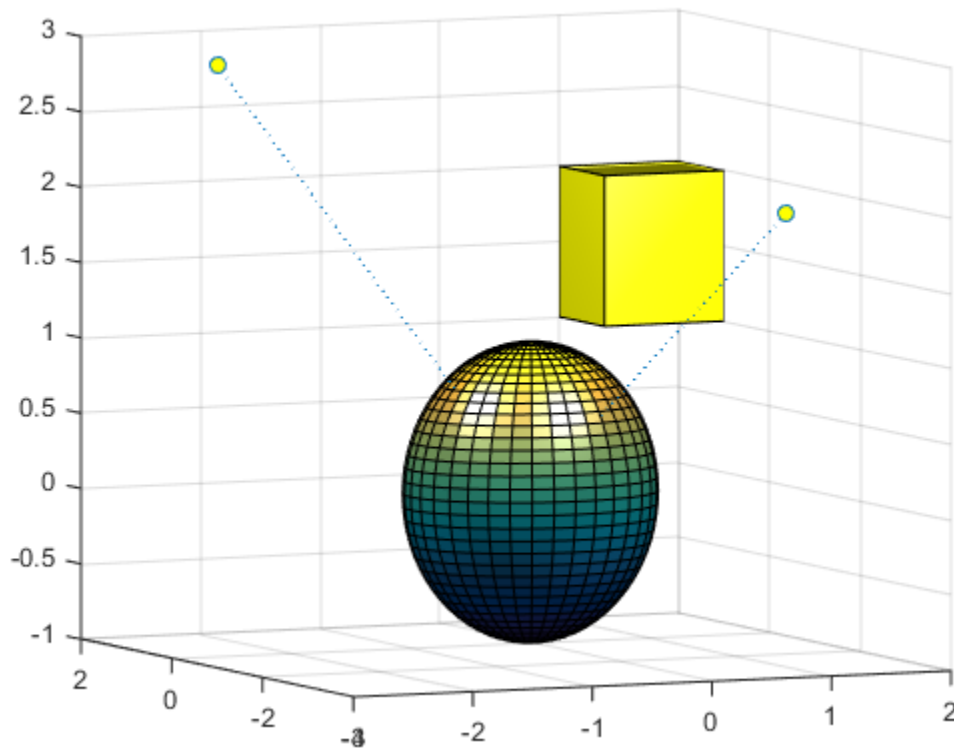
Transparency

- "Add Transparency to Graphics Objects" on page 12-2
- "Changing Transparency of Images, Patches or Surfaces" on page 12-9
- "Modify the Alphamap" on page 12-16

Add Transparency to Graphics Objects

In this section...
"What Is Transparency?" on page 12-2
"Graphics Objects that Support Transparency" on page 12-2
"Create Area Chart with Transparency" on page 12-3
"Create Bar Chart with Transparency" on page 12-4
"Create Scatter Chart with Transparency" on page 12-5
"Vary Transparency Using Alpha Data" on page 12-6
"Vary Surface Chart Transparency" on page 12-7
"Vary Patch Object Transparency" on page 12-7

What Is Transparency?

The transparency of a graphics object determines the degree to which you can see through it. Add transparency to graphics objects to customize the look of your charts or reveal details about an object that are otherwise hidden. This table shows the difference between an opaque and semitransparent surface.

Opaque Object	Semitransparent Object
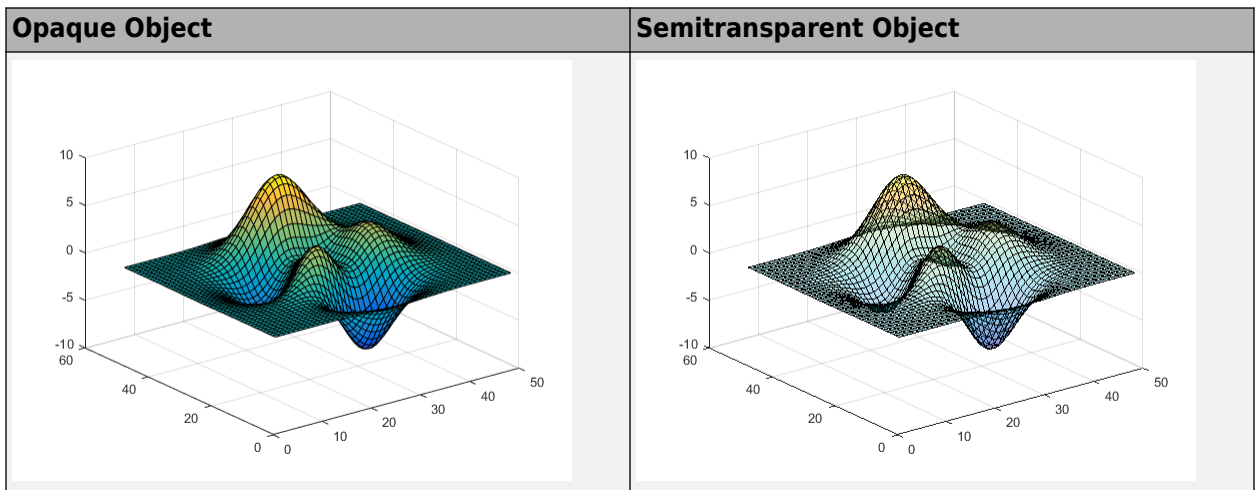
Graphics Objects that Support Transparency

Control the transparency of an object using the `alpha` function or by setting properties of the object related to transparency. Some graphics objects support using a different transparency value for the faces versus the edges of the object.

This table lists the objects that support transparency and the corresponding properties. Set the properties to a scalar value in the range `[0,1]`. A value of 0 means completely transparent, a value of 1 means completely opaque, and values between 0 and 1 are semitransparent.

Graphics Objects that Support Transparency	Properties for Uniform Transparency
Area	`FaceAlpha` `EdgeAlpha`
Bar series	`FaceAlpha` `EdgeAlpha`
Scatter series	`MarkerFaceAlpha` `MarkerEdgeAlpha`
BubbleChart series	`MarkerFaceAlpha` `MarkerEdgeAlpha`
Histogram	`FaceAlpha`
Histogram2	`FaceAlpha`
Chart surface	`FaceAlpha` `EdgeAlpha`
Primitive surface	`FaceAlpha` `EdgeAlpha`
Patch	`FaceAlpha` `EdgeAlpha`
Image	`AlphaData`

Tip Patch, surface, scatter, and image objects support using alpha data to vary the transparency across the object. For more information, see "Vary Transparency Using Alpha Data" on page 12-6.

Create Area Chart with Transparency

Combine two semitransparent area charts by setting the `FaceAlpha` and `EdgeAlpha` properties for each area object.

```
x = linspace(0,10);
y1 = 4 + sin(x).*exp(0.1*x);
area(x,y1,'FaceColor','b','FaceAlpha',.3,'EdgeAlpha',.3)

y2 = 4 + cos(x).*exp(0.1*x);
hold on
area(x,y2,'FaceColor','r','FaceAlpha',.3,'EdgeAlpha',.3)
hold off
```
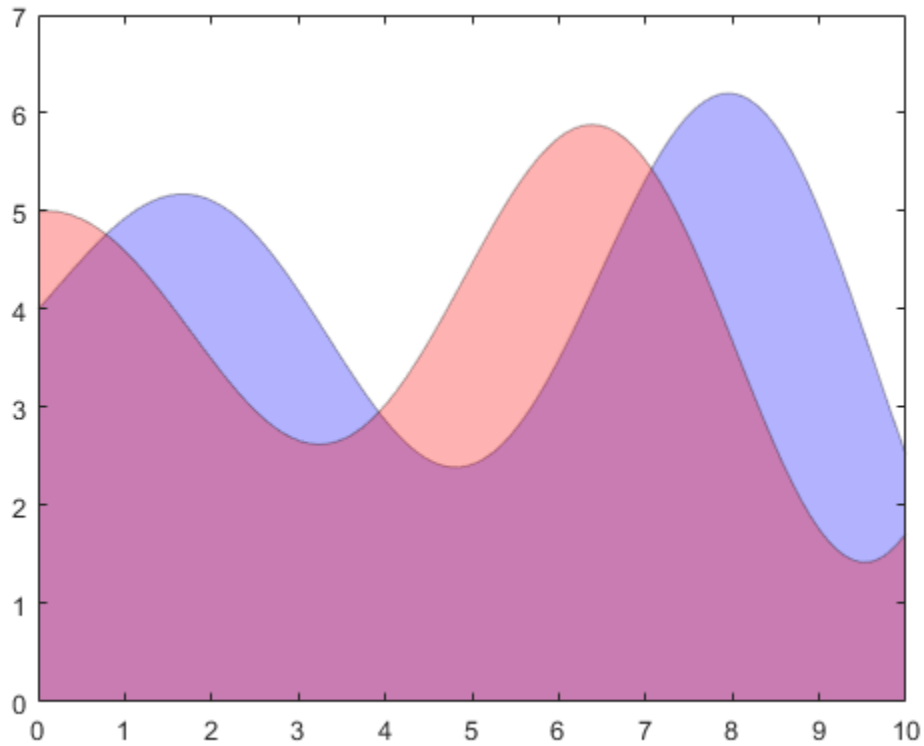
Create Bar Chart with Transparency

Create a semitransparent bar chart by setting the `FaceAlpha` property of the bar series object to a value between 0 and 1. Display the grid lines.

```
month = 1:5;
sales = [10 25 85 35 16];
bar(month,sales,'FaceAlpha',.5)
grid on
```
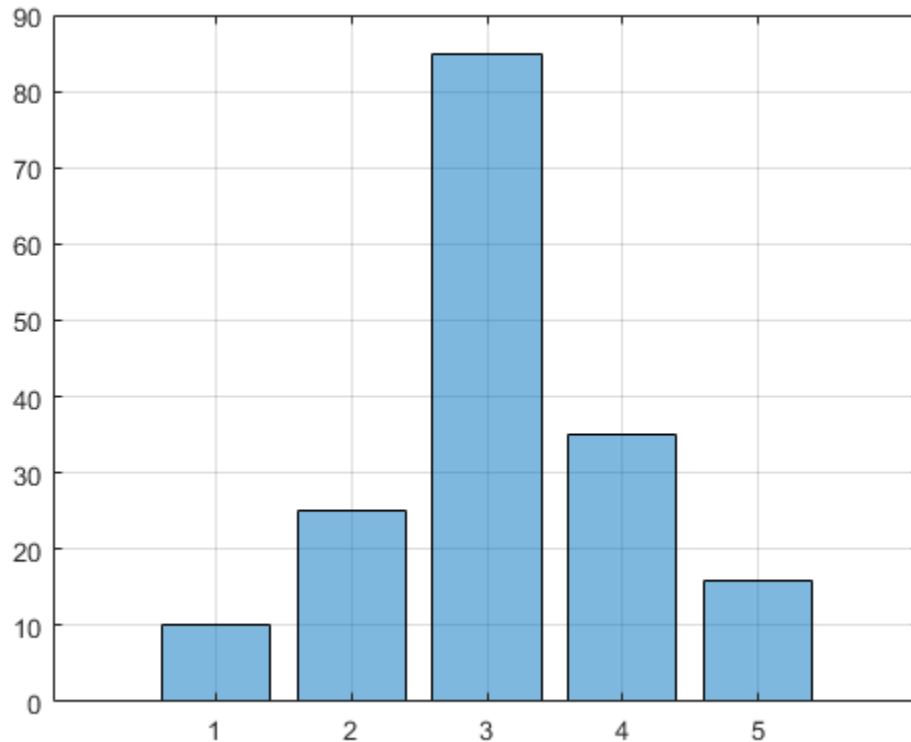
Create Scatter Chart with Transparency

Create a scatter plot using blue, semitransparent markers. Then, add a second scatter plot using red, semitransparent markers. Specify the marker color by setting the `MarkerFaceColor` and `MarkerEdgeColor` properties of the scatter series object. Specify the transparency by setting the `MarkerFaceAlpha` and `MarkerEdgeAlpha` properties to a scalar value between 0 and 1.

```
x = [5 10 11 13 18];
y1 = [40 80 150 80 50];
a1 = 100*[100 50 80 30 50];
scatter(x,y1,a1,'MarkerFaceColor','b','MarkerEdgeColor','b',...
    'MarkerFaceAlpha',.2,'MarkerEdgeAlpha',.2)
axis([0 20 0 200])

x = [2 6 8 11 13];
y2 = [30 40 100 60 140];
a2 = 100*[30 50 30 80 80];
hold on
scatter(x,y2,a2,'MarkerFaceColor','r','MarkerEdgeColor','r',...
    'MarkerFaceAlpha',.2,'MarkerEdgeAlpha',.2)
hold off
```
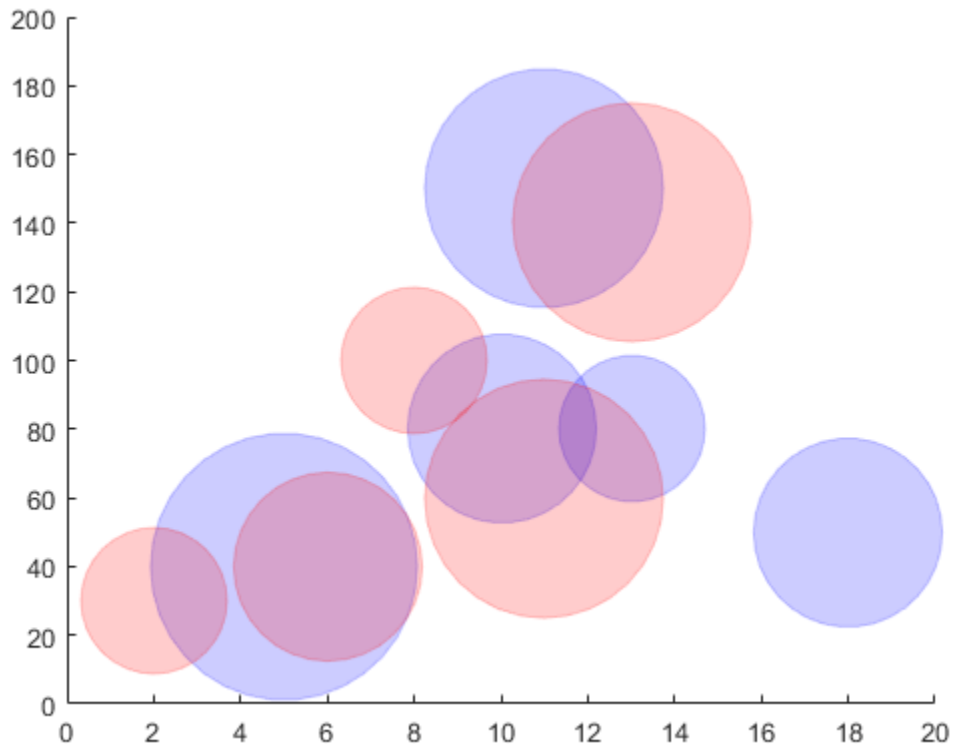
Vary Transparency Using Alpha Data

Patch, surface, and image objects have a few additional properties for varying the transparency across the object.

- Images — Specify a different transparency value for each image element. Specify the values by setting the `AlphaData` property to an array the same size as the `CData` property.

- Chart and primitive surfaces — Specify a different transparency value for each face and edge. Additionally, you can specify whether to use flat or interpolated transparency across each face or edge. First, specify the transparency values by setting the `AlphaData` property to an array the same size as the `ZData` property. Then, specify flat or interpolated transparency by setting the `FaceAlpha` and `EdgeAlpha` properties to either `'flat'` or `'interp'`.

- Patches — Specify a different transparency value for each face and edge. Additionally, you can specify whether to use flat or interpolated transparency across each face or edge. First, specify the transparency values by setting the `FaceVertexAlphaData` property to a column vector with length equal to either the number of faces (for flat transparency) or the number of vertices in the patch (for interpolated transparency). Then, specify flat or interpolated transparency by setting the `FaceAlpha` and `EdgeAlpha` properties to either `'flat'` or `'interp'`.

- Scatter plots — Specify a different transparency value for each marker. First, specify the transparency values by setting the `AlphaData` property to an array the same size as the `XData` property. Then, specify flat transparency by setting either the `MarkerFaceAlpha` or `MarkerEdgeAlpha` property to `'flat'`.

Use the `AlphaDataMapping` property to control how the objects interpret the alpha data values. See the property descriptions for more details.

Vary Surface Chart Transparency

Create a surface and vary the transparency based on the gradient of the `z` data. Use a flat transparency across each surface face by setting the `FaceAlpha` to `'flat'`. Set the surface color to blue to show how the transparency varies.

```
[x,y] = meshgrid(-2:.2:2);
z = x.*exp(-x.^2-y.^2);
a = gradient(z);

surf(x,y,z,'AlphaData',a,...
    'FaceAlpha','flat',...
    'FaceColor','blue')
```
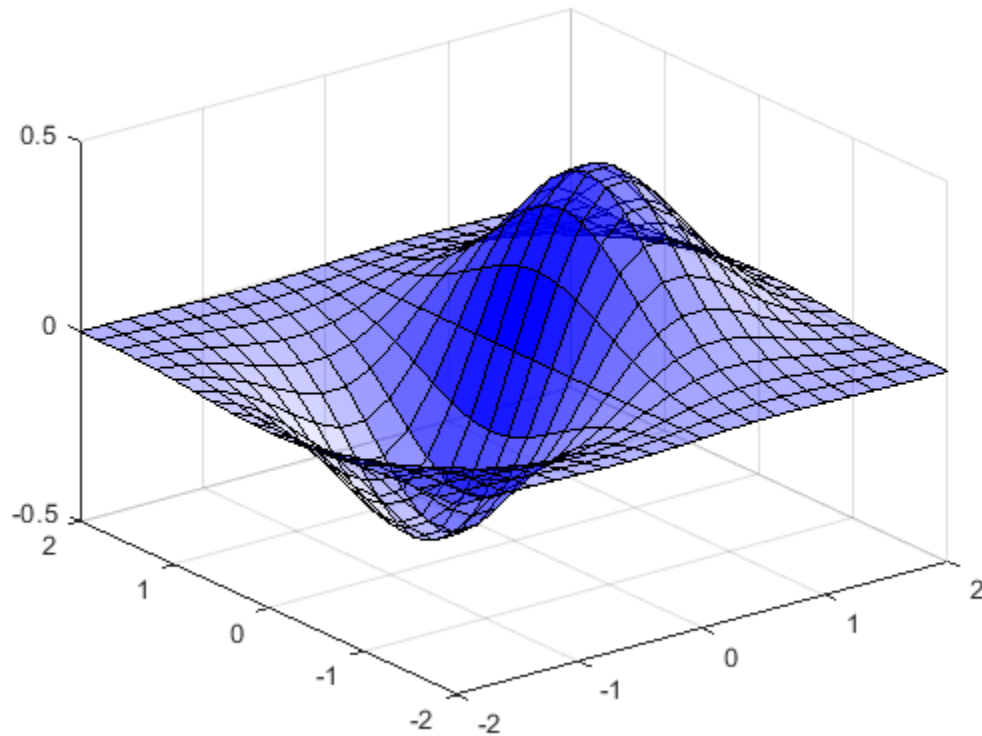

Vary Patch Object Transparency

Plot a line using the `patch` function. Set the last entry of `y` to `NaN` so that `patch` creates a line instead of a closed polygon.

Define one transparency value per vertex by setting the `FaceVertexAlphaData` property to a column vector. Interpret the values as transparency values (0 is invisible, 1 is opaque) by setting the

AlphaDataMapping property to `'none'`. Interpolate the transparency between vertices by setting the EdgeAlpha property to `'interp'`.

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

figure
alpha_values = linspace(0,1,10)';
patch(x,y,'red','EdgeColor','red',...
    'FaceVertexAlphaData',alpha_values,'AlphaDataMapping','none',...
    'EdgeAlpha','interp')
```
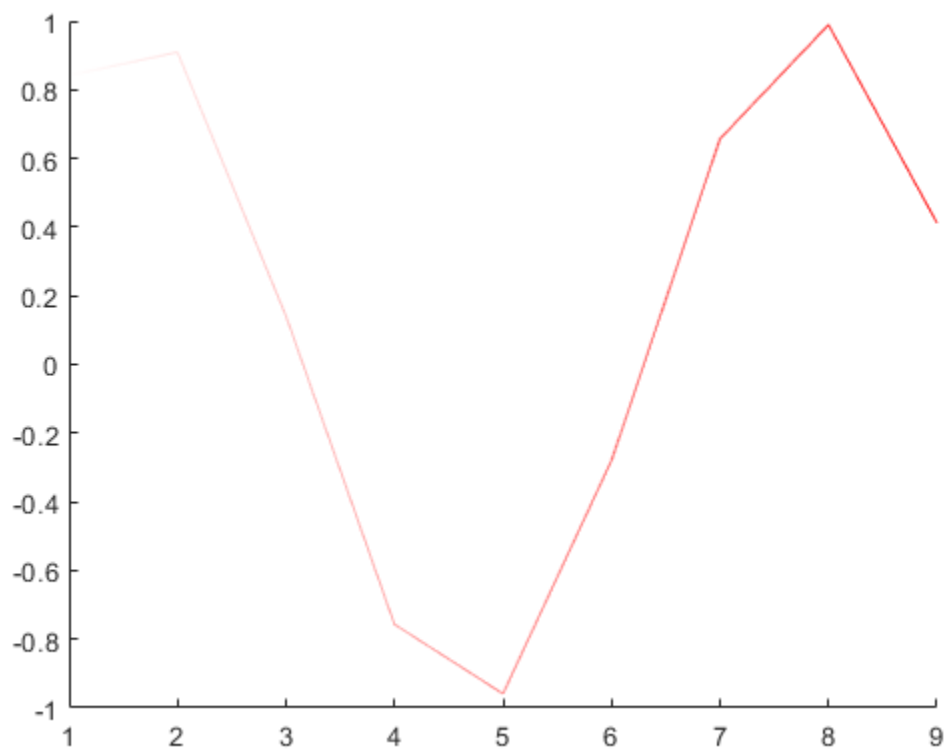

See Also

alim | alpha | alphamap | area | bar | image | patch | scatter | surf

Changing Transparency of Images, Patches or Surfaces

This example shows how to modify transparency of images, patches and surfaces.

Transparency for All Objects in Axes

Transparency values are referred to as alpha values. Use the `alpha` function to set the transparency for all image, patch, and surface objects in the current axes. Specify a transparency value between 0 (fully transparent) and 1 (fully opaque).

```
t = 0:0.1:2*pi;
x = sin(t);
y = cos(t);

figure
patch(x,y,'r')              % make a red circular patch
patch(x+0.8,y,'g')          % make a green circular path
patch(x+0.4,y+0.8,'b')      % make a blue circular path
axis square tight           % set axis to square

alpha(0.3)                  % set all patches transparency to 0.3
```
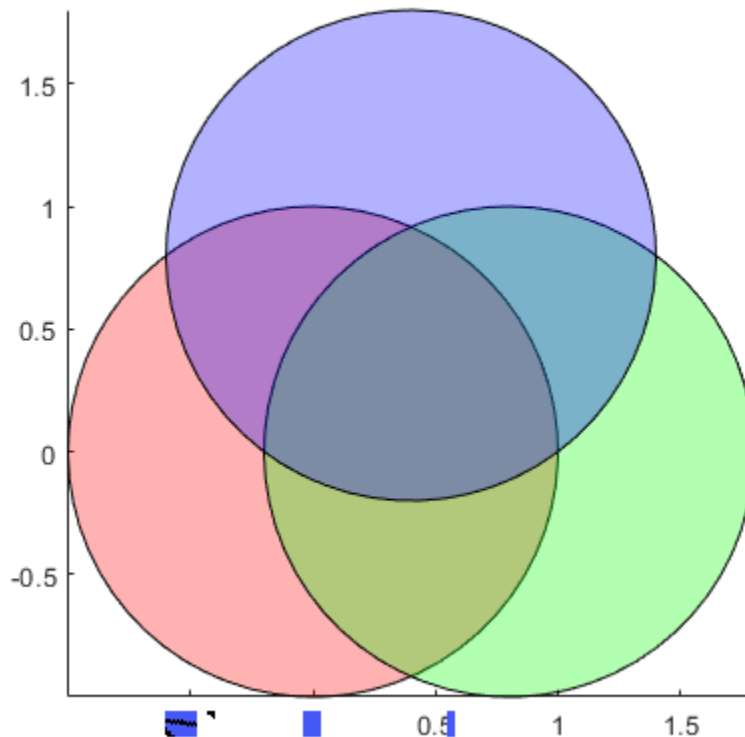

Transparency for Individual Surfaces

The transparency of a surface is defined by its `AlphaData` property. Set the alpha data as either a scalar value or a matrix of values specifying the transparency of each vertex of the surface. The

FaceAlpha property indicates how the transparency of the surface faces are determined from vertex transparency.

```
[X,Y,Z] = peaks(20);
s2 = surf(X,Y,Z);

s2.AlphaData = gradient(Z);     % set vertex transparencies
s2.FaceAlpha = 'flat';
```
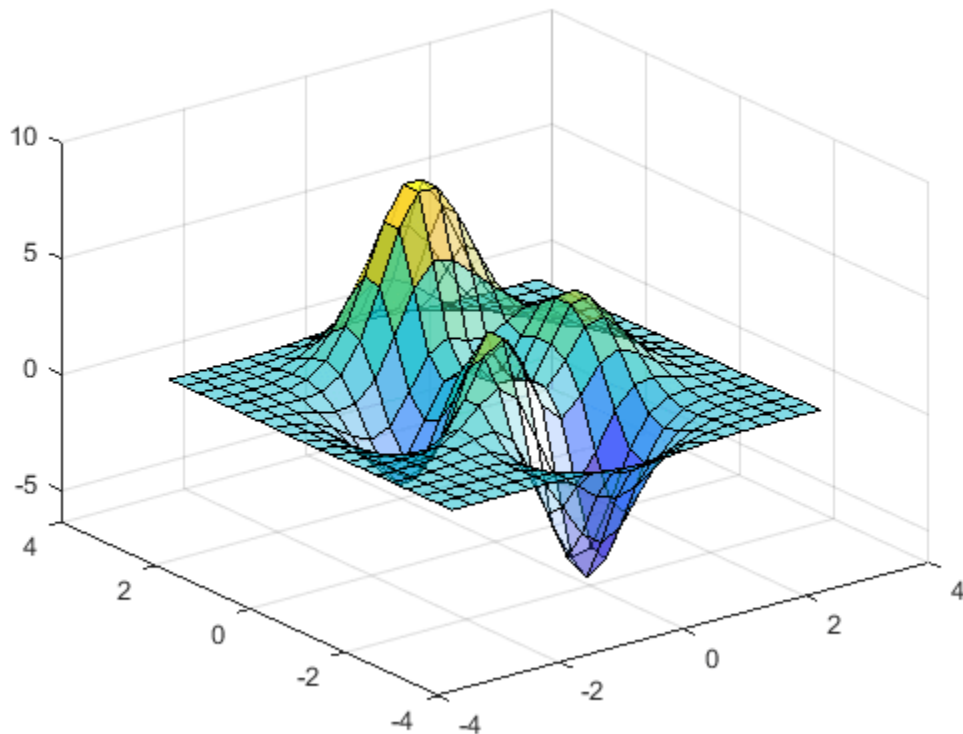

Transparency for Individual Images

Like surfaces, the transparency of an image is also defined by its AlphaData property. For images, set the alpha data as either a scalar value or a matrix of values specifying the transparency of each element in the image data.

For example, use transparency to overlay two images. First, display the image of the Earth.

```
earth = imread('landOcean.jpg');
image(earth)    % display Earth image
axis image
```
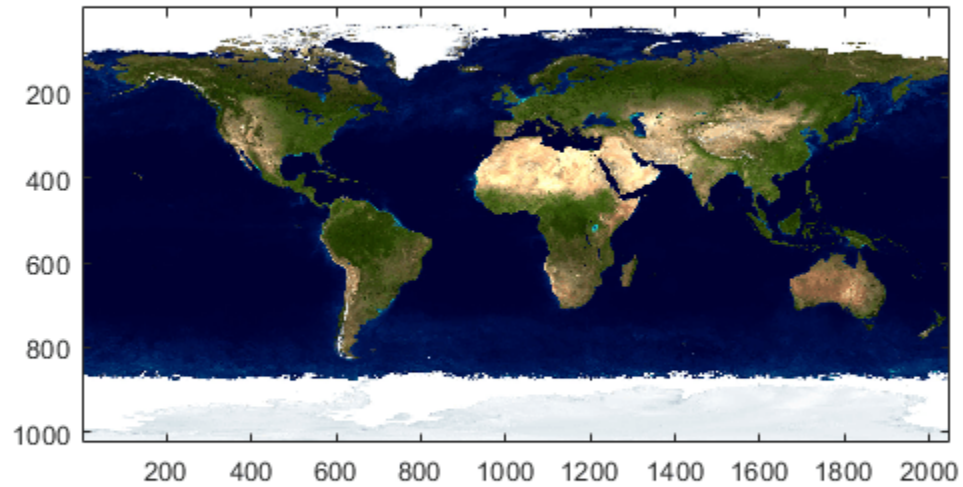
Then, add a cloud layer to the image of the Earth using transparency.

```
clouds = imread('cloudCombined.jpg');
image(earth)
axis image
hold on

im = image(clouds);
im.AlphaData = max(clouds,[],3);     % set transparency to maximum cloud value
hold off
```
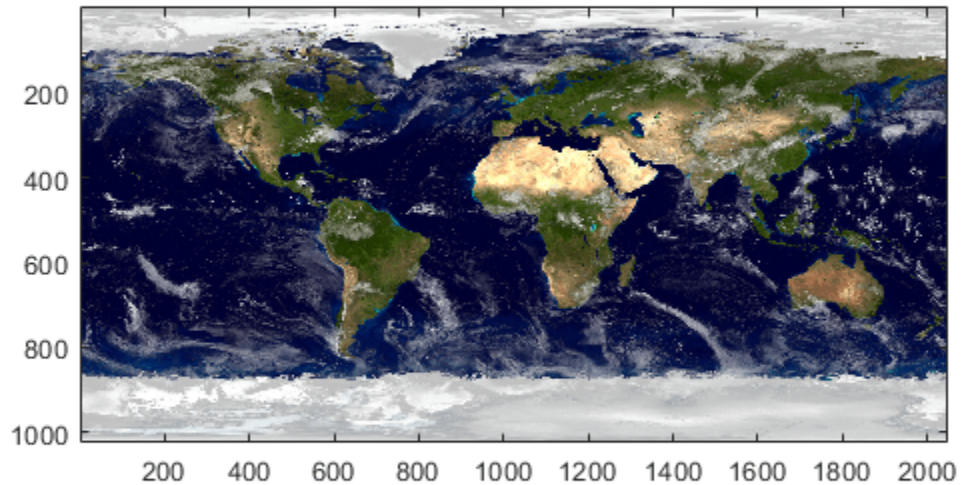
Transparency for Individual Patches

The transparency of a patch is defined by its `FaceAlpha` and `FaceVertexAlphaData` properties. For constant transparency across the entire patch, set the `FaceVertexAlphaData` to a constant between 0 (fully transparent) and 1 (fully opaque), and set the `FaceAlpha` property to `'flat'`.

```
cla
p1 = patch(x,y,'r');              % make a red circular patch
axis square tight                 % set axis to square

p1.FaceVertexAlphaData = 0.2;     % Set constant transparency
p1.FaceAlpha = 'flat' ;           % Interpolate to find face transparency
```
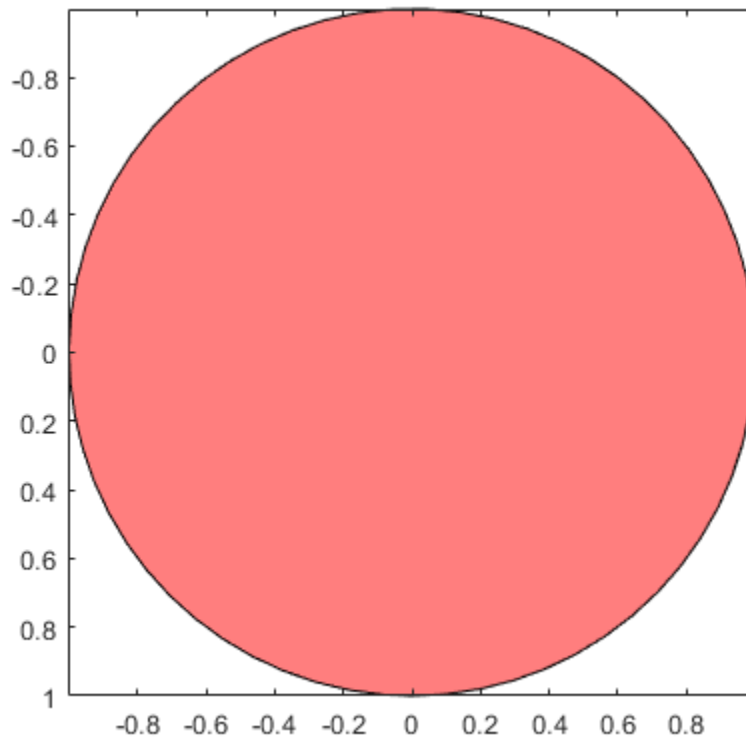
For transparency that varies across the patch, set the `FaceVertexAlphaData` to a matrix of values specifying the transparency at each vertex or each face of the patch. The `FaceAlpha` property then indicates how the face transparencies are determined using the `FaceVertexAlphaData`. If alpha data is specified for vertices, `FaceAlpha` must be set to `'interp'`.

```
p1.FaceVertexAlphaData = x';    % Set vertex transparency to x values
p1.FaceAlpha = 'interp' ;       % Interpolate to find face transparency
```
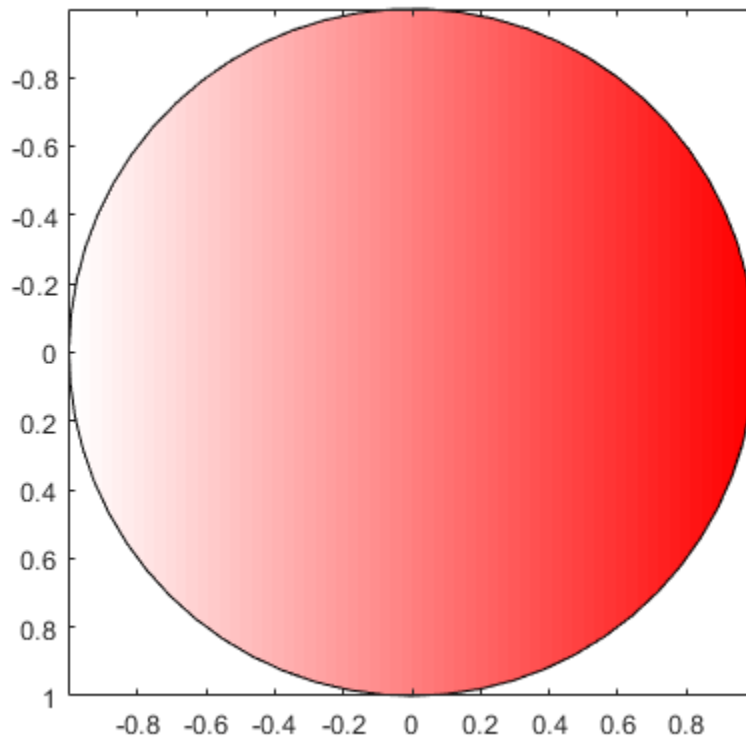
Transparency with Texture Mapping

Texture mapping maps a 2-D image onto a 3-D surface. An image can be mapped to a surface by setting the CData property to the image data and setting the FaceColor property to be 'texturemap'.

This example creates a 3-D view of the earth and clouds. It creates spherical surfaces and uses texture mapping to map the images of the earth and clouds onto the surfaces.

```
[px,py,pz] = sphere(50);                  % generate coordinates for a 50 x 50 sphere

cla
sEarth = surface(py, px ,flip(pz));
sEarth.FaceColor = 'texturemap';          % set color to texture mapping
sEarth.EdgeColor = 'none';                % remove surface edge color
sEarth.CData = earth;                     % set color data

hold on
sCloud = surface(px*1.02,py*1.02,flip(pz)*1.02);

sCloud.FaceColor = 'texturemap';          % set color to texture mapping
sCloud.EdgeColor = 'none';                % remove surface edge color
sCloud.CData = clouds;                    % set color data

sCloud.FaceAlpha = 'texturemap';          % set transparency to texture mapping
sCloud.AlphaData = max(clouds,[],3);      % set transparency data
hold off
```

```
view([80 2])                            % specify viewpoint
daspect([1 1 1])                        % set aspect ratio
axis off tight                          % remove axis and set limits to data range
```
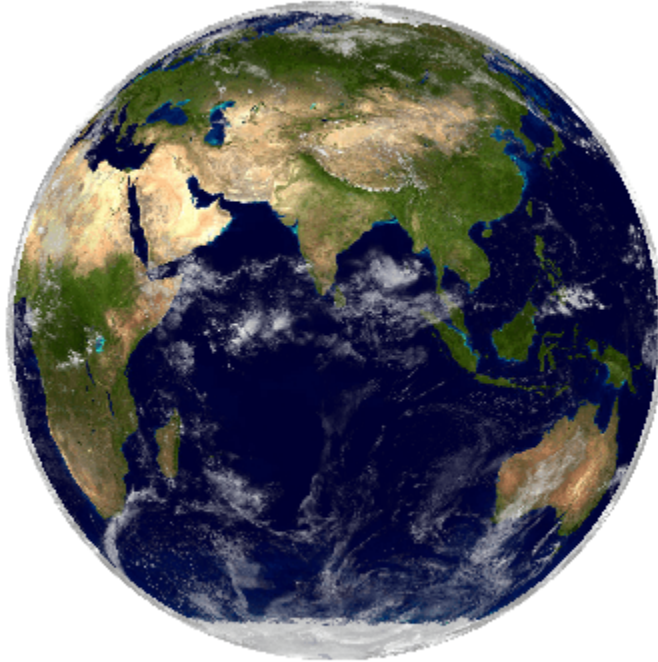

The images used in this example are from Visible Earth.

Credit: NASA Goddard Space Flight Center Image by Reto Stöckli (land surface, shallow water, clouds). Enhancements by Robert Simmon (ocean color, compositing, 3D globes, animation). Data and technical support: MODIS Land Group; MODIS Science Data Support Team; MODIS Atmosphere Group; MODIS Ocean Group Additional data: USGS EROS Data Center (topography); USGS Terrestrial Remote Sensing Flagstaff Field Center (Antarctica); Defense Meteorological Satellite Program (city lights).

See Also
alim | alpha | alphamap

Modify the Alphamap

Every figure has an associated alphamap, which is a vector of values ranging from 0 to 1. The default alphamap contains 64 values ranging linearly from 0 to 1. View or modify the alphamap using the `Alphamap` property of the figure or using the `alphamap` function.

Default Alpha Map

The default alphamap contains 64 values ranging linearly from 0 to 1, as shown in the following plot.

```
am = get(gcf,'Alphamap');
plot(am)
```
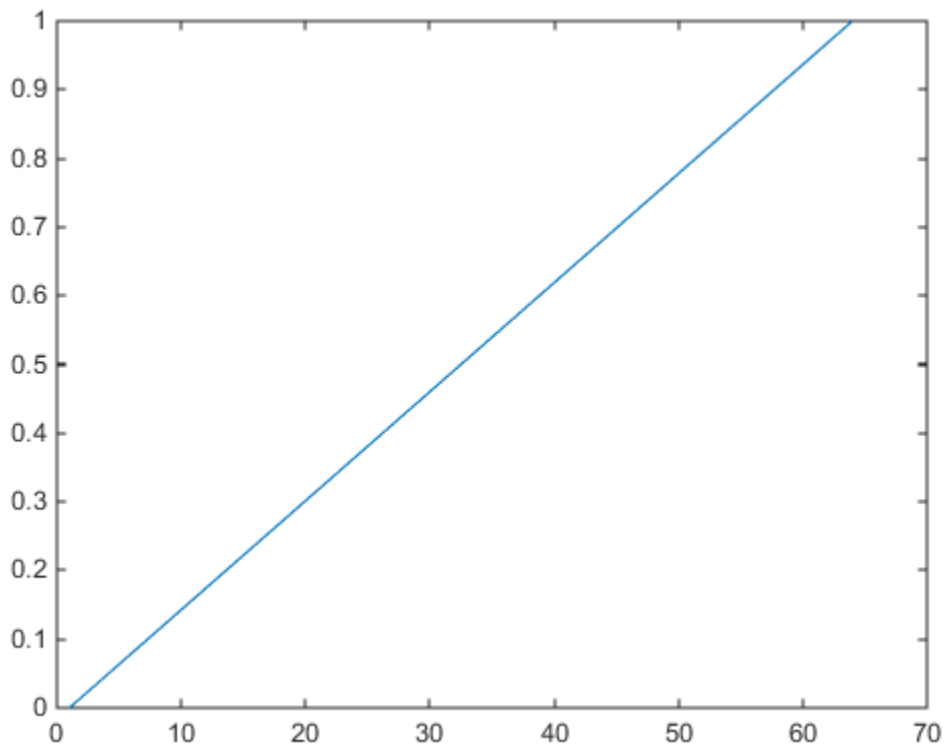

This alphamap displays the lowest alpha data values as completely transparent and the highest alpha data values as opaque.

The `alphamap` function creates some useful predefined alphamaps and also enables you to modify existing maps. For example,

```
figure;
alphamap('vup')
```

sets the figure `Alphamap` property to an alphamap whose values increase then decrease:

```
am = get(gcf,'Alphamap');
plot(am)
```
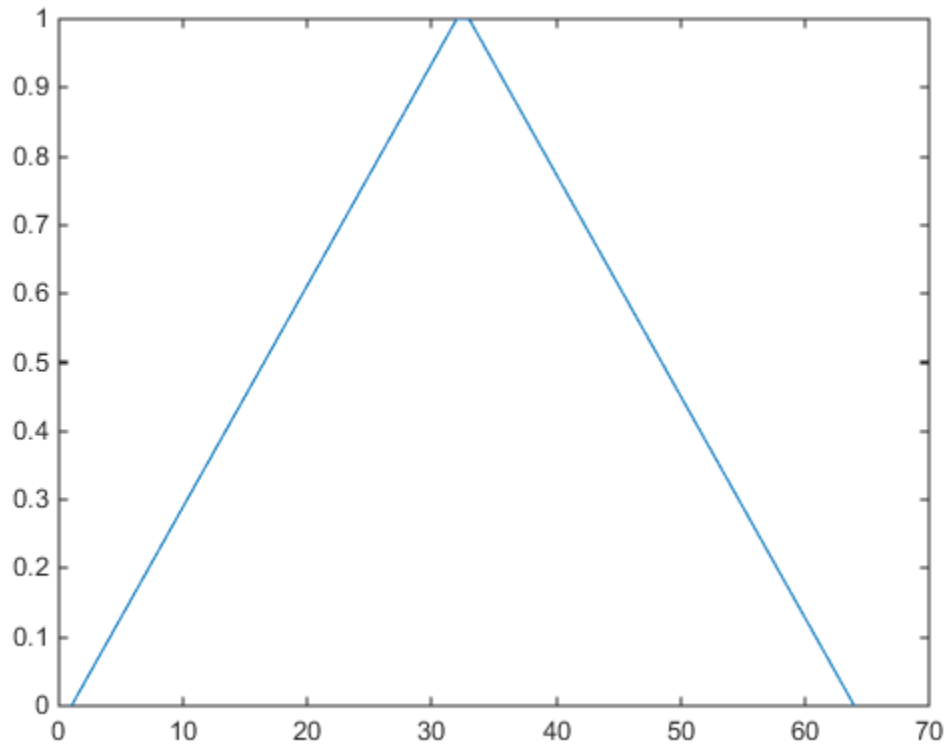
You can shift the values using the `increase` or `decrease` options. For example,

```
alphamap('increase',.4)
```

adds the value .4 to all values in the current figure's alphamap. Replotting the `'vup'` alphamap illustrates the change. The values are clamped to the range [0 1].

```
am = get(gcf,'Alphamap');
plot(am)
```
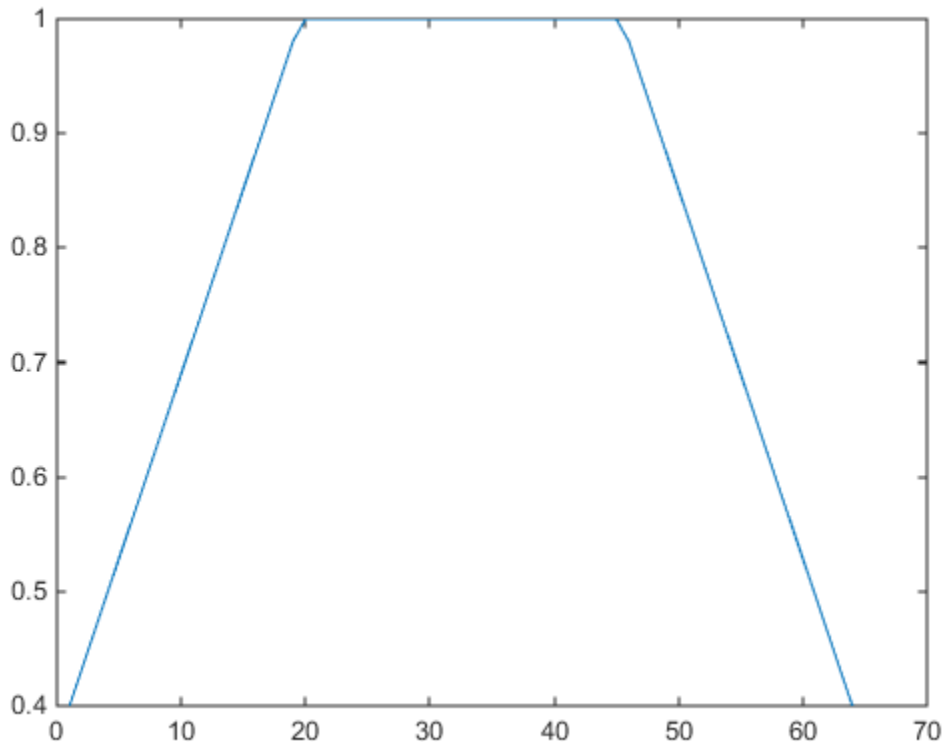
Example — Modifying the Alphamap

This example uses slice planes to examine volume data. The slice planes use the color data for alpha data and employ a rampdown alphamap (the values range from 1 to 0):

1 Create the volume data by evaluating a function of three variables.

```
[x,y,z] = meshgrid(-1.25:.1:-.25,-2:.2:2,-2:.1:2);
v = x.*exp(-x.^2-y.^2-z.^2);
```

2 Create the slice planes, set the alpha data equal to the color data, and specify interpolated FaceColor and FaceAlpha.

```
h = slice(x,y,z,v,[-1 -.75 -.5],[],[0]);
set(h,'EdgeColor','none',...
'FaceColor','interp',...
'FaceAlpha','interp')
alpha('color')
```

3 Install the rampdown alphamap and increase each value in the alphamap by .1 to achieve the desired degree of transparency. Specify the hsv colormap.

```
alphamap('rampdown')
alphamap('increase',.1)
colormap hsv
```

This alphamap displays the smallest values of the function (around zero) with the least transparency and the greatest values display with the most transparency. This enables you to see through the slice planes, while at the same time preserving the data around zero.
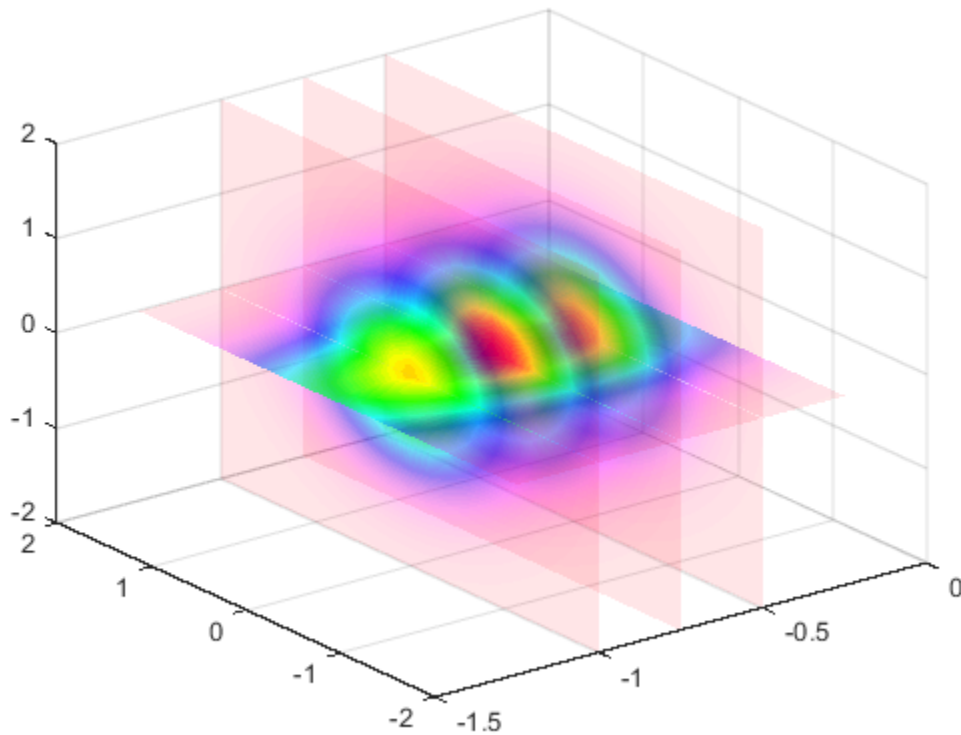
See Also

Related Examples

*   "Add Transparency to Graphics Objects" on page 12-2

Data Exploration

Interactively Explore Plotted Data

You can interactively explore and edit plotted data to improve the visual display of the data or reveal additional information about the data. The interactions available depend on the contents of the axes, but typically include zooming, panning, rotating, data tips, data brushing, and restoring the original view.

Some types of interactions are available through the axes toolbar. The toolbar appears at the top-right corner of the axes when you hover over the chart area.

Other types of interactions are built into the axes and correspond to gestures, such as dragging to pan or scrolling to zoom. These interactions are separate from those in the axes toolbar.

Note In R2018a and previous releases, the interaction options appear in the figure toolbar instead of the axes toolbar. Also, in previous releases, none of the gesture-based interactions are built into the axes.

Zoom, Pan, and Rotate Data

Zooming, panning, and rotating the axes let you explore different views of your data. By default, you can scroll or pinch to zoom in and out of the view of the axes. Also, you can drag to pan (2-D view) or drag to rotate (3-D view).

You can enable more interactions by clicking the zoom in ⊕, zoom out ⊖, pan 🖑, and rotate ⊛ buttons in the axes toolbar. For example, click the zoom-in button if you want to drag a rectangle to zoom into a region of interest.

Display Data Values Using Data Tips

To identify the values of data points in your chart, create data tips. Data tips appear temporarily as you hover over the data points in your chart. To display persistent (pinned) data tips, click one or more data points. Alternatively, select the data tips button ⊟ in the axes toolbar and then click a data point. To pin multiple data tips using the data tips button, hold down the **Shift** key. To bring a data tip in front of other data tips that overlap with it, click on it.
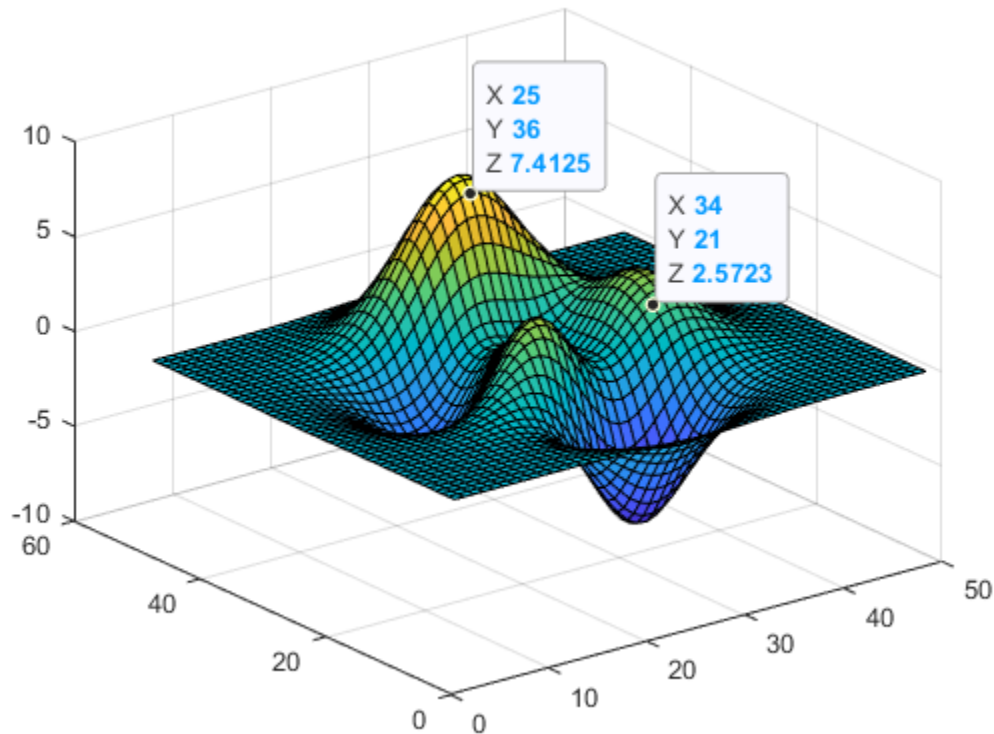
Note In MATLAB Online™, you might experience some differences in data tip interactivity. For example, in some cases, you cannot click to bring a data tip in front of other data tips that overlap with it.

Select and Modify Data Values Using Data Brushing

You can use data brushing to select, remove, or replace individual data values. To brush data, select the data brushing button ⊿ from the axes toolbar. Click a data point to highlight it or drag a rectangle to highlight all the data points within the rectangle. Use the **Shift** key to highlight additional data points.

After you highlight the desired data points, you can use the options in the right-click context menu to remove, replace, or copy the values. The displayed plot shows your changes. Also, you see the updates to the data properties of the plotted object update (such as XData). However, the original workspace variables are not updated. Then, if you want to update the workspace variables as well, you can use the **Link** option on the figure **Tools** menu to link the variables to the plot.

Remove Outliers from Plotted Data

This example shows how to use data brushing to delete an outlier from a plot of 100 data points.

First, plot the data containing a single outlier. Then, select the data brushing button ⊿ from the axes toolbar and drag a rectangle around the outlier.

