CHAPTER 12

# Advanced Plotting Techniques

## KEY TERMS

| | | |
|---|---|---|
| histogram | area plot | plot properties |
| stem plot | bin | core objects |
| pie chart | animation | text box |

## CONTENTS


12.1 Plot Functions and Customizing Plots ..... 428
12.2 3D Plots .... 439
12.3 Core Graphics Objects ..... 444
12.4 Plot Applications ..... 452
12.5 Saving and Printing Plots ..... 457
Summary ..... 458
Common Pitfalls ..... 458
Programming Style Guidelines ..... 458


In Chapter 3, we introduced the use of the function **plot** in the MATLAB® software to get simple, two-dimensional (2D) plots of $x$ and $y$ points represented by two vectors $x$ and $y$. We have also seen some functions that allow customization of these plots. In Chapter 11, we introduced handle classes, object handles, and methods of examining and modifying plot object properties. In this chapter, we will explore other types of plots, ways of customizing plots, and some applications that combine plotting with functions and file input. Additionally, animation, three-dimensional (3D) plots, and core graphics primitives will be introduced. Note that, in R2014b, the default colors were modified, so in versions prior to that the colors in the plots may differ from what is shown.

In the latest versions of MATLAB, the PLOTS tab can be used to very easily create advanced plots. The method is to create the variables in which the data are stored, and then select the PLOTS tab. The plot functions that can be used are then highlighted; simply clicking the mouse on one will plot the data using that function and open up the Figure Window with that plot. For example, by creating $x$ and $y$ variables, and highlighting them in the Workspace Window, the 2D plot types will become visible. If, instead, $x$, $y$, and $z$ variables are highlighted, the 3D plot types will become available. These are extremely fast methods for users to create plots in MATLAB. However, as this text focuses on programming concepts, the programmatic methodologies will be explained in this chapter.



MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00012-X

## 12.1 PLOT FUNCTIONS AND CUSTOMIZING PLOTS

So far, we have used **plot** to create 2D plots and **bar** to create bar charts. We have seen how to clear the Figure Window using **clf**, and how to create and number Figure Windows using **figure**. Labeling plots has been accomplished using **xlabel**, **ylabel**, **title**, and **legend**, and we have also seen how to customize the strings passed to these functions using **sprintf**. The **axis** function changes the axes from the defaults that would be taken from the data in the $x$ and $y$ vectors to the values specified. Finally, the **grid** and **hold** toggle functions print grids or not, or lock the current graph in the Figure Window so that the next plot will be superimposed.

Another function that is very useful with all types of plots is **subplot**, which creates a matrix of plots in the current Figure Window, as we have seen in Chapter 5. The **sprintf** function is frequently used to create customized axis labels and titles within the matrix of plots.

Besides **plot** and **bar**, there are many other plot types, such as ***histograms***, ***stem plots***, ***pie charts***, and ***area plots***, as well as other functions that customize graphs. Described in this section are some of the other plotting functions.

### 12.1.1 Bar, Barh, Area, and Stem Functions

The functions **bar**, **barh**, **area**, and **stem** essentially display the same data as the **plot** function, but in different forms. The **bar** function draws a bar chart (as we have seen before), **barh** draws a horizontal bar chart, **area** draws the plot as a continuous curve and fills in under the curve that is created, and **stem** draws a stem plot.

For example, the following script creates a Figure Window that uses a $2 \times 2$ **subplot** to demonstrate four plot types using the same data points (see Figure 12.1). Notice how the axes are set by default.

subplottypes.m

```
% Subplot to show plot types

year = 2016:2020;
pop = [0.9 1.4 1.7 1.3 1.8];
subplot(2,2,1)
plot(year,pop)
title('plot')
xlabel('Year')
ylabel('Population (mil)')
subplot(2,2,2)
```

```
bar(year,pop)
title('bar')
xlabel('Year')
ylabel('Population (mil)')
subplot(2,2,3)
area(year,pop)
title('area')
xlabel('Year')
ylabel('Population (mil)')
subplot(2,2,4)
stem(year,pop)
title('stem')
xlabel('Year')
ylabel('Population (mil)')
```

**Note**

The third argument in the call to the **subplot** function is a single index into the matrix created in the Figure Window; the numbering is rowwise (in contrast to the normal columnwise unwinding that MATLAB uses for matrices).