```
x = linspace(0,10);
y = exp(.1*x).*sin(3*x);
y(60) = 2.7;
plot(x,y)
```
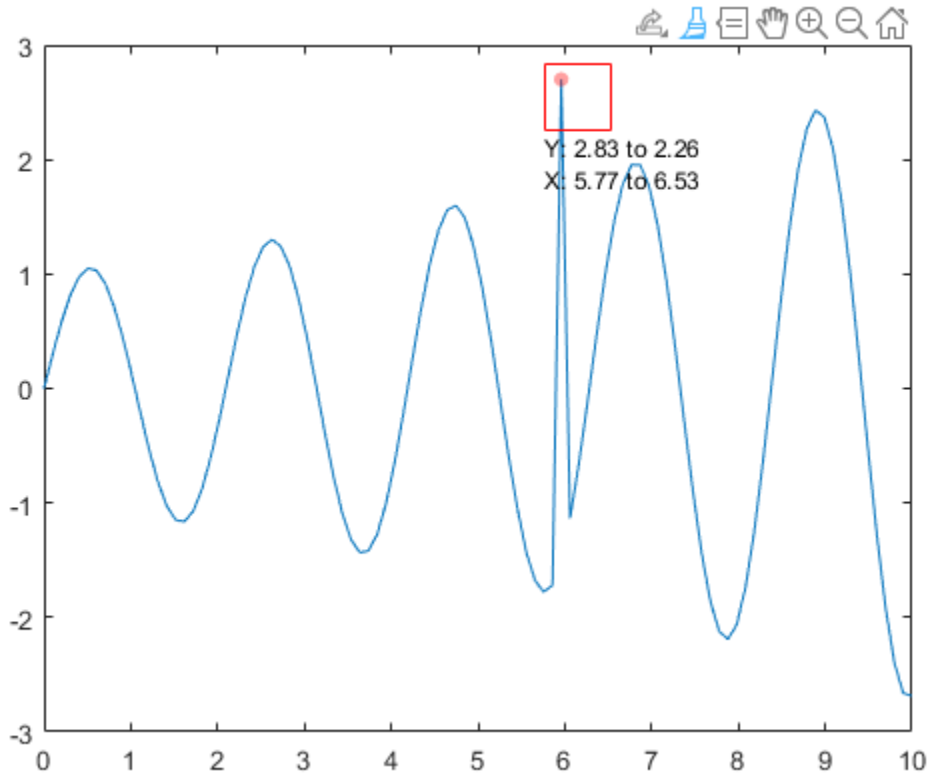

Right-click the brushed data point and select **Remove** from the context menu. Notice that the plot updates. However, the workspace variable does not change.

If you want to remove the point from the workspace variable, then select the **Link** option from the figure **Tools** menu before brushing the data.

Customize Plots Using Property Inspector

You can modify plots interactively by using the **Property Inspector**. When you open the Property Inspector and select a plot, the inspector displays a list of properties that you can edit. To open the inspector, use the `inspect` function or click the Property Inspector button ▤ on the figure toolbar.
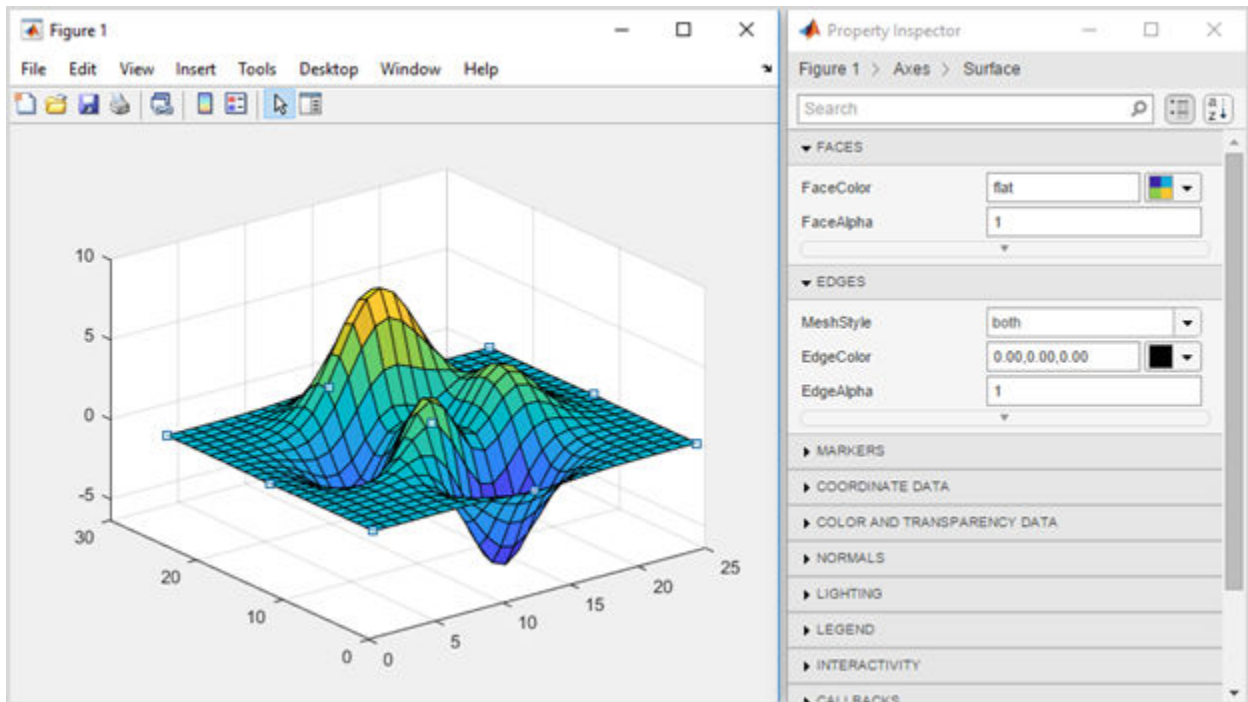
See Also

Property Inspector | brush | datacursormode | linkdata | pan | rotate3d | zoom

More About

- "Automatically Refresh Plot After Changing Data" on page 13-9
- "Create Custom Data Tips" on page 13-6

Create Custom Data Tips

Data tips appear when you hover over a data point. By default, data tips show the coordinates of the selected point. However, for some types of charts, you can customize the information that appears in the data tip, such as changing the labels or adding new rows.

Charts that support these customizations have a `DataTipTemplate` property, for example, `Line` objects created with the `plot` function.

Change Labels and Add Row

Modify the contents of data tips on a scatter plot. First, load sample accident data and create the scatter plot. Then, create a data tip interactively or by using the `datatip` function. By default, data tips show the coordinates of the data point.

```
load('accidents.mat','hwydata','statelabel')
s = scatter(hwydata(:,5),hwydata(:,4));
dt = datatip(s,11246.7,1493);
```
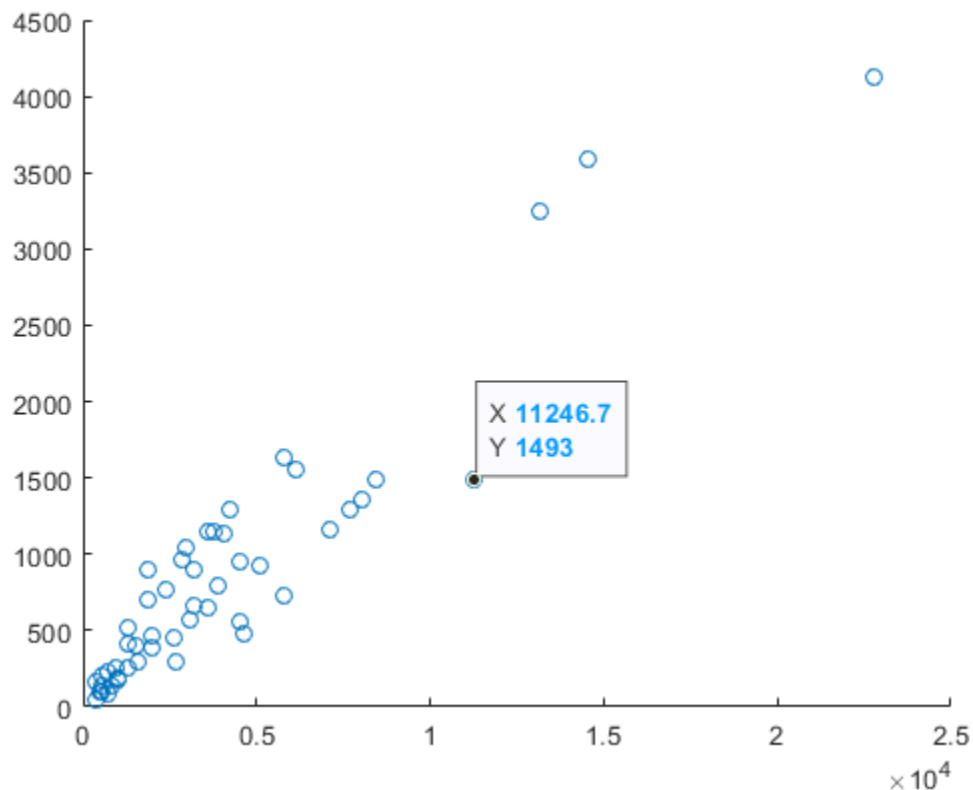

Change the data tip labels from X and Y to `Drivers (in thousands)` and `Fatalities` by accessing the `DataTipTemplate` property of the plotted object and setting the `Label` properties.

```
s.DataTipTemplate.DataTipRows(1).Label = 'Drivers (in thousands)';
s.DataTipTemplate.DataTipRows(2).Label = 'Fatalities';
```

Add a new row to the data tip. For the label, use `State`. For the value, use the state names contained in the `statelabel` variable in your workspace.

```
row = dataTipTextRow('State',statelabel);
s.DataTipTemplate.DataTipRows(end+1) = row;
```
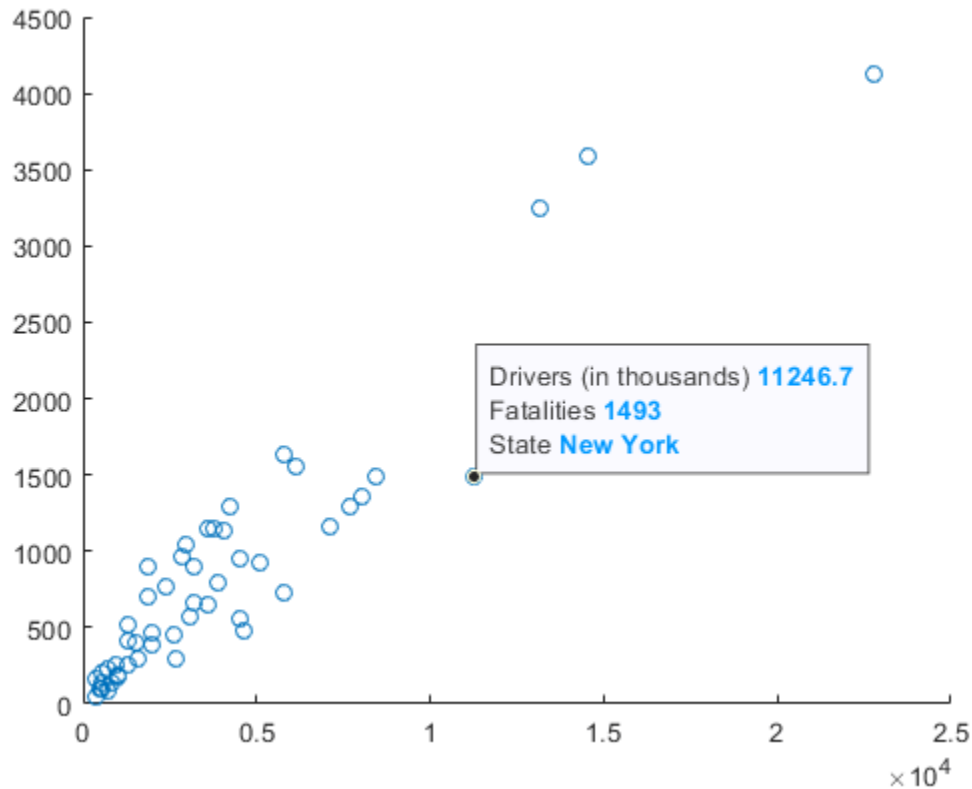

Show Table Values in Data Tips

Modify the contents of data tips for a scatter plot to include values from a table. First, create a table from a sample spreadsheet of patient data. Plot the data. Then, create a data tip interactively or by using the `datatip` function.

```
tbl = readtable('patients.xls');
s = scatter(tbl.Height,tbl.Weight);
dt = datatip(s,64,142);
```

Change the data tip labels from X and Y to `Height` and `Weight`. Then, add a new row to the data tip that uses the label `Age` and shows the values from the Age column of the table.

```
s.DataTipTemplate.DataTipRows(1).Label = 'Height';
s.DataTipTemplate.DataTipRows(2).Label = 'Weight';
row = dataTipTextRow('Age',tbl.Age);
s.DataTipTemplate.DataTipRows(end+1) = row;
```
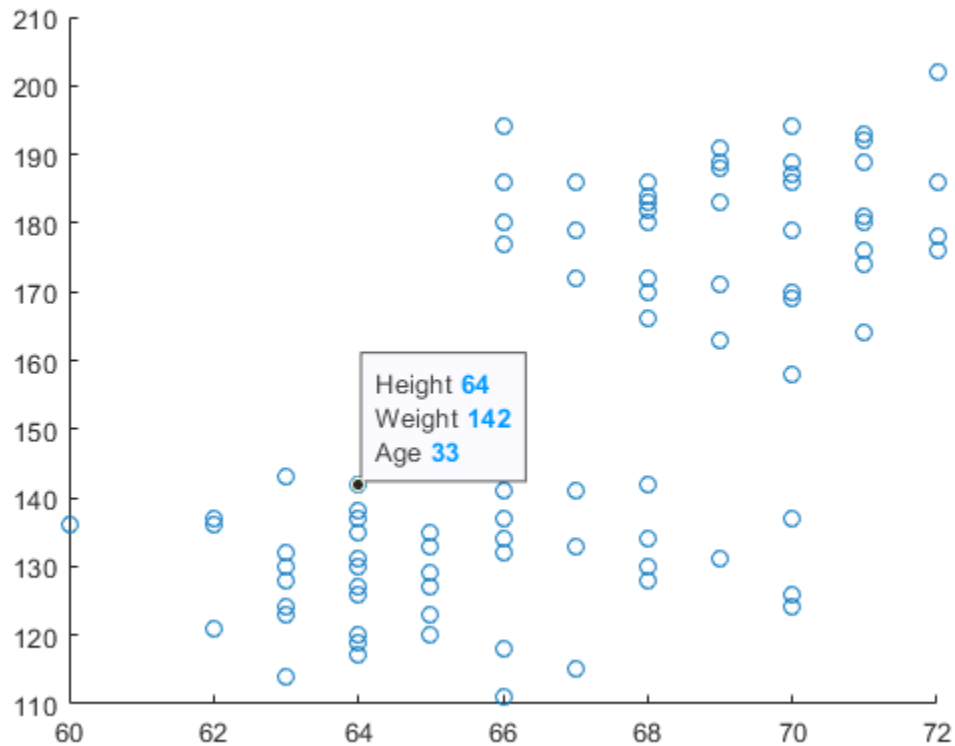
If you are using R2018b or earlier, customize the data tips by setting the `UpdateFcn` property of the `datacursormode` object instead of using the `DataTipTemplate` object.

See Also

DataTipTemplate | `dataTipTextRow` | `datatip`

More About

- "Interactively Explore Plotted Data" on page 13-2
- "Automatically Refresh Plot After Changing Data" on page 13-9

Automatically Refresh Plot After Changing Data

When you plot data from workspace variables, the plots contain copies of the variables. As a result, if you change the workspace variable (such as add or delete data) the plots do not automatically update. If you want the plot to reflect the change, you must replot it. However, you can use one of these techniques to link plots to the workspace variables they represent. When you link plots and workspace variables, changing the data in one place also changes it in the other.

- Use data linking to link the plot to workspace variables.
- Set the data source properties of the plotted object (such as the `XDataSource` property) to the names of the workspace variables. Then, call the `refreshdata` function to update the data properties indirectly. You can use this technique to create animations.

Update Plot Using Data Linking

Data linking keeps plots continuously synchronized with the workspace variables they depict.

For example, iteratively approximate `pi`. Create the variable `x` to represent the iteration number and `y` to represent the approximation. Plot the initial values of `x` and `y`. Turn on data linking using `linkdata on` so that the plot updates when the variables change. Then, update `x` and `y` in a for loop. The plot updates at half-second intervals.

```
x = [1 2];
y = [4 4];
plot(x,y);
xlim([0 100])
ylim([2.5 4])
xlabel('Iteration')
ylabel('Approximation for \pi')

linkdata on

denom = 1;
k = -1;
for t = 3:100
    denom = denom + 2;
    x(t) = t;
    y(t) = 4*(y(t-1)/4 + k/denom);
    k = -k;
end
```
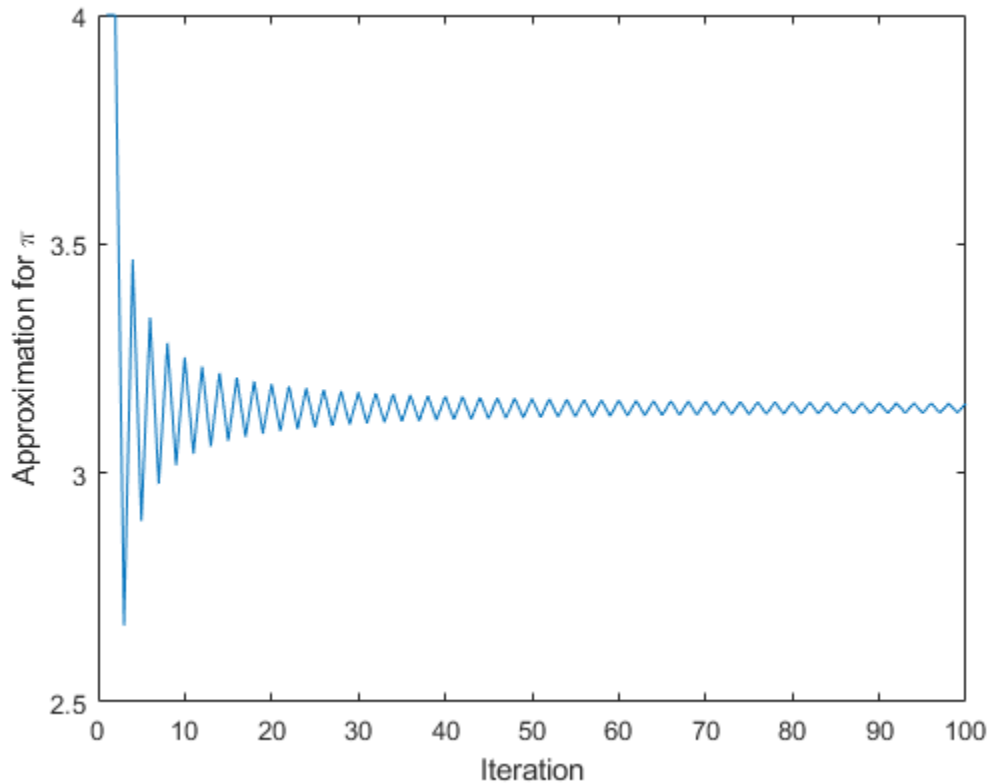
Update Plot Using Data Source Properties

Instead of using the data linking feature, you can keep the plot synchronized with the workspace variables by setting the data source properties of the plotted object. You can use this technique to create animations.

For example, iteratively approximate `pi`. Create the variable `x2` to represent the iteration number and `y2` to represent the approximation. Plot the initial values of `x2` and `y2`. Link the plot to the workspace variables by setting the data source properties of the plotted object to `'x2'` and `'y2'`. Then, update `x2` and `y2` in a for loop. Call `refreshdata` and `drawnow` each iteration to update the plot based on the updated data.

```
x2 = [1 2];
y2 = [4 4];
p = plot(x2,y2);
xlim([0 100])
ylim([2.5 4])
xlabel('Iteration')
ylabel('Approximation for \pi')

p.XDataSource = 'x2';
p.YDataSource = 'y2';

denom = 1;
k = -1;
for t = 3:100
```

```
    denom = denom + 2;
    x2(t) = t;
    y2(t) = 4*(y2(t-1)/4 + k/denom);
    refreshdata
    drawnow
    k = -k;
end
```
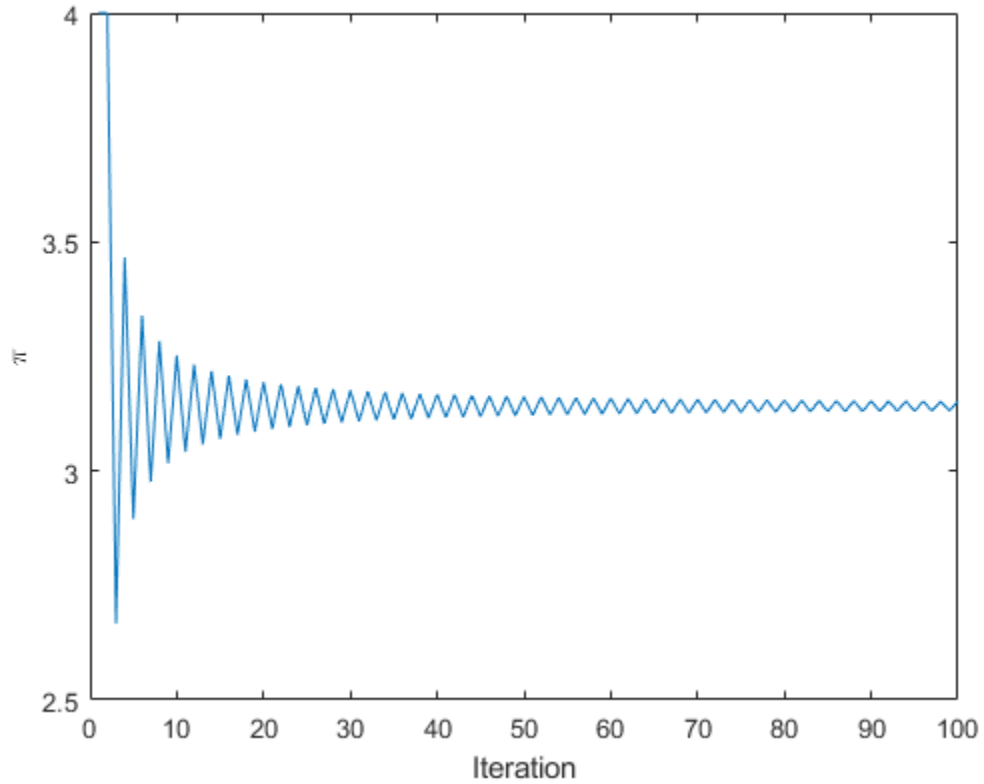

See Also

brush | linkaxes | linkdata | refreshdata

More About

- "Interactively Explore Plotted Data" on page 13-2

Control Chart Interactivity

You can interactively explore and edit plotted data to improve the visual display of the data or reveal additional information about the data. The interactions available depend on the contents of the axes, but typically include zooming, panning, rotating, data tips, data brushing, and restoring the original view.

Some types of interactions are available through the axes toolbar. The toolbar appears at the top-right corner of the axes when you hover over the chart area.

Other types of interactions are built into the axes and are available through gestures, such as dragging to pan or scrolling to zoom. These interactions are controlled separately from those in the axes toolbar.

When you create a chart, you can control the set of available interactions in several ways:

- Show or hide the axes toolbar on page 13-12.
- Customize the axes toolbar on page 13-12.
- Enable or disable built-in interactions on page 13-14.
- Customize the built-in interactions on page 13-14.

In R2018a and previous releases, many of the interaction options appear in the figure toolbar instead of the axes toolbar. Also, in previous releases, none of the interactions are built into the axes.

Show or Hide Axes Toolbar

To show or hide the axes toolbar, set the `Visible` property of the `AxesToolbar` object to `'on'` or `'off'`, respectively. For example, hide the toolbar for the current axes:

```
ax = gca;
ax.Toolbar.Visible = 'off';
```

Customize Axes Toolbar

You can customize the options available in the axes toolbar using the `axtoolbar` and `axtoolbarbtn` functions.

Note Custom toolbars do not appear in figures in the Live Editor. To see the custom toolbar, open the figure in a separate figure window.

For example, add a custom state button for the axes toolbar that turns on and off the axes grid lines. First, create a program file called `mycustomstatebutton.m`. Within the program file:

- Plot random data.
- Create a toolbar for the axes with options to zoom in, zoom out, and restore the view using the `axtoolbar` function.

- Add an empty state button to the toolbar using the `axtoolbarbtn` function. Return the `ToolbarStateButton` object.
- Specify the icon, tool tip, and callback function for the state button by setting the `Icon`, `Tooltip`, and `ValueChangedFcn` properties. This example uses the ⊞ icon, which you must first save as an image file called `mygridicon.png` on your path.

When you run the program file, click the icon to turn on and off the grid lines.

```matlab
function mycustomstatebutton

plot(rand(5))
ax = gca;
tb = axtoolbar(ax,{'zoomin','zoomout','restoreview'});

btn = axtoolbarbtn(tb,'state');
btn.Icon = 'mygridicon.png';
btn.Tooltip = 'Grid Lines';
btn.ValueChangedFcn = @customcallback;

    function customcallback(src,event)
        switch src.Value
            case 'off'
                event.Axes.XGrid = 'off';
                event.Axes.YGrid = 'off';
                event.Axes.ZGrid = 'off';
            case 'on'
                event.Axes.XGrid = 'on';
                event.Axes.YGrid = 'on';
                event.Axes.ZGrid = 'on';
        end
    end

end
```
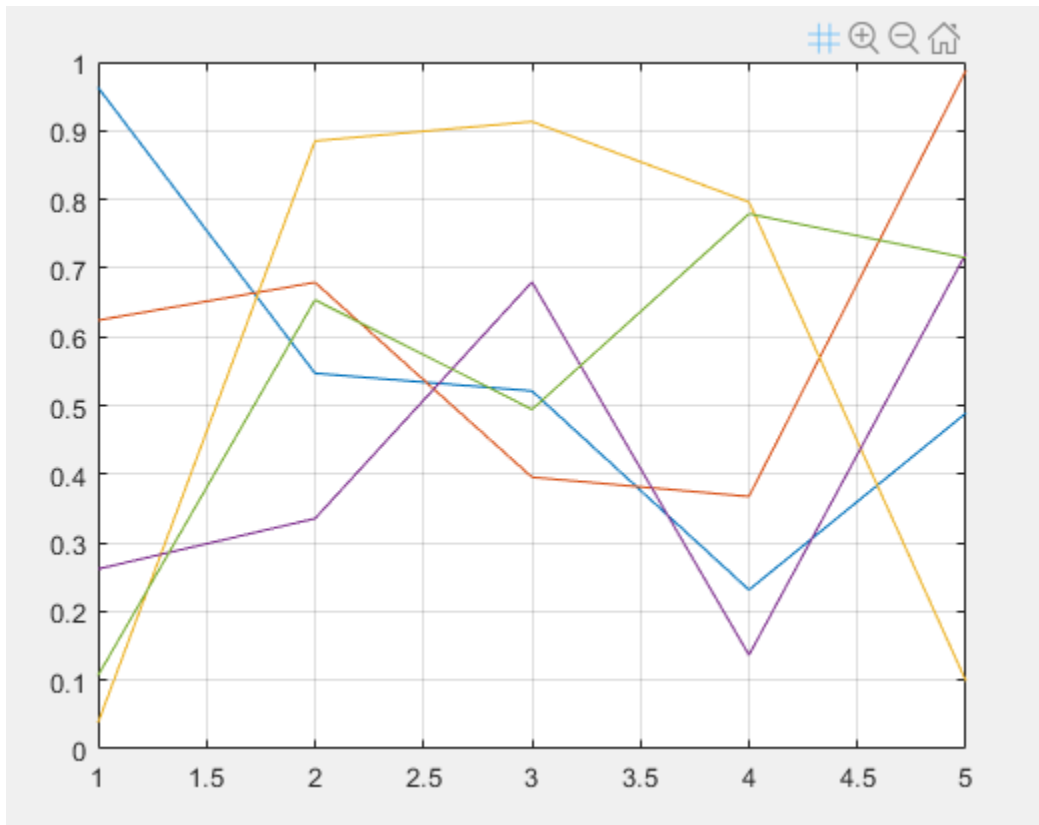
Enable or Disable Built-In Interactions

To control whether a set of built-in interactions is enabled within a chart, use the `disableDefaultInteractivity` and `enableDefaultInteractivity` functions. Sometimes MATLAB automatically disables the built-in interactions. For example, they might be disabled for charts that have special features, or when you implement certain callbacks such as a `WindowScrollWheelFcn`.

Customize Built-In Interactions

Most types of axes include a default set of built-in interactions that correspond to specific gestures. The interactions that are available depend on the contents of the axes. Most Cartesian axes include interactions for scrolling to zoom, hovering or clicking to display data tips, and dragging to pan (in a 2-D view) or rotate (in a 3-D view). You can replace the default set with a new set of interactions, but you cannot access or modify any of the interactions in the default set.

Note Customizing or replacing the built-in interactions is not supported in the Live Editor.

To replace the default interactions, set the `Interactions` property of the axes to an array of interaction objects. Choose a compatible combination of interaction objects from the following table. To delete all interactions from the axes, set the property to an empty array (`[]`).

Interaction Object	Description	Compatible Interactions
panInteraction	Pan within a chart by dragging.	All except regionZoomInteraction and rotateInteraction
rulerPanInteraction	Pan an axis by dragging it.	All
zoomInteraction	Zoom by scrolling or pinching.	All
regionZoomInteraction	Zoom into a rectangular region by dragging. (For 2-D Cartesian charts only)	All except panInteraction and rotateInteraction
rotateInteraction	Rotate a chart by dragging it.	All except panInteraction and regionZoomInteraction
dataTipInteraction	Display data tips by hovering, clicking, or tapping.	All

For example, create a plot containing 1000 scattered points.

```
x = 1:500;
y = randn(1,500);
y2 = 5*randn(1,500) + 10;
plot(x,y,'.',x,y2,'.')
```
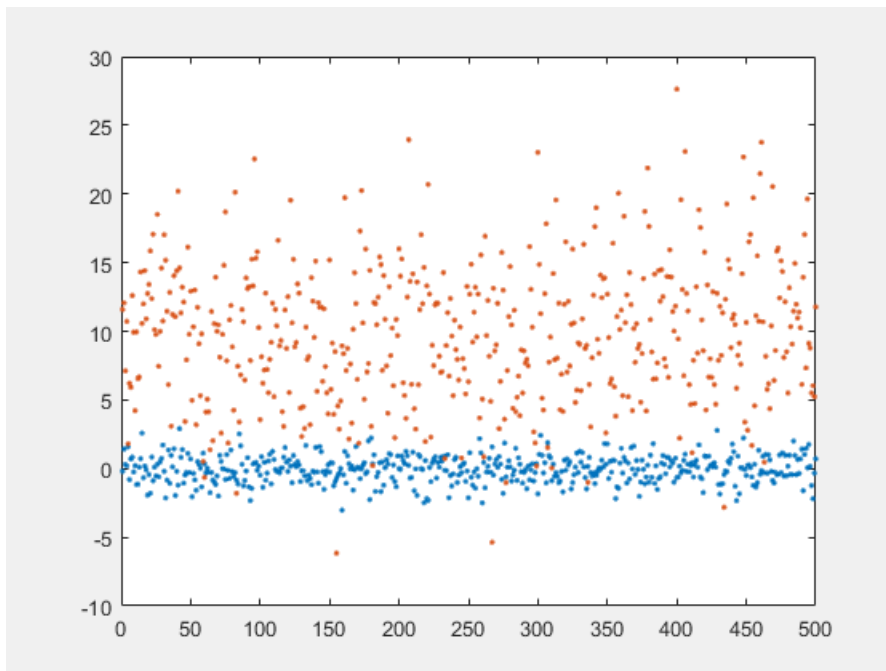

By default, this plot has a set of interactions that includes dragging to pan within the chart area. However, because the plot has a dense collection of points, a more useful set of interactions might include one that allows you to zoom into specific regions of the plot. The regionZoomInteraction object provides this functionality. Replace the default set of interactions for the current axes with an array that includes the regionZoomInteraction object.

```
ax = gca;
ax.Interactions = [zoomInteraction regionZoomInteraction rulerPanInteraction];
```

Now, dragging within the plot area defines a rectangular region of interest to zoom into.
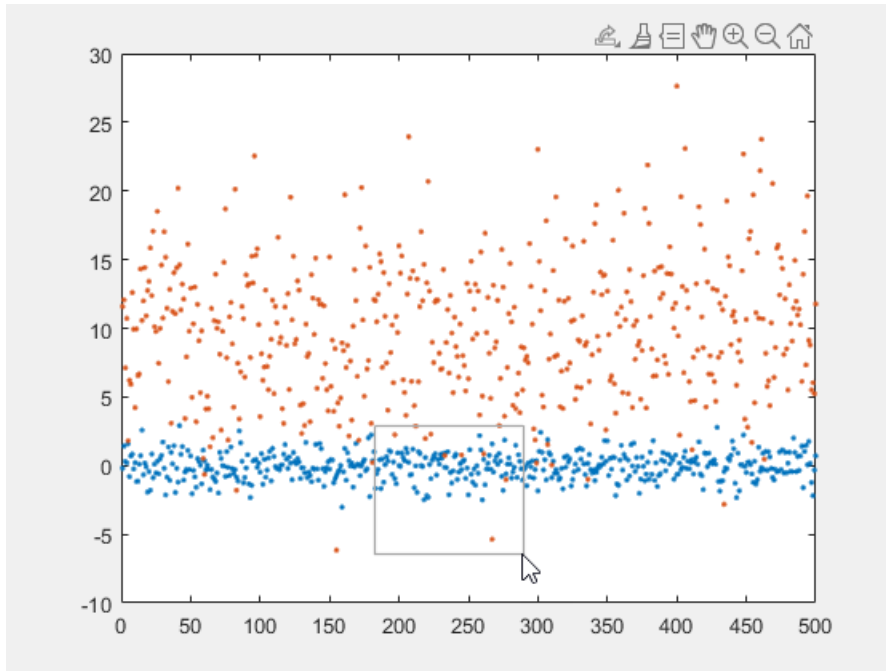
See Also

Functions
axtoolbar

Properties
Axes | AxesToolbar | ToolbarPushButton | ToolbarStateButton

More About

- "Interactively Explore Plotted Data" on page 13-2

Camera Views

View Overview

In this section...
"Viewing 3-D Graphs and Scenes" on page 14-2
"Positioning the Viewpoint" on page 14-2
"Setting the Aspect Ratio" on page 14-2
"Default Views" on page 14-2

Viewing 3-D Graphs and Scenes

The *view* is the particular orientation you select to display your graph or graphical scene. The term *viewing* refers to the process of displaying a graphical scene from various directions, zooming in or out, changing the perspective and aspect ratio, flying by, and so on.

This section describes how to define the various viewing parameters to obtain the view you want. Generally, viewing is applied to 3-D graphs or models, although you might want to adjust the aspect ratio of 2-D views to achieve specific proportions or make a graph fit in a particular shape.

MATLAB viewing is composed of two basic areas:

- Positioning the viewpoint to orient the scene
- Setting the aspect ratio and relative axis scaling to control the shape of the objects being displayed

Positioning the Viewpoint

- "Setting the Viewpoint with Azimuth and Elevation" on page 14-4 — Discusses how to specify the point from which you view a graph in terms of azimuth and elevation. This is conceptually simple, but does have limitations.
- "View Control with the Camera Toolbar" on page 14-9 — How to compose complex scenes using the MATLAB camera viewing model.
- "Moving the Camera Through a Scene" on page 14-19 — Programming techniques for moving the view around and through scenes.
- "Low-Level Camera Properties" on page 14-22 — The graphics properties that control the camera and illustrates the effects they cause.

Setting the Aspect Ratio

- "Understanding View Projections" on page 14-27 — Describes orthographic and perspective projection types and illustrates their use.
- "Manipulating Axes Aspect Ratio" on page 9-77 — How MATLAB sets the aspect ratio of the axes and how you can select the most appropriate setting for your graphs.

Default Views

MATLAB automatically sets the view when you create a graph. The actual view that MATLAB selects depends on whether you are creating a 2- or 3-D graph. See "Default Viewpoint Selection" on page

14-22 and "Default Aspect Ratio Selection" on page 9-78 for a description of how MATLAB defines the standard view.

Setting the Viewpoint with Azimuth and Elevation

Azimuth and Elevation

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

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

This diagram illustrates the coordinate system. The arrows indicate positive directions.
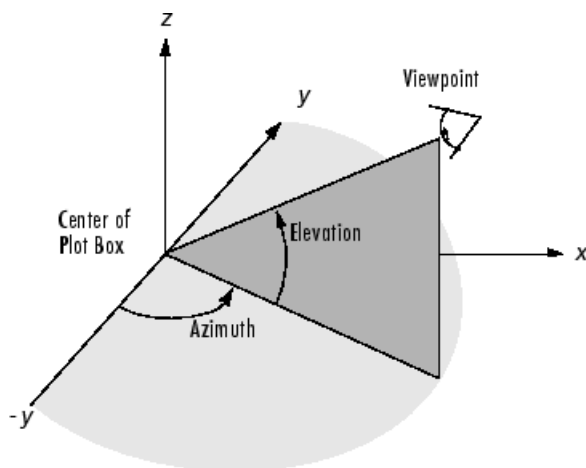
Default 2-D and 3-D Views

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

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

Examples of Views Specified with Azimuth and Elevation

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

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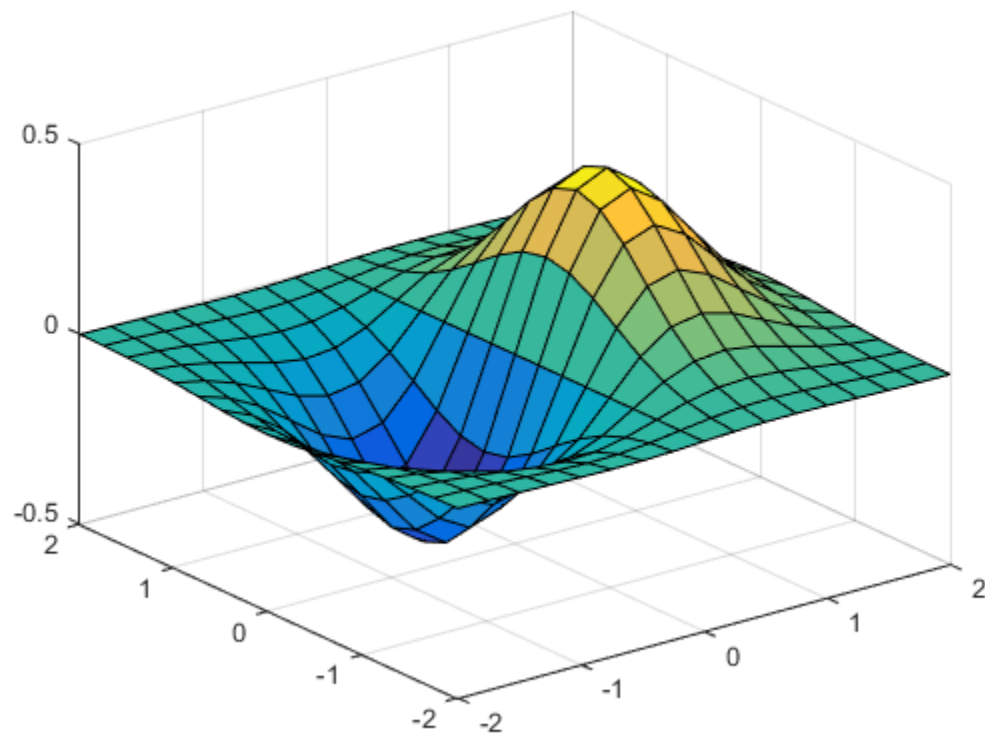
The statement

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

sets the viewpoint so you are looking in the negative *y*-direction with your eye at the $z = 0$ elevation.
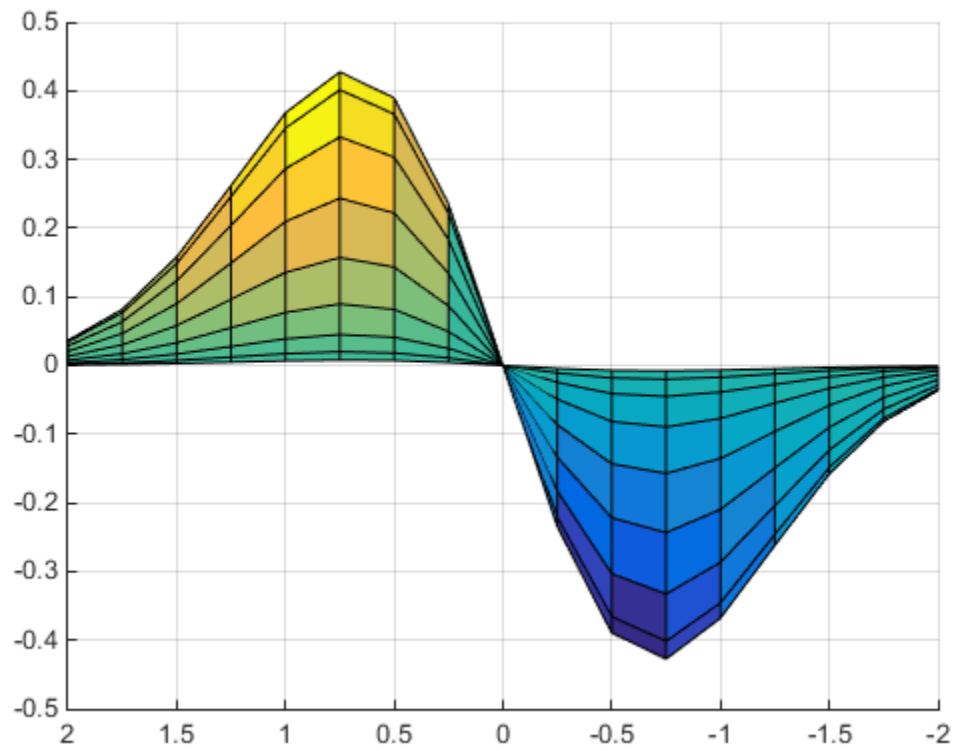
You can move the viewpoint to a location below the axis origin using a negative elevation.

```
view([-37.5 -30])
```
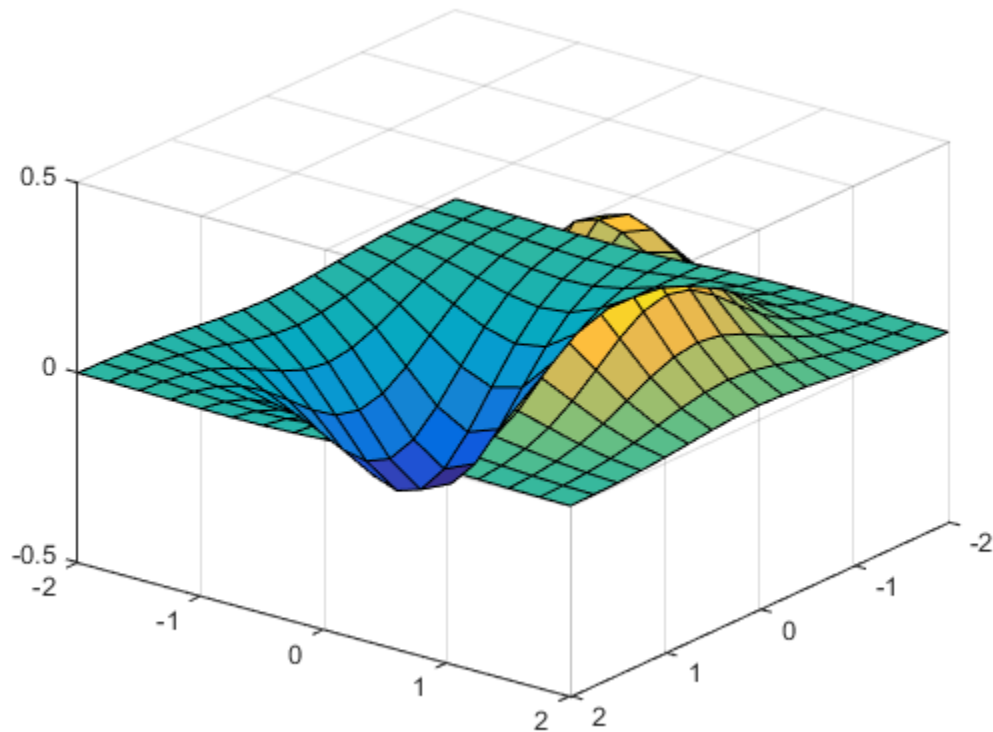
Limitations of Azimuth and Elevation

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

MATLAB camera graphics provides greater control than the simple adjustments allowed with azimuth and elevation.

See Also

More About

Camera Graphics Terminology

When you look at the graphics objects displayed in an axes, you are viewing a scene from a particular location in space that has a particular orientation with regard to the scene. MATLAB Graphics provides functionality, analogous to that of a camera with a zoom lens, that enables you to control the view of the scene created by MATLAB.

This picture illustrates how the camera is defined in terms of properties of the axes. The view is the 2-D projection of the plot box onto the screen.
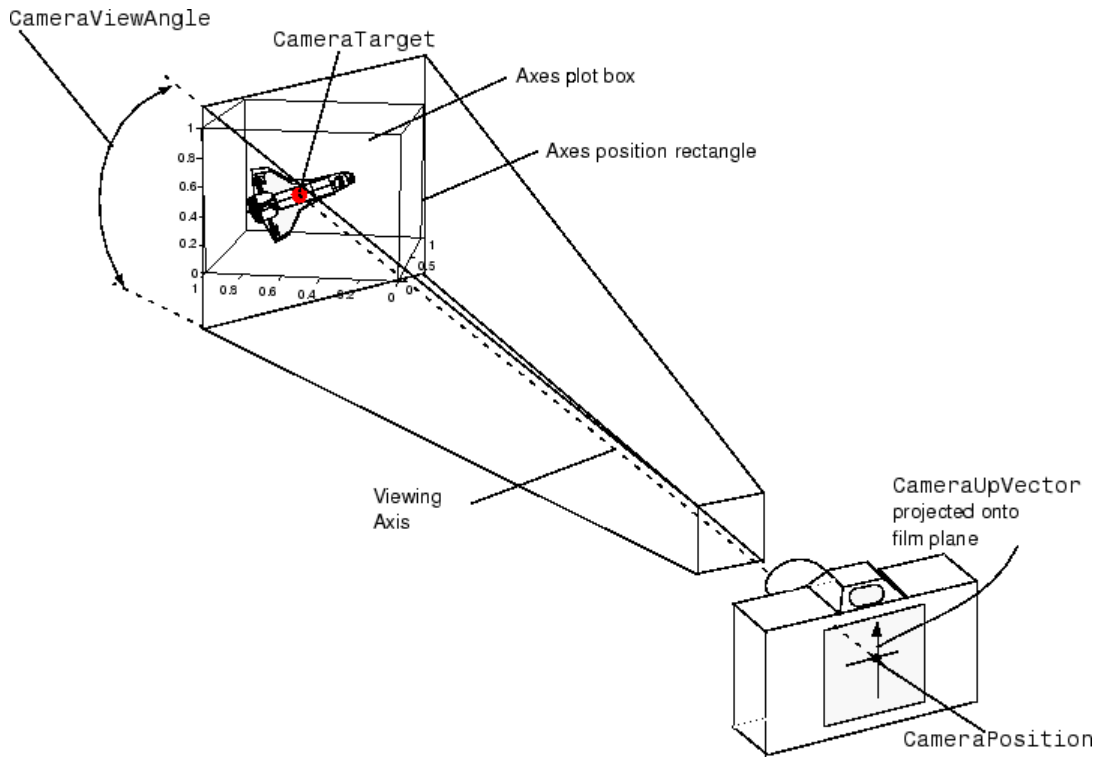
See Also

camdolly | camlookat | camorbit | campan | camproj | camroll | camtarget | camup | camva | camzoom

Related Examples

- "View Control with the Camera Toolbar" on page 14-9

View Control with the Camera Toolbar

In this section...

Camera Toolbar

The Camera toolbar enables you to perform a number of viewing operations interactively. To use the Camera toolbar,

- Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu or by typing `cameratoolbar` in the Command Window.
- Select the type of camera motion control you want to use by either clicking on the buttons or changing the `cameratoolbar` mode in the Command Window.
- Position the cursor over the figure window and click, hold down the right mouse button, then move the cursor in the desired direction.

The display updates immediately as you move the mouse.

The toolbar contains the following parts:
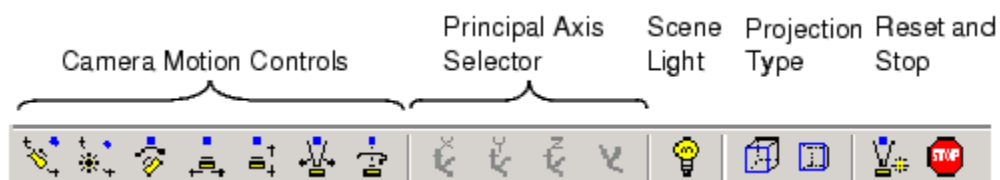
- Camera Motion Controls — These tools select which camera motion function to enable. You can also access the camera motion controls from the **Tools** menu.
- Principal Axis Selector — Some camera controls operate with respect to a particular axis. These selectors enable you to select the principal axis or to select nonaxis constrained motion. The selectors are grayed out when not applicable to the currently selected function. You can also access the principal axis selector from the **Tools** menu.
- Scene Light — The scene light button toggles a light source on or off in the scene (one light per axes).
- Projection Type — You can select orthographic or perspective projection types.
- Reset and Stop — Reset returns the scene to the view when interactions began. Stop causes the camera to stop moving (this can be useful if you apply too much cursor movement). You can also access an expanded set of reset functions from the **Tools** menu.

Principal Axes

The principal axis of a scene defines the direction that is oriented upward on the screen. For example, a MATLAB surface plot aligns the up direction along the positive z-axis.

Principal axes constrain camera-tool motion along axes that are (on the screen) parallel and perpendicular to the principal axis that you select. Specifying a principal axis is useful if your data is defined with respect to a specific axis. Z is the default principal axis, because this matches the MATLAB default 3-D view.

Two of the camera tools (Orbit and Pan/Tilt) allow you to select a principal axis as well as axis-free motion. On the screen, the axes of rotation are determined by a vertical and a horizontal line, both of which pass through the point defined by the `CameraTarget` property and are parallel and perpendicular to the principal axis.

For example, when the principal axis is z, movement occurs about

- A vertical line that passes through the camera target and is parallel to the z-axis
- A horizontal line that passes through the camera target and is perpendicular to the z-axis

This means the scene (or camera, as the case may be) moves in an arc whose center is at the camera target. The following picture illustrates the rotation axes for a z principal axis.
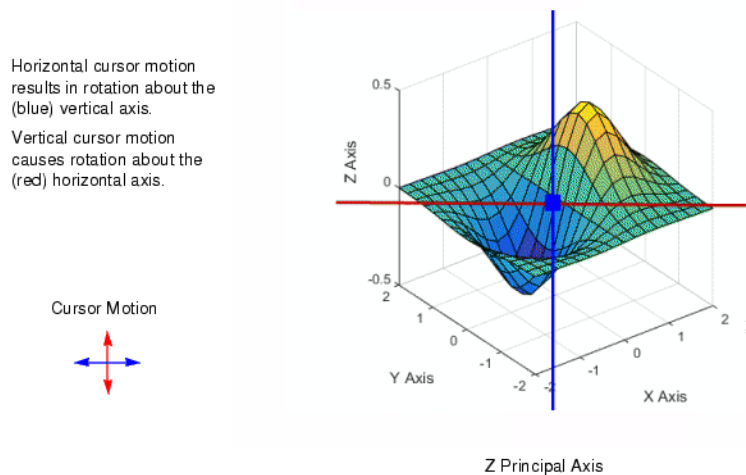
The axes of rotation always pass through the camera target.

Optimizing for 3-D Camera Motion

When you create a plot, MATLAB displays it with an aspect ratio that fits the figure window. This behavior might not create an optimum situation for the manipulation of 3-D graphics, as it can lead to distortion as you move the camera around the scene. To avoid possible distortion, it is best to switch to a 3-D visualization mode (enabled from the command line with the command `axis vis3d`). When using the Camera toolbar, MATLAB automatically switches to the 3-D visualization mode, but warns you first with the following dialog box.
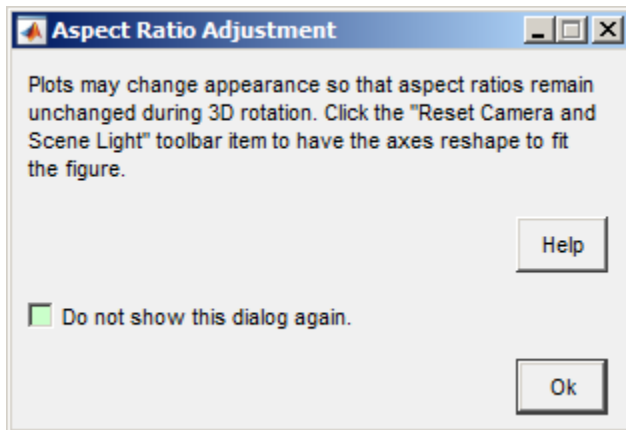
This dialog box appears only once per MATLAB session.

Camera Motion Controls

This section discusses the individual camera motion functions selectable from the toolbar.

Note When interpreting the following diagrams, keep in mind that the camera always points towards the camera target. See "Camera Graphics Terminology" on page 14-8 for an illustration of the graphics properties involved in camera motion.

Orbit Camera

Orbit Camera rotates the camera about the *z*-axis (by default). You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axis Selectors. When using no principal axis, you can rotate about an arbitrary axis.

Graphics Properties

Orbit Camera changes the `CameraPosition` property while keeping the `CameraTarget` fixed.
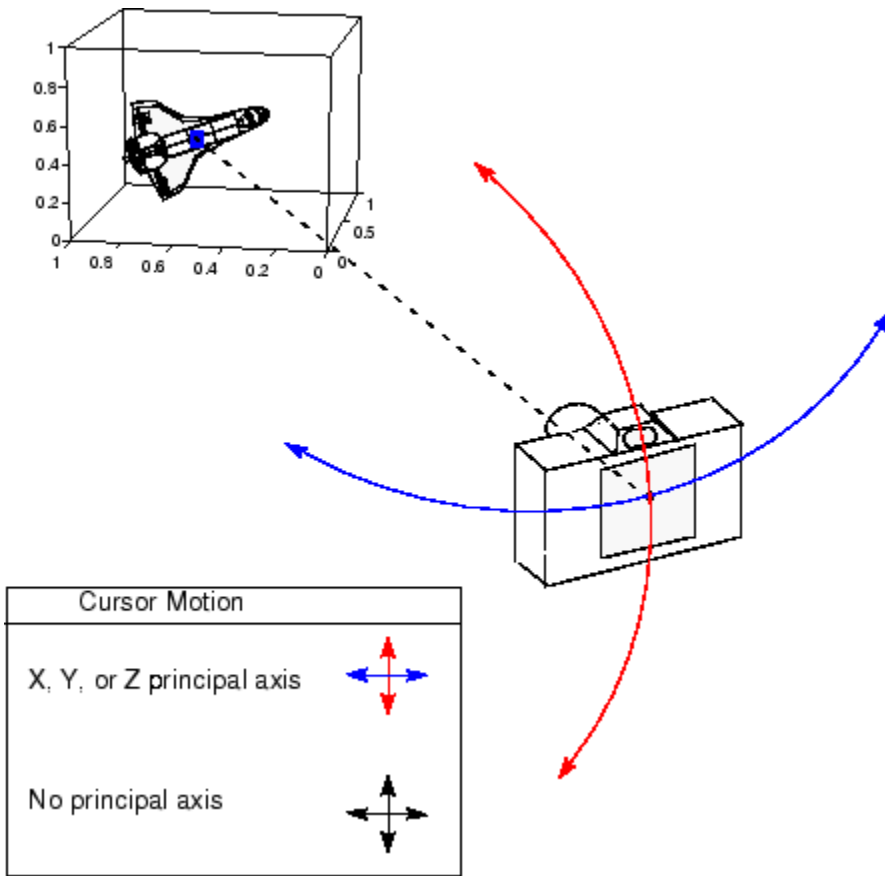
Orbit Scene Light

The scene light is a light source that is placed with respect to the camera position. By default, the scene light is positioned to the right of the camera (i.e., `camlight right`). Orbit Scene Light changes the light's offset from the camera position. There is only one scene light; however, you can add other lights using the `light` command.

Toggle the scene light on and off by clicking the yellow light bulb icon.

Graphics Properties

Orbit Scene Light moves the scene light by changing the light's `Position` property.

Pan/Tilt Camera

Pan/Tilt Camera moves the point in the scene that the camera points to while keeping the camera fixed. The movement occurs in an arc about the *z*-axis by default. You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axes Selectors.

Graphics Properties

Pan/Tilt Camera moves the point in the scene that the camera is pointing to by changing the `CameraTarget` property.
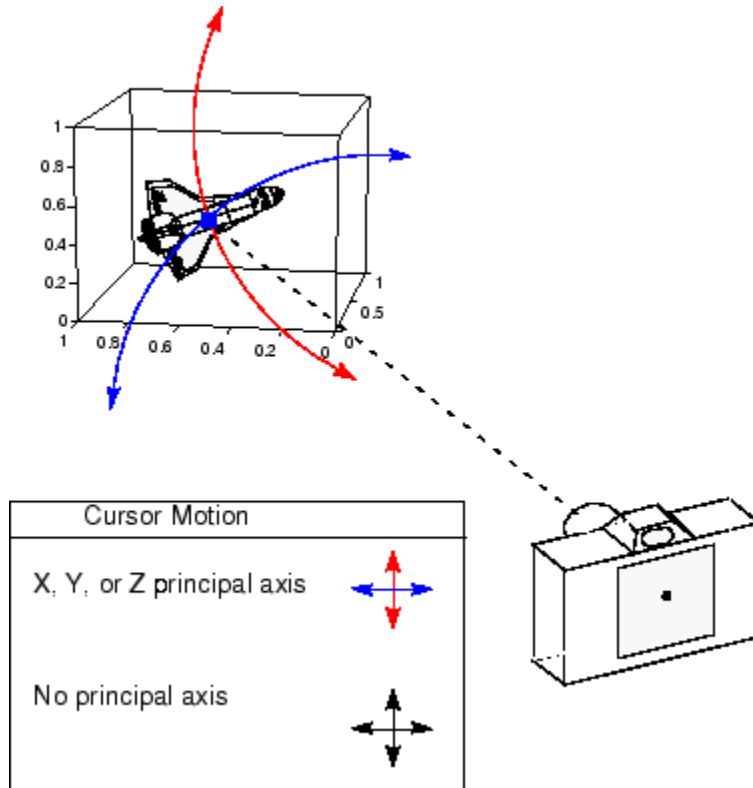
Move Camera Horizontally/Vertically

Moving the cursor horizontally or vertically (or any combination of the two) moves the scene in the same direction.

Graphics Properties

The horizontal and vertical movement is achieved by moving the `CameraPosition` and the `CameraTarget` in unison along parallel lines.
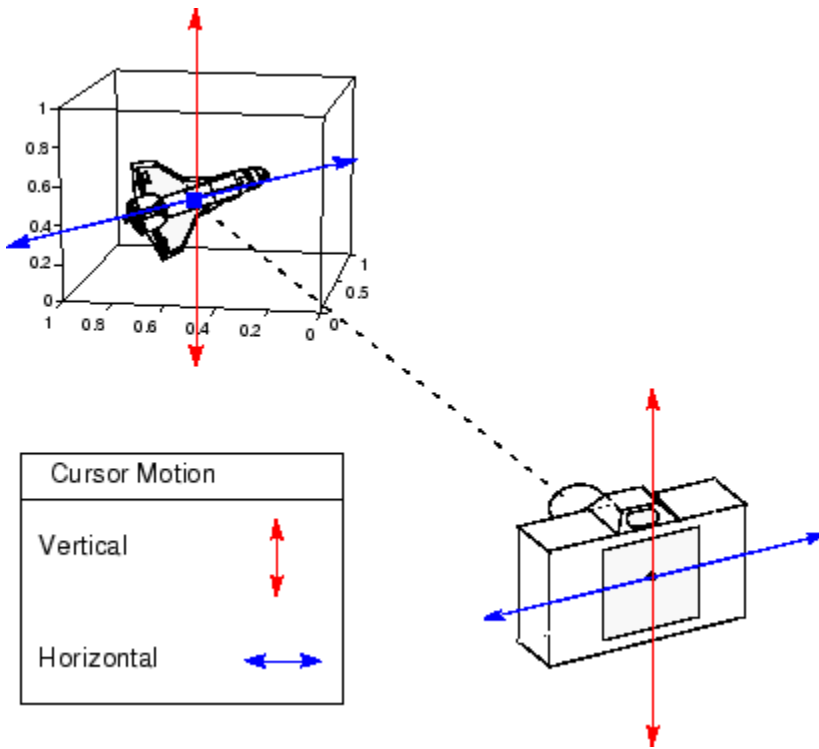
Move Camera Forward and Backward

Moving the cursor up or to the right moves the camera toward the scene. Moving the cursor down or to the left moves the camera away from the scene. It is possible to move the camera through objects in the scene and to the other side of the camera target.

Graphics Properties

This function moves the `CameraPosition` along the line connecting the camera position and the camera target.
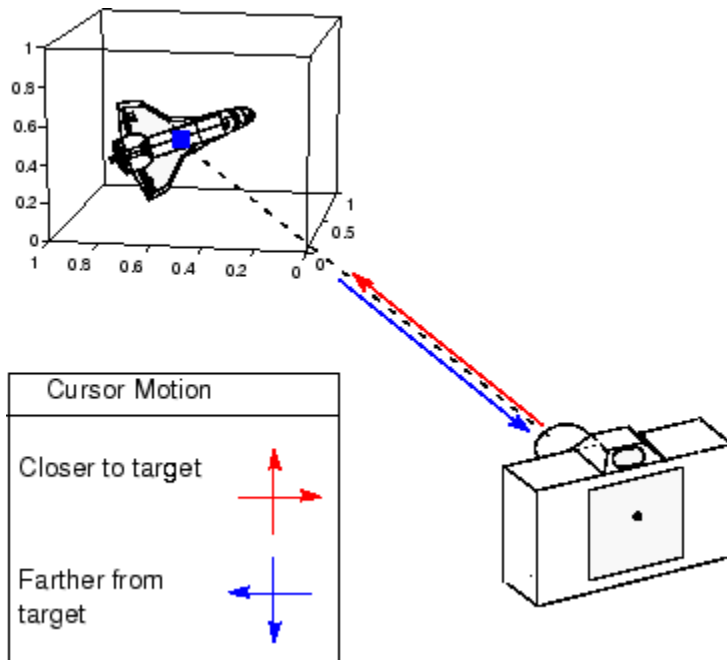
Zoom Camera

Zoom Camera makes the scene larger as you move the cursor up or to the right and smaller as you move the cursor down or to the left. Zooming does not move the camera and therefore cannot move the viewpoint through objects in the scene.

Graphics Properties

Zoom is implemented by changing the `CameraViewAngle`. The larger the angle, the smaller the scene appears, and vice versa.
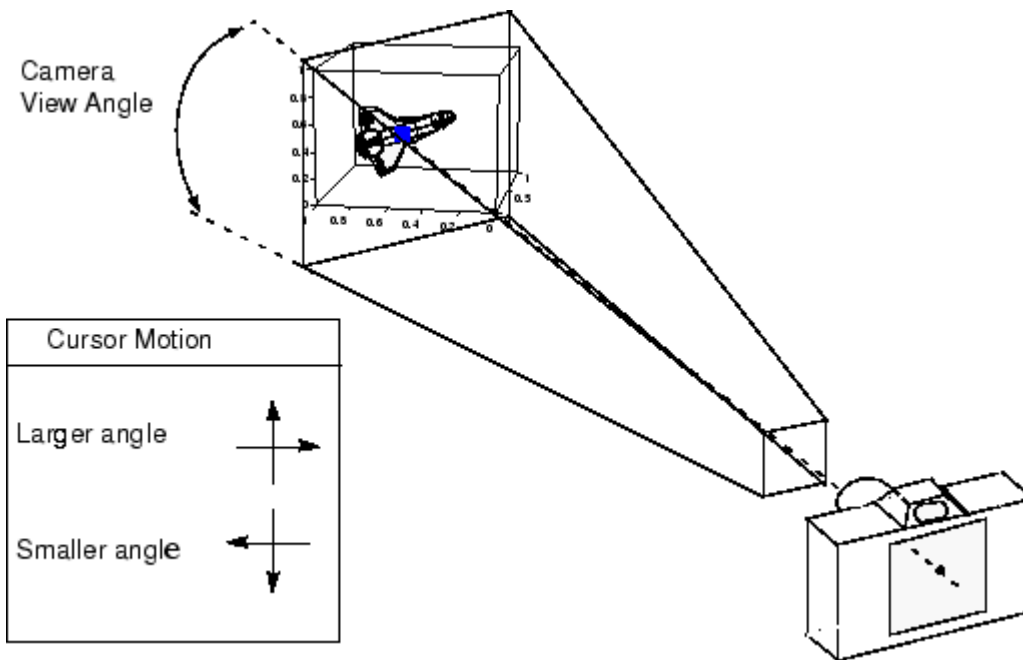
Camera Roll

Camera Roll rotates the camera about the viewing axis, thereby rotating the view on the screen.

Graphics Properties

Camera Roll changes the `CameraUpVector`.

Apparent View Rotation

Actual Camera Rotation

Cursor Motion

Counterclockwise
around camera target

Clockwise around
camera target

14-17
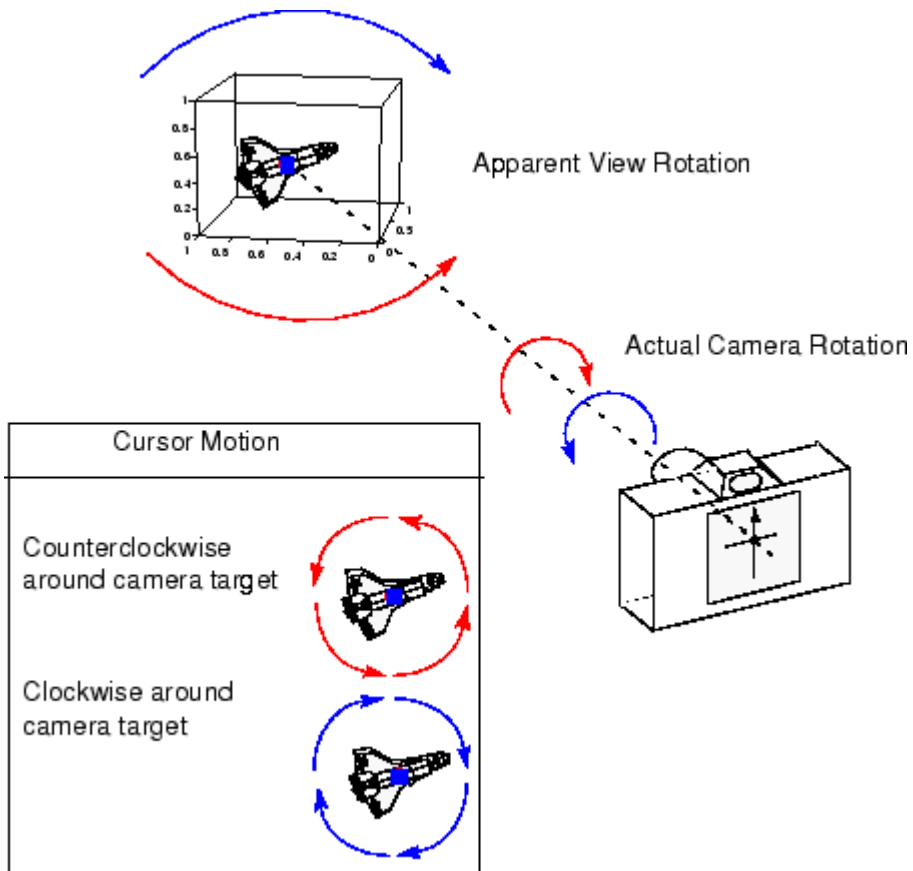
Dollying the Camera

In this section...
"Summary of Techniques" on page 14-18
"Implementation" on page 14-18

Summary of Techniques

In the camera metaphor, a dolly is a stage that enables movement of the camera from side to side with respect to the scene. The `camdolly` command implements similar behavior by moving both the position of the camera and the position of the camera target in unison (or just the camera position if you so desire).

This example illustrates how to use `camdolly` to explore different regions of an image. It shows how to use the following functions:

- `ginput` to obtain the coordinates of locations on the image
- The `camdolly data` coordinates option to move the camera and target to the new position based on coordinates obtained from `ginput`
- `camva` to zoom in and to fix the camera view angle, which is otherwise under automatic control

Implementation

First load the Cape Cod image and zoom in by setting the camera view angle (using `camva`).

```
load cape
image(X)
colormap(map)
axis image
camva(camva/2.5)
```

Then use `ginput` to select the *x*- and *y*-coordinates of the camera target and camera position.

```
while 1
  [x,y] = ginput(1);
  if ~strcmp(get(gcf,'SelectionType'),'normal')
    break
  end
  ct = camtarget;
  dx = x - ct(1);
  dy = y - ct(2);
  camdolly(dx,dy,ct(3),'movetarget','data')
  drawnow
end
```

Moving the Camera Through a Scene

In this section...
"Summary of Techniques" on page 14-19
"Graph the Volume Data" on page 14-19
"Set the View" on page 14-20
"Specify the Light Source" on page 14-20
"Select the Lighting Method" on page 14-20
"Define the Camera Path as a Stream Line" on page 14-20
"Implement the Fly-Through" on page 14-21

Summary of Techniques

A fly-through is an effect created by moving the camera through three-dimensional space, giving the impression that you are flying along with the camera as if in an aircraft. You can fly through regions of a scene that might be otherwise obscured by objects in the scene or you can fly by a scene by keeping the camera focused on a particular point.

To accomplish these effects you move the camera along a particular path, the *x*-axis for example, in a series of steps. To produce a fly-through, move both the camera position and the camera target at the same time.

The following example makes use of the fly-though effect to view the interior of an isosurface drawn within a volume defined by a vector field of wind velocities. This data represents air currents over North America.

This example employs a number of visualization techniques. It uses

- Isosurfaces and cone plots to illustrate the flow through the volume
- Lighting to illuminate the isosurface and cones in the volume
- Stream lines to define a path for the camera through the volume
- Coordinated motion of the camera position, camera target, and light

Graph the Volume Data

The first step is to draw the isosurface and plot the air flow using cone plots.

See `isosurface`, `isonormals`, `reducepatch`, and `coneplot` for information on using these commands.

Setting the data aspect ratio (`daspect`) to `[1,1,1]` before drawing the cone plot enables MATLAB software to calculate the size of the cones correctly for the final view.

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
figure
p = patch(isosurface(x,y,z,wind_speed,35));
isonormals(x,y,z,wind_speed,p)
p.FaceColor = [0.75,0.25,0.25];
```

```
p.EdgeColor = [0.6,0.4,0.4];

[f,vt] = reducepatch(isosurface(x,y,z,wind_speed,45),0.05);
daspect([1,1,1]);
hcone = coneplot(x,y,z,u,v,w,vt(:,1),vt(:,2),vt(:,3),2);
hcone.FaceColor = 'blue';
hcone.EdgeColor = 'none';
```

Set the View

You need to define viewing parameters to ensure the scene is displayed correctly:

- Selecting a perspective projection provides the perception of depth as the camera passes through the interior of the isosurface (`camproj`).

- Setting the camera view angle to a fixed value prevents MATLAB from automatically adjusting the angle to encompass the entire scene as well as zooming in the desired amount (`camva`).

  ```
  camproj perspective
  camva(25)
  ```

Specify the Light Source

Positioning the light source at the camera location and modifying the reflectance characteristics of the isosurface and cones enhances the realism of the scene:

- Creating a light source at the camera position provides a "headlight" that moves along with the camera through the isosurface interior (`camlight`).

- Setting the reflection properties of the isosurface gives the appearance of a dark interior (`AmbientStrength` set to 0.1) with highly reflective material (`SpecularStrength` and `DiffuseStrength` set to 1).

- Setting the `SpecularStrength` of the cones to 1 makes them highly reflective.

  ```
  hlight = camlight('headlight');
  p.AmbientStrength = 1;
  p.SpecularStrength = 1;
  p.DiffuseStrength = 1;
  hcone.SpecularStrength = 1;
  set(gcf,'Color','k')
  set(gca,'Color',[0,0,0.25])
  ```

Select the Lighting Method

Use `gouraud` lighting for smoother lighting:

```
lighting gouraud
```

Define the Camera Path as a Stream Line

Stream lines indicate the direction of flow in the vector field. This example uses the *x*-, *y*-, and *z*-coordinate data of a single stream line to map a path through the volume. The camera is then moved along this path. The tasks include

- Create a stream line starting at the point `x = 80`, `y = 30`, `z = 11`.
- Get the *x-*, *y-*, and *z*-coordinate data of the stream line.
- Delete the stream line (you could also use `stream3` to calculate the stream line data without actually drawing the stream line).

```
hsline = streamline(x,y,z,u,v,w,80,30,11);
xd = hsline.XData;
yd = hsline.YData;
zd = hsline.ZData;
delete(hsline)
```

Implement the Fly-Through

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

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

```
for i=1:length(xd)-5
    campos([xd(i),yd(i),zd(i)])
    camtarget([xd(i+5)+min(xd)/500,yd(i),zd(i)])
    camlight(hlight,'headlight')
    drawnow
end
```

See `coneplot` for a fixed visualization of the same data.

14-21

Low-Level Camera Properties

In this section...

Camera Properties You Can Set

Camera graphics is based on a group of axes properties that control the position and orientation of the camera. In general, the camera commands, such as `campos`, `camtarget`, and `camup`, make it unnecessary to access these properties directly.

Property	Description
`CameraPosition`	Specifies the location of the viewpoint in axes units.
`CameraPositionMode`	In `automatic` mode, the scene determines the position. In `manual` mode, you specify the viewpoint location.
`CameraTarget`	Specifies the location in the axes pointed to by the camera. Together with the `CameraPosition`, it defines the viewing axis.
`CameraTargetMode`	In `automatic` mode, MATLAB specifies the `CameraTarget` as the center of the axes plot box. In `manual` mode, you specify the location.
`CameraUpVector`	The rotation of the camera around the viewing axis is defined by a vector indicating the direction taken as up.
`CameraUpVectorMode`	In `automatic` mode, MATLAB orients the up vector along the positive *y*-axis for 2-D views and along the positive *z*-axis for 3-D views. In `manual` mode, you specify the direction.
`CameraViewAngle`	Specifies the field of view of the "lens." If you specify a value for CameraViewAngle, MATLAB does not stretch-the axes to fit the figure.
`CameraViewAngleMode`	In `automatic` mode, MATLAB adjusts the view angle to the smallest angle that captures the entire scene. In `manual` mode, you specify the angle. Setting `CameraViewAngleMode` to `manual` overrides stretch-to-fill behavior.
`Projection`	Selects either an orthographic or perspective projection.

Default Viewpoint Selection

When all the camera mode properties are set to `auto` (the default), MATLAB automatically controls the view, selecting appropriate values based on the assumption that you want the scene to fill the

position rectangle (which is defined by the width and height components of the axes `Position` property).

By default, MATLAB

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

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

Moving In and Out on the Scene

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

Moving Through a Scene

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

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

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

Example — Moving Toward or Away from the Target

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

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

```
function movecamera(dist) %dist in the range [-1 1]
set(gca,'CameraViewAngleMode','manual')
newcp = cpos - dist * (cpos - ctarg);
set(gca,'CameraPosition',newcp)
function out = cpos
out = get(gca,'CameraPosition');
function out = ctarg
out = get(gca,'CameraTarget');
```
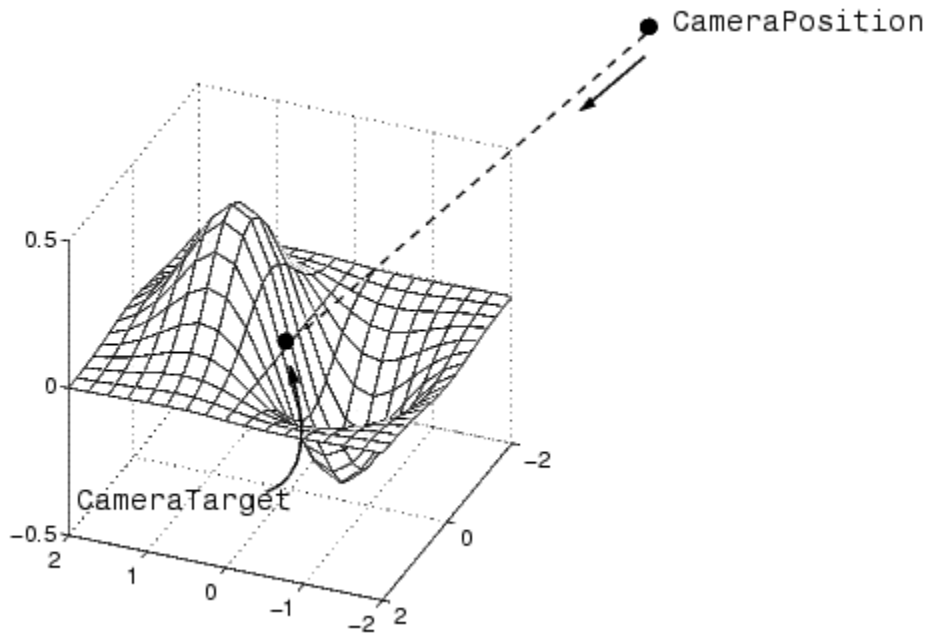
Setting the `CameraViewAngleMode` to `manual` can cause an abrupt change in the aspect ratio.

Making the Scene Larger or Smaller

Adjusting the `CameraViewAngle` property makes the view of the scene larger or smaller. Larger angles cause the view to encompass a larger area, thereby making the objects in the scene appear smaller. Similarly, smaller angles make the objects appear larger.

Changing `CameraViewAngle` makes the scene larger or smaller without affecting the position of the camera. This is desirable if you want to zoom in without moving the viewpoint past objects that will then no longer be in the scene (as could happen if you changed the camera position). Also, changing the `CameraViewAngle` does not affect the amount of perspective applied to the scene, as changing `CameraPosition` does when the figure `Projection` property is set to `perspective`.

Revolving Around the Scene

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

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

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

```
    drawnow
end
```

Rotation Without Resizing

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

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

Rotation About the Viewing Axis

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

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

In many cases, you might find it convenient to visualize the desired up vector in terms of angles with respect to the axes *x*-, *y*-, and *z*-axis. You can then use *direction cosines* to convert from angles to vector components. For a unit vector, the expression simplifies to
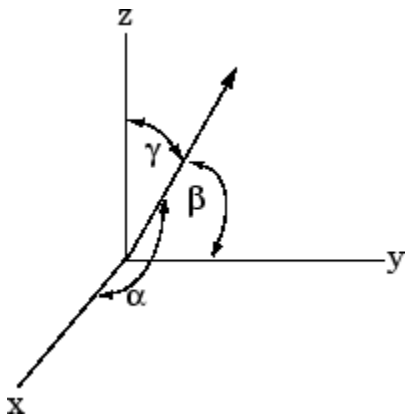
where the angles α, β, and γ are specified in degrees.

XComponent = cos(α*(pi/180));

YComponent = cos(β*(pi/180));

ZComponent = cos(γ*(pi/180));

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

Calculating a Camera Up Vector

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

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

and then set the `CameraUpVector` property.

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

Drawing a sphere with this orientation produces
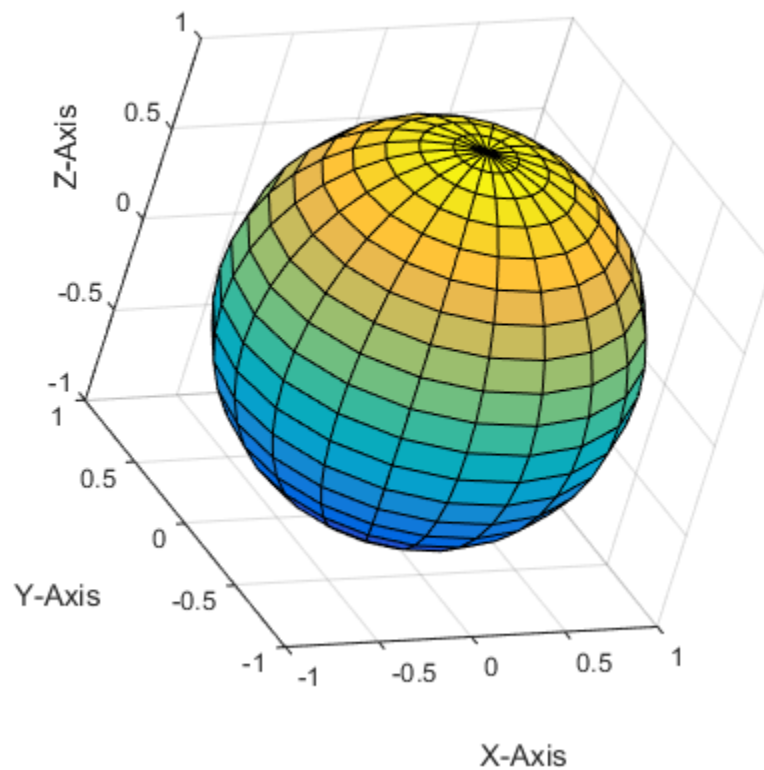
Understanding View Projections

In this section...
"Two Types of Projections" on page 14-27
"Projection Types and Camera Location" on page 14-28

Two Types of Projections

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

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

By default, MATLAB displays objects using orthographic projection. You can set the projection type using the `camproj` command.

These pictures show a drawing of a dump truck (created with `patch`) and a surface plot of a mathematical function, both using orthographic projection.
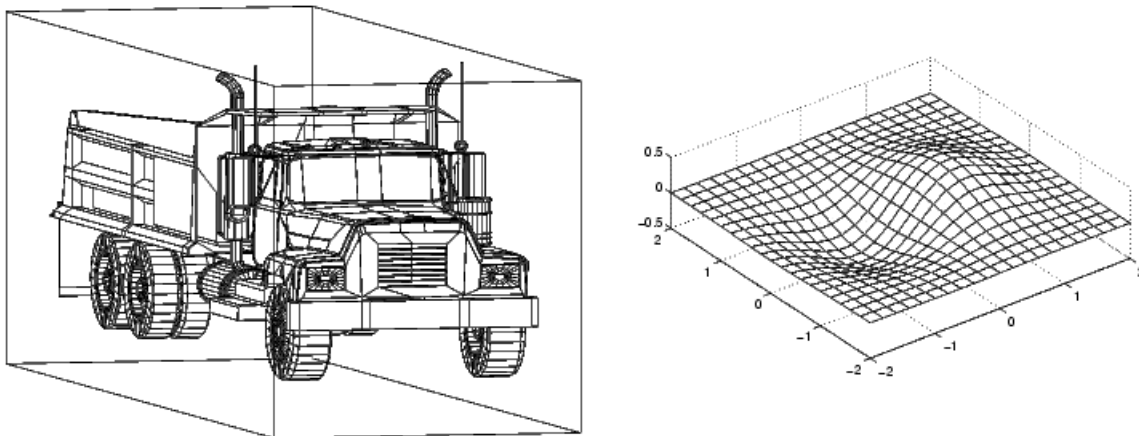
If you measure the width of the front and rear faces of the box enclosing the dump truck, you'll see they are the same size. This picture looks unnatural because it lacks the apparent perspective you see when looking at real objects with depth. On the other hand, the surface plot accurately indicates the values of the function within rectangular space.

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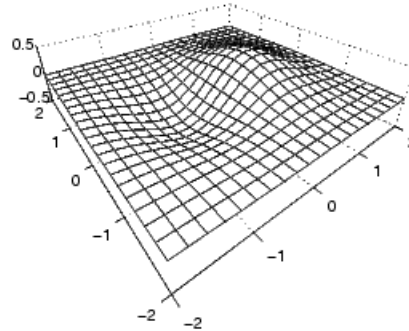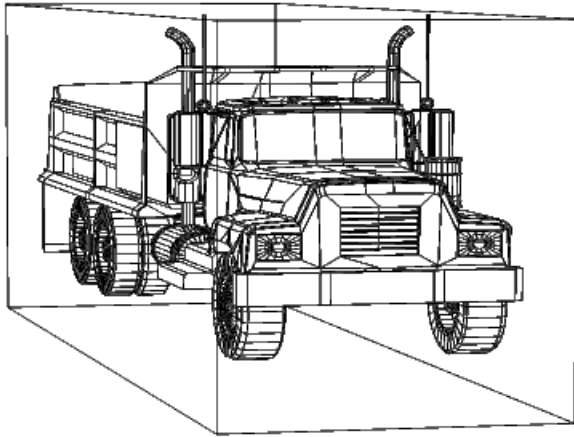
Projection Types and Camera Location

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

	Orthographic	**Perspective**
OpenGL®	`CameraViewAngle` determines extent of scene at `CameraTarget`.	`CameraViewAngle` determines extent of scene from `CameraPosition` to infinity.
Painters	All objects are displayed regardless of `CameraPosition`.	Not recommended if graphics objects are behind the `CameraPosition`.

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

Orthographic projection and Z-buffer renderer
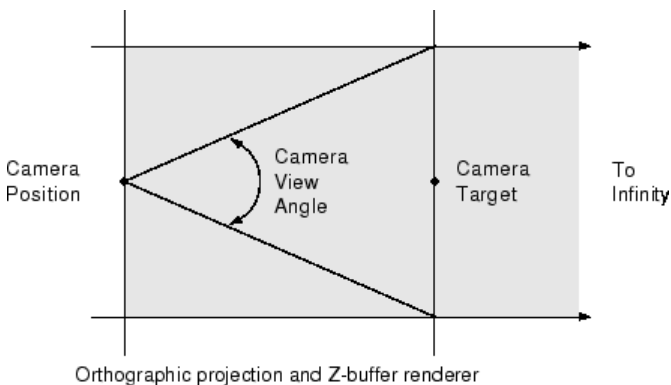
In perspective projection, you see only what is visible in the cone of the camera view angle.
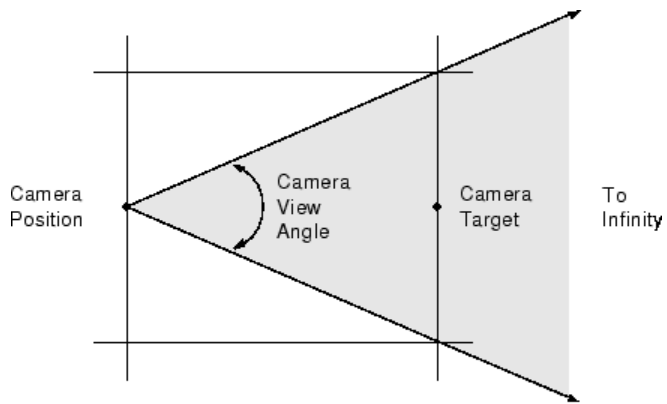
Perspective projection and Z-buffer renderer

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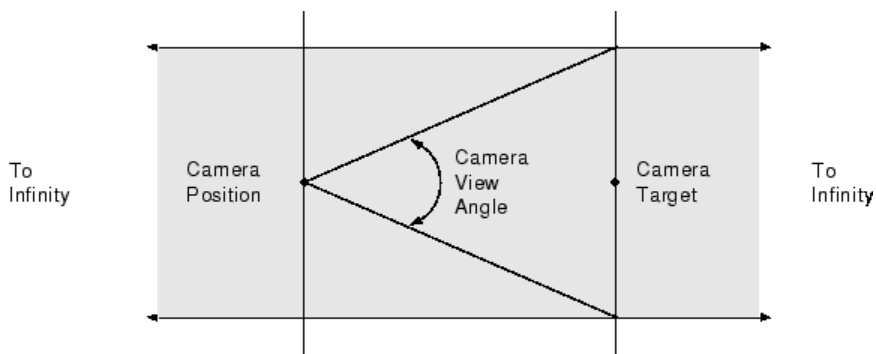
Orthographic projection and painters renderer

Printing 3-D Scenes

The same effects described in the previous section occur in hardcopy output. You should specify opengl printing explicitly to obtain the results displayed on the screen (use the `-opengl` option with the `print` command).

15

Displaying Bit-Mapped Images

Working with Images in MATLAB Graphics

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 15-8 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`.

Image Types

In this section...
"Indexed Images" on page 15-4
"Grayscale (Intensity) Images" on page 15-5
"RGB (Truecolor) Images" on page 15-6

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.
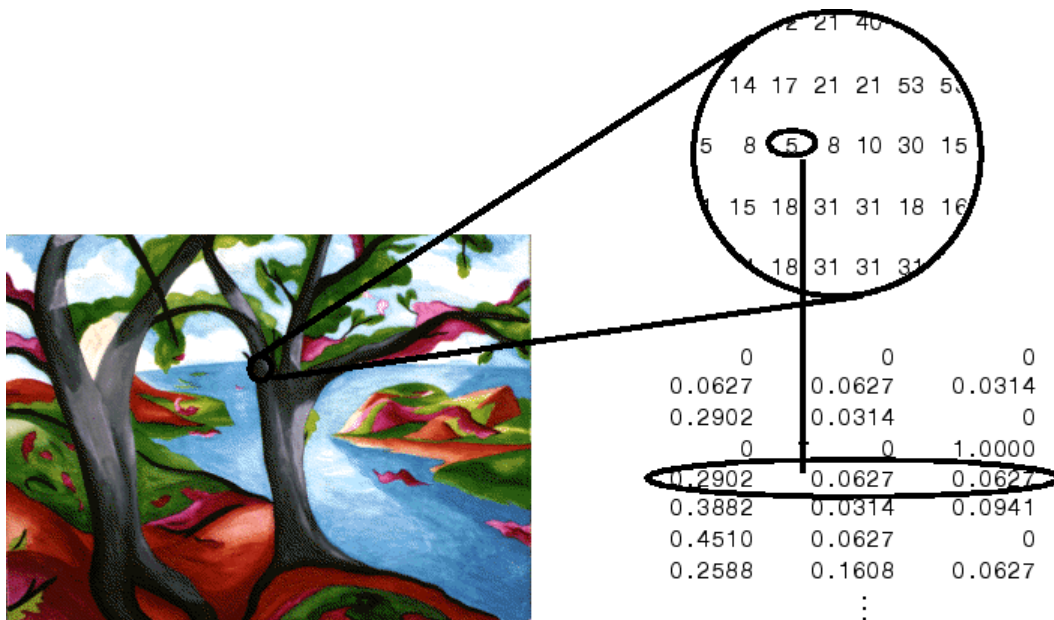
The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class double, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class uint8 or uint16, 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.

Grayscale (Intensity) Images

A grayscale image, sometimes referred to as an intensity image, is a data matrix `I` whose values represent intensities within some range. A grayscale 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 grayscale images are rarely saved with a colormap, a colormap is still used to display them. In essence, grayscale images are treated as indexed images.

This figure depicts a grayscale image of class `double`.
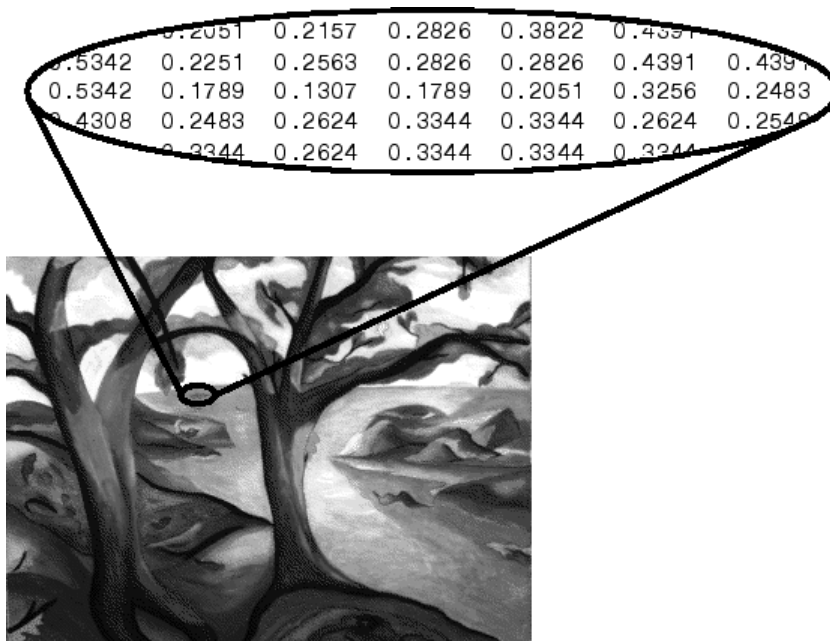
To display a grayscale image, use the `imagesc` ("image scale") function, which enables you to set the range of intensity values. `imagesc` scales the image data to use the full colormap. Use the two-input form of `imagesc` to display a grayscale image, for example:

```
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 grayscale images using a grayscale colormap, it is possible to use other colormaps. For example, the following statements display the grayscale 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 a grayscale image, use the single-argument form of `imagesc`. With one input argument, `imagesc` maps the minimum value of the data matrix to

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

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

RGB (Truecolor) Images

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

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

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

```
image(RGB)
```

The next figure shows an RGB image of class `double`.
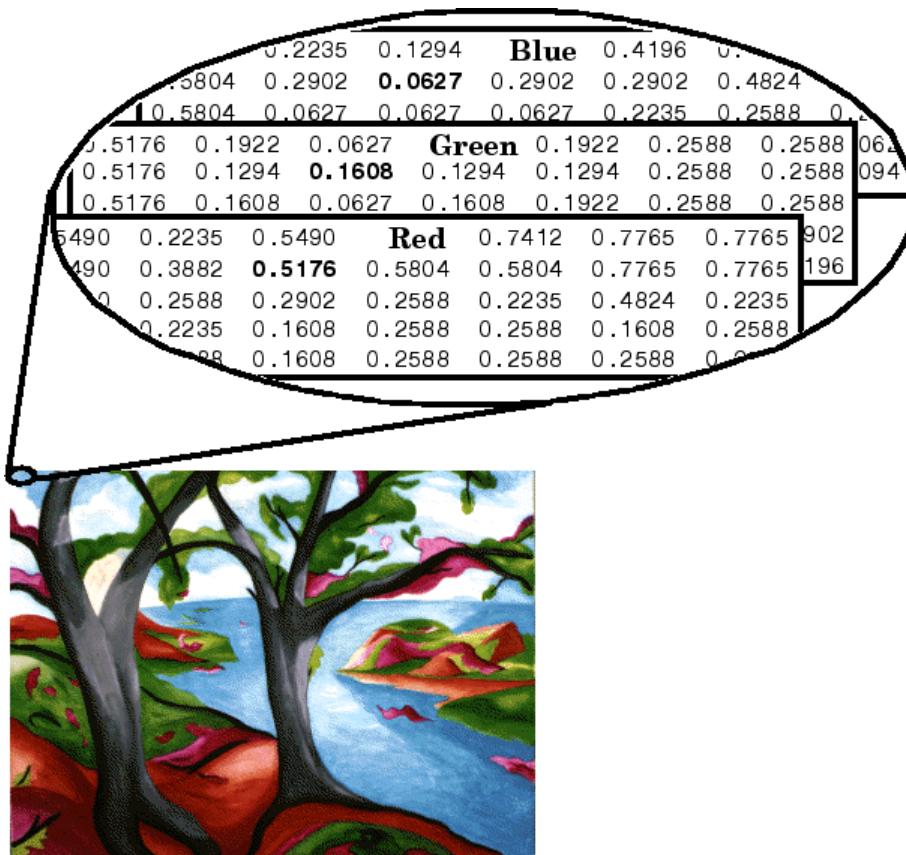
To determine the color of the pixel at (2,3), look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

8-Bit and 16-Bit Images

In this section...
"Indexed Images" on page 15-8
"Intensity Images" on page 15-9
"RGB Images" on page 15-9
"Mathematical Operations Support for uint8 and uint16" on page 15-9
"Other 8-Bit and 16-Bit Array Support" on page 15-10
"Converting an 8-Bit RGB Image to Grayscale" on page 15-10
"Summary of Image Types and Numeric Classes" on page 15-12

Indexed Images

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

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

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

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

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

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

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

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

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

Intensity Images

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

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

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

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

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

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

RGB Images

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

```
image(RGB);
```

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

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

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

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

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

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

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

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

Mathematical Operations Support for uint8 and uint16

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

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

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

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

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

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

MATLAB Integer Mathematics

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

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

Other 8-Bit and 16-Bit Array Support

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

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

Converting an 8-Bit RGB Image to Grayscale

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

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

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

axis image;
```

NGC 6543

HST · WFPC2
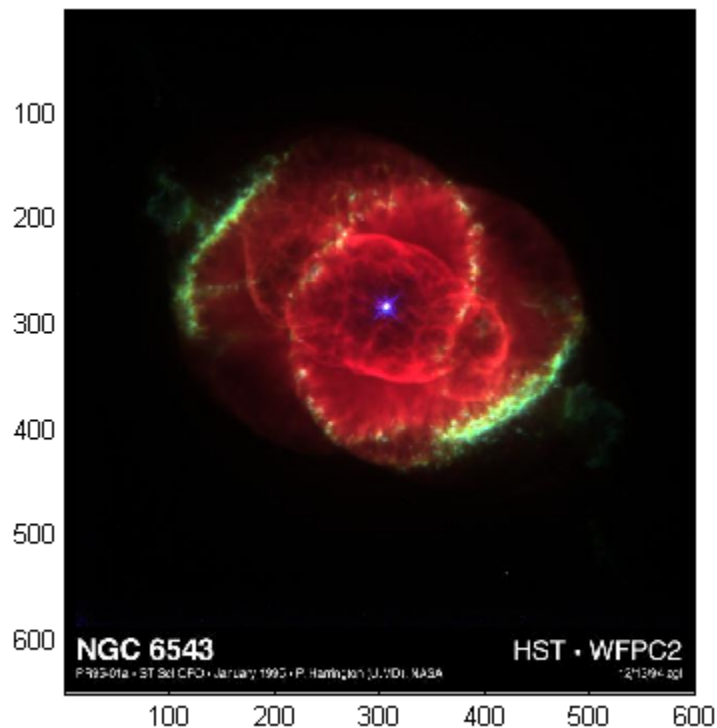
Note This image was created with the support of the Space Telescope Science Institute, operated by the Association of Universities for Research in Astronomy, Inc., from NASA contract NAs5-26555, and is reproduced with permission from AURA/STScI. Digital renditions of images produced by AURA/ STScI are 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;
```
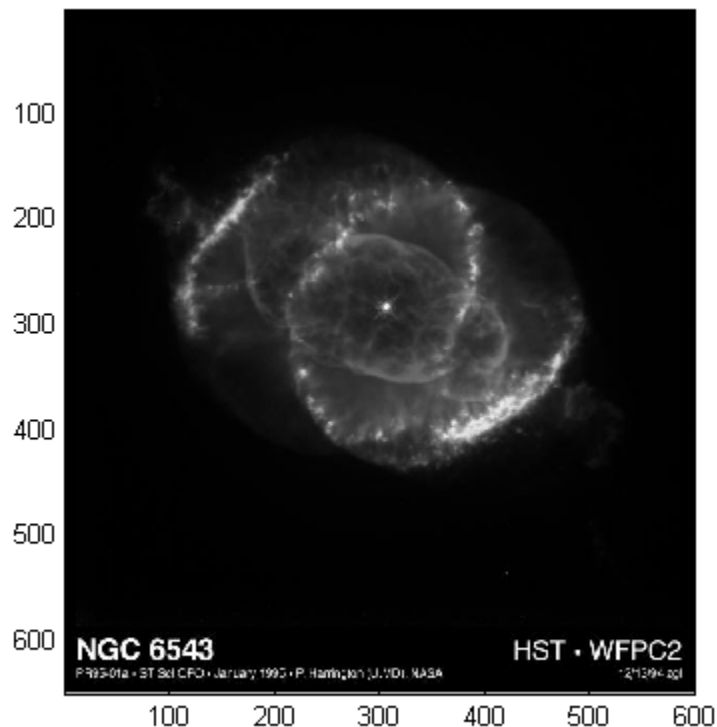

Related Information

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].

15-13

Read, Write, and Query Image Files

In this section...
"Working with Image Formats" on page 15-14
"Reading a Graphics Image" on page 15-14
"Writing a Graphics Image" on page 15-15
"Subsetting a Graphics Image (Cropping)" on page 15-15
"Obtaining Information About Graphics Files" on page 15-16

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`
	
	`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 15-17
"Controlling Aspect Ratio and Display Size" on page 15-18

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;
```
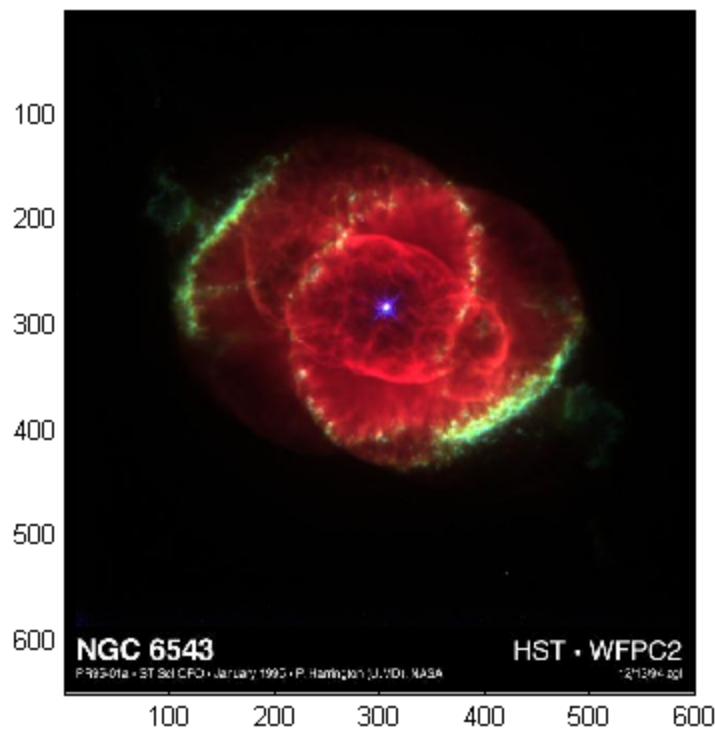

This table summarizes display methods for the three types of images.

Image Type	Display Commands	Uses Colormap Colors
Indexed	`image(X); colormap(map)`	Yes
Intensity	`imagesc(I,[0 1]); colormap(gray)`	Yes

Image Type	Display Commands	Uses Colormap Colors
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)
```
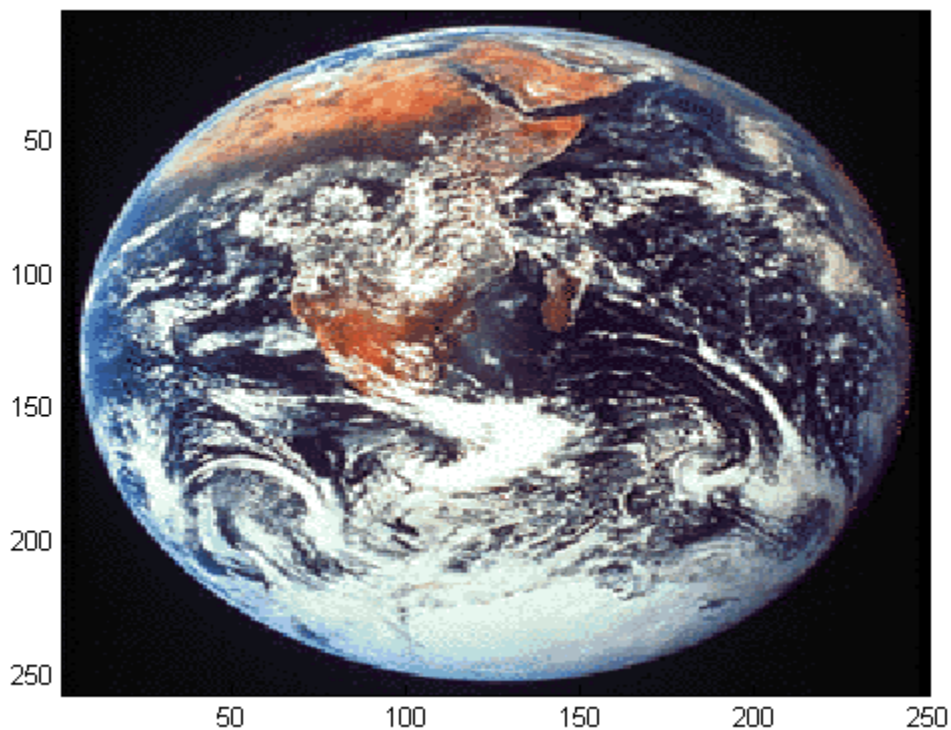

The elongated globe results from stretching the image display to fit the axes position. Use the `axis image` command to force the aspect ratio to be one-to-one.

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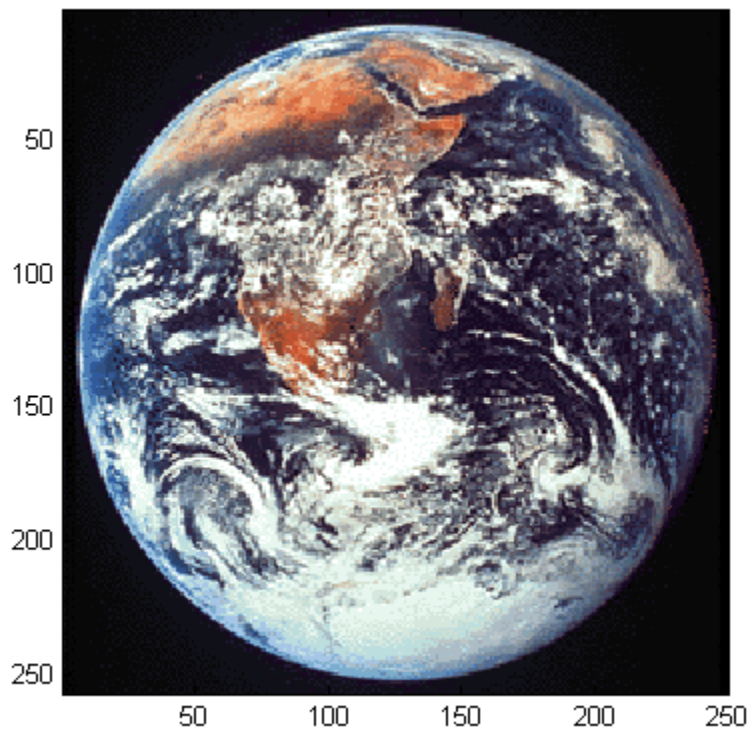
The `axis image` command works by setting the `DataAspectRatio` property of the axes object to [1 1 1]. See `axis` and `axes` for more information on how to control the appearance of axes objects.

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)
```

15-19

The Image Object and Its Properties

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)
```
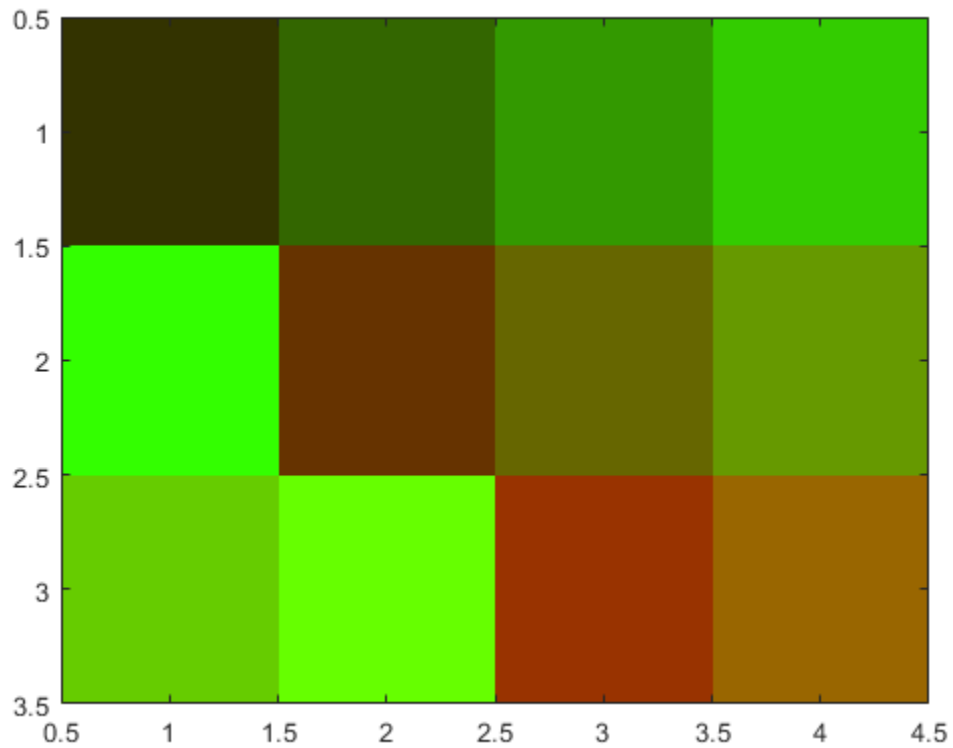
Specify Coordinate System

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)
```
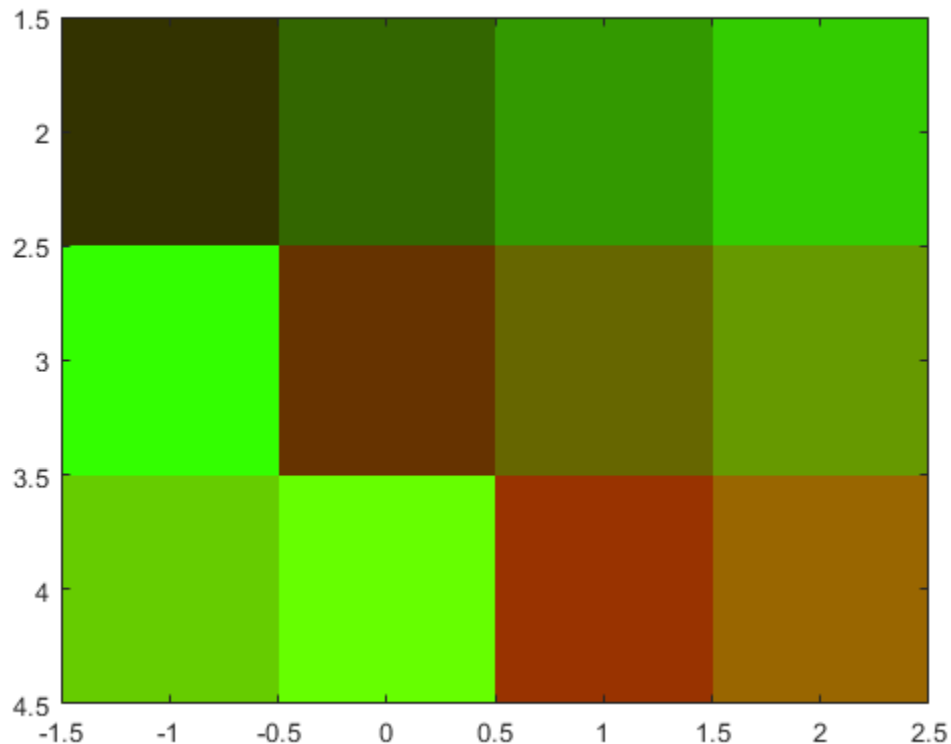
Add Text to Image Data

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
```
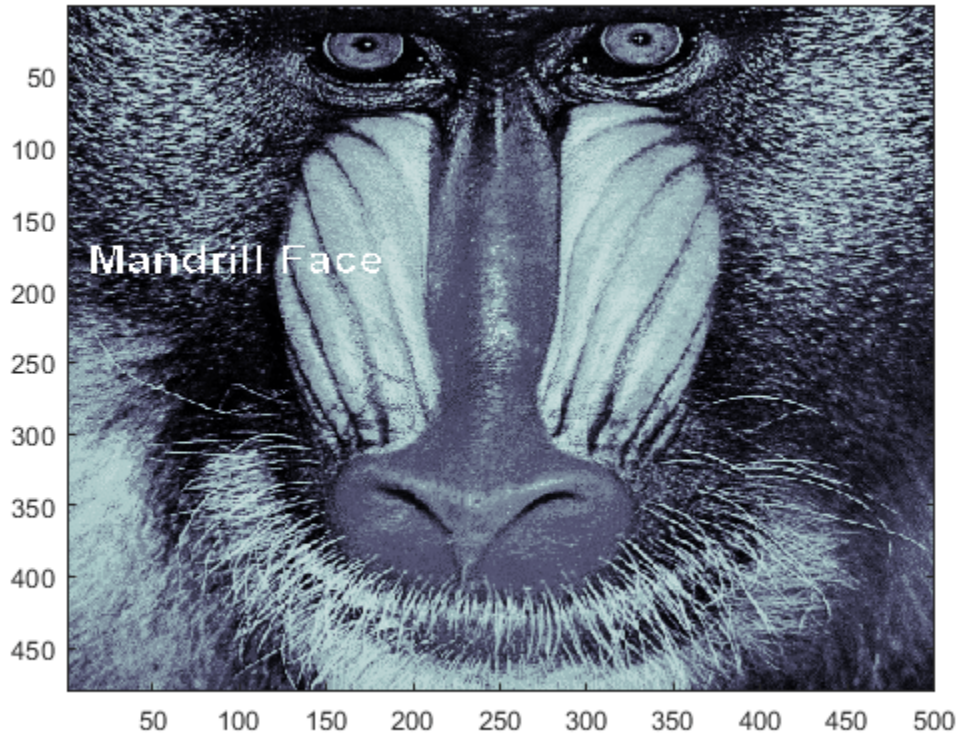

Additional Techniques for Fast Image Updating

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 15-4 and "Indexed Images" on page 15-8 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.

Displaying Image Data

This example shows how to read an RGB image into the workspace and display it. The example then converts the RGB image into a grayscale image and displays it. Finally, the example shows how to combine several individual images into one tiled image (or montage).

Read the Image

The sample file named `peppers.png` contains an RGB image. Read the image into the workspace using the `imread` function.

```
RGB = imread('peppers.png');
```

Display the Color Image

Display the image data using the `imshow` function.

```
imshow(RGB)
```


Convert to Grayscale

Convert the RGB image to grayscale using the `rgb2gray` function.

```
gray = rgb2gray(RGB);
```

15-29

Display the Grayscale Image

Display the grayscale image using the `imshow` function.

```
imshow(gray)
```


Create a Tiled Image from Multiple Images

Combine several individual images into a single tiled image and display the tiled image using the `imshow` function.

```
out = imtile({'peppers.png', 'ngc6543a.jpg'});
imshow(out);
```
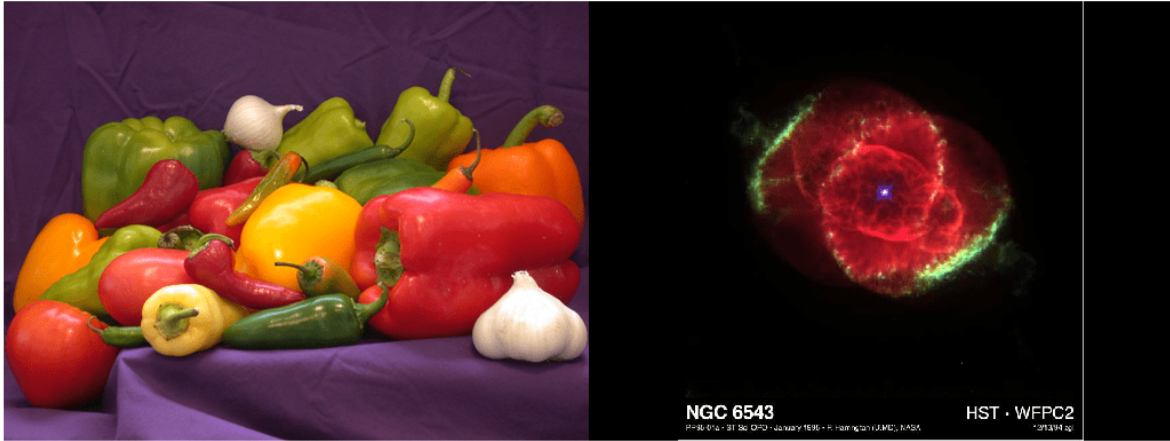
Printing and Saving

Print Figure from File Menu

In this section...
"Simple Printout" on page 16-2
"Preserve Background Color and Tick Values" on page 16-2
"Figure Size and Placement" on page 16-2
"Line Width and Font Size" on page 16-3

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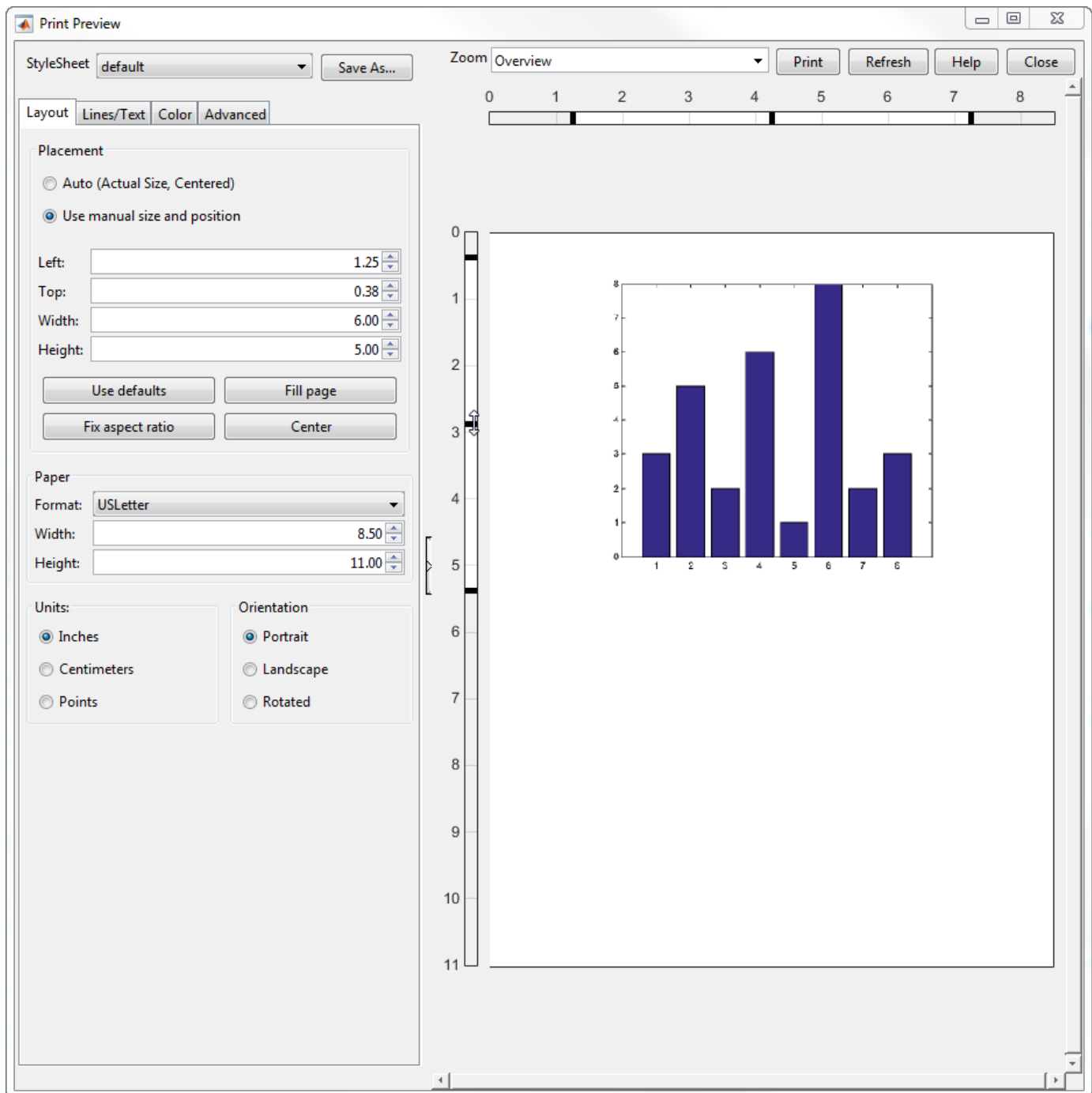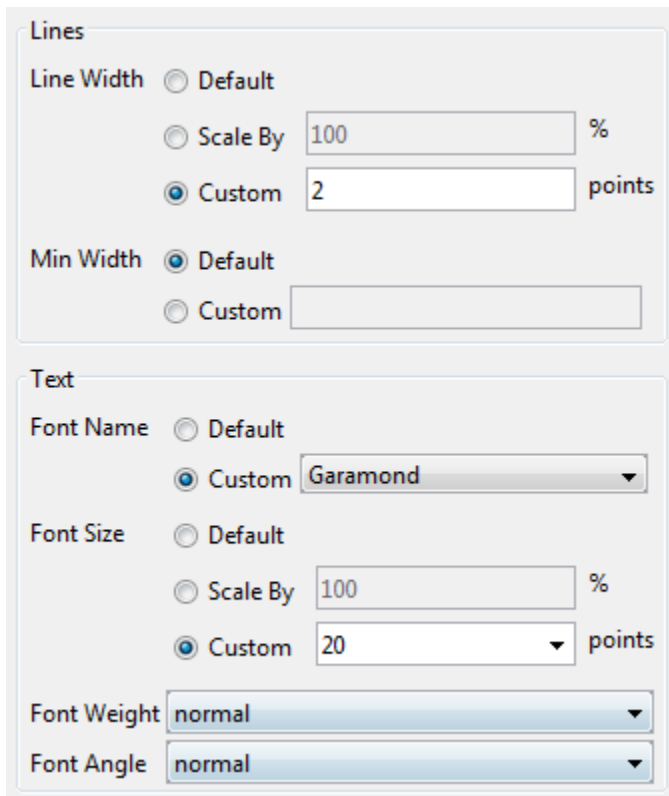
To specify the printed figure size and placement programmatically, use the `PaperPosition` property for the figure.

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.

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 16-5

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.

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')
```
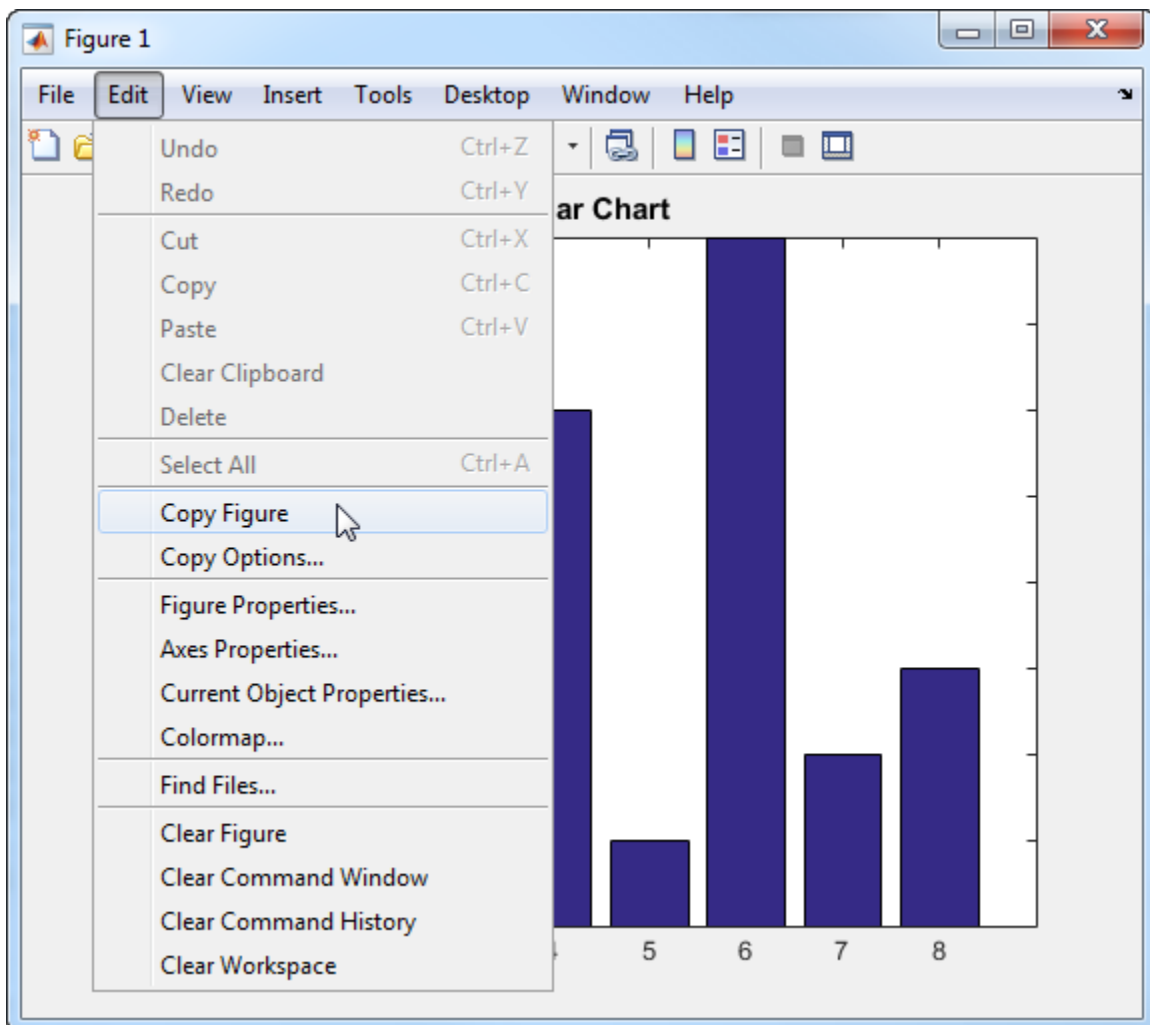

Paste the copied figure into other applications, typically by right-clicking. By default, MATLAB converts the background color of the copied figure to white.

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.
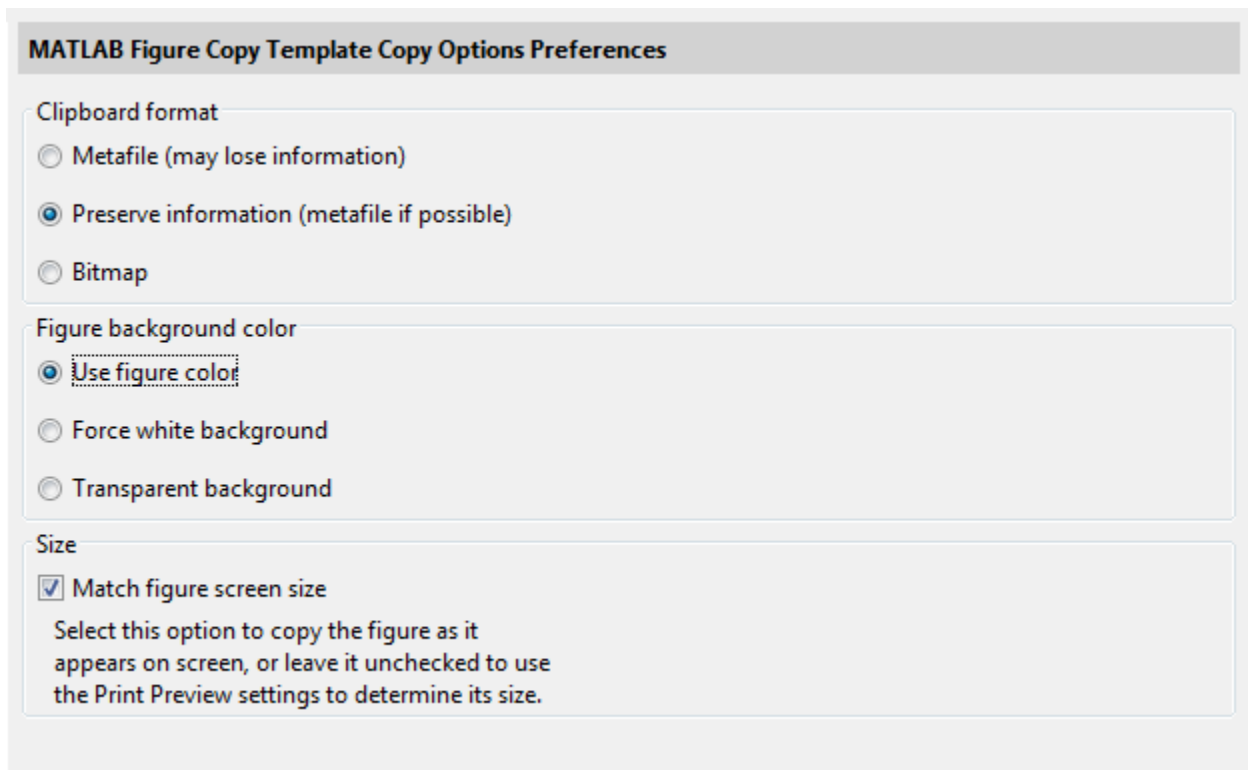
Set the clipboard format to one of these options:

- 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 Plot as Image or Vector Graphics File" on page 16-14

Customize Figure 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.

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 a `PositionConstraint` property set to `'outerposition'`.
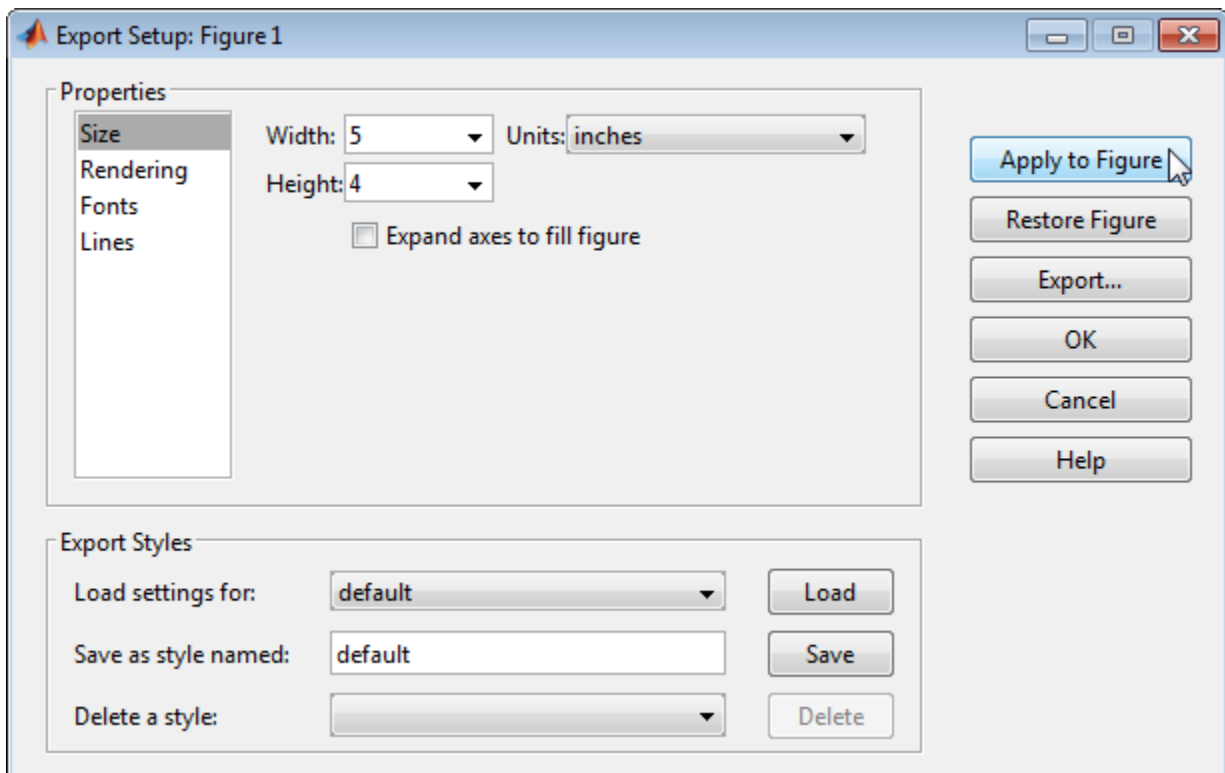
Click **Apply to Figure**. Applying the settings changes the appearance of the figure on the screen. All settings from the Export Setup dialog are applied to the figure. Thus, more than just the figure size can change. For example, by default, MATLAB converts the background color of the saved figure to white.
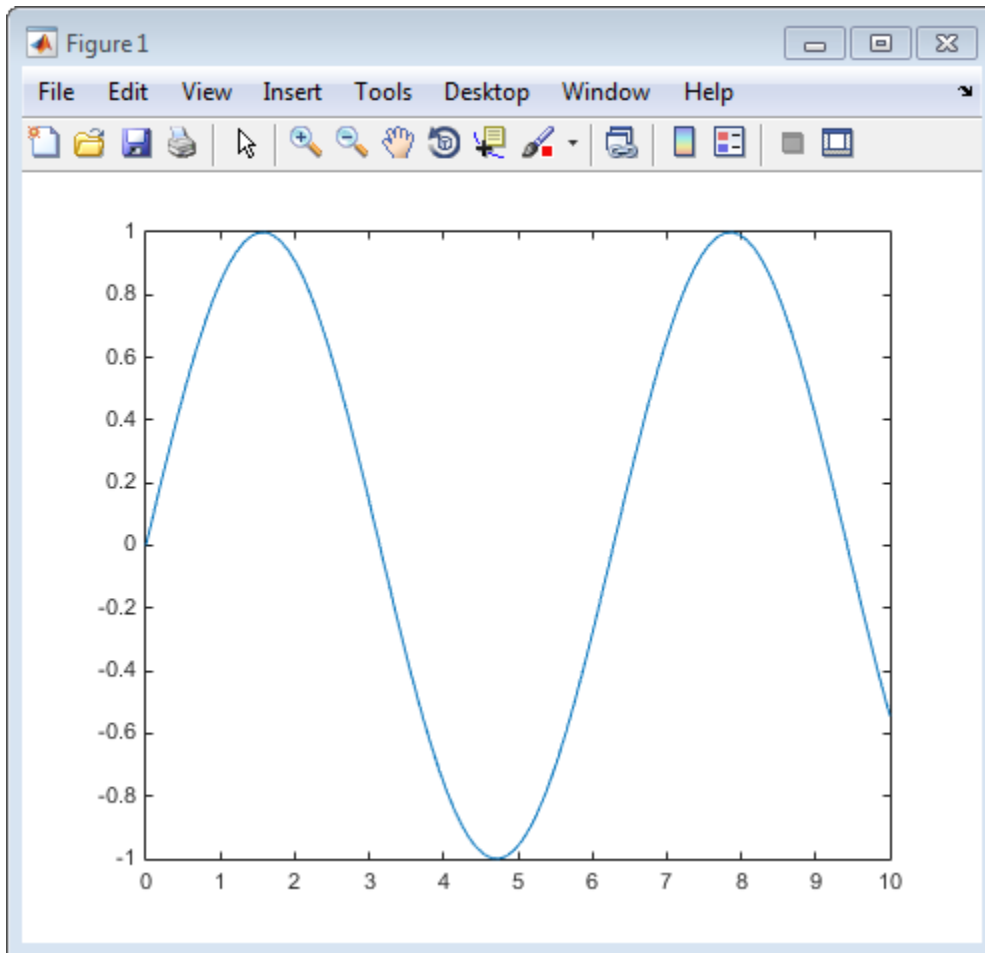
Set Figure Background Color

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.
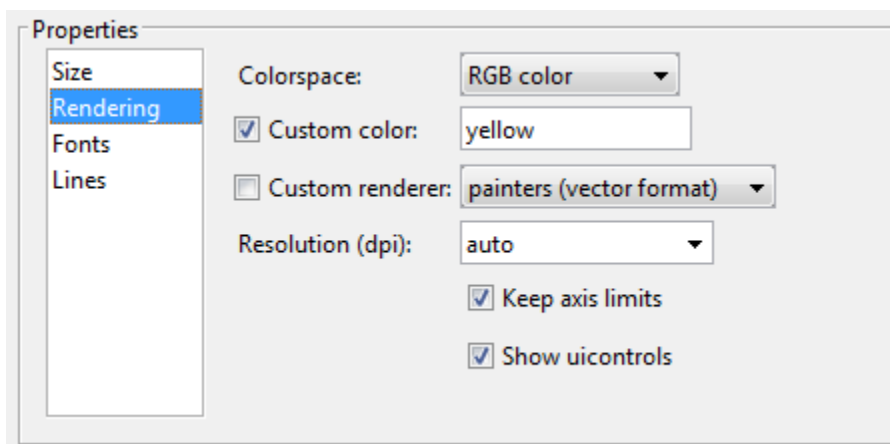
An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`, for example, `[0.4 0.6 0.7]`. This table lists some common RGB triplets that have corresponding color names. To specify 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.
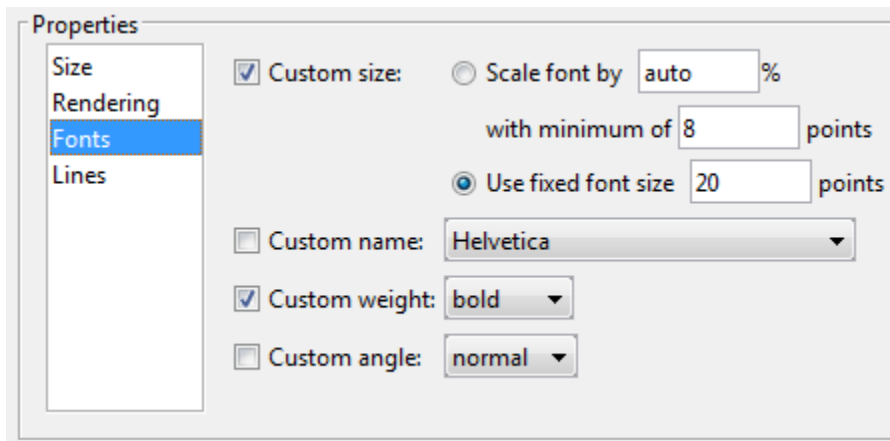
Change the line width by clicking the **Lines** property. Specify a fixed line width, for example, 2 points.
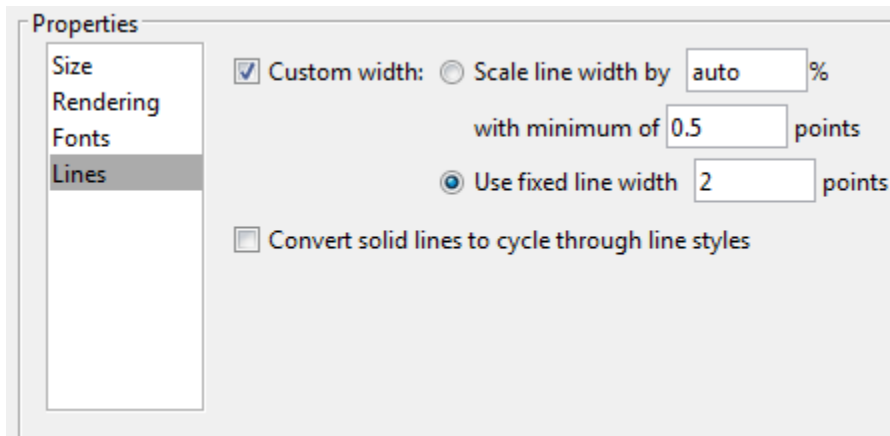
Click **Apply to Figure** on the right side of the Export Setup dialog.
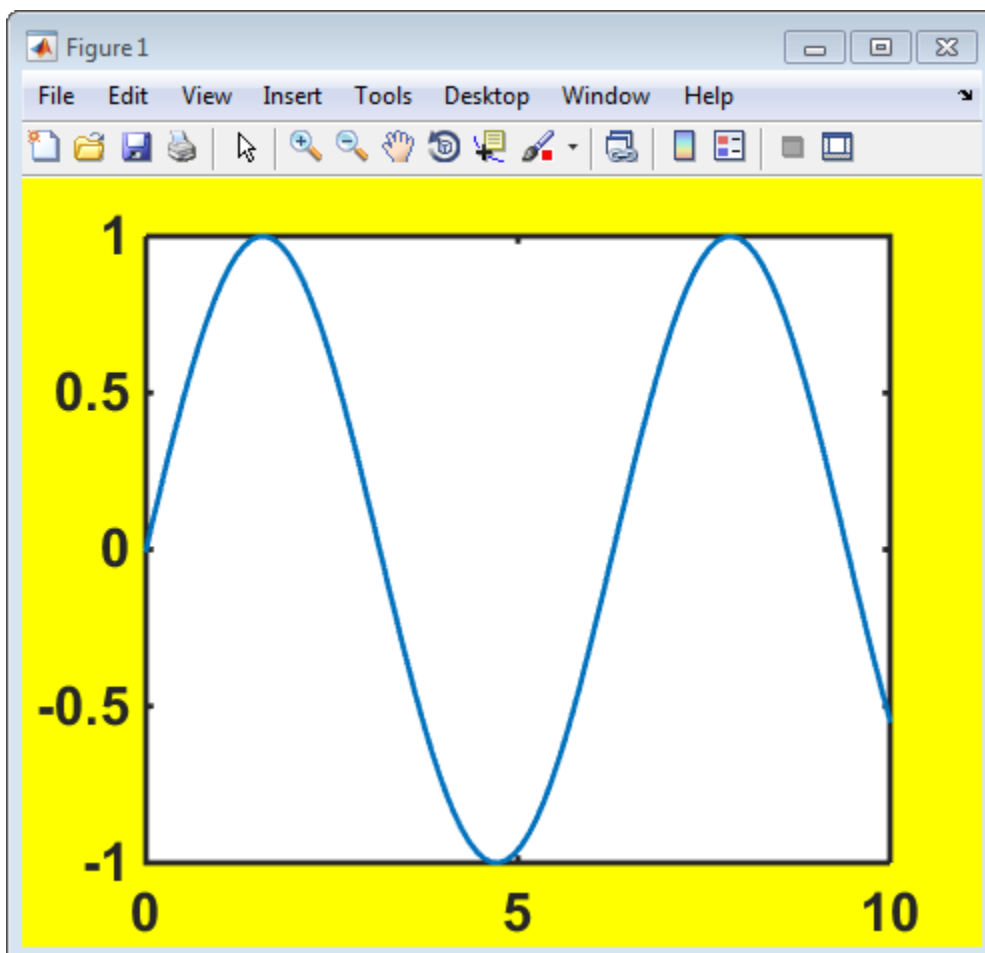
Save Figure to File

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`.
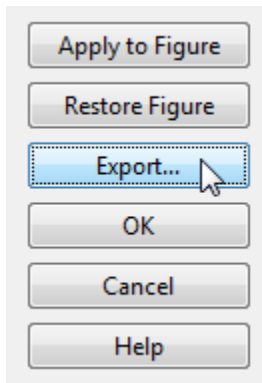
Save Figure Settings for Future Use

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.
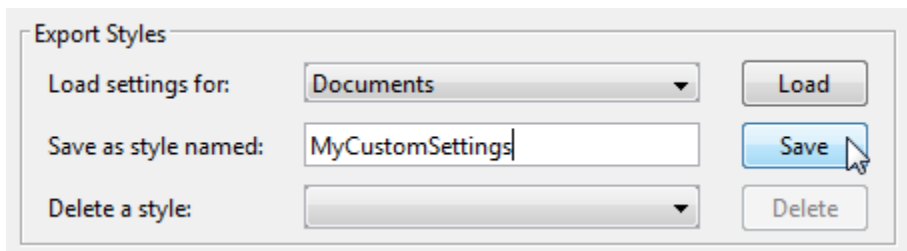
Apply Settings to Another Figure

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.
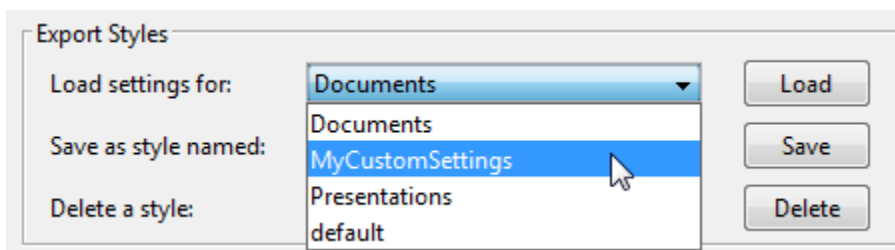
Restore Figure to Original Settings

Restore the figure on the screen to the original settings by clicking **Restore Figure**.
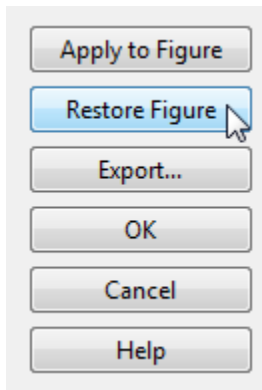
Customize Figure Programmatically

Alternatively, you can customize your figure programmatically. 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".

See Also
Property Inspector | `print` | `saveas`

Related Examples
- "Save Plot as Image or Vector Graphics File" on page 16-14
- "Save Figure with Specific Size, Resolution, or Background Color" on page 16-19

Save Plot as Image or Vector Graphics File

You can save plots as images or as vector graphics files using either the export button ✎ in the axes toolbar, or by calling the `exportgraphics` function. When deciding between the two types of content, consider the quality, file size, and formatting requirements for the document you are placing the file into.

Images are supported in most applications. They are useful for representing pictorial images and complex surfaces. However, because they made up of pixels, they do not always scale well when you print or display them on other devices that have different resolutions. In some cases, you might need to save an image with enough resolution to satisfy certain quality requirements. Higher resolution files tend to be larger, which can make them difficult to share in an email or upload to a server. It can also be difficult to edit the lines and text in an image without introducing artifacts.

Vector graphics files contain instructions for drawing lines, curves, and polygons. They are useful for representing content consisting of lines, curves, and regions of solid color. These files contain high quality content that is scalable to any size. However, some surfaces and mesh plots are too complicated to be represented using vector graphics. Some applications support extensive editing of vector graphics files, while other applications support only resizing the graphics.

Regardless of whether you save your plots as images or as vector graphics files, you can get the best results by finalizing your content in the MATLAB figure before saving your file.

Save Plots Interactively

To save a plot using interactive controls, use the export button ✎ in the axes toolbar. The toolbar appears when you hover over the upper right corner of the axes. The export button supports three image formats (PNG, JPEG, and TIFF), as well as PDF files, which can contain images or vector graphics, depending on the content in the axes.

For example, create a bar chart. Save the chart to a file by hovering over the export button ✎ in the axes toolbar and selecting the first item in the drop-down list.

```
bar([1 11 7 8 2 2 9 3 6])
```
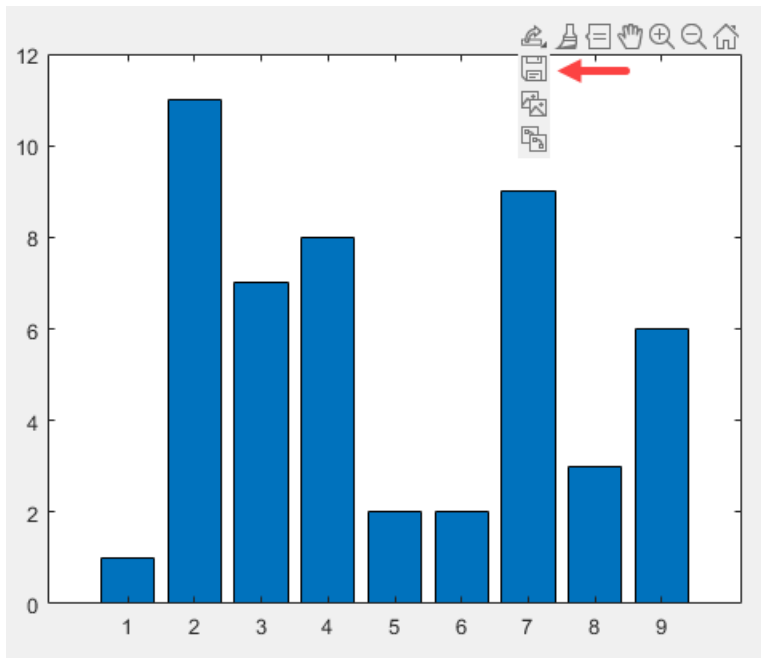
MATLAB displays the Save As dialog box with the file type options.
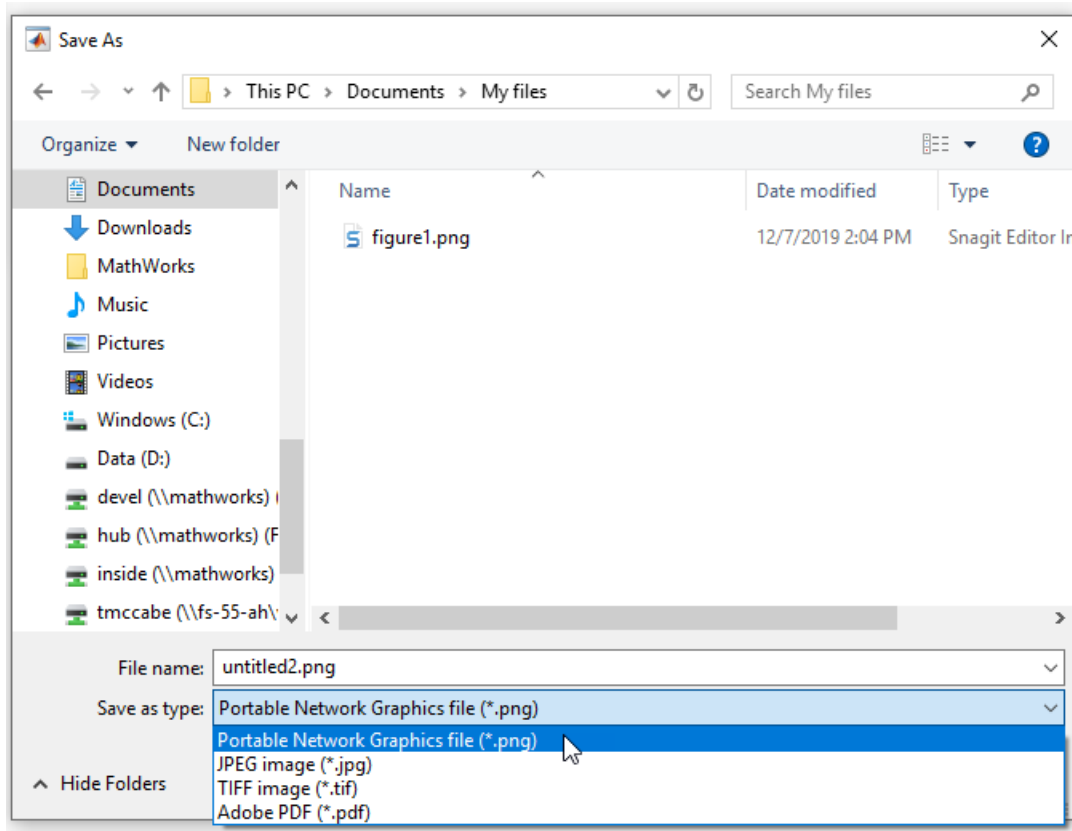
When you use the export button to save a plot, the output is tightly cropped around the axes content, including any legends or colorbars. The output does not include content outside the axes, such as other axes in the figure.

If the figure contains multiple plots in a tiled chart layout, you can save all the plots together by moving the toolbar to the layout. To move the toolbar, call the `axtoolbar` function and specify the `TiledChartLayout` object as an input argument. Then hover over the export button in the toolbar. The toolbar appears when you hover over the upper right corner of the layout

Save Plots Programmatically

Note The following examples use the `exportgraphics` function, which is available starting in R2020a. If you are using an earlier release, see Save Plot as Image or Vector Graphics File (19b).

To save plots programmatically, use the `exportgraphics` function, which is new in R2020a. The saved content is tightly cropped around the axes with minimal white space. All UI components and adjacent containers such as panels are excluded from the saved content. The `exportgraphics` function supports three image formats (PNG, JPEG and TIFF) and three formats that support both vector and image content (PDF, EPS, and EMF). The PDF format supports embedding fonts.

For example, create a bar chart and get the current figure. Then save the figure as a PNG file. In this case, specify an output resolution of 300 dots per inch (DPI).

```
bar([1 11 7 8 2 2 9 3 6])
f = gcf;
```

```
% Requires R2020a or later
exportgraphics(f,'barchart.png','Resolution',300)
```
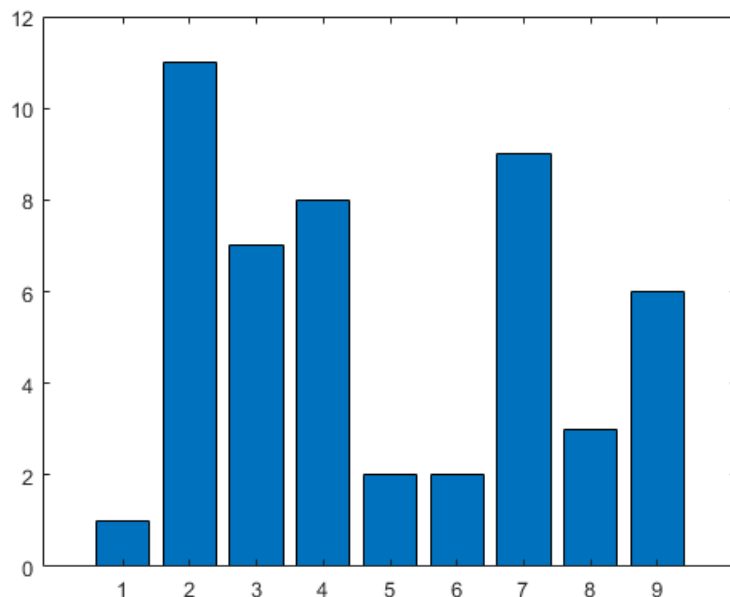

If you specify a file name with a `.pdf`, `.eps`, or `.emf` extension, MATLAB stores either an image or vector graphics depending on the content in the figure.

You can control whether the file contains an image or vector graphics by specifying the `'ContentType'` name-value pair argument. For example, save the content in the current figure as a PDF containing vector graphics.

```
% Requires R2020a or later
exportgraphics(gcf,'vectorfig.pdf','ContentType','vector')
```

To save multiple plots in a figure, create a tiled chart layout and pass the `TileChartLayout` object to the `exportgraphics` function. For example, create a 2-by-1 tiled chart layout `t`. Place two axes in the layout by calling the `nexttile` function, and plot into the axes. Then, save both plots as an EPS file by calling the `exportgraphics` function with `t` as the first argument.

```
t = tiledlayout(2,1);
nexttile
plot([0 1 0 1])
nexttile
plot([1 0 1 0])
```

```
% Requires R2020a or later
exportgraphics(t,'twoplots.eps')
```
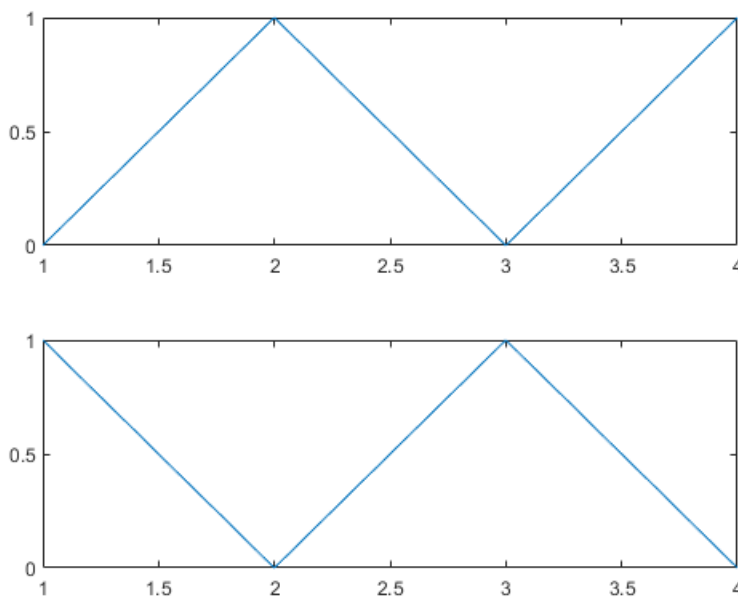

Open Saved Plots in Other Applications

You can open the files you save in other applications such as Microsoft Word or LaTeX.

To add a plot to a LaTeX document, first save the plot as an EPS file using the `exportgraphics` function. Then add the `\includegraphics` element to the LaTeX document. For example:

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{twoplots.eps}}
\caption{Bar Chart from MATLAB}
```

```
\end{figure}

\end{document}
```

See Also
copygraphics | exportgraphics | nexttile | tiledlayout

Related Examples
- "Save Figure with Specific Size, Resolution, or Background Color" on page 16-19
- "Saving and Copying Plots with Minimal White Space" on page 16-25
- "Save Figure to Reopen in MATLAB Later" on page 16-23

Save Figure with Specific Size, Resolution, or Background Color

In this section...
"Specify Resolution" on page 16-19
"Specify Size" on page 16-20
"Specify Background Color" on page 16-21
"Preserve Axis Limits and Tick Values" on page 16-22

Since R2020a. Replaces Save Figure at Specific Size and Resolution (R2019b) and Save Figure Preserving Background Color (R2019b).

To save plots for including in documents, such as publications or slide presentations, use the `exportgraphics` function. This function enables you to save plots at the appropriate size, resolution, and background color for your document. The saved content is tightly cropped around the axes with minimal white space. All UI components and adjacent containers such as panels are excluded from the saved content.

Specify Resolution

To save a figure as an image at a specific resolution, call the `exportgraphics` function, and specify the `'Resolution'` name-value pair argument. By default, images are saved at 150 dots per inch (DPI).

For example, create a bar chart and get the current figure. Then save the figure as a 300-DPI PNG file.

```
bar([1 11 7 8 2 2 9 3 6])
f = gcf;
exportgraphics(f,'barchart.png','Resolution',300)
```
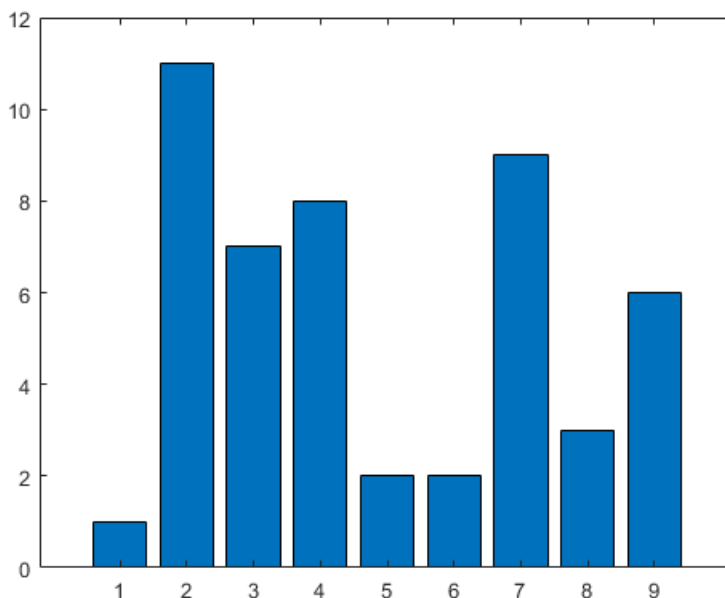
Alternatively, you can specify the axes instead of the figure as the first argument to the `exportgraphics` function.

```
ax = gca;
exportgraphics(ax,'barchartaxes.png','Resolution',300)
```

Specify Size

The `exportgraphics` function captures content at the same width and height as it is displayed on your screen. If you want to change the width and height, then adjust the size of the content displayed in the figure. One way to do this is to create the plot in a tiled chart layout at the desired size without any padding. Then pass the layout to the `exportgraphics` function.

For example, to save a bar chart as a 3-by-3 inch square image, start by creating a 1-by-1 tiled chart layout `t`, and set the `'Padding'` name-value pair argument to `'tight'`.

```
t = tiledlayout(1,1,'Padding','tight');
```

Before R2021a, set `'Padding'` *to* `'none'`.

Set the `Units` property of `t` to inches. Then set the `OuterPosition` property of `t` to [0.25 0.25 3 3]. The first two numbers in the vector position the layout at 0.25 inches from the left and bottom edges of the figure. The last two numbers set the width and height of the layout to 3 inches.

```
t.Units = 'inches';
t.OuterPosition = [0.25 0.25 3 3];
```

Next, create an axes object by calling the `nexttile` function. Then create a bar chart in the axes.

```
nexttile;
bar([1 11 7 8 2 2 9 3 6])
```

Save the layout as a 300-DPI JPEG file by passing `t` to the `exportgraphics` function. The resulting image is approximately 3 inches square.

```
exportgraphics(t,'bar3x3.jpg','Resolution',300)
```
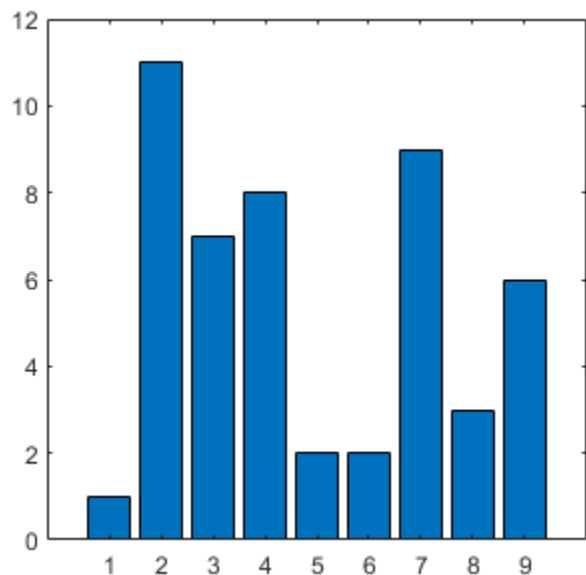
An alternative way to change the size is to save the content as a vector graphics file. Then you can resize the content in your document. To save the content as a vector graphics file, call the `exportgraphics` function and set the `'ContentType'` name-value pair argument to `'vector'`. For example, create a bar chart, and save the figure as a PDF file containing vector graphics. All embeddable fonts are included in the PDF.

```
bar([1 11 7 8 2 2 9 3 6])
f = gcf;
exportgraphics(f,'barscalable.pdf','ContentType','vector')
```

Specify Background Color

By default, the `exportgraphics` function saves content with a white background. You can specify a different background by setting the `BackgroundColor` name-value pair argument. These are the possible values:

- `'current'` — Uses the color of the axes parent container (such as a figure or a panel).
- `'none'` — Sets the background color to transparent or white, depending on the file format and the value of `ContentType`:
  - Transparent — For files with `ContentType='vector'`
  - White — For image files, or when `ContentType='image'`
- A custom color, specified as an RGB triplet such as `[1 0 0]`, a hexadecimal color code such as `#FF0000`, or a named color such as `'red'`.

For example, create a bar chart, and save the figure as a PDF file with a transparent background.

```
bar([1 11 7 8 2 2 9 3 6])
f = gcf;
exportgraphics(f,'bartransparent.pdf','ContentType','vector',...
               'BackgroundColor','none')
```
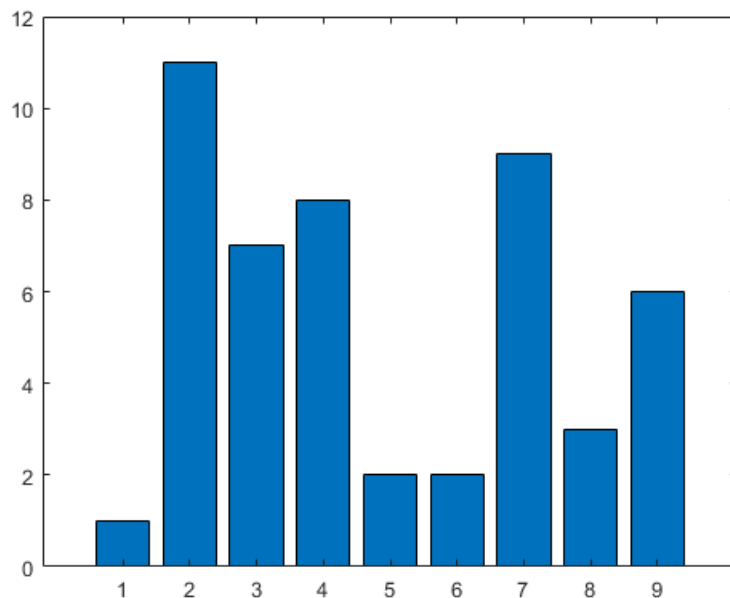
Preserve Axis Limits and Tick Values

Occasionally, the `exportgraphics` function saves your content with different axis limits or tick values depending on the size of the font and the resolution of the file. To keep the axis limits and tick values from changing, set the tick value mode and limit mode properties on the axes to `'manual'`. For example, when plotting into Cartesian axes, set the tick value and limit mode properties for the *x*-, *y*-, and *z*-axis.

```
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';
exportgraphics(ax,'barticks.png')
```
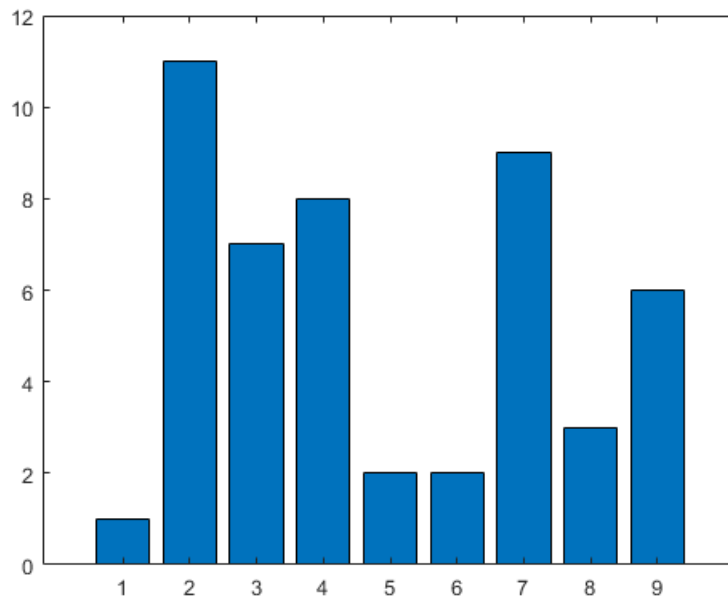

For polar plots, set the `RTickMode`, `ThetaTickMode`, `RLimMode`, and `ThetaLimMode` properties on the polar axes to `'manual'`.

See Also

Functions
copygraphics | exportgraphics | nexttile | tiledlayout

Properties
Axes | PolarAxes | TiledChartLayout Properties

More About

- "Save Plot as Image or Vector Graphics File" on page 16-14
- "Saving and Copying Plots with Minimal White Space" on page 16-25

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.

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)')
```
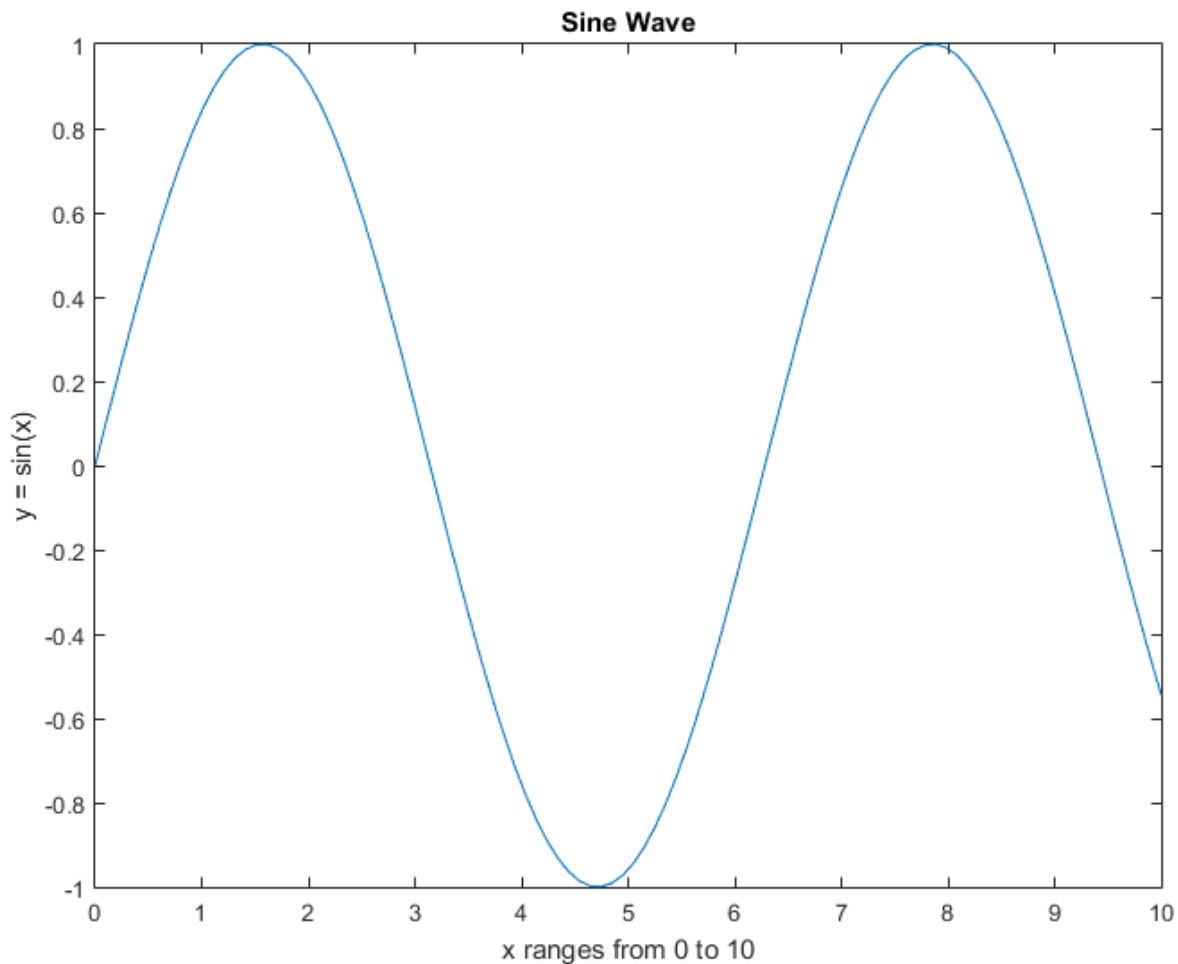

Save the figure to a FIG-file using the `savefig` function. The FIG-file stores the information required to recreate the figure.

```
savefig('SineWave.fig')
```

16-23

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 Plot as Image or Vector Graphics File" on page 16-14.

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 Plot as Image or Vector Graphics File" on page 16-14

Saving and Copying Plots with Minimal White Space

One way to minimize the white space when saving or copying the contents of a plot is to use the axes toolbar, which appears when you hover over the upper right corner of the axes. An alternative method is to use the `exportgraphics` and `copygraphics` functions, which provide more flexibility.

Note The following examples use the `exportgraphics` and `copygraphics` functions, which are new in R2020a. If you are using an earlier release, see Save Plots with Minimal White Space (19b).

Saving or Copying a Single Plot

Create a contour plot of the `peaks` function with a title and a colorbar.

```
contour(peaks)
colorbar
title('Peaks Function')
```

Save the plot to a file by hovering over the export button ✎ in the axes toolbar and selecting the first item in the drop-down list. If you want to copy the contents of the plot to the clipboard, select either the second or the third item in the drop-down list. The second item copies the content as an image, and the third items copies the content as a vector graphic. The content you save or copy is tightly cropped around the title, the axes, and the colorbar.
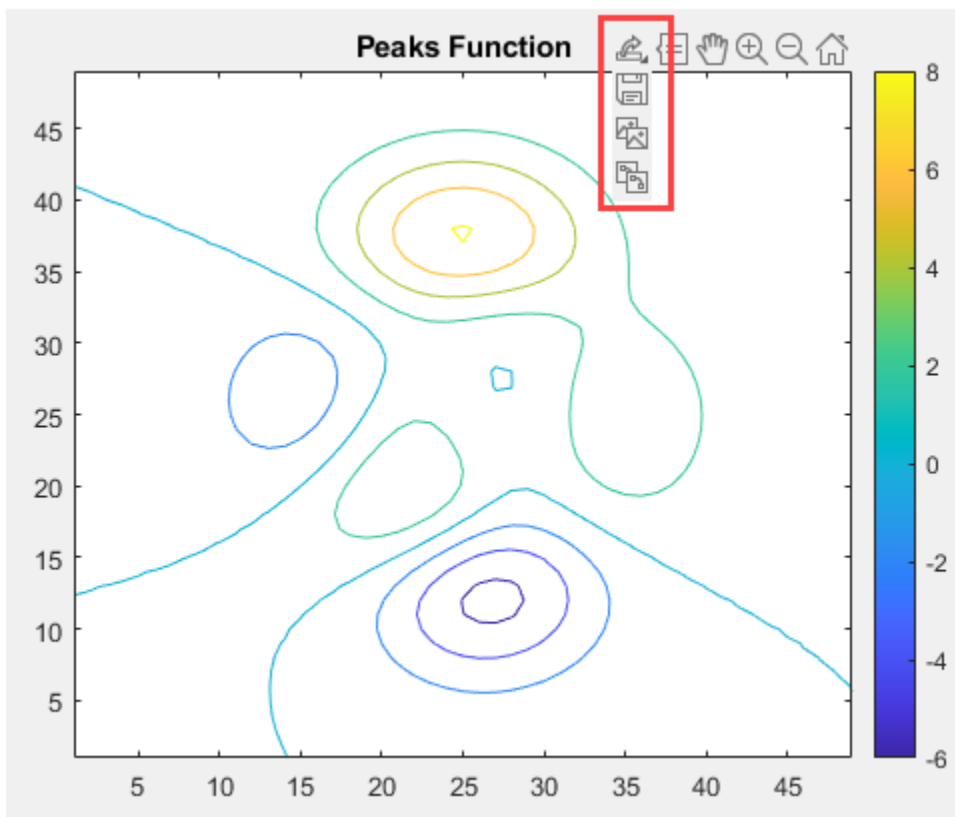
Alternatively, you can save the content using the `exportgraphics` function, which is available starting in R2020a. This function provides the same tight cropping around your content, and it also provides additional options. For example, you can save an image file and specify the resolution.

```matlab
ax = gca;
% Requires R2020a or later
exportgraphics(ax,'myplot.png','Resolution',300)
```

The `copygraphics` function provides similar functionality for copying content to the clipboard.

```matlab
ax = gca;
% Requires R2020a or later
copygraphics(ax,'Resolution',300)
```

Saving or Copying Multiple Plots in a Figure

Starting in R2019b, you can create a tiling of plots in a figure using the `tiledlayout` function. This function has options for minimizing the space around the plots. (If you are using an earlier release, you can use the `subplot` function to create a tiling of plots. However, the `subplot` function does not have options for controlling the space around the plots.)

Create a 2-by-2 tiled chart layout by calling the `tiledlayout` function. To minimize the space between the plots, set the `'TileSpacing'` name-value pair argument to `'compact'`. To minimize the space around the perimeter of the layout, set the `'Padding'` name-value pair argument to `'compact'`. Next, call the `nexttile` function to create the first axes, and call the `plot` function to plot into the axes. Then, create three more axes and plots.

```matlab
% Requires R2019b or later
t = tiledlayout(2,2,'TileSpacing','Compact','Padding','Compact');
nexttile
plot([0 1])
nexttile
plot([1 0])
nexttile
plot([0 1 0 1])
nexttile
plot([1 0 1 0])
```
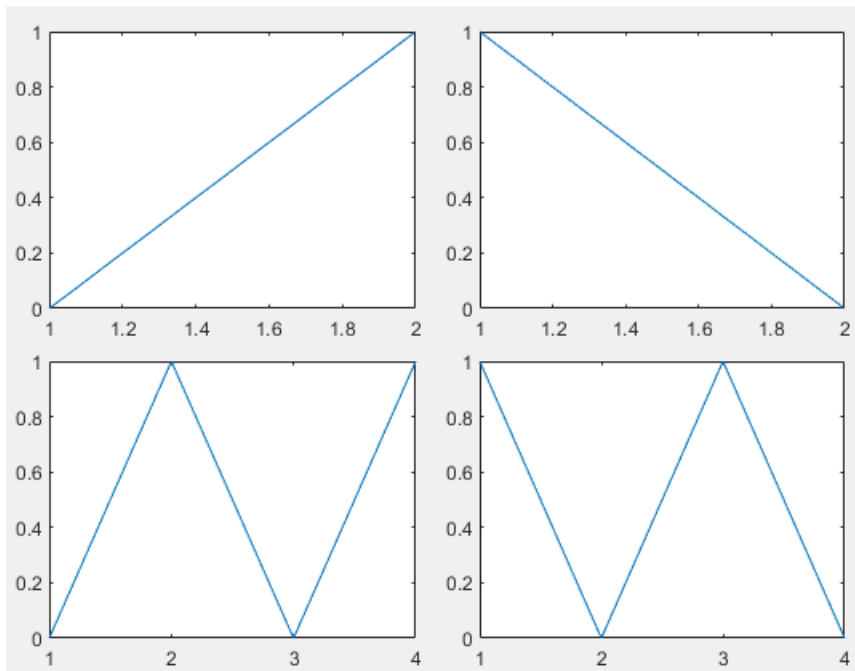
Save the layout as a PDF file by passing the tiled chart layout (`t`) to the `exportgraphics` function. In this case, save the PDF with a transparent background.

```
% Requires R2020a or later
exportgraphics(t,'fourplots.pdf','BackgroundColor','none')
```

Alternatively, you can copy the layout to the clipboard using the `copygraphics` function.

```
% Requires R2020a or later
copygraphics(t,'BackgroundColor','none')
```

See Also

Functions
copygraphics | exportgraphics | nexttile | tiledlayout

Properties
TiledChartLayout Properties

More About

- "Save Plot as Image or Vector Graphics File" on page 16-14

Graphics Properties

Modify Graphics Objects

This example shows how to create, display, and modify graphics objects in MATLAB®.

Graphics Objects

When MATLAB creates a plot, it creates a series of graphics objects. Figures, axes, lines, patches, and text are examples of graphics objects. The figure below has three graphics objects -- an axes, a line, and a text object. Use an optional output argument to store the graphics object that is created.

```
x = -pi:pi/20:pi;
y = sin(x);

f = figure;
p = plot(x,y);
txt1 = text(0.2,0,'sin(x)');
```
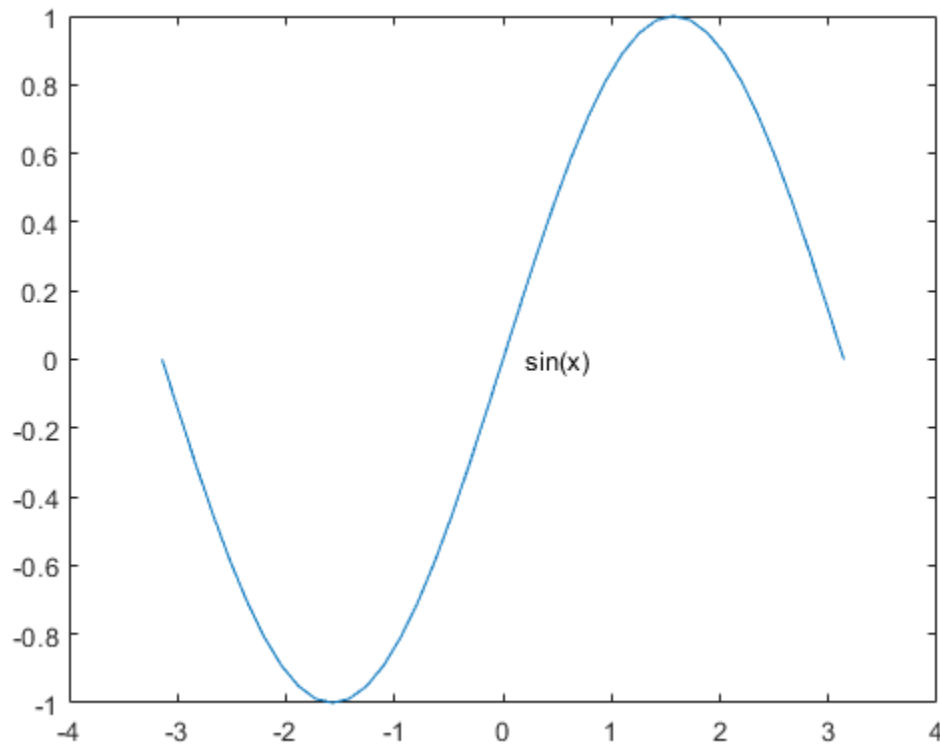

All graphics objects have properties that you can view and modify. These properties have default values. The display of the line object, `p`, shows the most commonly used line properties, such as `Color`, `LineStyle`, and `LineWidth`.

```
p
```

```
p =
  Line with properties:

            Color: [0 0.4470 0.7410]
```

```
         LineStyle: '-'
         LineWidth: 0.5000
            Marker: 'none'
        MarkerSize: 6
   MarkerFaceColor: 'none'
             XData: [1x41 double]
             YData: [1x41 double]
             ZData: [1x0 double]

  Show all properties
```

MATLAB shows the same display if the semicolon is missing from the command that creates the object.

```
txt2 = text(x(end), y(end), 'pi')
```
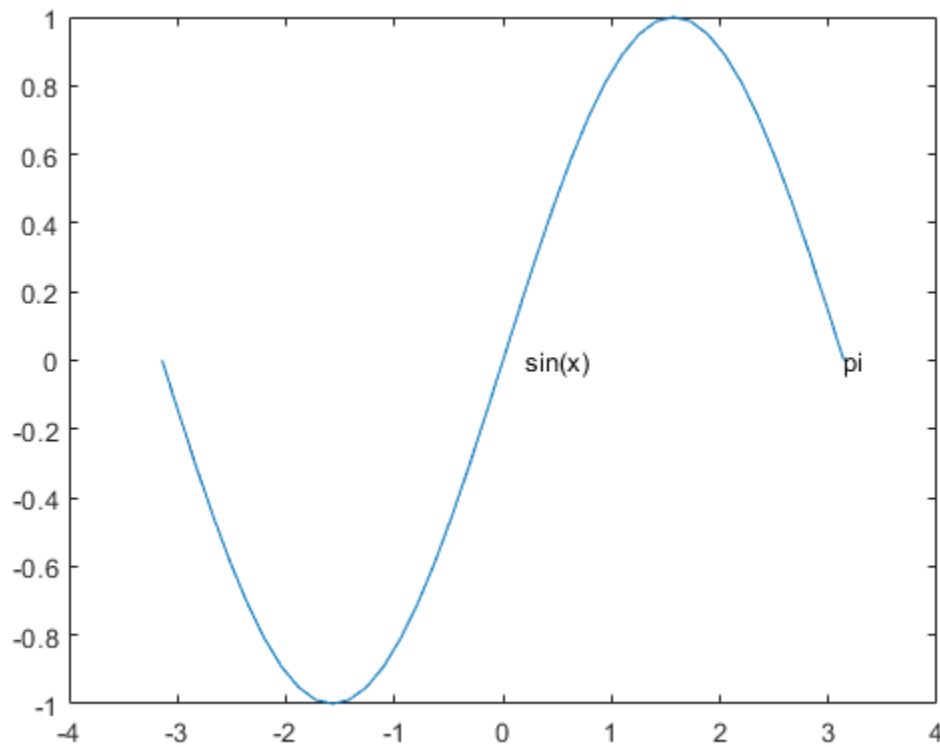


```
txt2 = 
  Text (pi) with properties:

               String: 'pi'
             FontSize: 10
           FontWeight: 'normal'
             FontName: 'Helvetica'
                Color: [0 0 0]
  HorizontalAlignment: 'left'
             Position: [3.1416 1.2246e-16 0]
                Units: 'data'
```

```
Show all properties
```

Get Graphics Object Properties

To access individual properties of a graphics object, use dot notation syntax `object.PropertyName`. For example, return the `LineWidth` property for the line object.

```
pcol = p.LineWidth
```

```
pcol = 0.5000
```

Change the line color to red by setting its `Color` property.

```
p.Color = 'red';
```
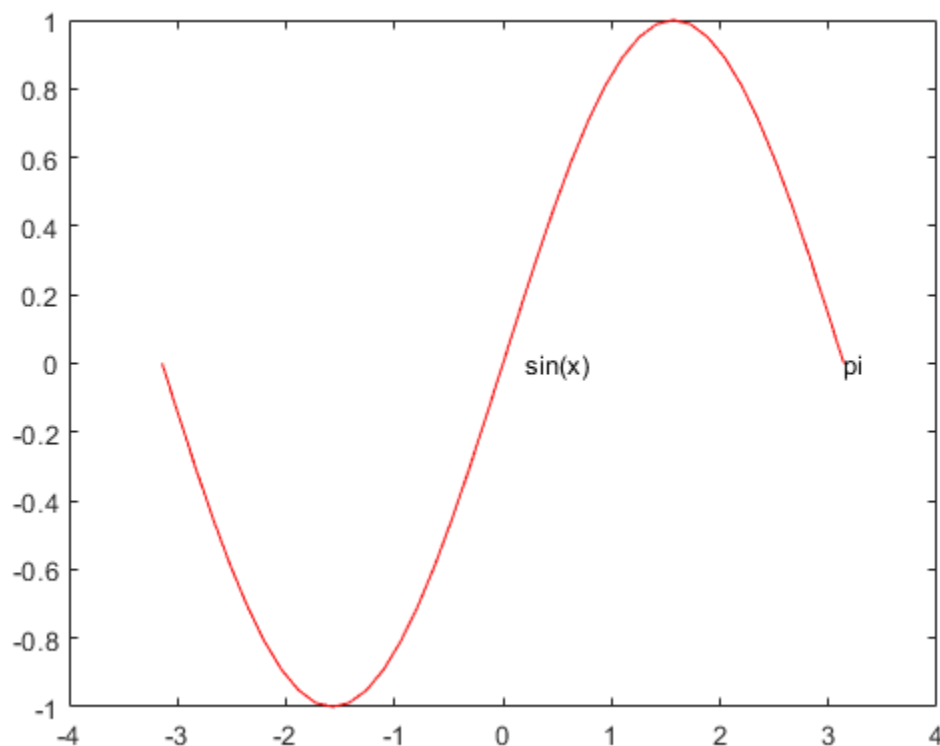

Parents and Children

MATLAB arranges graphics objects in a hierarchy. The top of the hierarchy is a special object called the *graphics root*. To access the graphics root, use the `groot` function.

```
groot
```

```
ans =
  Graphics Root with properties:

        CurrentFigure: [1x1 Figure]
```

```
      ScreenPixelsPerInch: 96
                ScreenSize: [1 1 1280 1024]
          MonitorPositions: [1 1 1280 1024]
                     Units: 'pixels'

  Show all properties
```

All graphics objects (except the root) have a parent. For example, the parent of an axes is a figure.

```
ax = gca;
ax.Parent

ans =
  Figure (1) with properties:

        Number: 1
          Name: ''
         Color: [1 1 1]
      Position: [360 502 560 420]
         Units: 'pixels'

  Show all properties
```

Many objects also have children. This axes has three children - the two text objects and the line object.

```
ax.Children

ans =
  3x1 graphics array:

  Text     (pi)
  Text     (sin(x))
  Line
```

Since the axes has multiple children, the value of the `Children` property is an array of graphics objects. To access an individual child of the axes, index into the array. You can then set properties of the child object.

```
t = ax.Children(2);       % get the 'sin(x)' text object
t.FontWeight = 'bold';    % set the font to bold
```
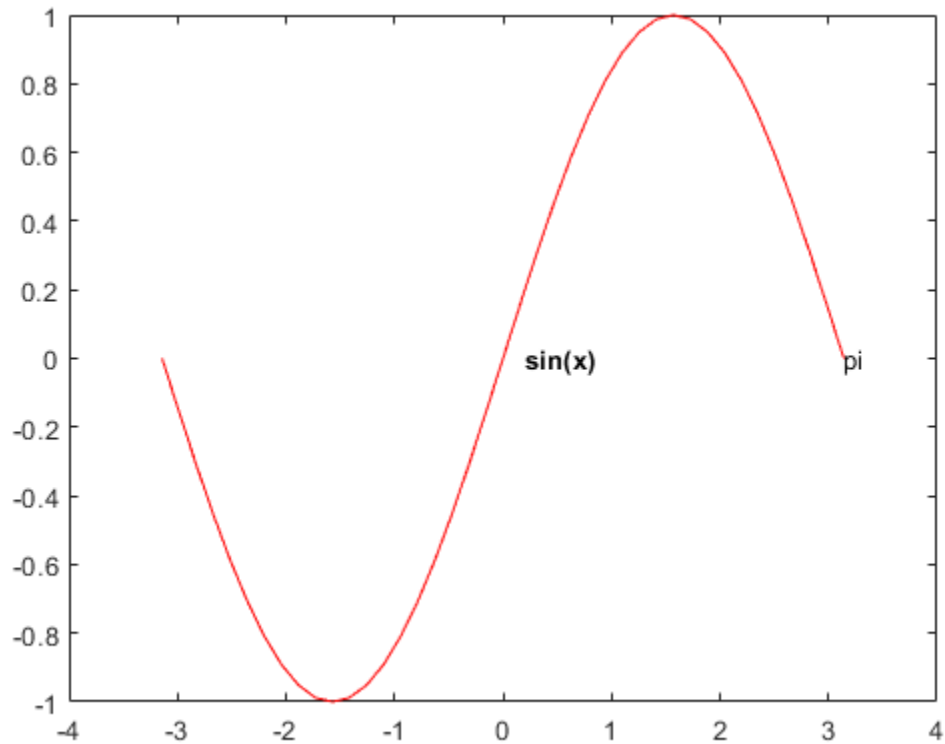
Preallocate Graphics Objects Array

It is a best practice in MATLAB to preallocate an array before using it. Use the `gobjects` command to preallocate an array of graphics objects. You can then add graphics objects to the array.

```
objarray = gobjects(1,5);
objarray(1) = f;
objarray(2) = ax;
objarray(3) = p;
objarray(4) = txt1;
objarray(5) = txt2;
objarray

objarray =
  1x5 graphics array:

    Figure    Axes       Line       Text       Text
```

Get All Object Properties

Graphics objects in MATLAB have many properties. To see all the properties of an object, use the `get` command.

```
get(f)

              Alphamap: [1x64 double]
          BeingDeleted: off
```

```
              BusyAction: 'queue'
          ButtonDownFcn: ''
               Children: [1x1 Axes]
               Clipping: on
        CloseRequestFcn: 'closereq'
                  Color: [1 1 1]
               Colormap: [256x3 double]
            ContextMenu: [0x0 GraphicsPlaceholder]
              CreateFcn: ''
            CurrentAxes: [1x1 Axes]
       CurrentCharacter: ''
          CurrentObject: [0x0 GraphicsPlaceholder]
           CurrentPoint: [0 0]
              DeleteFcn: ''
           DockControls: on
               FileName: ''
      GraphicsSmoothing: on
       HandleVisibility: 'on'
                   Icon: ''
          InnerPosition: [360 502 560 420]
          IntegerHandle: on
          Interruptible: on
          InvertHardcopy: on
            KeyPressFcn: ''
          KeyReleaseFcn: ''
                MenuBar: 'none'
                   Name: ''
               NextPlot: 'add'
                 Number: 1
            NumberTitle: on
          OuterPosition: [360 502 560 420]
        PaperOrientation: 'portrait'
          PaperPosition: [1.3333 3.3125 5.8333 4.3750]
        PaperPositionMode: 'auto'
              PaperSize: [8.5000 11]
              PaperType: 'usletter'
             PaperUnits: 'inches'
                 Parent: [1x1 Root]
                Pointer: 'arrow'
        PointerShapeCData: [16x16 double]
      PointerShapeHotSpot: [1 1]
               Position: [360 502 560 420]
               Renderer: 'opengl'
           RendererMode: 'auto'
                 Resize: on
             Scrollable: off
          SelectionType: 'normal'
         SizeChangedFcn: ''
                    Tag: ''
                ToolBar: 'none'
                   Type: 'figure'
                  Units: 'pixels'
               UserData: []
                Visible: off
    WindowButtonDownFcn: ''
  WindowButtonMotionFcn: ''
      WindowButtonUpFcn: ''
       WindowKeyPressFcn: ''
```

```
         WindowKeyReleaseFcn: ''
       WindowScrollWheelFcn: ''
                 WindowState: 'normal'
                 WindowStyle: 'normal'
```

Graphics Object Hierarchy

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.
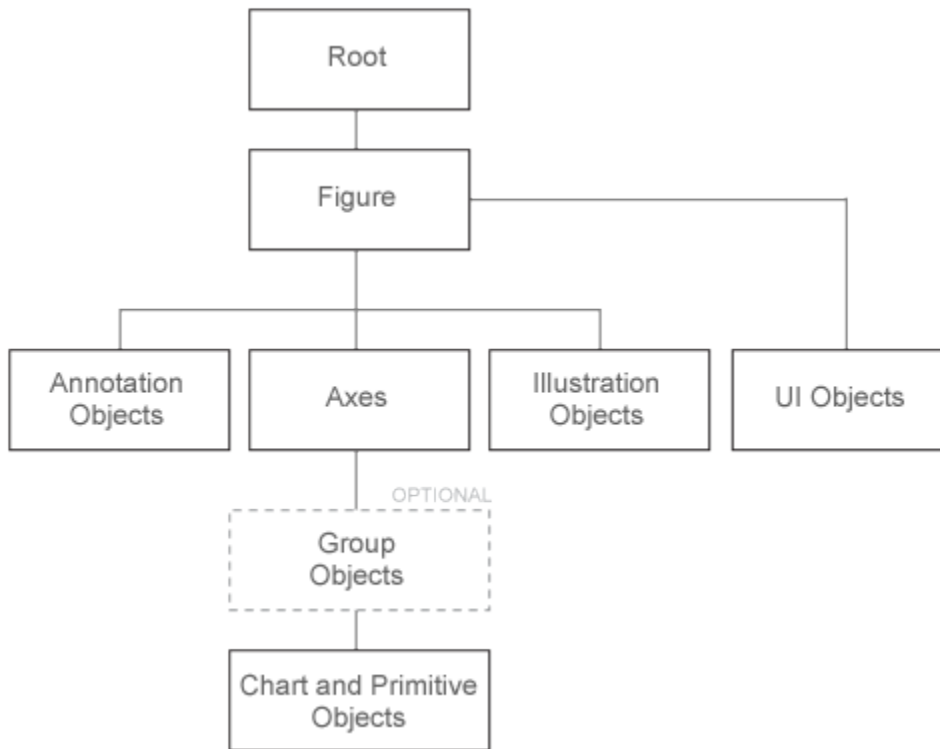
The hierarchical nature of graphics objects reflects the containment of objects by other objects. Each object plays a specific role in the graphics display.

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.
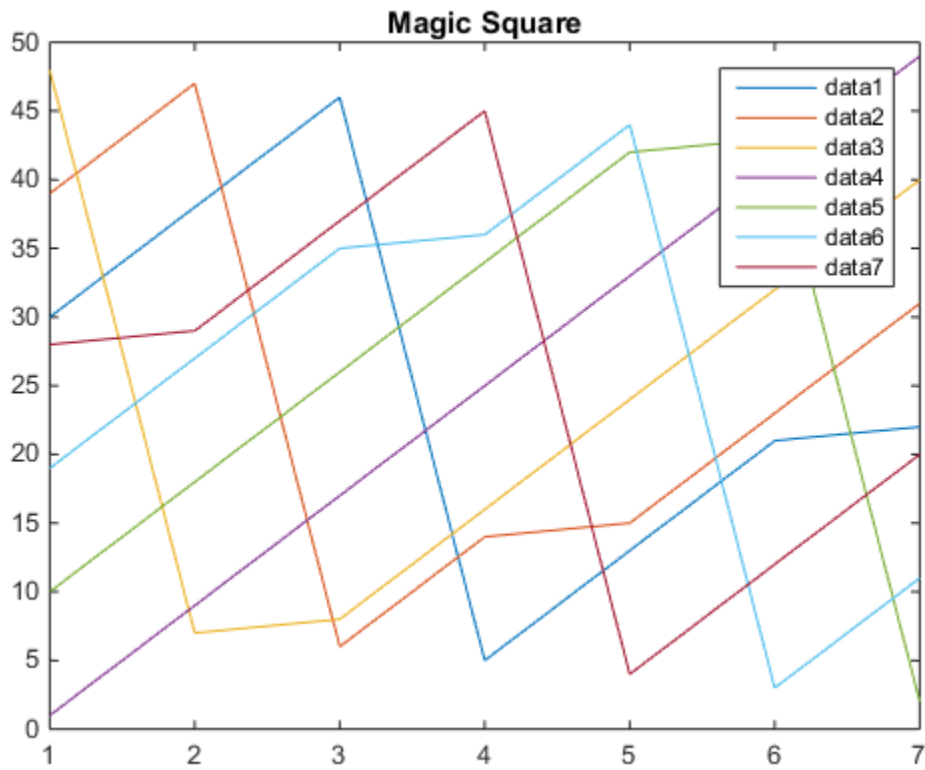
This graph forms a hierarchy of objects.
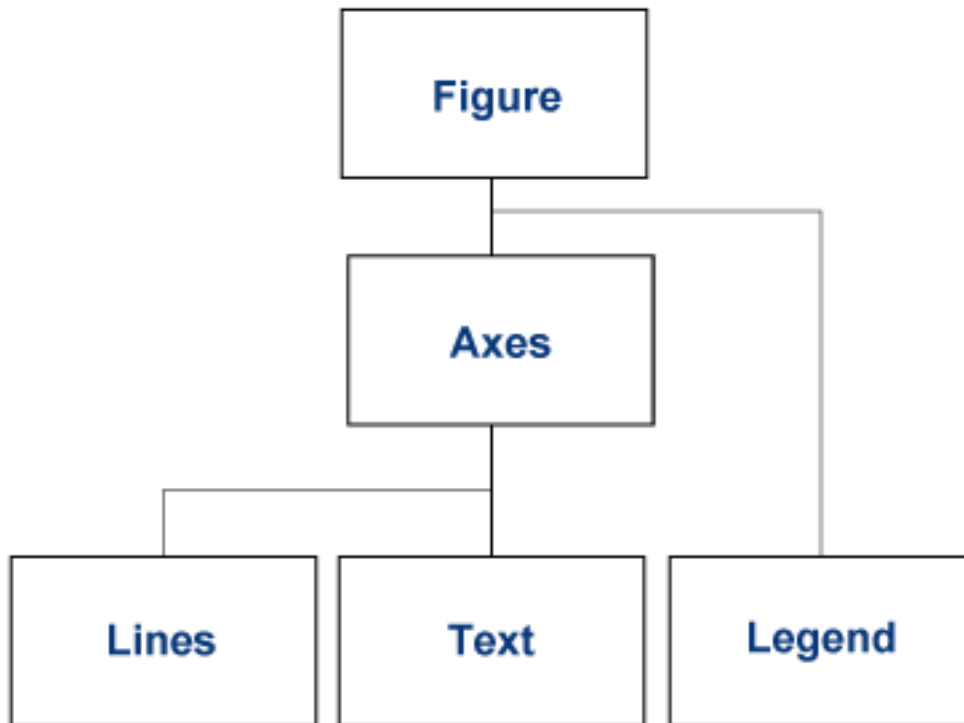
Parent-Child Relationship

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)
```

Access Property Values

In this section...

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.

The dot notation 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'
```

Features Controlled by Graphics Objects

In this section...

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

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

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')
```
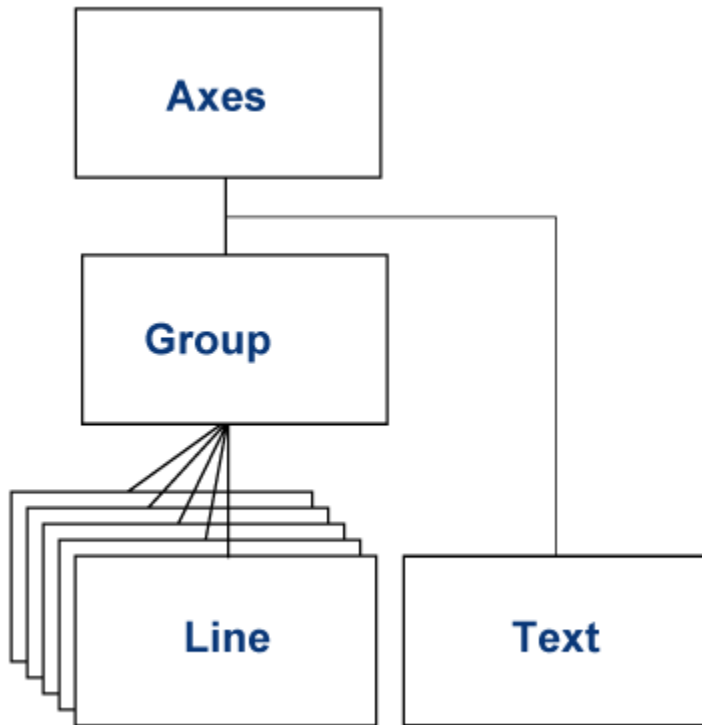
Annotation Objects

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.

Default Property Values

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 functions 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 current MATLAB session
- Figure — use for default values applied to children of the figure defining the defaults.
- Axes — use for default values applied only to children of the axes defining the defaults and only when using low-level functions (`light`, `line`, `patch`, `rectangle`, `surface`, `text`, and the low-level form of `image`).

For example, specify a default figure color only on the root level.

```
set(groot,'defaultFigureColor','b')
```

List Default Values

Use `get` to determine what default values are currently set on any given object level:

```
get(groot,'default')
```

returns all default values set in your current MATLAB session.

Set Properties to the Current Default

Specifying a property value of `'default'` sets the property to the first encountered default value defined for that property. For example, these statements result in a green surface `EdgeColor`:

```
set(groot,'defaultSurfaceEdgeColor','k')
h = surface(peaks);
set(gcf,'defaultSurfaceEdgeColor','g')
set(h,'EdgeColor','default')
```

Because a default value for surface `EdgeColor` exists on the figure level, MATLAB encounters this value first and uses it instead of the default `EdgeColor` defined on the root.

Remove Default Values

Specifying a property value of `'remove'` gets rid of user-defined default values. The statement

```
set(groot,'defaultSurfaceEdgeColor','remove')
```

removes the definition of the default surface `EdgeColor` from the root.

Set Properties to Factory-Defined Values

Specifying a property value of `'factory'` sets the property to its factory-defined value. For example, these statements set the `EdgeColor` of surface `h` to black (its factory setting), regardless of what default values you have defined:

```
set(gcf,'defaultSurfaceEdgeColor','g')
h = surface(peaks);
set(h,'EdgeColor','factory')
```

List Factory-Defined Property Values

You can list factory values:

- `get(groot,'factory')` — List all factory-defined property values for all graphics objects
- `get(groot,'factoryObjectType')` — List all factory-defined property values for a specific object
- `get(groot,'factoryObjectTypePropertyName')` — List factory-defined value for the specified property.

Reserved Words

Setting a property value to `default`, `remove`, or `factory` produces the effects described in the previous sections. To set a property to one of these words (for example, a text `String` property set to the word `default`), precede the word with the backslash character:

```
h = text('String','\default');
```

Default Values for Automatically Calculated Properties

In this section...
"What Are Automatically Calculated Properties" on page 17-26
"Default Values for Automatically Calculated Properties" on page 17-26

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]`:
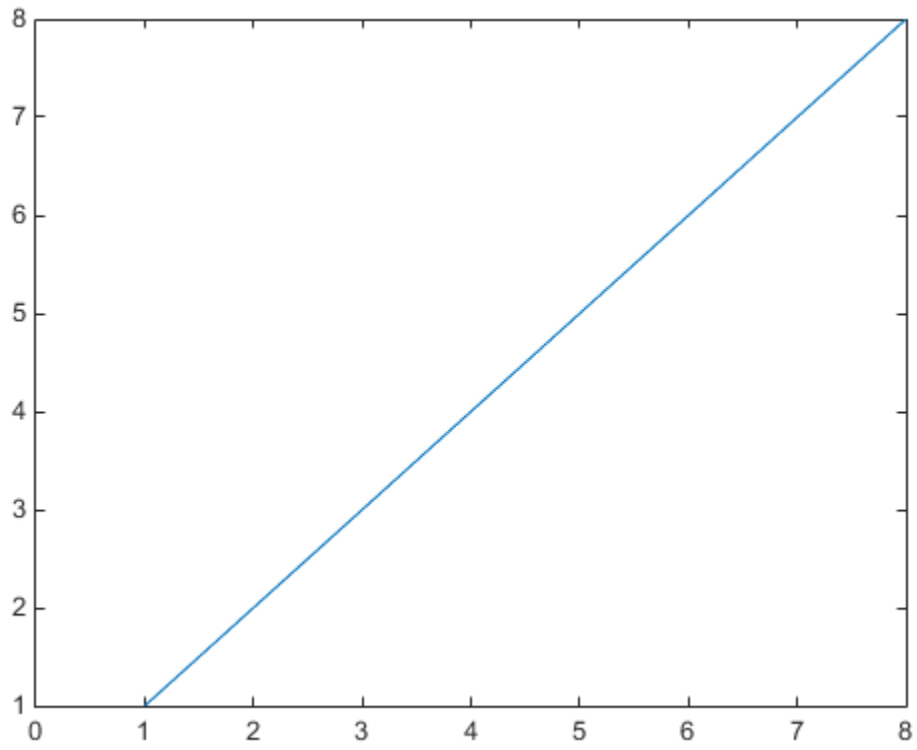
How MATLAB Finds Default Values

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')
```

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')
```
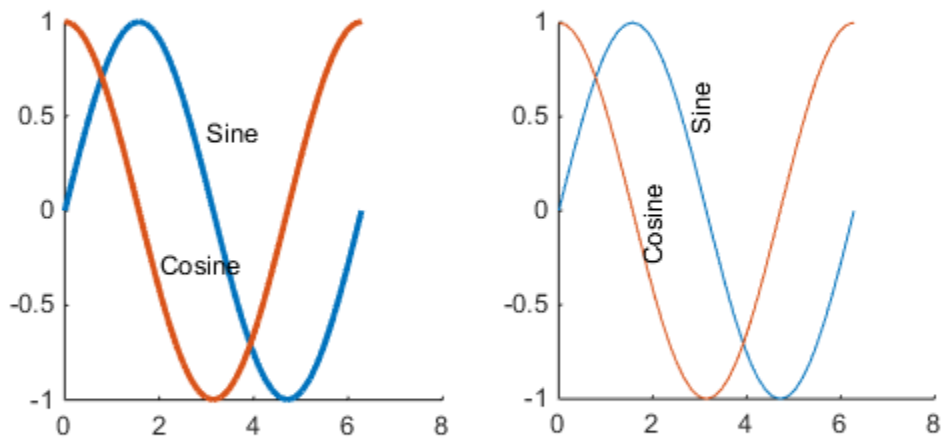

Issuing the same `plot` and `text` statements to each subplot region results in a different display, reflecting different default values defined for the axes. The default defined on the figure applies to both axes.

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.

Object Identification

Special Object Identifiers

In this section...
"Getting Handles to Special Objects" on page 18-2
"The Current Figure, Axes, and Object" on page 18-2
"Callback Object and Callback Figure" on page 18-3

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 22-11.

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 20-3

Find Objects

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.

Value of the Sine from 0 to 2π
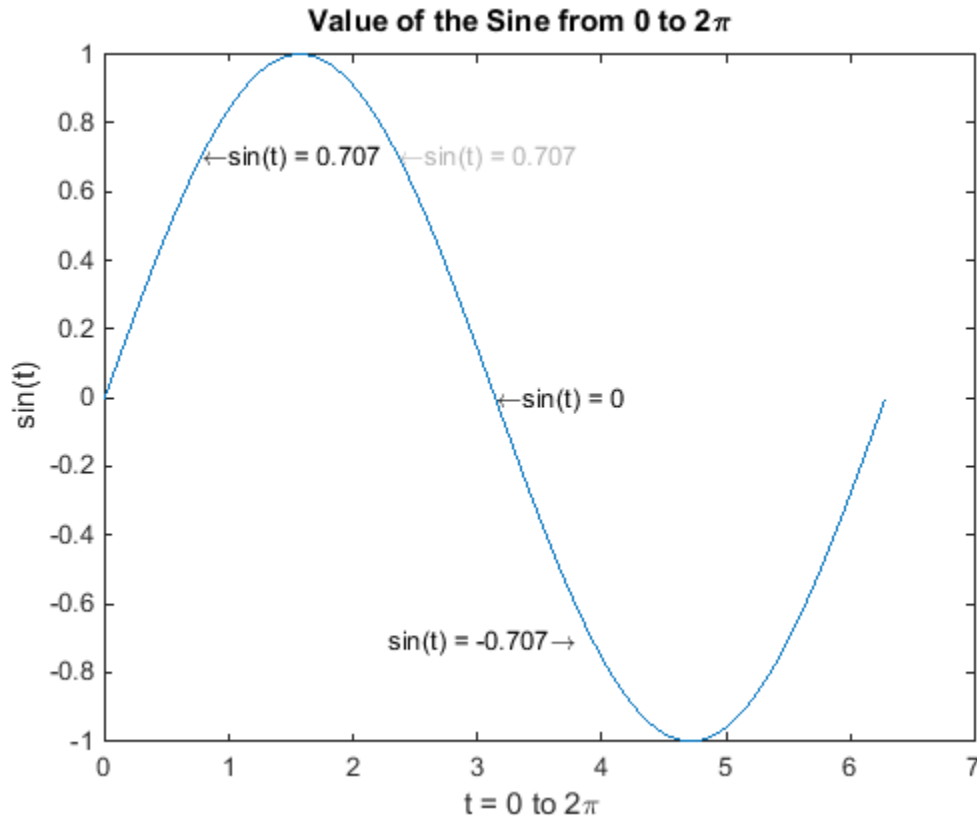
Suppose that you want to move the text labeling the value sin(t) = .707 from its current location at `[pi/4,sin(pi/4)]` to the point `[3*pi/4,sin(3*pi/4)]` where the function has the same value (shown in light gray out in the graph).

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';
```
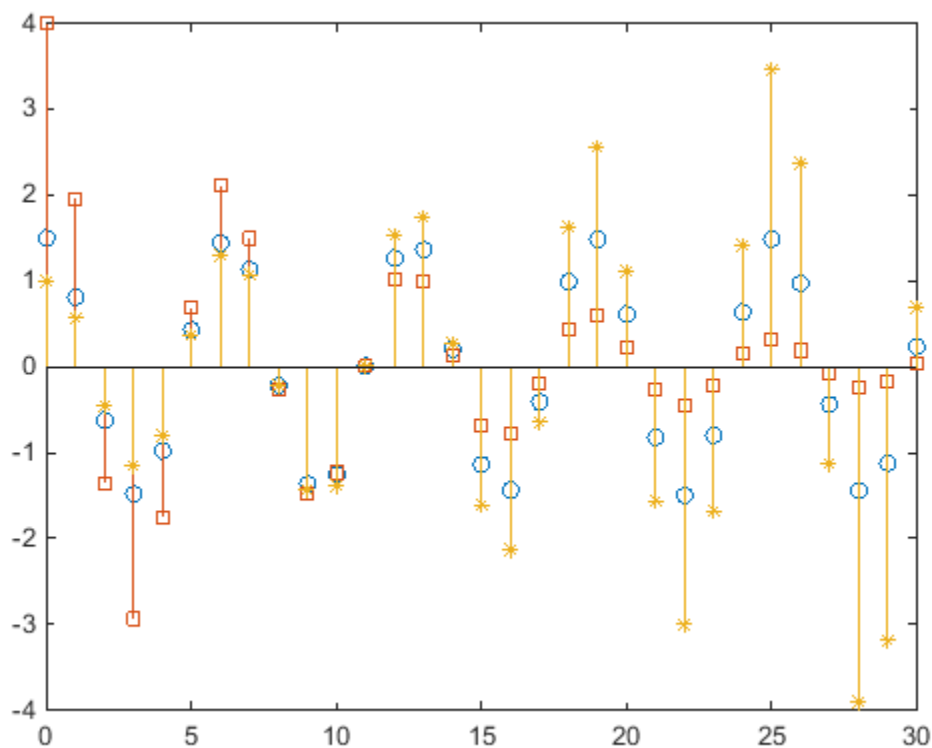

Passing a regular expression to `findobj` enables you to match specific patterns. For example, suppose that you want to set the value of the `MarkerFaceColor` property to green on all stem objects that do *not* have their `Tag` property set to `'Steady State'` (that is, stems that represent decaying 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)]';
tiledlayout(3,1)
ax1 = nexttile;
stem(x,y(:,1))
ax2 = nexttile;
stem(x,y(:,2))
ax3 = nexttile;
stem(x,y(:,3))
```

Set the marker face color of the stems in the third axes only.

```
h = findobj(ax3,'Type','stem');
h.MarkerFaceColor = 'red';
```
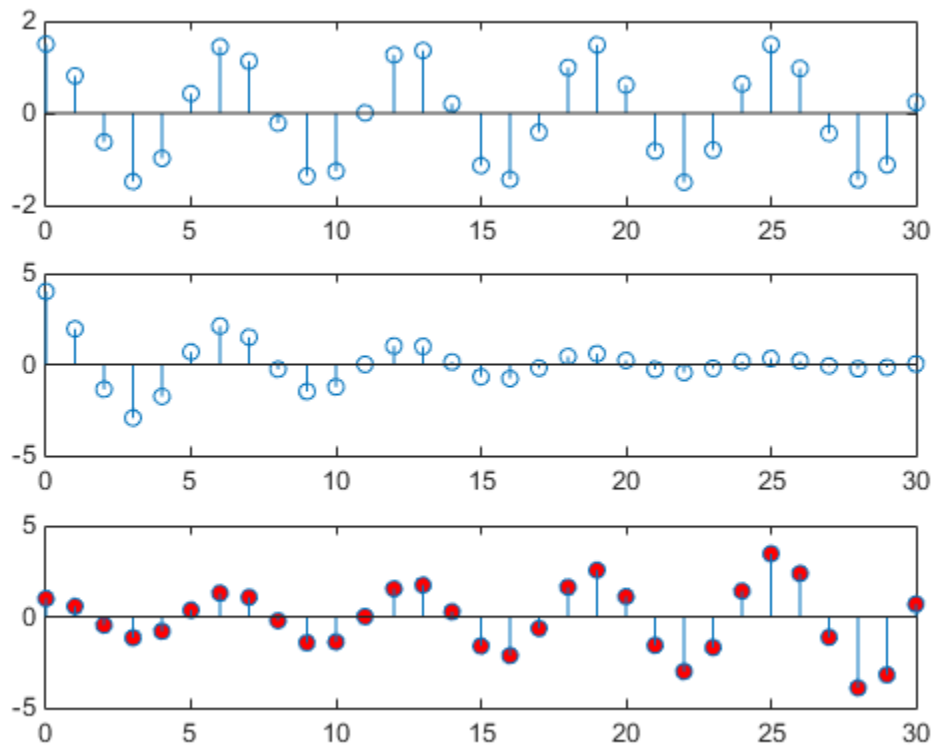

For more information on limiting the scope and depth of an object search, see `findobj` and `findall`.

Copy Objects

In this section...
"Copying Objects with copyobj" on page 18-8
"Copy Single Object to Multiple Destinations." on page 18-8
"Copying Multiple Objects" on page 18-8

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 18-10
"Handles to Deleted Objects" on page 18-11

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 18-4

Working with Graphics Objects

Graphics Object Handles

In this section...
"What You Can Do with Handles" on page 19-2
"What You Cannot Do with Handles" on page 19-2

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.

See Also

More About

- "Graphics Arrays" on page 19-8
- "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:

```
tiledlayout(2,2)
for k=1:4
    h(k) = nexttile;
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

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
```

Graphics Object Callbacks

Callbacks — Programmed Response to User Action

In this section...
"What Are Callbacks?" on page 20-2
"Window Callbacks" on page 20-2

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 20-4.

Window Callbacks

Figures have additional properties that execute callbacks with specific user actions. These additional properties are not available in MATLAB Online.

- `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 20-3
"Callback Function Syntax" on page 20-3
"Related Information" on page 20-4
"Define a Callback as a Default" on page 20-4

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 20-3.

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:

```matlab
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:

```matlab
plot(x,y,'ButtonDownFcn',{@lineCallback,'--','*'})
```

You can use an anonymous function to pass additional arguments. For example:

```matlab
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`.

```matlab
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 20-5
"How to Define a Button Down Callback" on page 20-5

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.

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 `ContextMenu` object by calling the `uicontextmenu` function with an output argument.
- 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 `ContextMenu` object to the `ContextMenu` 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 `ContextMenu` property of the line objects as they are created by the `plot` function:

```
plot(rand(1,5),'ContextMenu',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 20-3

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.

```matlab
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.

```matlab
figure('DeleteFcn',@figDelete)
figure('DeleteFcn',@figDelete)
```

Capturing Mouse Clicks

In this section...
"Properties That Control Response to Mouse Clicks" on page 20-9
"Combinations of PickablePart/HitTest Values" on page 20-9
"Passing Mouse Click Up the Hierarchy" on page 20-10

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 `ContextMenu` 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.
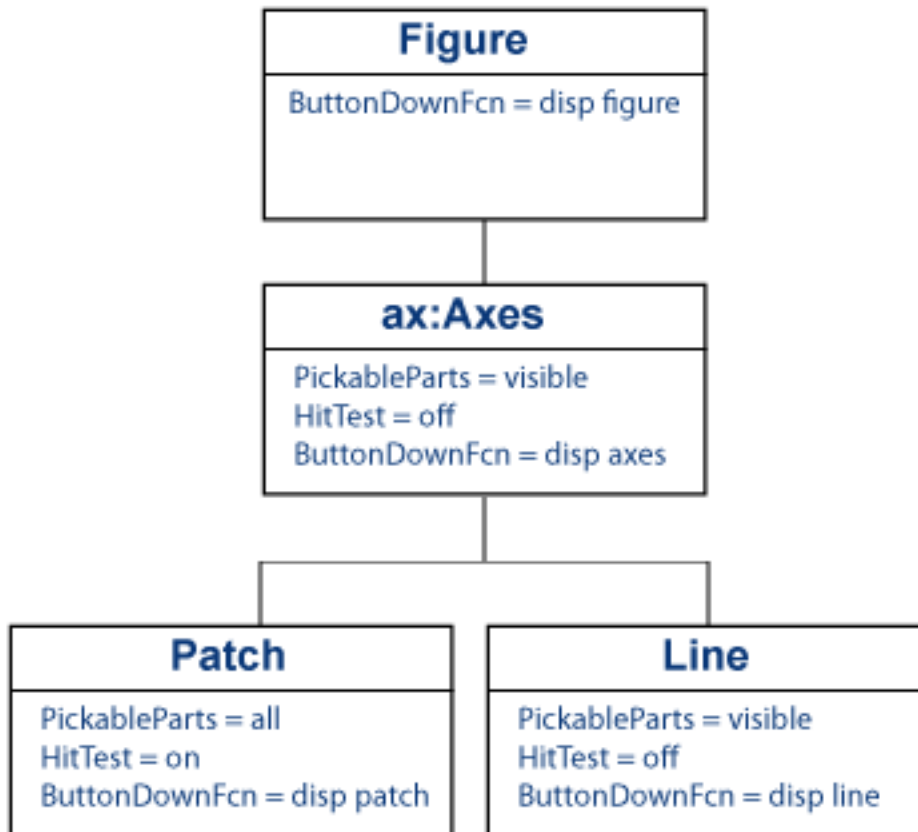
This code creates the hierarchy:

```
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.

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.
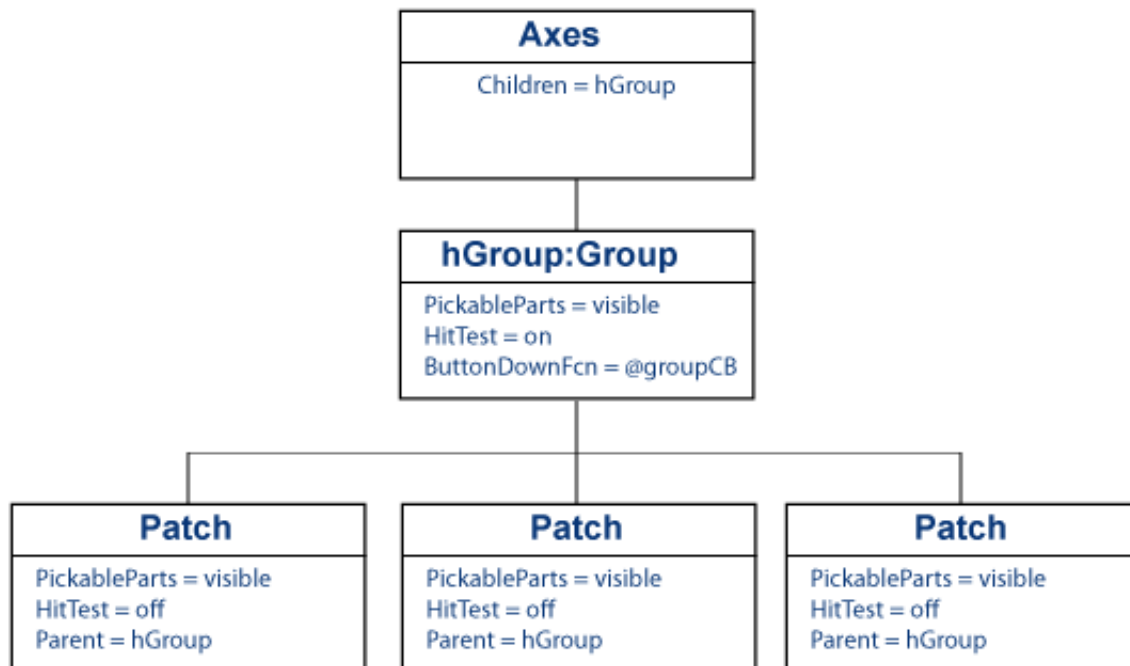
MATLAB Code

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.

```matlab
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.

```matlab
function groupCB(src,~)
    s = src.Children;
    set(s,'FaceColor',rand(1,3))
end
```

For more information on callback functions, see "Callback Definition" on page 20-3

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.

```matlab
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:

```matlab
bar(1:20,'PickableParts','none')
```

Create the context menu for the current axes:

```matlab
ax = gca;
ax.ContextMenu = axesHoldCM
```

Right-click over the bars in the graph and display the axes context menu:
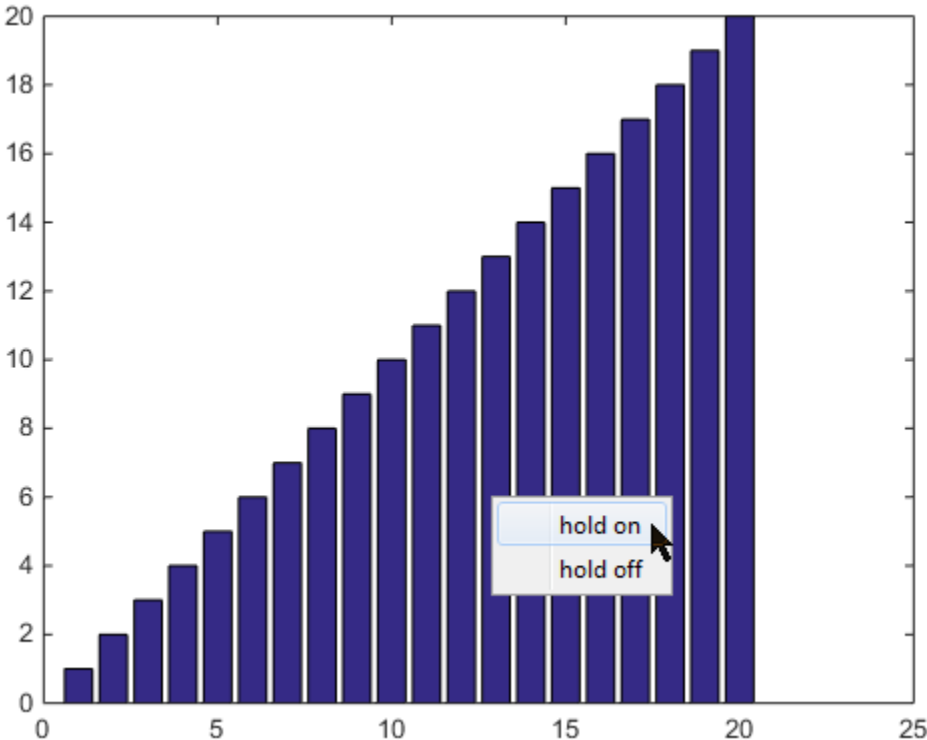
Group Objects

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 21-12. 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 21-3
"Visible and Selected Properties of Group Children" on page 21-4

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 21-9 and "Nest Transforms for Complex Movements" on page 21-12.

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

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 21-12.

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 21-8.

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_x 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 21-7.

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

See Also
eye | hgtransform | makehgtform

More About
- "Nest Transforms for Complex Movements" on page 21-12
- "Undoing Transform Operations" on page 21-8
- "Combining Transforms into One Matrix" on page 21-7

Rotate About an Arbitrary Axis

This example shows how to rotate an object about an arbitrary axis.

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
```
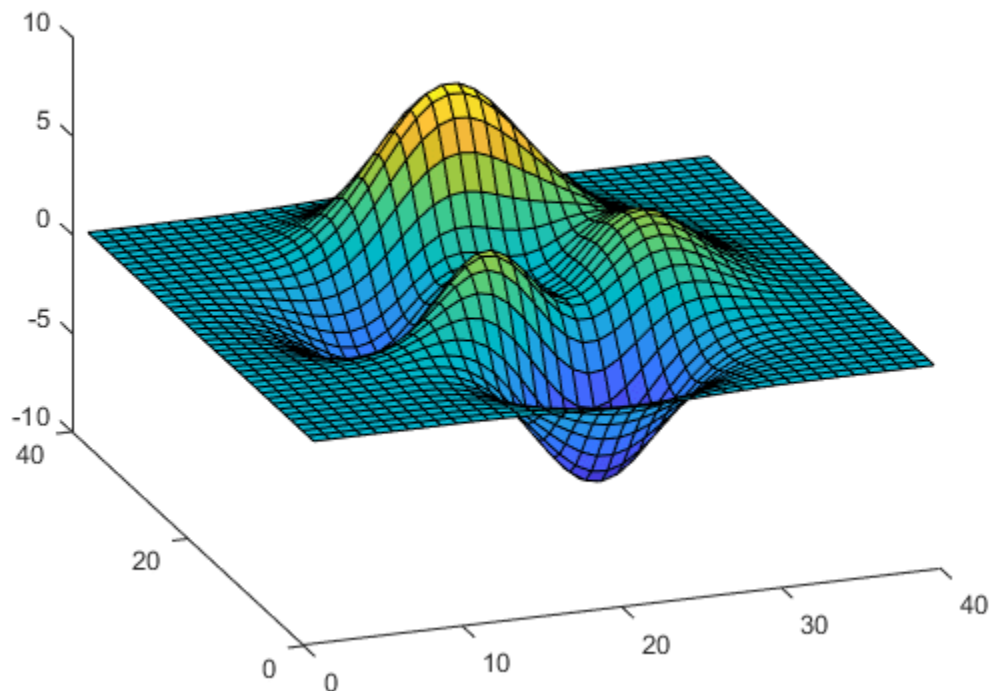

Create Transform

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;
```
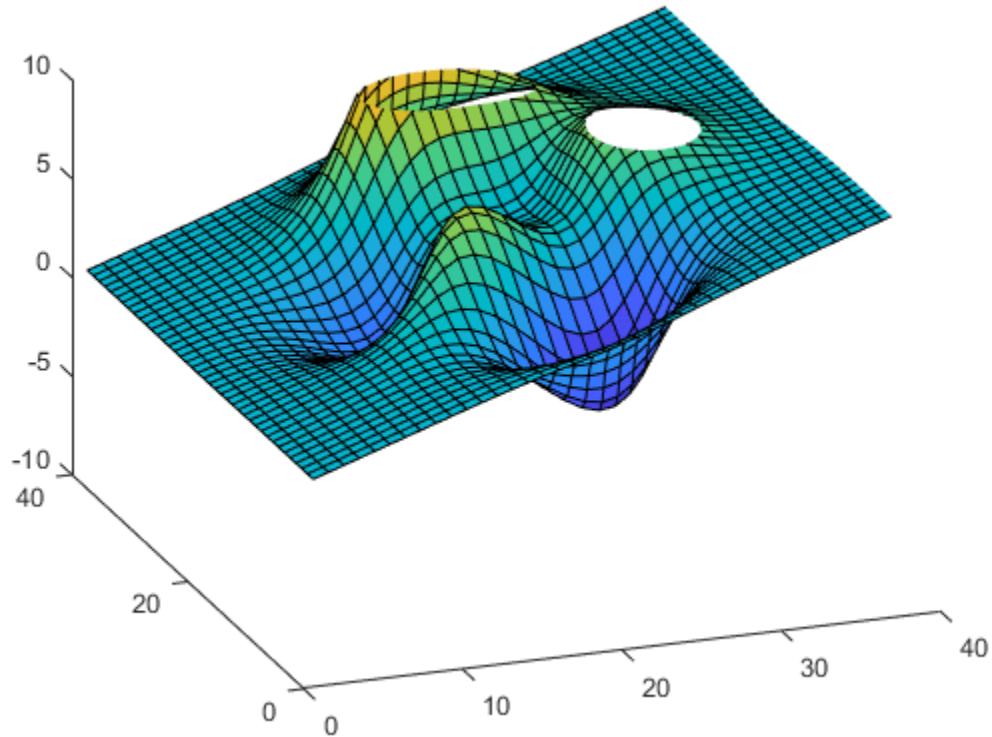

The surface rotated -15 degrees about the *y*-axis that passes through the origin.

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;
```
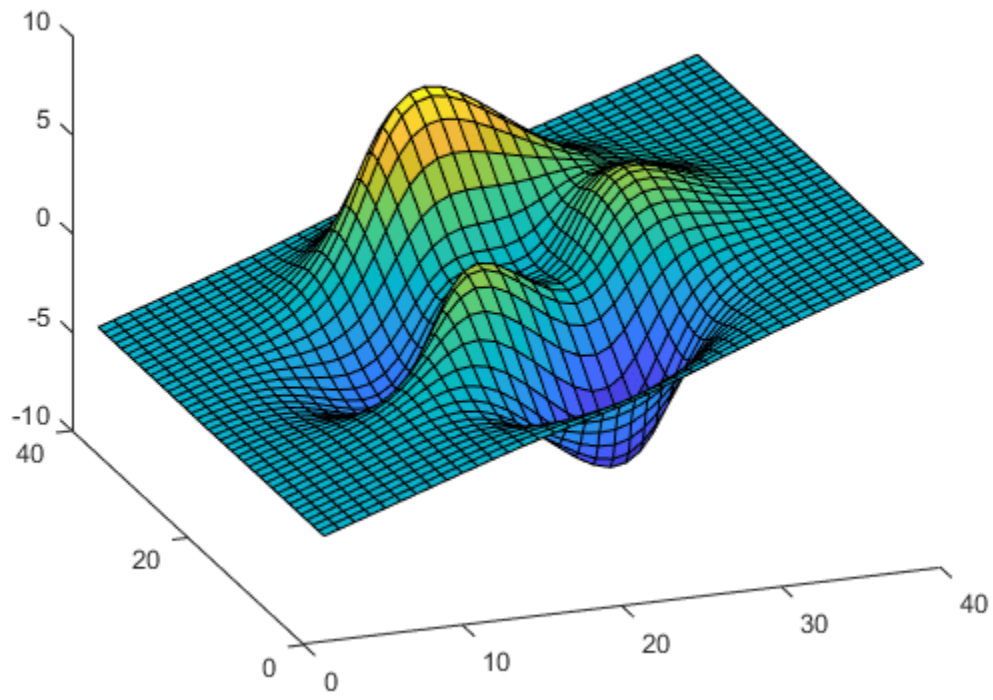
Nest Transforms for Complex Movements

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.
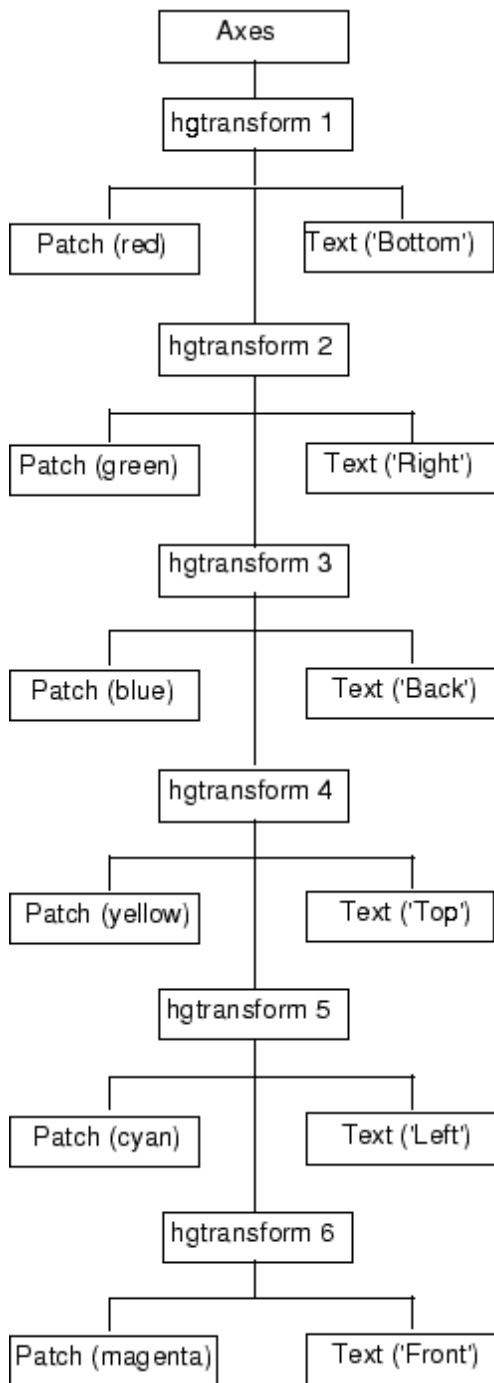
The diagram on the left represents the object hierarchy in the picture below.

Through a series of simple rotations and translations, the six squares are folded into a cube.
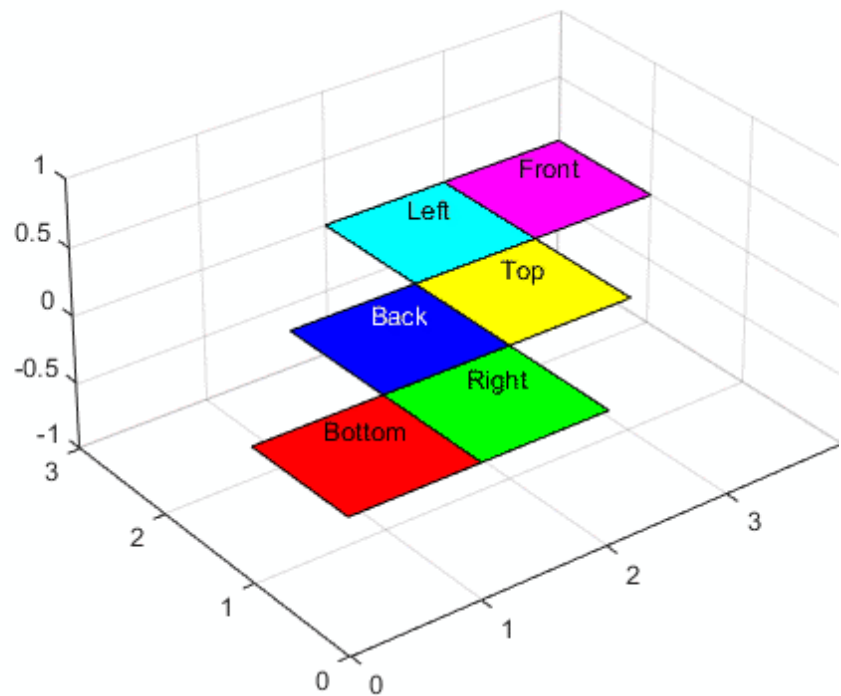
The `transform_foldbox` function implements the transform hierarchy. The `doUpdate` function renders each step. Place both functions in a file named `transform_foldbox.m` and execute `transform_foldbox`.

```
function transform_foldbox
    % Create six square and fold
    % them into a cube
```

```matlab
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]);
```

```matlab
    % 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
```

Controlling Graphics Output

Control Graph Display

In this section...
"What You Can Control" on page 22-2
"Targeting Specific Figures and Axes" on page 22-2

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 two axes, `ax1` and `ax2`.

```
tiledlayout(1,2)
ax1 = nexttile;
ax2 = nexttile;
```

Call `plot` with the axes object as the first argument:

```
plot(ax1,1:10)
```

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',ax2)
```

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 front-most 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 front-most 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 22-4
"How the NextPlot Properties Control Behavior" on page 22-4
"Control Behavior of User-Written Plotting Functions" on page 22-5

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`.

NextPlot	Figure	Axes
`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.

Figure NextPlot Property Value	newplot Function
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 dot notation to set properties.

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.

```matlab
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:

```matlab
x = 1:10;
y = peaks(10);
myPlot2D(x,y)
```
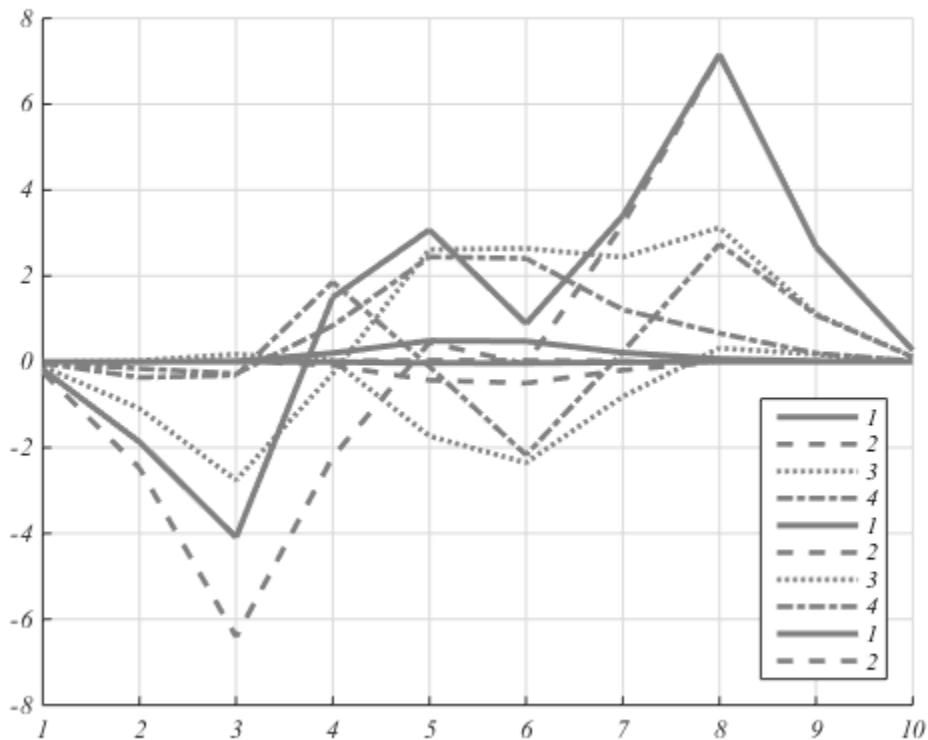
The `myPlot2D` function shows the basic structure of a user-written plotting functions:

- 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.

```matlab
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`.

```matlab
[x,y,z] = peaks(20);
myPlot3D(x,y,z)
hold on
myPlot3D(x,y)
```
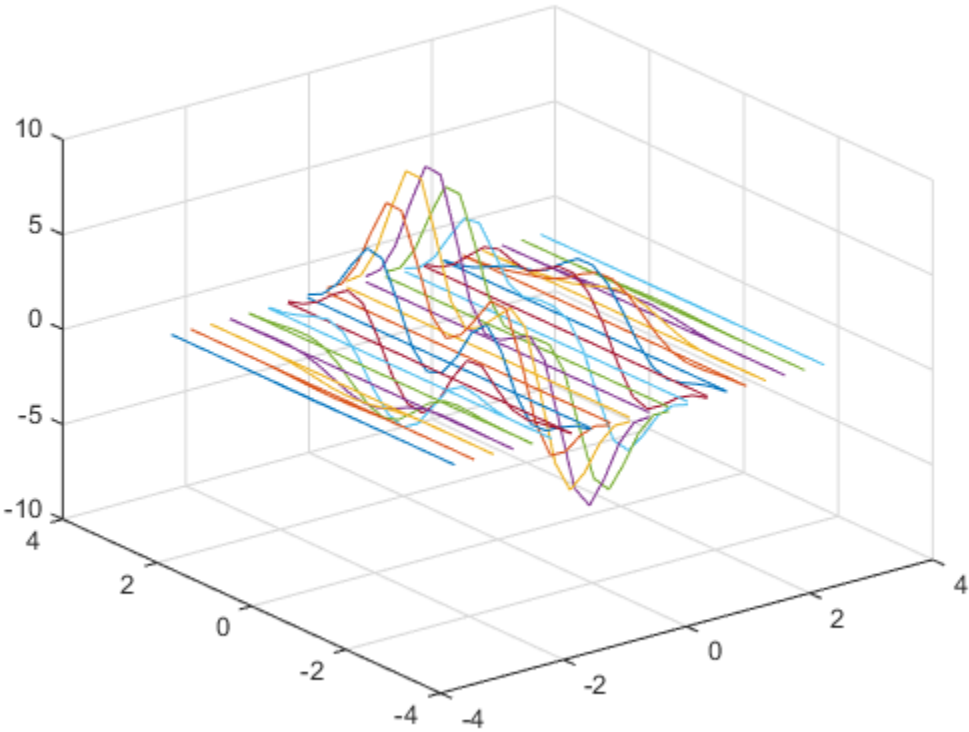
Prevent Access to Figures and Axes

In this section...

Why Prevent Access

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.

Developing Classes of Chart Objects

Chart Development Overview

Charting functions such as `plot`, `scatter`, and `bar` enable you to quickly visualize your data with basic control over aspects such as color and line style. To create custom charts, you can combine multiple graphics objects, set properties on those objects, or call additional functions. In R2019a and earlier releases, a common way to store your customization code and share it with others is to write a script or a function.

Starting in R2019b, you can create a class implementation for your charts by defining a subclass of the `ChartContainer` base class. Creating a class enables you to:

- Provide a convenient interface for your users — When users want to customize an aspect of your chart, they can set a property rather than having to modify and rerun your graphics code. Users can modify properties at the command line or inspect them in the Property Inspector.

- Encapsulate algorithms and primitive graphics objects — You implement methods that perform calculations and manage the underlying graphics objects. Organizing your code in this way allows you to hide implementation details from your users.

When you define a chart that derives from this base class, instances of your chart are members of the graphics object hierarchy. As a result, your charts are compatible with many aspects of the graphics system. For example, the `gca` and `findobj` functions can return instances of your chart.

Structure of a Chart Class

The first line of a chart class specifies the `matlab.graphics.chartcontainer.ChartContainer` class as the superclass. For example, the first line of a class called `ConfidenceChart` looks like this:

```
classdef ConfidenceChart < matlab.graphics.chartcontainer.ChartContainer
```

In addition to specifying the superclass, include the following components in your class definition.

Component	Description
Public property block on page 23-3 (recommended)	This block defines all the properties that you want your users to have access to. Together, these properties make up the user interface of your chart.
Private property block on page 23-3 (recommended)	This block stores the underlying graphics objects and other implementation details that you do not want your users to access. In this block, set these attribute values: • `Access = private` • `Transient` • `NonCopyable`
`setup` method on page 23-4 (required)	This method sets the initial state of the chart. It executes once when MATLAB constructs the object. Define this method in a protected block so that only your class can execute it.

Component	Description
`update` method on page 23-5 (required)	This method updates the underlying objects in your chart. It executes during the next `drawnow` execution, after the user changes one or more property values. Define this method in the same protected block as the `setup` method.

Implicit Constructor Method

You do not have to write a constructor method for your class because a constructor is inherited from the `ChartContainer` base class. The constructor accepts optional input arguments: a parent container and any number of name-value pair arguments for setting properties on the chart. For example, if you define a class called `ConfidenceChart` that has the public properties `XData` and `YData`, you can create an instance of your class using either of these commands:

```
c = ConfidenceChart(gcf,'XData',[1 2 3],'YData',[4 5 6])
c = ConfidenceChart('XData',[1 2 3],'YData',[4 5 6])
```

If you want to provide an interface that accepts input arguments in the same way as a typical function does, you can define a custom constructor method. See "Write Constructors for Chart Classes" on page 23-9 for more information.

Public and Private Property Blocks

Divide your class properties between at least two blocks:

- A public block for storing the components of the user-facing interface
- A private block for storing the implementation details that you want to hide

The properties that go in the public block store the input values provided by the user. For example, a chart that displays a line might store the *x*- and *y*-coordinate vectors in two public properties. Since the property name-value pair arguments are optional inputs to the implicit constructor method, the recommended approach is to initialize the public properties to default values. If you define public properties that store coordinate values, initializing them to `NaN` values or empty arrays constructs an empty chart if the user calls the constructor without any inputs.

The properties that go in the private block store the underlying graphics objects that make up your chart, in addition to any calculated values you want to store. Eventually, your class will use the data in the public properties to configure the underlying objects. By including the `Transient` and `NonCopyable` attributes for the private block, you avoid storing redundant information if the user copies or saves an instance of the chart.

For example, here are the property blocks for a chart that displays a `Line` object and a `Patch` object. The public property block stores values that the user can control: the *x*- and *y*-coordinates of the line, a confidence margin value, a marker symbol, and a color value. The private property block stores the `Line` and `Patch` objects.

```
properties
    XData = NaN
    YData = NaN
    ConfidenceMargin = 0.15
    MarkerSymbol = 'o'
    Color = [1 0 0]
```

```
end

properties(Access = private,Transient,NonCopyable)
    LineObject
    PatchObject
end
```

Setup Method

The `setup` method executes once when MATLAB constructs the chart object. Any property values passed as name-value pair arguments to the constructor method are assigned after this method executes.

Use the `setup` method to:

- Call plotting functions to create the primitive graphics objects you want to use in the chart.
- Store the primitive objects returned by the plotting functions as private properties on the chart object.
- Configure the primitive graphics objects.
- Configure the axes.

Many graphics functions have an optional input argument for specifying the target axes object. These functions include plotting functions (such as `plot`, `scatter`, and `bar`) and functions that modify the axes (such as `hold`, `grid`, and `title`). When you call these types of functions from within a class method, you must specify the target axes object. You can access the axes object by calling the `getAxes` method. This method returns the axes object, or it creates a Cartesian axes object if the chart does not already contain an axes object.

Caution Calling plotting functions, or functions that modify the axes, without specifying the target axes might produce unexpected results.

When you call plotting functions in the `setup` method, specify temporary values (such as `NaN`) for the coordinate data. Also, specify temporary values for other arguments that correspond to public properties of your class. Doing so avoids setting the same property values in both the `setup` and the `update` methods.

If you want to display multiple primitive objects in the axes, call the `hold` function between plotting commands. Set the hold state back to `'off'` after your last plotting command.

For example, consider a chart that displays a line and a patch. It has these properties:

- Two public properties called `XData` and `YData` for storing the *x*- and *y*-coordinates of the line
- Two private properties called `LineObject` and `PatchObject`

The `setup` method gets the axes object by calling the `getAxes` method. Then it calls the `patch` function and stores the output in the `PatchObject` property. The next line of code set the hold state of the axes to `'on'` before calling the `plot` function to create the `LineObject` property. The last line of code sets the axes hold state back to `'off'`.

```
function setup(obj)
    % Get the axes
    ax = getAxes(obj);
```

```matlab
    % Create Patch and Line objects
    obj.PatchObject = patch(ax,NaN,NaN,'r','FaceAlpha',0.2,...
        'EdgeColor','none');
    hold(ax,'on')
    obj.LineObject = plot(ax,NaN,NaN);

    % Turn hold state off
    hold(ax,'off')
end
```

Update Method

When the user changes one or more property values on the chart object, MATLAB marks the chart object for updating. The `update` method runs for the first time after the `setup` method runs. Then it runs the next time `drawnow` executes. The `drawnow` function automatically executes periodically, based on the state of the graphics environment in the user's MATLAB session. Thus, there might be a delay between changing property values and seeing the results of those changes.

Use the `update` method to reconfigure the underlying graphics objects in your chart based on the new values of the public properties. Typically, this method does not distinguish which of the public properties changed. It reconfigures all aspects of the underlying graphics objects that depend on the public properties.

For example, consider a chart that has these properties:

- Two public properties called `XData` and `Color`
- Two private properties called `LineObject` and `PatchObject`

The `update` method updates the `XData` and `Color` properties of the `Line` and `Patch` objects.

```matlab
function update(obj)

    % Update XData of line object
    obj.LineObject.XData = obj.XData;

    % Update patch XData
    x = obj.XData;
    obj.PatchObject.XData = [x x(end:-1:1)];

    % Update line object colors
    obj.LineObject.Color = obj.Color;
    obj.PatchObject.FaceColor = obj.Color;

end
```

Example: Confidence Bounds Chart

This example shows how to create a chart for plotting a line with confidence bounds. Create a class definition file named `ConfidenceChart.m` in a folder that is on the MATLAB path. Define the class by following these steps.

Step	Implementation
Derive from the `ChartContainer` base class.	```classdef ConfidenceChart < matlab.graphics.chartcontainer.ChartContainer```
Define public properties.	```properties XData = NaN YData = NaN ConfidenceMargin = 0.15 MarkerSymbol = 'o' Color = [1 0 0] end```
Define private properties.	```properties(Access = private,Transient,NonCopyable) LineObject PatchObject end```
Implement the `setup` method. In this case, call the `plot` and `patch` functions to create the `Patch` and `Line` objects respectively. Store those objects in the corresponding private properties. Turn the hold state of the axes back to `'off'` before exiting the method.	```methods(Access = protected) function setup(obj) % get the axes ax = getAxes(obj); % Create Patch and Line objects obj.PatchObject = patch(ax,NaN,NaN,'r','FaceAlpha',0.2,... 'EdgeColor','none'); hold(ax,'on') obj.LineObject = plot(ax,NaN,NaN); % Turn hold state off hold(ax,'off') end```
Implement the `update` method. In this case, update the *x*- and *y*-coordinates, color, and marker symbol of the underlying objects.	``` function update(obj) % Update XData and YData of Line obj.LineObject.XData = obj.XData; obj.LineObject.YData = obj.YData; % Update patch XData and YData x = obj.XData; obj.PatchObject.XData = [x x(end:-1:1)]; y = obj.YData; c = obj.ConfidenceMargin; obj.PatchObject.YData = [y+c y(end:-1:1)-c]; % Update colors obj.LineObject.Color = obj.Color; obj.PatchObject.FaceColor = obj.Color; % Update markers obj.LineObject.Marker = obj.MarkerSymbol; end end end```

Next, create an instance of the chart by calling the implicit constructor method with a few of the public properties:

```
x = 0:0.2:10;
y = besselj(0,x);
c = ConfidenceChart('XData',x,'YData',y,'ConfidenceMargin',0.15);
```
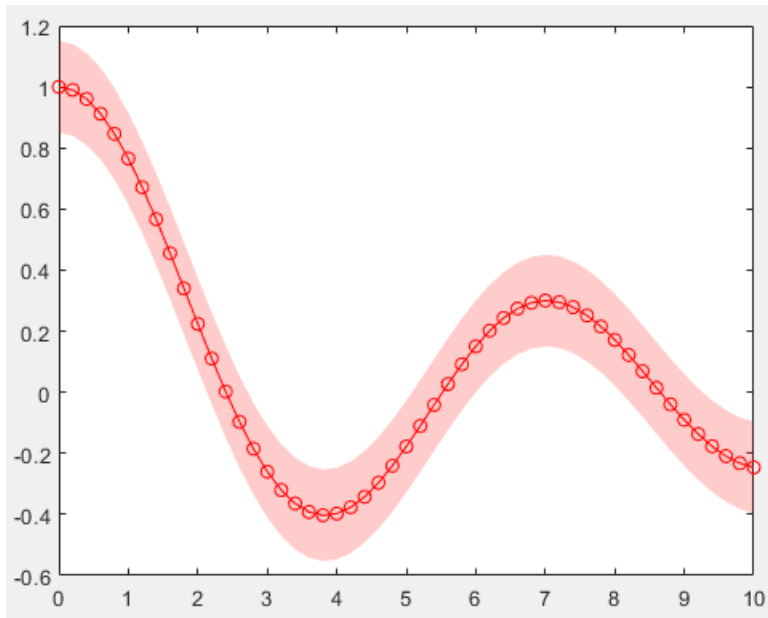

Change the color.

```
c.Color = [0 0 1];
```
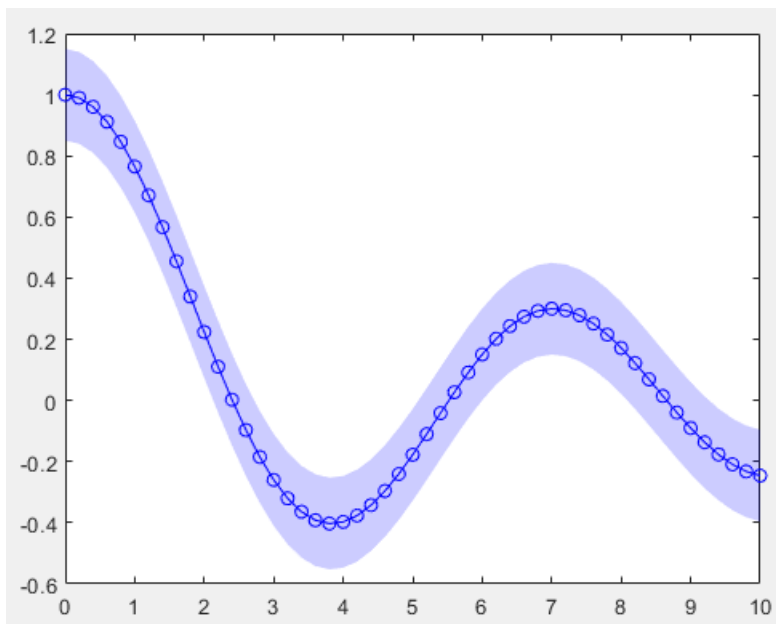
Support Common Graphics Features

By default, instances of your charts support much of the functionality that is common to all MATLAB charts. For example, the `gca` and `findobj` functions can return instances of your chart. You can also pass instances of your chart to the `set` and `get` functions, and you can configure the properties of the chart in the Property Inspector.

The features described in this table are supported only if you enable them for your chart.

Feature	Description	More Information
Legend	Enable the `legend` function and the legend tool in the figure toolbar.	`matlab.graphics.chartcontainer.mixin.Legend`
Colorbar	Enable the `colorbar` function and the colorbar tool in the figure toolbar.	`matlab.graphics.chartcontainer.mixin.Colorbar`
Different types of axes, or multiple axes	Display one or more Cartesian, polar, or geographic plots.	"Develop Charts With Polar Axes, Geographic Axes, or Multiple Axes" on page 23-13
functions	Enable functions that set properties on the axes, such as `title`, `xlim`, and `ylim` functions.	"Enabling Convenience Functions for Setting Axes Properties" on page 23-25
Saving and loading	Store changes after users interact with the chart, so that they can save the chart and restore its state when they load it back into MATLAB.	"Saving and Loading Instances of Chart Classes" on page 23-31

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

Functions
`patch` | `plot`

Properties
Line | Patch

More About

- "Class Syntax Guide"
- "Develop Charts With Polar Axes, Geographic Axes, or Multiple Axes" on page 23-13

Write Constructors for Chart Classes

When you develop a chart as a subclass of the `ChartContainer` base class, the base class provides a default constructor that accepts optional name-value pair arguments for setting chart properties. For example, this command creates an instance of a class called `ConfidenceChart`.

```
ConfidenceChart('XData',x,'YData',y,'ConfidenceMargin',0.15,'Color',[1 0 0])
```

By writing a custom constructor method, you can provide an interface that accepts individual argument values and optional name-value pair arguments. For example, you can design a custom constructor to change the calling syntax for `ConfidenceChart` so that both of these commands are valid ways to create the chart:

```
ConfidenceChart(x,y,0.15)
ConfidenceChart(x,y,0.15,'Color',[1 0 0])
```

When you write the constructor method:

- Specify the input arguments you want to support in the function declaration. Include `varargin` as the last input argument to capture any property name-value pair arguments that the user specifies.
- Call the `ChartContainer` constructor before all other references to the chart object.

For example, the following constructor method for the `ConfidenceChart` class performs these tasks:

- Checks the number of input arguments and returns an error if the number is less than three.
- Converts the `x`, `y`, and `margin` values to the name-value pair arguments that the `ChartContainer` constructor accepts, and stores the results in `args`.
- Appends any user-specified name-value pair arguments to the end of `args`.
- Passes `args` to the `ChartContainer` constructor method.

```
methods
    function obj = ConfidenceChart(x,y,margin,varargin)
        % Check for at least three inputs
        if nargin < 3
            error('Not enough inputs');
        end

        % Convert x, y, and margin into name-value pairs
        args = {'XData', x, 'YData', y, 'ConfidenceMargin', margin};

        % Combine args with user-provided name-value pairs
        args = [args varargin];

        % Call superclass constructor method
        obj@matlab.graphics.chartcontainer.ChartContainer(args{:});
    end
end
```

Example: Confidence Bounds Chart with Custom Constructor

This example shows how to develop a chart that has a custom constructor that accepts single-value input arguments and optional name-value pair arguments. The chart plots a line with markers and a surrounding confidence margin.

Create a program file named `ConfidenceChart.m` in a folder that is on the MATLAB path. Define the class by following these steps.

Step	Implementation
Derive from the `ChartContainer` base class.	```classdef ConfidenceChart < matlab.graphics.chartcontainer.ChartContainer```
| Define public properties. | ```matlab
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        ConfidenceMargin (1,1) double = 0.15
        MarkerSymbol (1,:) char = 'o'
        Color (1,3) double {mustBeGreaterThanOrEqual(Color,0),...
            mustBeLessThanOrEqual(Color,1)} = [1 0 0]
    end
``` |
| Define private properties. | ```matlab
    properties(Access = private,Transient,NonCopyable)
        LineObject (1,1) matlab.graphics.chart.primitive.Line
        PatchObject (1,1) matlab.graphics.primitive.Patch
    end
``` |
| Implement the custom constructor method that accepts the `x`, `y`, and `margin` values and optional property name-value pair arguments. | ```matlab
    methods
        function obj = ConfidenceChart(x,y,margin,varargin)
            % Check for at least three inputs
            if nargin < 3
                error('Not enough inputs');
            end

            % Convert x, y, and margin into name-value pairs
            args = {'XData', x, 'YData', y, 'ConfidenceMargin', margin};

            % Combine args with user-provided name-value pairs.
            args = [args varargin];

            % Call superclass constructor method
            obj@matlab.graphics.chartcontainer.ChartContainer(args{:});
        end
    end
``` |
| Implement the `setup` method. | ```matlab
    methods(Access = protected)
        function setup(obj)
            % get the axes
            ax = getAxes(obj);

            % Create Patch and objects
            obj.PatchObject = patch(ax,NaN,NaN,'r','FaceAlpha',0.2,...
                'EdgeColor','none');
            hold(ax,'on')
            obj.LineObject = plot(ax,NaN,NaN);
            hold(ax,'off')
        end
``` |

Step	Implementation
| Implement the `update` method. | ```matlab
        function update(obj)
            % Update XData and YData of Line
            obj.LineObject.XData = obj.XData;
            obj.LineObject.YData = obj.YData;

            % Update patch XData and YData
            x = obj.XData;
            obj.PatchObject.XData = [x x(end:-1:1)];
            y = obj.YData;
            c = obj.ConfidenceMargin;
            obj.PatchObject.YData = [y+c y(end:-1:1)-c];

            % Update colors
            obj.LineObject.Color = obj.Color;
            obj.PatchObject.FaceColor = obj.Color;

            % Update markers
            obj.LineObject.Marker = obj.MarkerSymbol;
        end
    end
end
``` |

Next, create an instance of a `ConfidenceChart`. Specify the *x*- and *y*-coordinates, the margin value, and a marker symbol.

```matlab
x = 0:0.2:10;
y = besselj(0,x);
ConfidenceChart(x,y,0.20,'MarkerSymbol','>');
```
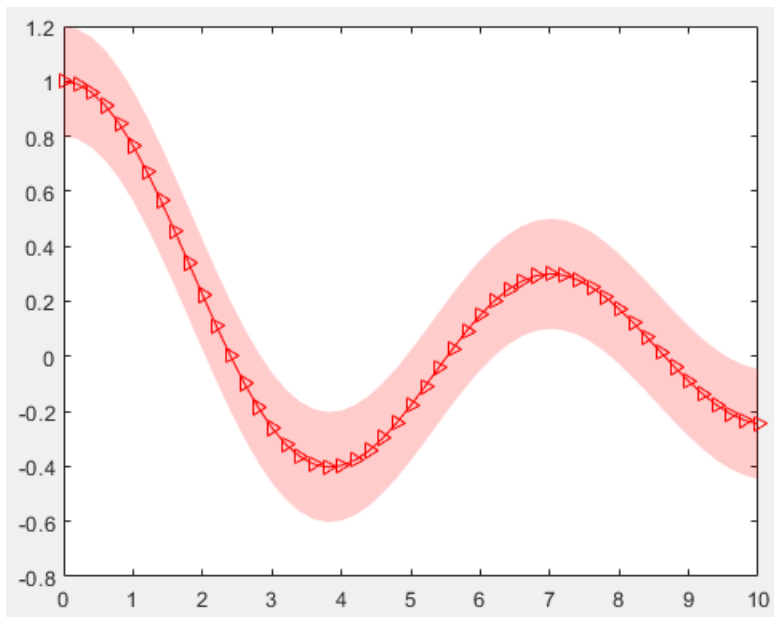

See Also

Classes
matlab.graphics.chartcontainer.ChartContainer

Functions
patch | plot

Properties
Line | Patch

More About

- "Class Constructor Methods"
- "Call Superclass Methods on Subclass Objects"
- "Chart Development Overview" on page 23-2

Develop Charts With Polar Axes, Geographic Axes, or Multiple Axes

For charts you develop as a subclass of the `ChartContainer` base class, the `getAxes` method provides a way to support a single Cartesian axes object. If you want to support polar axes, geographic axes, or multiple axes, you must create and configure the axes as children of a `TiledChartLayout` object, which is stored in the chart object.

Create a Single Polar or Geographic Axes Object

To include a single polar axes or geographic axes object in your chart:

1 Define a private property to store the axes.
2 In the `setup` method:

- Call the `getLayout` method to get the `TiledChartLayout` object.
- Call the `polaraxes` or `geoaxes` function to create the axes, and specify the `TiledChartLayout` object as the parent object.

For example, here is a basic class that contains a polar axes object.

```
classdef SimplePolar < matlab.graphics.chartcontainer.ChartContainer
    properties(Access = private,Transient,NonCopyable)
        PolarAx matlab.graphics.axis.PolarAxes
    end

    methods(Access = protected)
        function setup(obj)
            % Get the layout and create the axes
            tcl = getLayout(obj);
            obj.PolarAx = polaraxes(tcl);

            % Other setup code
            % ...
        end
        function update(obj)
            % Update the chart
            % ...
        end
    end
end
```

Create a Tiling of Multiple Axes Objects

To display a tiling of multiple axes:

1 Define private properties that store the axes objects. You can also define one property that stores an array of axes objects.
2 In the `setup` method:

- Call the `getLayout` method to get the `TiledChartLayout` object.
- Set the `GridSize` property of the `TiledChartLayout` object so that it has at least one tile for each of the axes.

- Call the `axes`, `polaraxes`, or `geoaxes` function to create the axes objects, and specify the `TiledChartLayout` object as the parent object.
- Move each of the axes to the desired tile by setting the `Layout` property on each axes object. By default, the axes appear in the first tile.

For example, here is a basic class that contains two Cartesian axes:

```matlab
classdef TwoAxesChart < matlab.graphics.chartcontainer.ChartContainer
    properties(Access = private,Transient,NonCopyable)
        Ax1 matlab.graphics.axis.Axes
        Ax2 matlab.graphics.axis.Axes
    end

    methods(Access = protected)
        function setup(obj)
            % Get the layout and set the grid size
            tcl = getLayout(obj);
            tcl.GridSize = [2 1];

            % Create the axes
            obj.Ax1 = axes(tcl);
            obj.Ax2 = axes(tcl);

            % Move the second axes to the second tile
            obj.Ax2.Layout.Tile = 2;
        end
        function update(obj)
            % Update the chart
            % ...
        end
    end
end
```

Example: Chart Containing Geographic and Cartesian Axes

This example shows how to define a class of charts for visualizing geographic and categorical data using two axes. The left axes contains a map showing the locations of several cellular towers. The right axes shows the distribution of the towers by category.

The following `TowerChart` class definition shows how to:

- Define a public property called `TowerData` that stores a table.
- Validate the contents of the table using a local function called `mustHaveRequiredVariables`.
- Define two private properties called `MapAxes` and `HistogramAxes` that store the axes.
- Implement a `setup` method that gets the `TiledChartLayout` object, specifies the grid size of the layout, and positions the axes.

To define the class, copy this code into the editor and save it with the name `TowerChart.m` in a writable folder.

```matlab
classdef TowerChart < matlab.graphics.chartcontainer.ChartContainer
    properties
        TowerData (:,:) table {mustHaveRequiredVariables} = table([],...
```

```matlab
                [],[],'VariableNames',{'STRUCTYPE','Latitude','Longitude'})
        end

        properties (Access = private,Transient,NonCopyable)
            MapAxes matlab.graphics.axis.GeographicAxes
            HistogramAxes matlab.graphics.axis.Axes
            ScatterObject matlab.graphics.chart.primitive.Scatter
            HistogramObject matlab.graphics.chart.primitive.categorical.Histogram
        end

        methods (Access = protected)
            function setup(obj)
                % Configure layout and create axes
                tcl = getLayout(obj);
                tcl.GridSize = [1 2];
                obj.MapAxes = geoaxes(tcl);
                obj.HistogramAxes = axes(tcl);

                % Move histogram axes to second tile
                obj.HistogramAxes.Layout.Tile = 2;

                % Create Scatter and Histogram objects
                obj.ScatterObject = geoscatter(obj.MapAxes,NaN,NaN,'.');
                obj.HistogramObject = histogram(obj.HistogramAxes,categorical.empty,...
                    'Orientation','horizontal');

                % Add titles to the axes
                title(obj.MapAxes,"Tower Locations")
                title(obj.HistogramAxes,"Tower Types")
                xlabel(obj.HistogramAxes,"Number of Towers")
            end

            function update(obj)
                % Update Scatter object
                obj.ScatterObject.LatitudeData = obj.TowerData.Latitude;
                obj.ScatterObject.LongitudeData = obj.TowerData.Longitude;

                % Get tower types from STRUCTYPE table variable
                towertypes = obj.TowerData.STRUCTYPE;

                % Check for empty towertypes before updating histogram
                if ~isempty(towertypes)
                    obj.HistogramObject.Data = towertypes;
                    obj.HistogramObject.Categories = categories(towertypes);
                else
                    obj.HistogramObject.Data = categorical.empty;
                end
            end
        end
    end

function mustHaveRequiredVariables(tbl)
% Return error if table does not have required variables
assert(all(ismember({'STRUCTYPE','Latitude','Longitude'},...
    tbl.Properties.VariableNames)),...
    'MATLAB:TowerChart:InvalidTable',...
    'Table must have STRUCTYPE, Latitude, and Longitude variables.');
```

23-15

```
    end
```

After saving the class file, load the table stored in `cellularTowers.mat`. Then create an instance of the chart by passing the table to the `TowerChart` constructor method as a name-value pair argument.

```
load cellularTowers.mat
TowerChart('TowerData',cellularTowers);
```
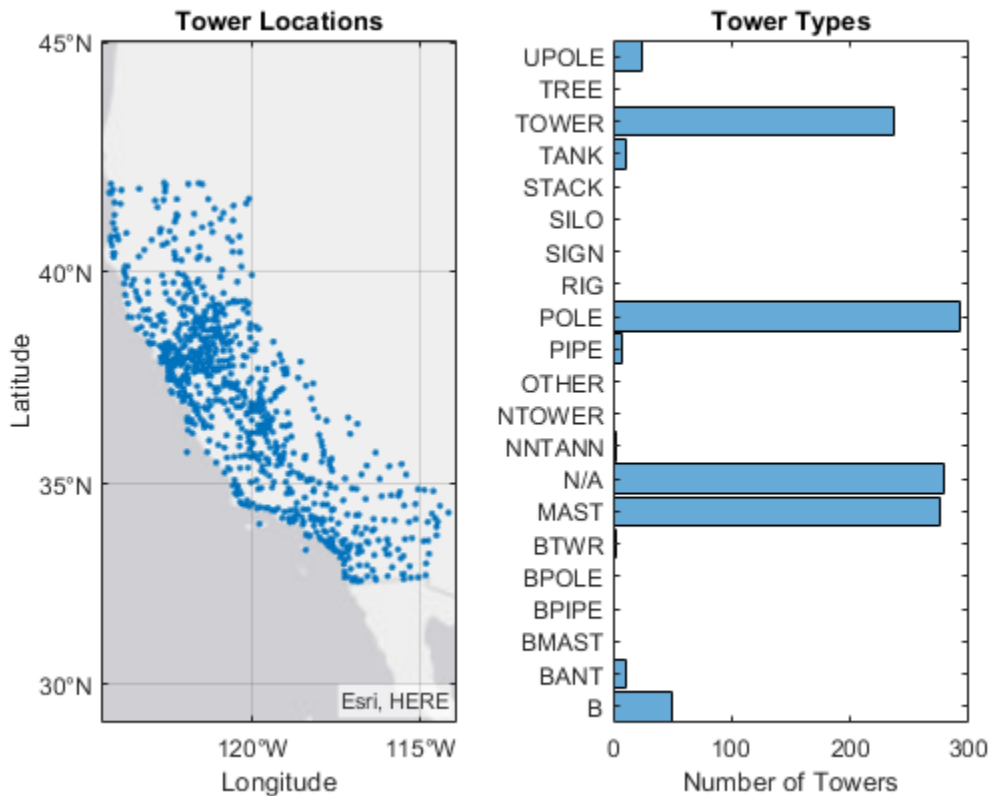

See Also

Functions
getLayout

Classes
matlab.graphics.chartcontainer.ChartContainer

Properties
TiledChartLayout Properties

More About

- "Chart Development Overview" on page 23-2

Managing Properties of Chart Classes

When you develop a custom chart as a subclass of the `ChartContainer` base class, you can use certain techniques to make your code more robust, efficient, and tailored to the needs of your users. These techniques focus on how you define and manage the properties of your class. Use any that are helpful for the type of visualization you want to create and the user experience you want to provide.

- Initialize property values on page 23-17 — Set the default state of the chart in case your users call the implicit constructor without any input arguments.

- Validate property values on page 23-17 — Ensure that the values are valid before using them to perform a calculation or configure one of the underlying graphics objects in your chart.

- Customize the property display on page 23-18 — Provide a customized list of properties when a user references the chart object without semicolon.

- Optimize the `update` method on page 23-19 — Improve the performance of the `update` method when only a subset of your properties are used in a time-consuming calculation.

Initialize Property Values

Assign default values for all of the public properties of your class. Doing so configures a valid chart if the user omits some of the name-value pair arguments when they call the constructor method.

For properties that store coordinate data, set the initial values to `NaN` values or empty arrays so that the default chart is empty when the user does not specify the coordinates. Choose the default coordinates according to the requirements of the plotting functions you plan to call in your class methods. To learn about the requirements, see the documentation for the plotting functions you plan to use.

Validate Property Values

A good practice is to verify the values of your class properties before your code uses those values. A convenient way to do this is to validate the size and class of the properties as you define them. For example, this property block validates the size and class of four properties.

```
properties
        IsoValue (1,1) double = 0.5
        Enclose {mustBeMember(Enclose,{'above','below'})} = 'below'
        CapVisible (1,1) matlab.lang.OnOffSwitchState = 'on'
        Color (1,3) double {mustBeGreaterThanOrEqual(Color,0),...
            mustBeLessThanOrEqual(Color,1)} = [.2 .5 .8]
end
```

- `IsoValue` must be a 1-by-1 array of class `double`.

- `Enclose` must have a value of either `'above'` or `'below'`.

- `CapVisible` must be a 1-by-1 array of class `matlab.lang.OnOffSwitchState`.

- `Color` must be a 1-by-3 array of class `double`, where each value is in the range `[0,1]`.

You can also validate properties that store the underlying graphics objects in your chart. To determine the class name of an object, call the corresponding plotting function at the command line, and then call the `class` function to get the class name. For example, if you plan to call the `patch` function in your `setup` method, call the `patch` function at the command line with an output

argument (the input arguments do not matter). Then pass the output to the `class` function to get its class name.

```
x = patch(NaN,NaN,NaN);
class(x)
```

```
ans =

    'matlab.graphics.primitive.Patch'
```

Use the output of the `class` function to validate the class for the corresponding property in your class. For example, each of the following properties stores a `Patch` object.

```
properties (Access = private,Transient,NonCopyable)
    IsoPatch (1,1) matlab.graphics.primitive.Patch
    CapPatch (1,1) matlab.graphics.primitive.Patch
end
```

Occasionally, you might want to define a property that can store different shapes and classes of values. For example, if you define a property that can store a character vector, cell array of character vectors, or string array, omit the size and class validation or use a custom property validation method.

For more information about validating properties, see "Validate Property Values".

Customize the Property Display

One of the benefits of defining your chart as a subclass of the `ChartContainer` base class is that it also inherits from the `matlab.mixin.CustomDisplay` class. Thus, you can customize the list of properties MATLAB displays in the Command Window when you reference the chart without a semicolon. To customize the property display, overload the `getPropertyGroups` method. Within that method, you can customize which properties are listed and the order of the list. For example, consider an `IsoSurfCapChart` class that has the following public properties.

```
properties
        IsoValue (1,1) double = 0.5
        Enclose {mustBeMember(Enclose,{'above','below'})} = 'below'
        CapVisible (1,1) matlab.lang.OnOffSwitchState = 'on'
        Color (1,3) double {mustBeGreaterThanOrEqual(Color,0),...
            mustBeLessThanOrEqual(Color,1)} = [.2 .5 .8]
end
```

The following `getPropertyGroups` method specifies the scalar object property list as `Color`, `IsoValue`, `Enclose`, and `CapVisible`.

```
function propgrp = getPropertyGroups(obj)
    if ~isscalar(obj)
        % List for array of objects
        propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
    else
        % List for scalar object
        propList = {'Color','IsoValue','Enclose','CapVisible'};
        propgrp = matlab.mixin.util.PropertyGroup(propList);
    end
end
```

When the user references an instance of this chart without a semicolon, MATLAB displays the customized list.

```
c = IsoSurfCapChart

c =

  IsoSurfCapChart with properties:

        Color: [0.2000 0.5000 0.8000]
      IsoValue: 0.5000
       Enclose: 'below'
    CapVisible: on
```

For more information about customizing the property display, see "Customize Property Display".

Optimize the update Method

In most cases, the `update` method of your class reconfigures all the relevant aspects of your chart that depend on the public properties. Sometimes, the reconfiguration involves an expensive calculation that is time consuming. If the calculation involves only a subset of the properties, you can design your class to execute that code only when it is necessary.

One way to optimize the `update` method is to add these components to your class:

* Define a private property called `ExpensivePropChanged` that accepts a `logical` value. This property indicates whether any of the properties used in the expensive calculation have changed.
* Write a `set` method for each property involved in the expensive calculation. Within each `set` method, set the `ExpensivePropChanged` property to `true`.
* Write a protected method that performs the expensive calculation.
* Write a conditional statement in the `update` method that checks the value of `ExpensivePropChanged`. If the value is `true`, execute the method that performs the expensive calculation.

The following code provides a simplified implementation of this design.

```
classdef OptimizedChart < matlab.graphics.chartcontainer.ChartContainer

    properties
        Prop1
        Prop2
    end
    properties(Access=private,Transient,NonCopyable)
        ExpensivePropChanged (1,1) logical = true
    end

    methods(Access = protected)
        function setup(obj)
            % Configure chart
            % ...
        end
        function update( obj )
            % Perform expensive computation if needed
            if obj.ExpensivePropChanged
                doExpensiveCalculation(obj);
                obj.ExpensivePropChanged = false;
            end
```

```
            % Update other aspects of chart
            % ...
        end
        function doExpensiveCalculation(obj)
            % Expensive code
            % ...
        end
    end

    methods
        function set.Prop2(obj,val)
            obj.Prop2 = val;
            obj.ExpensivePropChanged = true;
        end
    end
end
```

In this case, `Prop2` is involved in the expensive calculation. The `set.Prop2` method sets the value of `Prop2`, and then it sets `ExpensivePropChanged` to `true`. Thus, the next time the `update` method runs, it calls `doExpensiveCalculation` only if `ExpensivePropChanged` is `true`. Then the `update` method continues to update other aspects of the chart.

Example: Optimized Isosurface Chart with Customized Property Display

Define an `IsoSurfCapChart` class for displaying an `isosurface` with the associated `isocaps`. Include the following features:

- Properties that use size and class validation
- A customized property display
- An optimized `update` method that recalculates the `isosurface` and `isocaps` only if one or more of the relevant properties changed

To define this class, create a program file named `IsoSurfCapChart.m` in a folder that is on the MATLAB path. Then implement the class by following the steps in the table.

Step	Implementation
Derive from the `ChartContainer` base class.	`classdef` IsoSurfCapChart < matlab.graphics.chartcontainer.ChartContainer

Step	Implementation
| Define the public properties using class and size validation.<br><br>• VolumeData, IsoValue, and Color are parameters for the isosurface.<br><br>• Enclose, WhichCapPlane, and CapVisible are parameters for the isocaps. | ```matlab
properties
    VolumeData double = rand(25,25,25)
    IsoValue (1,1) double = 0.5
    Enclose {mustBeMember(Enclose,{'above','below'})} = 'below'
    WhichCapPlane {mustBeMember(WhichCapPlane,{'all','xmin',...
        'xmax','ymin','ymax','zmin','zmax'})} = 'all'
    CapVisible (1,1) matlab.lang.OnOffSwitchState = 'on'
    Color (1,3) double {mustBeGreaterThanOrEqual(Color,0),...
        mustBeLessThanOrEqual(Color,1)} = [.2 .5 .8]
end
``` |
| Define the private properties.<br><br>• IsoPatch and CapPatch store the Patch objects for the isosurface and isocaps.<br><br>• SmoothData stores a smoothed version of the volume data.<br><br>• ExpensivePropChanged indicates whether the update method needs to recalculate the isosurface and isocaps. | ```matlab
properties(Access = private,Transient,NonCopyable)
    IsoPatch (1,1) matlab.graphics.primitive.Patch
    CapPatch (1,1) matlab.graphics.primitive.Patch
    SmoothData double = [];
    ExpensivePropChanged (1,1) logical = true
end
``` |
| Implement the setup method. In this case, call the patch function twice to create the Patch objects for the isosurface and isocaps. Store the objects in the corresponding properties, and configure the axes. | ```matlab
methods(Access = protected)
    function setup(obj)
        ax = getAxes(obj);

        % Create two Patch objects
        obj.IsoPatch = patch(ax,NaN,NaN,NaN, 'EdgeColor', 'none', ...
            'FaceColor',[.2 .5 .8],'FaceAlpha',0.9);
        hold(ax,'on');
        obj.CapPatch = patch(ax,NaN,NaN,NaN,'EdgeColor', 'none', ...
            'FaceColor','interp');

        % Configure the axes
        view(ax,3)
        camlight(ax, 'infinite');
        camlight(ax,'left');
        lighting(ax, 'gouraud');
        hold(ax,'off');
    end
``` |

23-21

Step	Implementation

Implement the `update` method. Decide whether to call the `doExpensiveCalculation` method by testing the value of `ExpensivePropChanged`. Then continue updating the other (less expensive) aspects of the chart.	```matlab
	function update(obj)
	    % Perform expensive computation if needed
	    if obj.ExpensivePropChanged
	        doExpensiveCalculation(obj);
	        obj.ExpensivePropChanged = false;
	    end
	
	    % Update visibility of CapPatch and update color
	    obj.CapPatch.Visible = obj.CapVisible;
	    obj.IsoPatch.FaceColor = obj.Color;
	end```

Implement the `doExpensiveCalculation` method, which smooths the volume data and recalculates the faces and vertices of the `isosurface` and `isocaps`.	```matlab
	function doExpensiveCalculation(obj)
	    % Update isosurface
	    obj.SmoothData = smooth3(obj.VolumeData,'box',7);
	    [F,V] = isosurface(obj.SmoothData, obj.IsoValue);
	    set(obj.IsoPatch,'Faces',F,'Vertices',V);
	    isonormals(obj.SmoothData,obj.IsoPatch);
	
	    % Update isocaps
	    [m,n,p] = size(obj.SmoothData);
	    [Xc,Yc,Zc] = meshgrid(1:n,1:m,1:p);
	    [Fc,Vc,Cc] = isocaps(Xc,Yc,Zc,obj.SmoothData,obj.IsoValue,...
	        obj.Enclose,obj.WhichCapPlane);
	    set(obj.CapPatch,'Faces',Fc,'Vertices',Vc,'CData',Cc);
	end```

Implement the `getPropertyGroups` method to customize the property display.	```matlab
	function propgrp = getPropertyGroups(obj)
	    if ~isscalar(obj)
	        % List for array of objects
	        propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj)
	
	    else
	        % List for scalar object
	        propList = {'Color','IsoValue','Enclose','CapVisible',...
	            'WhichCapPlane','VolumeData'};
	        propgrp = matlab.mixin.util.PropertyGroup(propList);
	    end
	end```

Implement the `set` methods for each expensive property (`VolumeData`, `IsoValue`, and `Enclose`). Within each method, set the corresponding property value, and then set `ExpensivePropChanged` to `true`.	```matlab
	methods
	    function set.VolumeData(obj,val)
	        obj.VolumeData = val;
	        obj.ExpensivePropChanged = true;
	    end
	    function set.IsoValue(obj, val)
	        obj.IsoValue = val;
	        obj.ExpensivePropChanged = true;
	    end
	    function set.Enclose(obj, val)
	        obj.Enclose = val;
	        obj.ExpensivePropChanged = true;
	    end
	end
	end```

Next, create an array of volume data, and then create an instance of `IsoSurfCapChart`.

```
[X,Y,Z] = meshgrid(-2:0.1:2);
v = (1/9)*X.^2 + (1/16)*Y.^2 + Z.^2;
c = IsoSurfCapChart('VolumeData',v,'IsoValue',0.5)

c =

  IsoSurfCapChart with properties:

             Color: [0.2000 0.5000 0.8000]
           IsoValue: 0.5000
            Enclose: 'below'
         CapVisible: on
      WhichCapPlane: 'all'
         VolumeData: [41×41×41 double]
```


Change the color of `c` and hide the `isocaps`.

```
c.Color = [1 0.60 0];
c.CapVisible = false;
```

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

Functions
`isocaps` | `isosurface`

More About

- "Validate Property Values"
- "Customize Property Display"
- "Property Set Methods"
- "Chart Development Overview" on page 23-2

Enabling Convenience Functions for Setting Axes Properties

When you develop a chart as a subclass of the `ChartContainer` class, consider enabling some of the MATLAB convenience functions for setting properties on the axes. For example, you can design your class to support the `title` function. By enabling convenience functions, you provide a user experience that is consistent with the MATLAB plotting functions.

Support for Different Types of Properties

The way you enable a convenience function depends on whether the function controls a noncomputed property or a computed property. This table lists the convenience functions you can support.

Convenience Function	Associated Axes Property	Type of Property
`title`, `subtitle`	`Title`, `Subtitle`	Noncomputed
`xlabel`, `ylabel`, `zlabel`	`XLabel`, `YLabel`, and `ZLabel`, respectively	Noncomputed
`xlim`, `ylim`, `zlim`	`XLim`, `YLim`, and `ZLim`, respectively	Computed
`xticks`, `yticks`, `zticks`	`XTick`, `YTick`, and `ZTick`, respectively	Computed
`xticklabels`, `yticklabels`, `zticklabels`	`XTickLabel`, `YTickLabel`, and `ZTickLabel`, respectively	Computed
`view`	`View`	Computed

Enable Functions for Noncomputed Properties

Noncomputed properties are fixed values. They do not change until a user or your code changes them explicitly.

To enable a convenience function for a noncomputed property, define a public property in your class that stores the value of axes property you want to control. Then define a public method that has the same name and supports the same calling syntaxes as the convenience function you want to support. Add a line of code to the method that sets the value of the property. For example, consider a class that has a public property called `TitleText` for storing the title. The following code shows the `title` method for the class.

```
function title(obj,txt)
      obj.TitleText = txt;
end
```

Next, add a line of code to the `update` method that calls the MATLAB convenience function to set the corresponding axes property.

```
title(getAxes(obj),obj.TitleText);
```

After you perform the preceding steps and save your class file, you can create an instance of your chart and call the `title` function to display a title. Doing so triggers this calling sequence:

1   The `title` method on the class sets the `TitleText` property, which marks the chart for updating.

2 The next time `drawnow` executes, the `update` method executes and calls the `title` function on the axes.

3 The `title` function updates the `Title` property on the axes.

Enable Functions for Computed Properties

Computed properties are controlled by the axes. The axes recomputes their values depending on the content of the axes and the underlying data.

To enable a convenience function for a computed property, define a method that has the same name and calling syntax as the convenience function you want to enable. Inside that method, call the convenience function and specify the axes as the first argument. For example, to enable the `xlim` function, define a method called `xlim` in your class. Since the `xlim` function accepts a variable number of input arguments, you must specify `varargin` as the second input argument. The `xlim` function also supports a variable number of output arguments, so you must specify `[varargout{1:nargout}]` to support those arguments.

```
function varargout = xlim(obj,varargin)
    ax = getAxes(obj);
    [varargout{1:nargout}] = xlim(ax,varargin{:});
end
```

To provide access to the corresponding property values on your chart, define two dependent properties on your class. The first property provides access to the value that the convenience function controls. The other property provides access to the mode property, which indicates how the first property is controlled. The mode property can have a value of `'auto'` or `'manual'`. Define these properties as dependent so that the chart does not store the values. The axes controls and stores these values. For example, to provide access to the `XLim` and `XLimMode` properties on the axes, define a pair of dependent properties called `XLimits` and `XLimitsMode`.

```
properties (Dependent)
    XLimits (1,2) double
    XLimitsMode {mustBeMember(XLimitsMode,{'auto','manual'})}
end
```

Next, define the `set` and `get` methods for each dependent property. Within each method, set the corresponding axes property. The following code shows the `set` methods and `get` methods for the `XLimits` and `XLimitsMode` properties.

```
function set.XLimits(obj,xlm)
    ax = getAxes(obj);
    ax.XLim = xlm;
end
function xlm = get.XLimits(obj)
    ax = getAxes(obj);
    xlm = ax.XLim;
end
function set.XLimitsMode(obj,xlmmode)
    ax = getAxes(obj);
    ax.XLimMode = xlmmode;
end
function xlm = get.XLimitsMode(obj)
    ax = getAxes(obj);
    xlm = ax.XLimMode;
end
```

After you perform the preceding steps and save your class file, you can create an instance of your chart and call the `xlim` function to change the *x*-axis limits in the chart. The `xlim` method executes, which in turn calls the `xlim` function to update the XLim property on the axes.

Note By default, MATLAB does not store any changes when the user calls the `xlim` and `ylim` functions. To provide support for preserving these changes when the user saves and loads your chart, see "Saving and Loading Instances of Chart Classes" on page 23-31.

Chart Class That Supports title, xlim, and ylim Functions

This example shows how to define a class of charts that supports the `title`, `xlim`, and `ylim` functions. The following code demonstrates how to:

- Define a `TitleText` property and implement a `title` method so that instances of the chart support the `title` function.
- Implement the `xlim` and `ylim` methods so that instances of the chart support the `xlim` and `ylim` functions.
- Define properties that allow the user to get and set the *x*- and *y*-axis limits.
- Combine `Bar` and `ErrorBar` objects into a single chart.

To define the class, copy this code into the editor and save it with the name `BarErrorBarChart.m` in a writable folder.

```matlab
classdef BarErrorBarChart < matlab.graphics.chartcontainer.ChartContainer
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        EData (1,:) double = NaN
        TitleText (:,:) char = ''
    end
    properties (Dependent)
        % Provide properties to support setting & getting
        XLimits (1,2) double
        XLimitsMode {mustBeMember(XLimitsMode,{'auto','manual'})}
        YLimits (1,2) double
        YLimitsMode {mustBeMember(YLimitsMode,{'auto','manual'})}
    end
    properties (Access = private)
        BarObject (1,1) matlab.graphics.chart.primitive.Bar
        ErrorBarObject (1,1) matlab.graphics.chart.primitive.ErrorBar
    end

    methods(Access = protected)
        function setup(obj)
            ax = getAxes(obj);
            obj.BarObject = bar(ax,NaN,NaN);
            hold(ax,'on')
            obj.ErrorBarObject = errorbar(ax,NaN,NaN,NaN);
            obj.ErrorBarObject.LineStyle = 'none';
            obj.ErrorBarObject.LineWidth = 2;
            obj.ErrorBarObject.Color = [0.6 0.7 1];
            hold(ax,'off');
```

```matlab
        end
        function update(obj)
            % Update Bar and ErrorBar XData and YData
            obj.BarObject.XData = obj.XData;
            obj.BarObject.YData = obj.YData;
            obj.ErrorBarObject.XData = obj.XData;
            obj.ErrorBarObject.YData = obj.YData;

            % Update ErrorBar delta values
            obj.ErrorBarObject.YNegativeDelta = obj.EData;
            obj.ErrorBarObject.YPositiveDelta = obj.EData;

            % Update axes title
            ax = getAxes(obj);
            title(ax,obj.TitleText);
        end
    end

    methods
        % xlim method
        function varargout = xlim(obj,varargin)
            ax = getAxes(obj);
            [varargout{1:nargout}] = xlim(ax,varargin{:});
        end
        % ylim method
        function varargout = ylim(obj,varargin)
            ax = getAxes(obj);
            [varargout{1:nargout}] = ylim(ax,varargin{:});
        end
        % title method
        function title(obj,txt)
            obj.TitleText = txt;
        end

        % set and get methods for XLimits and XLimitsMode
        function set.XLimits(obj,xlm)
            ax = getAxes(obj);
            ax.XLim = xlm;
        end
        function xlm = get.XLimits(obj)
            ax = getAxes(obj);
            xlm = ax.XLim;
        end
        function set.XLimitsMode(obj,xlmmode)
            ax = getAxes(obj);
            ax.XLimMode = xlmmode;
        end
        function xlm = get.XLimitsMode(obj)
            ax = getAxes(obj);
            xlm = ax.XLimMode;
        end

        % set and get methods for YLimits and YLimitsMode
        function set.YLimits(obj,ylm)
            ax = getAxes(obj);
            ax.YLim = ylm;
        end
        function ylm = get.YLimits(obj)
```

```
            ax = getAxes(obj);
            ylm = ax.YLim;
        end
        function set.YLimitsMode(obj,ylmmode)
            ax = getAxes(obj);
            ax.YLimMode = ylmmode;
        end
        function ylm = get.YLimitsMode(obj)
            ax = getAxes(obj);
            ylm = ax.YLimMode;
        end
    end
end
```

After saving `BarErrorBarChart.m`, create an instance of the chart.

```
BarErrorBarChart('XData',[1 2 3 4],'YData',[11 22 31 41],'EData',[2 2 2 2]);
```
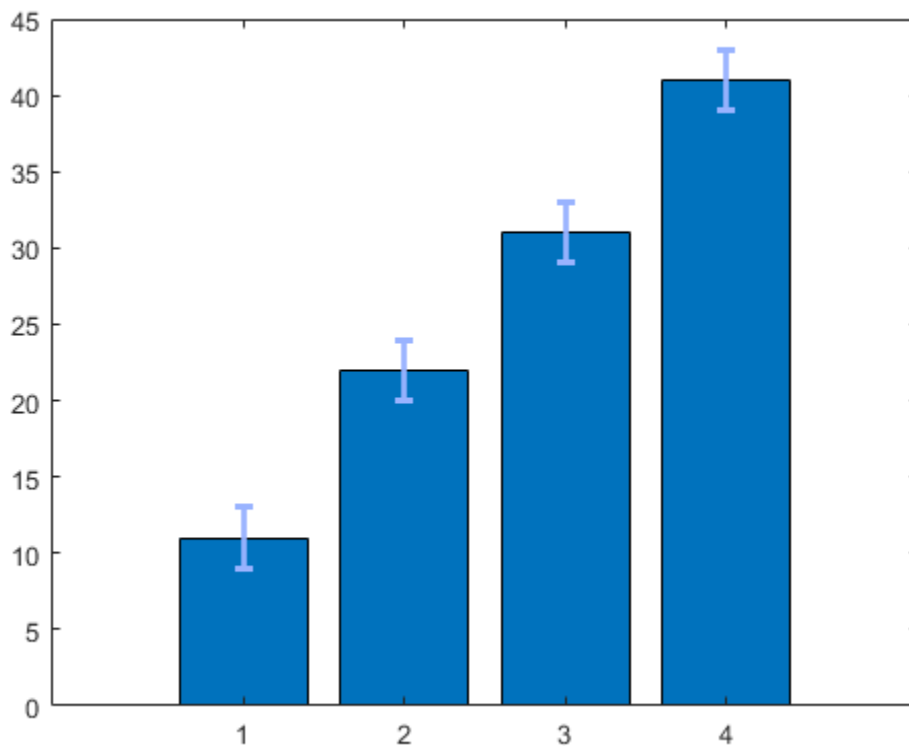

Specify a title by calling the `title` function. Then zoom into the last three bars by calling the `xlim` function.

```
title('Top Three Contributors')
xlim([1.5 5])
```
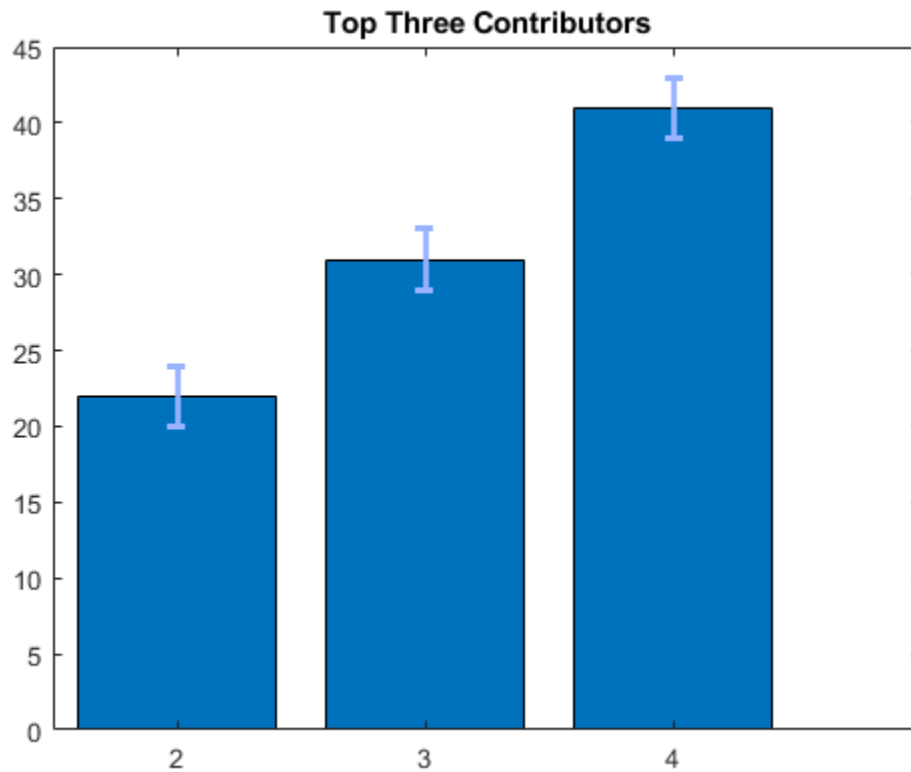
See Also

Classes
matlab.graphics.chartcontainer.ChartContainer

Functions
bar | errorbar | title | xlim | ylim

Properties
Axes | Bar | ErrorBar

More About

- "Chart Development Overview" on page 23-2
- "Property Get Methods"
- "Property Set Methods"

Saving and Loading Instances of Chart Classes

Charts that inherit from the `ChartContainer` base class follow the same rules for saving and loading as other MATLAB objects. However in some cases, you might want your objects to save and load additional information. For example, to provide support for saving and loading the result of interactive changes, such as rotating or zooming, you must store the modified view of the axes in a property on your class. By defining properties and methods for storing and retrieving these kinds of changes, you enable users to save and reload instances of your chart with their changes preserved.

Coding Pattern for Saving and Loading Axes Changes

The built-in axes interactions change certain properties on the axes. For example, dragging to rotate a 3-D chart changes the `View` property. Similarly, scrolling to zoom within a chart changes the `XLim`, `YLim`, (and possibly `ZLim`) properties on the axes. To preserve the changes when the user saves and reloads the chart, add these components to your class:

- Define a protected property for storing the chart state on page 23-31 — This property provides a place to store the axes changes when MATLAB saves the chart object. For example, you might name this property `ChartState`.
- Define a `get` method for retrieving the chart state on page 23-31 — This method does either of two things depending on whether MATLAB is saving or loading the chart object. When MATLAB saves the chart object, the method returns the relevant axes changes so they can be saved. When MATLAB loads the chart object, the method returns the axes changes that are stored in the `ChartState` property.
- Define a protected method that updates the axes on page 23-32 — When the chart object loads into MATLAB, this method calls the `get` method for the `ChartState` property and then updates the relevant axes properties for the chart.

Define a Protected Property for Storing the Chart State

Define a protected property to store the relevant axes information. This property is empty except when MATLAB sets its value during the save process, or when MATLAB loads a saved instance of the chart. Define the property with a name that is useful and easy to recognize. For example, define a property called `ChartState`.

```
properties (Access = protected)
    ChartState = []
end
```

Define a get Method for Retrieving the Chart State

Define a public `get` method for the `ChartState` property. Like all `set` and `get` methods, this method automatically inherits the access permissions of the `ChartState` property. MATLAB calls this method when it saves an instance of the chart.

Within this method, create a variable called `isLoadedStateAvailable` that stores a `logical` value. This value is `true` when the `ChartState` property is not empty.

Next, write a conditional statement that checks the value of `isLoadedStateAvailable`. Divide the statement into clauses:

- `if...then` clause — The `isLoadedStateAvailable` value is `true`. Return the contents of the `ChartState` property.
- `else` clause — The `isLoadedStateAvailable` value is `false`. Create a structure and get the axes object. Add the `XLim`, `YLim`, and `ZLim` fields to the structure only if the `XLim`, `YLim`, and `ZLim` properties on the axes changed. To test whether the axes properties changed, check to see if the corresponding mode properties are set to `'manual'`. Since there is no mode property associated with the axes `View` property, add the `View` field to the structure without checking anything.

```
methods
    function data = get.ChartState(obj)
        isLoadedStateAvailable = ~isempty(obj.ChartState);

        if isLoadedStateAvailable
            data = obj.ChartState;
        else
            data = struct;
            ax = getAxes(obj);

            % Get axis limits only if mode is manual.
            if strcmp(ax.XLimMode,'manual')
                data.XLim = ax.XLim;
            end
            if strcmp(ax.YLimMode,'manual')
                data.YLim = ax.YLim;
            end
            if strcmp(ax.ZLimMode,'manual')
                data.ZLim = ax.ZLim;
            end

            % No ViewMode to check. Store the view anyway.
            data.View = ax.View;
        end
    end
end
```

Define a Protected Method That Updates the Axes

Define a protected method called `loadstate`. In this method, perform these steps:

- Query the `ChartState` property and store the returned value as `data`.
- Check for the existence of the `XLim`, `YLim`, `ZLim`, and `View` fields before updating the corresponding properties on the axes.
- Clear the contents of the `ChartState` property.

After you create this method, call it near the end of the `setup` method (after creating the graphics objects that make up your chart). The `setup` method executes when MATLAB creates a new instance of the chart or when it loads an instance of a chart.

```
function loadstate(obj)
    data=obj.ChartState;
    ax = getAxes(obj);

    % Look for states that changed
    if isfield(data, 'XLim')
```

```
            ax.XLim=data.XLim;
        end
        if isfield(data, 'YLim')
            ax.YLim=data.YLim;
        end
        if isfield(data, 'ZLim')
            ax.ZLim=data.ZLim;
        end
        if isfield(data, 'View')
            ax.View=data.View;
        end

        % Reset ChartState to empty
        obj.ChartState=[];
    end
end
```

Example: 3-D Plot That Stores Axis Limits and View

Define a `MeshGradientChart` class for displaying a mesh plot with *x* and *y* gradient vectors at the grid points. Design this class so that the `XLim`, `YLim`, `ZLim`, and `View` properties of the axes are preserved when the user saves and reloads an instance of the chart.

To define this class, create a program file named `MeshGradientChart.m` in a folder that is on the MATLAB path. Then implement the class by following the steps in the table.

Step	Implementation
Derive from the `ChartContainer` base class.	```classdef MeshGradientChart < matlab.graphics.chartcontainer.ChartContainer```

Define the public properties.	```properties```
	``` XData (:,:) double = []```
	``` YData (:,:) double = []```
	``` ZData (:,:) double = []```
	```end```

Define the private properties. One property stores a `Surface` object, and the other stores a `Quiver` object.	```properties (Access = private,Transient,NonCopyable)```
	``` SurfaceObject (1,1) matlab.graphics.chart.primitive.Surface```
	``` QuiverObject (1,1) matlab.graphics.chart.primitive.Quiver```
	```end```

Define a protected `ChartState` property for storing the axes state.	```properties (Access = protected)```
	``` ChartState = []```
	```end```

Step	Implementation
| Implement the `setup` method. In this case, call the `mesh` and `quiver3` functions to create the `Surface` and `Quiver` objects respectively. Store the objects in the corresponding properties, and turn the hold state of the axes to `'off'`. Then call the `loadstate` method to update the state of the axes. | ```matlab
methods(Access = protected)
    function setup(obj)
        ax = getAxes(obj);

        % Create Mesh and Quiver objects.
        obj.SurfaceObject=mesh(ax,[],[],[],'FaceColor','none');
        hold(ax,'on')
        obj.QuiverObject=quiver3(ax,[],[],[],[],'Color','r','LineWidth
        hold(ax,'off')

        % Load state of the axes.
        loadstate(obj);
    end
``` |
| Implement the `update` method. In this case, update the *x*- and *y*-coordinates of the mesh plot and the tails of the gradient vectors. Then update the lengths and directions of the vectors. | ```matlab
    function update(obj)
        % Update Mesh data.
        obj.SurfaceObject.XData = obj.XData;
        obj.SurfaceObject.YData = obj.YData;
        obj.SurfaceObject.ZData = obj.ZData;

        % Update locations of vector tails.
        obj.QuiverObject.XData = obj.XData;
        obj.QuiverObject.YData = obj.YData;
        obj.QuiverObject.ZData = obj.ZData;

        % Update lengths and directions of vectors.
        [gradx,grady] = gradient(obj.ZData);
        obj.QuiverObject.UData = gradx;
        obj.QuiverObject.VData = grady;
        obj.QuiverObject.WData = zeros(size(obj.ZData));
    end
``` |

Step	Implementation

Implement the `loadstate` method, which updates the axes and resets the `ChartState` property to an empty array.	```matlab
	        function loadstate(obj)
	            data=obj.ChartState;
	            ax = getAxes(obj);
	
	            % Look for states that changed.
	            if isfield(data, 'XLim')
	                ax.XLim=data.XLim;
	            end
	            if isfield(data, 'YLim')
	                ax.YLim=data.YLim;
	            end
	            if isfield(data, 'ZLim')
	                ax.ZLim=data.ZLim;
	            end
	            if isfield(data, 'View')
	                ax.View=data.View;
	            end
	
	            % Reset ChartState to empty.
	            obj.ChartState=[];
	        end
	    end
	``

Implement the `ChartState` get method, which returns the axes state information.	```matlab
	    methods
	        function data = get.ChartState(obj)
	            isLoadedStateAvailable = ~isempty(obj.ChartState);
	
	            % Return ChartState content if loaded state is available.
	            % Otherwise, return current axes state.
	            if isLoadedStateAvailable
	                data = obj.ChartState;
	            else
	                data = struct;
	                ax = getAxes(obj);
	
	                % Get axis limits only if mode is manual.
	                if strcmp(ax.XLimMode,'manual')
	                    data.XLim = ax.XLim;
	                end
	                if strcmp(ax.YLimMode,'manual')
	                    data.YLim = ax.YLim;
	                end
	                if strcmp(ax.ZLimMode,'manual')
	                    data.ZLim = ax.ZLim;
	                end
	
	                % No ViewMode to check. Store the view anyway.
	                data.View = ax.View;
	            end
	        end
	    end
	end
	``

Next, create an instance of the chart. Then rotate or zoom into the chart and save it. The object preserves the interactive changes when you load the chart back into MATLAB.

23-35

Create an instance of the chart

```
[X,Y] = meshgrid(-5:5);
Z = X.^2 + Y.^2;
c = MeshGradientChart('XData',X,'YData',Y,'ZData',Z);
```


When you create the chart:

- The setup method calls the loadstate method.
- The loadstate method performs these tasks, which ultimately have no effect on the chart object or the underlying axes object.

  - Call the get.ChartState method, which returns a structure containing the current value of the axes View property.
  - Reset the View property on the axes to the value stored in the structure.
  - Clear the contents of the ChartState property.

Rotate or zoom into the chart and save it

```
savefig(gcf,'mychart.fig')
```

When you save the chart, MATLAB calls the get.ChartState method, which returns a structure containing:

- The values of the XLim, YLim, or ZLim properties on the axes, but only if they changed
- The value of the View property on the axes

After MATLAB retrieves the structure, it stores the structure in the ChartState property of the chart object that is being saved.

Load the chart that you saved

```
openfig('mychart.fig')
```

When you load the chart:

- The `setup` method calls the `loadstate` method.
- The `loadstate` method performs these tasks:
  - Call the `get.ChartState` method, which returns the structure from the `ChartState` property.
  - Reset the `XLim`, `YLim`, `ZLim`, and `View` properties on the axes, but only if the structure contains the corresponding fields.
  - Clear the contents of the `ChartState` property.

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

Functions
`mesh` | `quiver3`

Properties
Axes | Chart Surface | Quiver

More About

- "Save and Load Process for Objects"
- "Chart Development Overview" on page 23-2
- "Property Get Methods"

Chart Class with Custom Property Display

This example shows how to define a class of charts with a custom property display that lists only a subset of the properties. The following code demonstrates how to overload the `getPropertyGroups` method of the `matlab.mixin.CustomDisplay` class. The example also demonstrates the basic coding pattern for charts that derive from the `matlab.graphics.chartcontainer.ChartContainer` base class. You can use this example to become familiar with the coding techniques of chart development, or as the basis for a class you plan to develop.

To define the class, copy the following code into the editor and save it with the name `SmoothPlotCustomDisplay.m` in a writable folder.

```
classdef SmoothPlotCustomDisplay < matlab.graphics.chartcontainer.ChartContainer
    % c = SmoothPlotCustomDisplay('XData',X,'YData',Y,Name,Value,...)
    % plots a dotted line of the coordinates in X and Y with a smoothed
    % version of the line. You can also specify additonal name-value
    % arguments, such as 'SmoothColor' and 'SmoothWidth'.
    properties
        XData (1,:) double = NaN
        YData (1,:) double = NaN
        SmoothColor {validatecolor} = [0.9290 0.6940 0.1250]
        SmoothWidth (1,1) double = 2
    end
    properties(Access = private,Transient,NonCopyable)
        OriginalLine (1,1) matlab.graphics.chart.primitive.Line
        SmoothLine (1,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(obj)
            % Get the axes
            ax = getAxes(obj);

            % Create the original and smooth lines
            obj.OriginalLine = plot(ax,NaN,NaN,'LineStyle',':');
            hold(ax,'on')
            obj.SmoothLine = plot(ax,NaN,NaN,'LineStyle','-',...
                'Color',[0.9290 0.6940 0.1250],'LineWidth',2);
            hold(ax,'off')
        end
        function update(obj)
            % Update line data
            obj.OriginalLine.XData = obj.XData;
            obj.OriginalLine.YData = obj.YData;
            obj.SmoothLine.XData = obj.XData;
            obj.SmoothLine.YData = createSmoothData(obj);

            % Update line color and width
            obj.SmoothLine.Color = obj.SmoothColor;
            obj.SmoothLine.LineWidth = obj.SmoothWidth;
        end
        function propgrp = getPropertyGroups(obj)
            if ~isscalar(obj)
                % List for array of objects
```

```
                    propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);

            else
                % List for scalar object
                propList = {'SmoothColor','XData','YData'};
                propgrp = matlab.mixin.util.PropertyGroup(propList);
            end
        end
        function sm = createSmoothData(obj)
            % Calculate smoothed data
            v = ones(1,10)*0.1;
            sm = conv(obj.YData,v,'same');
        end
    end
end
```

After saving the class file, you can create an instance of the chart. Omit the semicolon when you create the chart to see the customized display.

```
x = 1:1:100;
y = 10*sin(x/15)+8*sin(10*x+0.5);
c = SmoothPlotCustomDisplay('XData',x,'YData',y)


c =

  SmoothPlotCustomDisplay with properties:

    SmoothColor: [0.9290 0.6940 0.1250]
          XData: [1x100 double]
          YData: [1x100 double]

  Use GET to show all properties
```

See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

More About

- "Customize Property Display"
- "Managing Properties of Chart Classes" on page 23-17

Chart Class with Variable Number of Lines

This example shows how to define a class of charts that can display any number of lines based on the size of the user's data. The chart displays as many lines as there are columns in the YData matrix. For each line, the chart calculates the local extrema and indicates their locations with circular markers. The following code demonstrates how to:

- Define two properties called PlotLineArray and ExtremaArray that store the objects for the lines and the markers, respectively.

- Implement an update method that replaces the contents of the PlotLineArray and ExtremaArray properties with the new objects. Because this method executes all the plotting and configuration commands, the setup method is empty. This is a simple way to create any number of lines. To learn how to create this chart more efficiently, by reusing existing line objects, see "Optimized Chart Class for Displaying Variable Number of Lines" on page 23-44.

To define the class, copy this code into the editor and save it with the name LocalExtremaChart.m in a writable folder.

```matlab
classdef LocalExtremaChart < matlab.graphics.chartcontainer.ChartContainer
    % c = LocalExtremaChart('XData',X,'YData',Y,Name,Value,...)
    % plots one line with markers at local extrema for every column of matrix Y.
    % You can also specify the additonal name-value arguments, 'MarkerColor'
    % and 'MarkerSize'.

    properties
        XData (1,:) double = NaN
        YData (:,:) double = NaN
        MarkerColor {validatecolor} = [1 0 0]
        MarkerSize (1,1) double = 5
    end
    properties(Access = private,Transient,NonCopyable)
        PlotLineArray (:,1) matlab.graphics.chart.primitive.Line
        ExtremaArray (:,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(~)
        end
        function update(obj)
            % get the axes
            ax = getAxes(obj);

            % Plot Lines and the local extrema
            obj.PlotLineArray = plot(ax,obj.XData,obj.YData);
            hold(ax,'on')

            % Replicate x-coordinate vectors to match size of YData
            newx = repmat(obj.XData(:),1,size(obj.YData,2));

            % Find local minima and maxima and plot markers
            tfmin = islocalmin(obj.YData,1);
            tfmax = islocalmax(obj.YData,1);
            obj.ExtremaArray = plot(ax,newx(tfmin),obj.YData(tfmin),'o',...
                newx(tfmax),obj.YData(tfmax),'o',...
                'MarkerEdgeColor','none',...
```

```
                'MarkerFaceColor',obj.MarkerColor,...
                'MarkerSize',obj.MarkerSize);
            hold(ax,'off')
        end
    end
end
```

After saving the class file, you can create an instance of the chart. For example:

```
x = linspace(0,3);
y1 = cos(5*x)./(1+x.^2);
y2 = -cos(5*x)./(1+x.^3);
y3 = sin(x)./2;
y = [y1' y2' y3'];
c = LocalExtremaChart('XData',x,'YData',y);
```
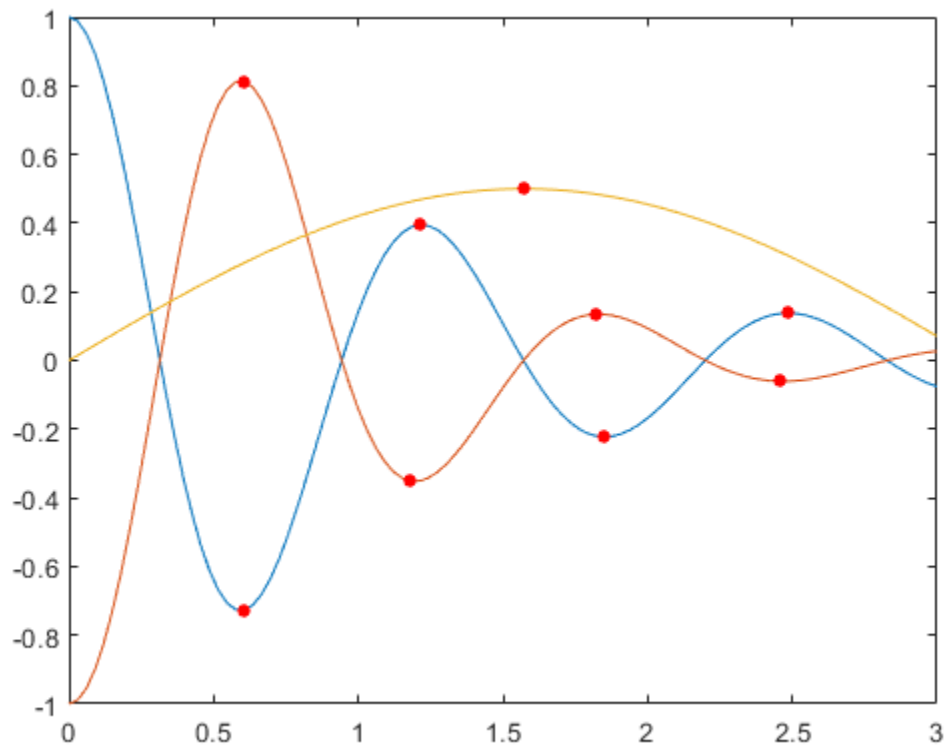

Change the marker size to 8.

```
c.MarkerSize = 8;
```
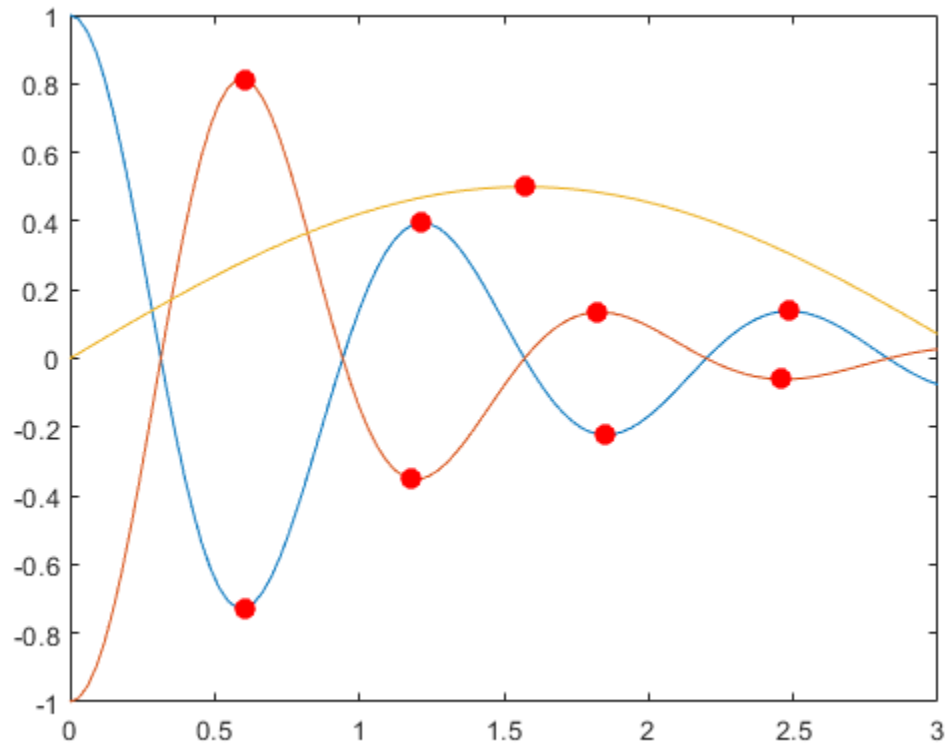
See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

More About

*   "Chart Development Overview" on page 23-2
*   "Optimized Chart Class for Displaying Variable Number of Lines" on page 23-44

Optimized Chart Class for Displaying Variable Number of Lines

This example shows how to optimize a chart class for displaying a variable number of lines. It reuses existing line objects, which can improve the performance of the chart, especially if the number of lines does not change frequently. For a simpler version of this chart without the optimization, see "Chart Class with Variable Number of Lines" on page 23-41.

The chart displays as many lines as there are columns in the YData matrix, with circular markers at the local extrema. The following code demonstrates how to:

- Define two properties called `PlotLineArray` and `ExtremaLine` that store the objects for the lines and the markers, respectively.
- Implement a `setup` method that initializes the `ExtremaLine` object.
- Implement an `update` method that gets the size of the `PlotLineArray`, and then adds or subtracts objects from that array according to the number of columns in YData.

To define the class, copy this code into the editor and save it with the name `OptimLocalExtremaChart.m` in a writable folder.

```matlab
classdef OptimLocalExtremaChart < matlab.graphics.chartcontainer.ChartContainer
    % c = OptimLocalExtremaChart('XData',X,'YData',Y,Name,Value,...)
    % plots one line with markers at local extrema for every column of matrix Y.
    % You can also specify the additonal name-value arguments, 'MarkerColor'
    % and 'MarkerSize'.

    properties
        XData (:,1) double = NaN
        YData (:,:) double = NaN
        MarkerColor {validatecolor} = [1 0 0]
        MarkerSize (1,1) double = 5
    end
    properties(Access = private,Transient,NonCopyable)
        PlotLineArray (:,1) matlab.graphics.chart.primitive.Line
        ExtremaLine (:,1) matlab.graphics.chart.primitive.Line
    end

    methods(Access = protected)
        function setup(obj)
            obj.ExtremaLine = matlab.graphics.chart.primitive.Line(...
                'Parent', obj.getAxes(), 'Marker', 'o', ...
                'MarkerEdgeColor', 'none', 'LineStyle',' none');
        end
        function update(obj)
            % Get the axes
            ax = getAxes(obj);

            % Create extra lines as needed
            p = obj.PlotLineArray;
            nPlotLinesNeeded = size(obj.YData, 2);
            nPlotLinesHave = numel(p);
            for n = nPlotLinesHave+1:nPlotLinesNeeded
                p(n) = matlab.graphics.chart.primitive.Line('Parent', ax, ...
                    'SeriesIndex', n, 'LineWidth', 2);
            end
```

```matlab
            % Update the lines
            for n = 1:nPlotLinesNeeded
                p(n).XData = obj.XData;
                p(n).YData = obj.YData(:,n);
            end

            % Delete unneeded lines
            delete(p((nPlotLinesNeeded+1):numel(p)))
            obj.PlotLineArray = p(1:nPlotLinesNeeded);

            % Replicate x-coordinate vectors to match size of YData
            newx = repmat(obj.XData(:),1,size(obj.YData,2));

            % Find local minima and maxima and plot markers
            tfmin = islocalmin(obj.YData,1);
            tfmax = islocalmax(obj.YData,1);
            obj.ExtremaLine.XData = [newx(tfmin); newx(tfmax)];
            obj.ExtremaLine.YData = [obj.YData(tfmin); obj.YData(tfmax)];
            obj.ExtremaLine.MarkerFaceColor = obj.MarkerColor;
            obj.ExtremaLine.MarkerSize = obj.MarkerSize;

            % Make sure the extrema are on top
            uistack(obj.ExtremaLine, 'top');
        end
    end
end
```

After saving the class file, you can create an instance of the chart. For example:

```matlab
x = linspace(0,2)';
y = cos(5*x)./(1+x.^2);
c = OptimLocalExtremaChart('XData',x,'YData',y);
```
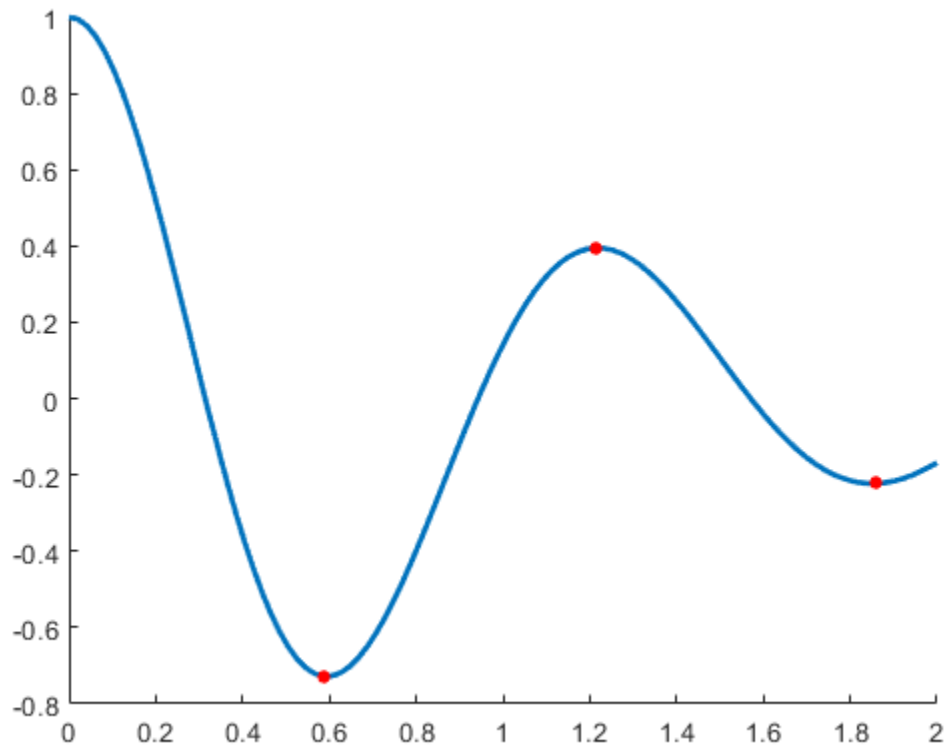
Now, create a `for` loop that adds an additional line to the plot at every iteration. The chart object keeps all the existing lines, and adds one additonal line for each `i`.

```
for i=1:10
    y = cos(5*x+i)./(1+x.^2);
    c.YData = [c.YData y];
end
```
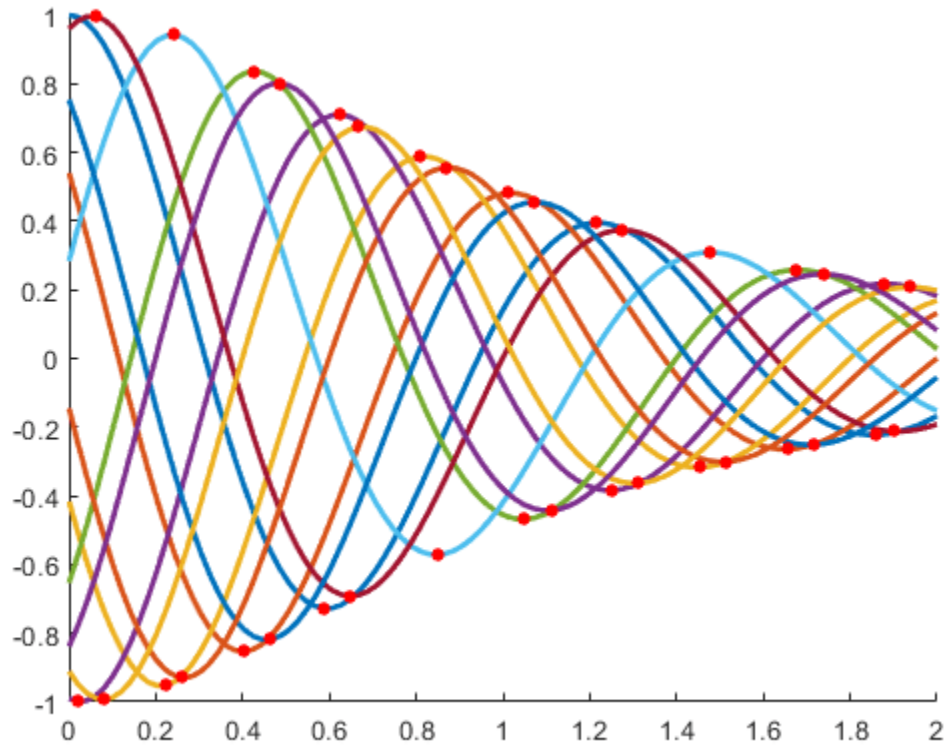
See Also

Classes
`matlab.graphics.chartcontainer.ChartContainer`

Related Examples

*   "Chart Class with Variable Number of Lines" on page 23-41
*   "Managing Properties of Chart Classes" on page 23-17

Chart Class for Displaying Variable Size Tiling of Plots

This example shows how to define a class for creating a tiling of plots that can be any size, depending on the size of the user's data. The chart has a public `Data` property that accepts an m-by-n matrix. The chart displays an n-by-n square tiling of scatter plots and histograms. The scatter plots show the different columns of the data plotted against each other. The histograms show the distribution of the values within each column of the data.

The `update` method in this class recreates the histograms and scatter plots to reflect changes in the data. If the grid size of the layout conflicts with the size of the data, then all the axes are deleted and the `GridSize` property is updated to match the size of the data. Then a new set of axes objects is created.

To define the class, copy the following code into the editor and save it with the name `TrellisChart.m` in a writable folder.

```matlab
classdef TrellisChart < matlab.graphics.chartcontainer.ChartContainer

    properties
        Data(:,:) {mustBeNumeric}
        ColNames(1,:) string
        TitleText(1,:) string
    end

    methods (Access = protected)
        function setup(obj)
            % Use one toolbar for all of the axes
            axtoolbar(getLayout(obj),'default');
        end

        function update(obj)
            % Get the layout and store it as tcl
            tcl = getLayout(obj);
            numvars = size(obj.Data,2);

            % Reconfigure layout if needed
            if numvars ~= tcl.GridSize(1)
                % Delete layout contents to change the grid size
                delete(tcl.Children);
                if numvars>0
                    tcl.GridSize = [numvars numvars];
                    for i = 1:numvars^2
                        nexttile(tcl,i);
                    end
                end
            end

            % Populate the layout with the axes
            ax = gobjects(numvars,numvars);
            for col = 1:numvars
                for row = 1:numvars
                    % Get the axes at the current row/column
                    t = col + (row-1) * numvars;
                    ax(row,col)=nexttile(tcl,t);
                    if col==row
```

```matlab
                    % On the diagonal, draw histograms
                    histogram(ax(row,col),obj.Data(:,col));
                    ylabel(ax(row,col),'Count')
                else
                    % Off the diagonal, draw scatters
                    scatter(ax(row,col),obj.Data(:,col),...
                        obj.Data(:,row),'filled','MarkerFaceAlpha',0.6)
                    if length(obj.ColNames) >= row
                        ylabel(ax(row,col),obj.ColNames(row));
                    end
                end

                if length(obj.ColNames) >= col
                    xlabel(ax(row,col),obj.ColNames(col));
                end
            end

            % Link the x-axis for each column, so that panning or zooming
            % affects all axes in the column.
            linkaxes(ax(:,col),'x')
        end

        % Chart title
        title(tcl,obj.TitleText,'FontSize',16);
    end
end
end
```

After saving the class file, create an instance of the chart.

```matlab
load patients
chartTitle = "Height, Weight, and Diastolic Blood Pressure";
c = TrellisChart('Data',[Height Weight Diastolic], ...
    'colNames', ["Height" "Weight" "Diastolic"],...
    'TitleText',chartTitle);
```

Height, Weight, and Diastolic Blood Pressure
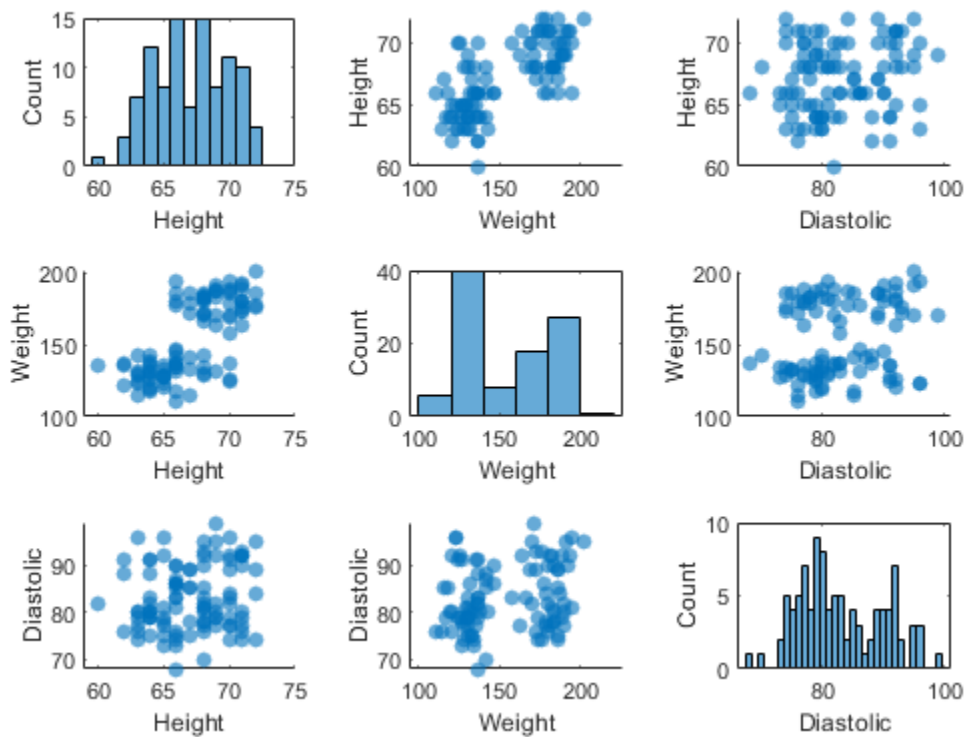
See Also

Functions
getLayout

Classes
matlab.graphics.chartcontainer.ChartContainer

Properties
TiledChartLayout Properties

More About

•   "Develop Charts With Polar Axes, Geographic Axes, or Multiple Axes" on page 23-13

Chart Class Containing Two Interactive Plots

This example shows how to define a class for visualizing timetable data using two axes with interactive features. The top axes has panning and zooming enabled along the *x* dimension, so the user can examine a region of interest. The bottom axes displays a plot over the entire time range. The bottom axes also displays a light blue time window, which indicates the time range in the top axes. The class defines the following properties, methods, and local functions.

Properties:

- `Data` - A public and dependent property that stores a timetable.
- `TimeLimits` - A public property that sets the limits of the top axes and the width of the time window in the bottom axes.
- `SavedData` - A protected property that enables the user to save and load instances of the chart and preserve the data.
- `TopAxes` and `BottomAxes` - Private properties that store the axes objects.
- `TopLine` and `BottomLine` - Private properties that store the line objects.
- `TimeWindow` - A patch object displayed in the bottom axes, which indicates the time range of the top axes.

Methods:

- `set.Data` and `get.Data` - Enable the user to save and load instances of the chart and preserve the data.
- `setup` - Executes once when the chart is created. It configures the layout and the axes, the line objects, and the patch object.
- `update` - Executes after the `setup` method and after the user changes one or more properties on the chart.
- `panZoom` - Updates the time limits of the chart when the user pans or zooms within the top axes. This causes the time window to update to reflect the new limits.
- `click` - Recalculates the time limits when the user clicks the bottom axes.

Local Functions:

- `updateDataTipTemplate` - Called from within in the `update` method. It creates rows in the data tips that correspond to the variables in the timetable.
- `mustHaveOneNumericVariable` - Validates the `Data` property. This function ensures that the timetable specified by the user has at least one numeric variable.

To define the class, copy the following code into the editor and save it with the name `TimeTableChart.m` in a writable folder.

```
classdef TimeTableChart < matlab.graphics.chartcontainer.ChartContainer
    properties (Dependent)
        Data timetable {mustHaveOneNumericVariable} = ...
            timetable(datetime.empty(0,1),zeros(0,1))
    end

    properties
        TimeLimits (1,2) datetime = [NaT NaT]
```

```matlab
        end

        properties (Access = protected)
            SavedData timetable = timetable(datetime.empty(0,1),zeros(0,1))
        end

        properties (Access = private, Transient, NonCopyable)
            TopAxes matlab.graphics.axis.Axes
            TopLine matlab.graphics.chart.primitive.Line
            BottomAxes matlab.graphics.axis.Axes
            BottomLine matlab.graphics.chart.primitive.Line
            TimeWindow matlab.graphics.primitive.Patch
        end

        methods
            function set.Data(obj, tbl)
                % Reset the time limits if the row times have changed.
                oldTimes = obj.SavedData.Properties.RowTimes;
                newTimes = tbl.Properties.RowTimes;
                if ~isequal(oldTimes, newTimes)
                    obj.TimeLimits = [NaT NaT];
                end

                % Store the new table.
                obj.SavedData = tbl;
            end

            function tbl = get.Data(obj)
                tbl = obj.SavedData;
            end
        end

        methods (Access = protected)
            function setup(obj)
                % Create two axes. The top axes is 3x taller than bottom axes.
                tcl = getLayout(obj);
                tcl.GridSize = [4 1];
                obj.TopAxes = nexttile(tcl, 1, [3 1]);
                obj.BottomAxes = nexttile(tcl, 4);

                % Add a shared toolbar on the layout, which removes the
                % toolbar from the individual axes.
                axtoolbar(tcl, 'default');

                % Create one line to show the zoomed-in data.
                obj.TopLine = plot(obj.TopAxes, NaT, NaN);

                % Create one line to show an overview of the data, and disable
                % HitTest so the ButtonDownFcn on the bottom axes works.
                obj.BottomLine = plot(obj.BottomAxes, NaT, NaN, ...
                    'HitTest', 'off');

                % Create a patch to show the current time limits.
                obj.TimeWindow = patch(obj.BottomAxes, ...
                    'Faces', 1:4, ...
                    'Vertices', NaN(4,2), ...
                    'FaceColor', obj.TopLine.Color, ...
                    'FaceAlpha', 0.3, ...
```

```matlab
            'EdgeColor', 'none', ...
            'HitTest', 'off');

        % Constrain axes panning/zooming to only the X-dimension.
        obj.TopAxes.Interactions = [ ...
            dataTipInteraction;
            panInteraction('Dimensions','x');
            rulerPanInteraction('Dimensions','x');
            zoomInteraction('Dimensions','x')];

        % Disable pan/zoom on the bottom axes.
        obj.BottomAxes.Interactions = [];

        % Add a listener to XLim to respond to zoom events.
        addlistener(obj.TopAxes, 'XLim', 'PostSet', @(~, ~) panZoom(obj));

        % Add a callback for clicks on the bottom axes.
        obj.BottomAxes.ButtonDownFcn = @(~, ~) click(obj);
    end

    function update(obj)
        % Extract the time data from the table.
        tbl = obj.Data;
        t = tbl.Properties.RowTimes;

        % Extract the numeric variables from the table.
        S = vartype('numeric');
        numericTbl = tbl(:,S);

        % Update the data on both lines.
        set([obj.BottomLine obj.TopLine], 'XData', t, 'YData', numericTbl{:,1});

        % Create a dataTipTextRow for each variable in the timetable.
        updateDataTipTemplate(obj.TopLine, tbl)

        % Update the top axes limits.
        obj.TopAxes.YLimMode = 'auto';
        if obj.TimeLimits(1) < obj.TimeLimits(2)
            obj.TopAxes.XLim = obj.TimeLimits;
        else
            % Current time limits are invalid, so set XLimMode to auto and
            % let the axes calculate limits based on available data.
            obj.TopAxes.XLimMode = 'auto';
            obj.TimeLimits = obj.TopAxes.XLim;
        end

        % Update time window to reflect the new time limits.
        xLimits = ruler2num(obj.TimeLimits, obj.BottomAxes.XAxis);
        yLimits = obj.BottomAxes.YLim;
        obj.TimeWindow.Vertices = [xLimits([1 1 2 2]); yLimits([1 2 2 1])]';
    end

    function panZoom(obj)
        % When XLim on the top axes changes, update the time limits.
        obj.TimeLimits = obj.TopAxes.XLim;
    end

    function click(obj)
```

23-53

```matlab
                % When clicking on the bottom axes, recenter the time limits.

                % Find the center of the click using CurrentPoint.
                center = obj.BottomAxes.CurrentPoint(1,1);

                % Convert from numeric units into datetime using num2ruler.
                center = num2ruler(center, obj.BottomAxes.XAxis);

                % Find the width of the current time limits.
                width = diff(obj.TimeLimits);

                % Recenter the current time limits.
                obj.TimeLimits = center + [-1 1]*width/2;
            end
        end
    end

    function updateDataTipTemplate(obj, tbl)

    % Create a dataTipTextRow for each variable in the timetable.
    timeVariable = tbl.Properties.DimensionNames{1};
    rows = dataTipTextRow(timeVariable, tbl.(timeVariable));
    for n = 1:numel(tbl.Properties.VariableNames)
        rows(n+1,1) = dataTipTextRow(...
            tbl.Properties.VariableNames{n}, tbl{:,n});
    end
    obj.DataTipTemplate.DataTipRows = rows;

    end

    function mustHaveOneNumericVariable(tbl)

    % Validation function for Data property.
    S = vartype('numeric');
    if width(tbl(:,S)) < 1
        error('TimeTableChart:InvalidTable', ...
            'Table must have at least one numeric variable.')
    end

    end
```

After saving the class file, create an instance of the chart. In this case, use the chart to examine a few weeks within a year of bicycle traffic data.

```matlab
bikeTbl = readtimetable('BicycleCounts.csv');
bikeTbl = bikeTbl(169:8954,:);
tlimits = [datetime(2015,8,6) datetime(2015,8,27)];
TimeTableChart('Data',bikeTbl,'TimeLimits',tlimits);
```
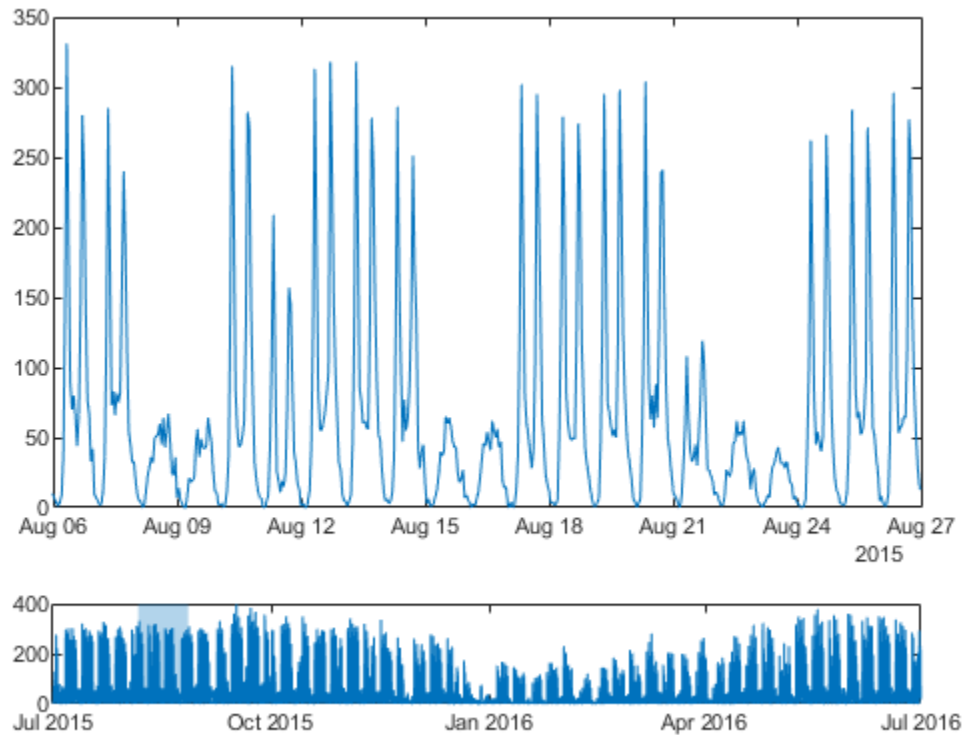
See Also

Functions
getLayout | timetable

Classes
matlab.graphics.chartcontainer.ChartContainer

Properties
DataTipTemplate Properties | TiledChartLayout Properties

More About

- "Develop Charts With Polar Axes, Geographic Axes, or Multiple Axes" on page 23-13
- "Overview Events and Listeners"
- "Create Custom Data Tips" on page 13-6

Optimize Performance of Graphics Programs

Finding Code Bottlenecks

Use the 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:

Profile Summary (Total Time: 0.794 s)

▼ Flame Graph

graphics\private\clo
cla
newplot>ObserveAxesN...
newplot
newplotwrapper
myPlot
Profile Summary

myPlot (93.8%, 0.745 s)

Generated 14-Nov-2019 13:52:01 using performance time.
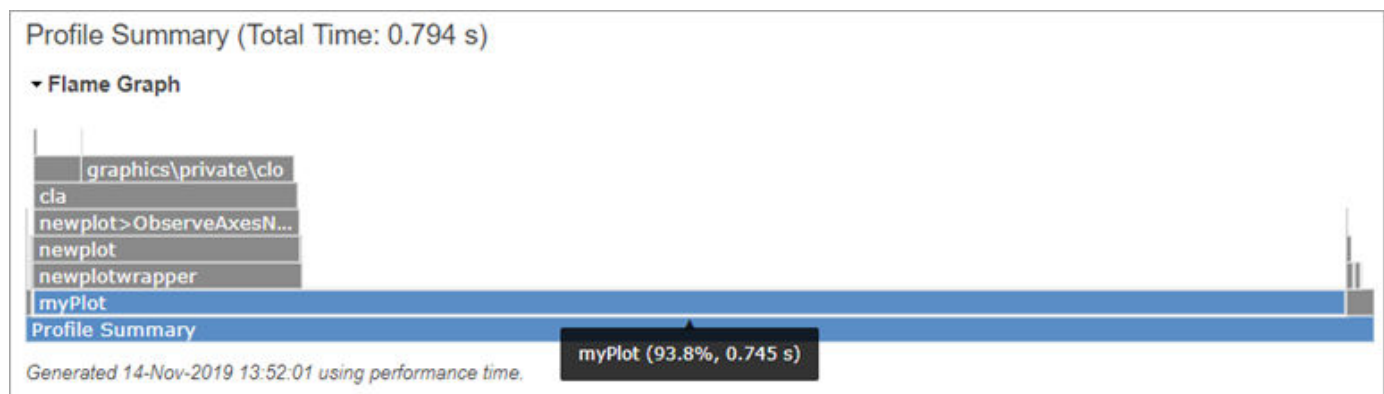
Because the x and y arrays contain 1000 columns of data, the plot function creates 1000 line objects. In this case, you can achieve the same results by creating one line with 10000 data points:

```
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:

Profile Summary (Total Time: 0.070 s)
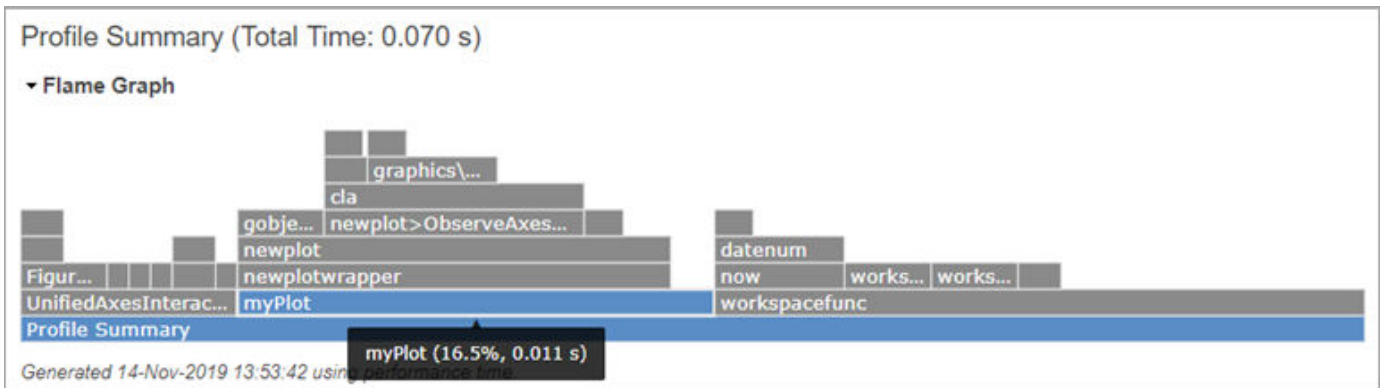
You can often achieve improvements in execution speed by understanding how to avoid or minimize inherently slow operations. For information on how to improve performance using this tool, see the documentation for the `profile` function.

What Affects Code Execution Speed

In this section...
"Potential Bottlenecks" on page 24-4
"How to Improve Performance" on page 24-4

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 24-5.
- Updating an object that has some percentage of static data? See "Avoid Updating Static Data" on page 24-12.
- Searching for object handles. See "Avoid Repeated Searches for Objects" on page 24-7.
- Rotating, translating, or scaling objects? See "Transforming Objects Efficiently" on page 24-14.
- Querying and setting properties in the same loop? See "Getting and Setting Properties" on page 24-10.

Judicious Object Creation

In this section...
"Object Overhead" on page 24-5
"Do Not Create Unnecessary Objects" on page 24-5
"Use NaNs to Simulate Multiple Lines" on page 24-5
"Modify Data Instead of Creating New Objects" on page 24-6

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.

```matlab
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.

```matlab
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 18-4

Screen Updates

In this section...
"MATLAB Graphics System" on page 24-8
"Managing Updates" on page 24-8

MATLAB Graphics System

MATLAB graphics is implemented using multiple threads of execution. The following diagram illustrates how the main and renderer threads interact during the update process. The MATLAB side contains the graphics model, which describes the geometry rendered by the graphics hardware. The renderer side has a copy of the geometry in its own memory system. The graphics hardware can render the screen without blocking MATLAB execution.
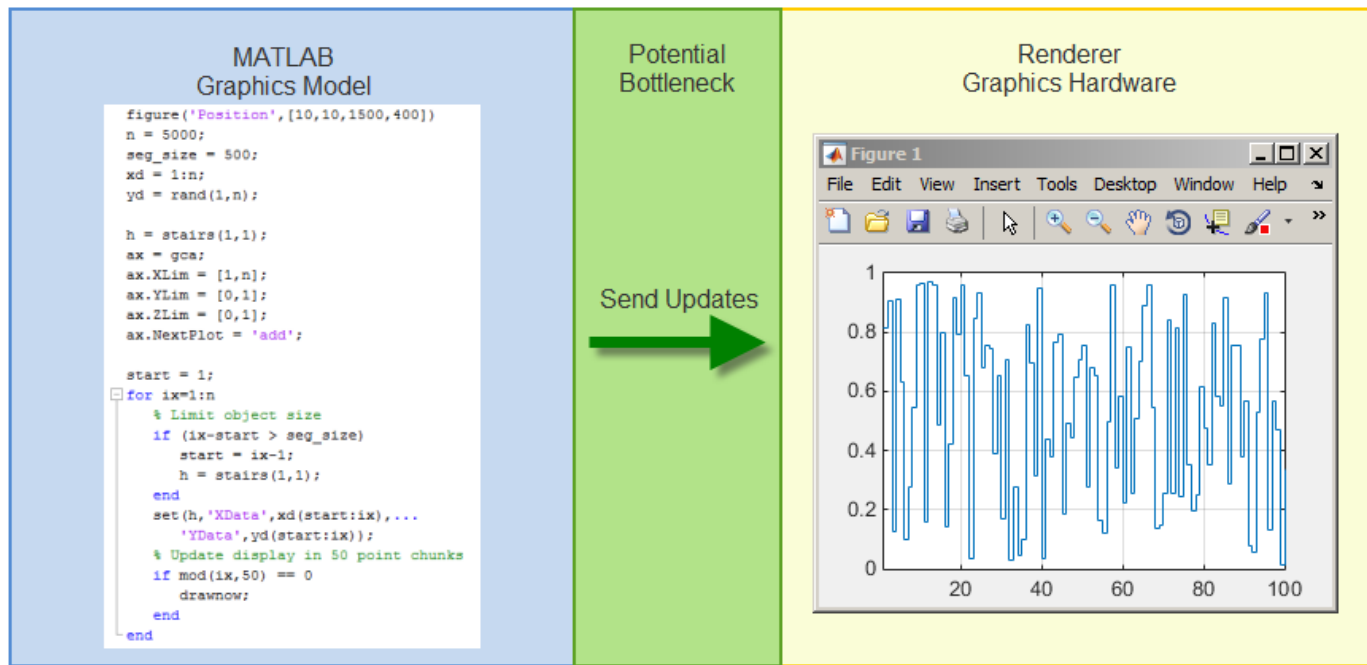
When the graphics model changes, these updates must be passed to the graphics hardware. Sending updates can be a bottleneck because the graphics hardware does not support all MATLAB data types. The update process must convert the data into the correct form.

When geometry is in the graphics hardware memory, you can realize performance advantages by using this data and minimizing the data sent in an update.

Managing Updates

Updates involve these steps:

- Collecting changes that require an update to the screen, such as property changes and objects added.

- Updating dependencies within the graphics model.
- Sending these updates to the renderer.
- Waiting for the renderer to accept these updates before returning execution to MATLAB.

You initiate an update by calling the `drawnow` function. `drawnow` completes execution when the renderer accepts the updates, which can happen before the renderer completes updating the screen.

Explicit Updates

During function execution, adding graphics objects to a figure or changing properties of existing objects does not necessarily cause an immediate update of the screen. The update process occurs when there are changes to graphics that need to be updated, and the code:

- Calls `drawnow`, `pause`, `figure`, or other functions that effectively cause an update (see `drawnow`).
- Queries a property whose value depends on other properties (see "Automatically Calculated Properties" on page 24-10).
- Completes execution and returns control to the MATLAB prompt or debugger.

Getting and Setting Properties

In this section...
"Automatically Calculated Properties" on page 24-10
"Inefficient Cycles of Sets and Gets" on page 24-10
"Changing Text Extent to Rotate Labels" on page 24-11

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.

Code with Poor Performance	Code with Better Performance
```matlab\nh = gobjects(1,500);\np = zeros(500,3);\nfor ix = 1:500\n h(ix) = text(ix/500,ix/500,num2str(ix));\nend\ndrawnow\n\n% Gets and sets in the same loop,\n% prompting a full update at each pass\nfor ix = 1:500\n pos = get(h(ix),'Position');\n ext = get(h(ix),'Extent');\n p(ix,:) = [pos(1)+(ext(3)+ext(1)), ...\n pos(2)+ext(2)+ext(4),0];\n set(h(ix),'Position',p(ix,:))\nend\ndrawnow\n```	```matlab\nh = gobjects(1,500);\np = zeros(500,3);\nfor ix = 1:500\n h(ix) = text(ix/500,ix/500,num2str(ix));\nend\ndrawnow\n\n% Get and save property values\nfor ix=1:500\n pos = get(h(ix),'Position');\n ext = get(h(ix),'Extent');\n p(ix,:) = [pos(1)+(ext(3)+ext(1)), ...\n pos(2)+ext(2)+ext(4),0];\nend\n\n% Set the property values and\n% call a drawnow after the loop\nfor ix=1:500\n set(h(ix),'Position',p(ix,:));\nend\ndrawnow\n```
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. • Each set of the `Position` property makes a full update necessary when the next get of the `Extent` property occurs.	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.

Changing Text Extent to Rotate Labels

In cases where you change the text `Extent` property to rotate axes labels, it is more efficient to use the axes properties `XTickLabelRotation`, `YTickLabelRotation`, and `ZTickLabelRotation`.

Avoid Updating Static Data

If only a small portion of the data defining a graphics scene changes with each update of the screen, you can improve performance by updating only the data that changes. The following example illustrates this technique.

Code with Poor Performance	Code with Better Performance
In this example, a marker moves along the surface by creating both objects with each pass through the loop.	Create the surface, then update the XData, YData, and ZData of the marker in the loop. Only the marker data changes in each iteration.

Code with Poor Performance:
```
[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
```

Code with Better Performance:
```
[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
```

Segmenting Data to Reduce Update Times

Consider the case where an object's data grows very large while code executes in a loop, such as a line tracing a signal over time.

With each call to drawnow, the updates are passed to the renderer. The performance decreases as the data arrays grow in size. If you are using this pattern, adopt the segmentation approach described in the example on the right.

Code with Poor Performance	Code with Better Performance
| ```matlab
% 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];
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
``` | ```matlab
% 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];
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. |

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.

Code with Poor Performance	Code with Better Performance
When object data is large, the update bottleneck becomes a limiting factor.	Using `hgtransform` applies the transform on the renderer side of the bottleneck.

```matlab
% 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
```

```matlab
% 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
```

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 24-16
"Recommended System Requirements" on page 24-16
"Upgrade Your Graphics Drivers" on page 24-16
"Graphics Features That Have Specific Requirements" on page 24-16

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 That Have Specific Requirements" on page 24-16. For the best performance, OpenGL 4.0 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 `rendererinfo`.

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 Macintosh systems, the graphics drivers are part of the operating system. Use the latest updates provided.

Graphics Features That Have Specific Requirements

Most graphics features work on all systems. However, support for some graphics features depends on:

- Whether you are using a hardware, basic hardware, or software implementation of the graphics renderer. By default, MATLAB uses hardware-accelerated graphics 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 `rendererinfo`.
- The version of the renderer implementation, for example, OpenGL 2.1.

This table lists the advanced graphics features and the circumstances under which they are supported. For more information on the features, see `rendererinfo`.

Graphics Feature	Hardware OpenGL	Basic Hardware OpenGL	Software OpenGL on Windows	Software OpenGL on Linux	WebGL™
Graphics Smoothing	Supported for OpenGL 2.1 or higher	Supported for OpenGL 2.1 or higher	Not supported	Not supported	Supported
Depth Peel Transparency	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Supported	Supported
Align Vertex Centers	Supported for OpenGL 2.1 or higher	Disabled	Not supported	Not supported	Supported
Hardware-accelerated markers	Supported for OpenGL 4.0 or higher	Disabled	Not supported	Not supported	Supported

See Also

Functions
opengl | rendererinfo

More About

- "Resolving Low-Level Graphics Issues" on page 24-18

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.

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 4.0 or later is recommended. For more information on determining your graphics hardware, see `rendererinfo`.

Choose a Renderer Implementation for Your System

MATLAB displays graphics using a hardware-accelerated, basic hardware-accelerated, or software implementation of the graphics renderer. 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 implementation:

- If your system does not have the necessary graphics hardware.
- If you are using a graphics driver with known issues, an older graphics driver, or graphics virtualization. Update your graphics drivers to the latest versions available.
- If a previous MATLAB session crashed due to a graphics issue. If the previous session was using software OpenGL and crashed, then subsequent sessions use a more stable version of software OpenGL that has fewer capabilities.

The availability of hardware-accelerated graphics when using remote desktop on Windows systems varies. If you try to use hardware-accelerated graphics when it is not supported, 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-accelerated graphics.

To determine which implementation MATLAB is using, call the `rendererinfo` function. For example, this command gets the information for the current axes and stores it in a structure called `info`.

```
info = rendererinfo(gca)
```

This structure also provides the name of the graphics renderer in the `GraphicsRenderer` field. For example, if MATLAB is using hardware-accelerated OpenGL, the field returns `'OpenGL Hardware'`. If it is using software OpenGL, the field returns `'OpenGL Software'`.

Specify OpenGL Implementation for Current Session

To specify an 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:

- The output returned by `info = rendererinfo(gca)`.
- Whether your code runs without error when using software OpenGL.
- Whether your code runs without error on a different computer. Provide the output of `rendererinfo` 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 https://www.mathworks.com/support/contact_us.

See Also
`opengl` | `rendererinfo`

More About
- "System Requirements for Graphics" on page 24-16