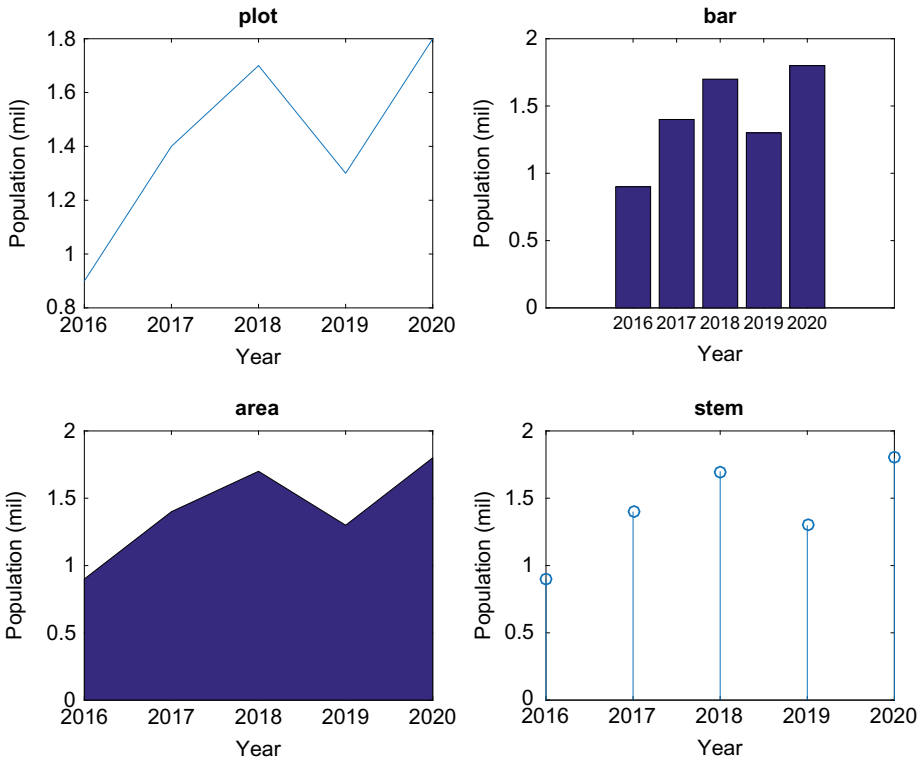


**FIGURE 12.1**
Subplot to display **plot**, **bar**, **area**, and **stem** plots.

## QUICK QUESTION!

Could we produce this **subplot** using a loop?

**Answer:** Yes, we can store the names of the plots in a cell array. These names are put in the titles, and also converted to function handles so that the functions can be called.

loopsubplot.m

```
% Demonstrates evaluating plot type names in order to
% use the plot functions and put the names in titles

year = 2013:2017;
pop = [0.9   1.4   1.7   1.3   1.8];
titles = {'plot', 'bar', 'area', 'stem'};
for i = 1:4
    subplot(2,2,i)
    fn = str2func(titles{i});
    fn(year,pop)
    title(titles{i})
    xlabel('Year')
    ylabel('Population (mil)')
end
```

## QUICK QUESTION!

What are some different options for plotting more than one graph?

**Answer:** There are several methods, depending on whether you want them in one Figure Window superimposed (using **hold on**), in a matrix in one Figure Window (using **subplot**), or in multiple Figure Windows (using **figure(n)**).

For a matrix, the **bar** and **barh** functions will group together the values in each row. For example:

```
>> groupages = [8 19 43 25; 35 44 30 45]
groupages =
      8     19     43     25
     35     44     30     45
>> bar(groupages)
>> xlabel('Group')
>> ylabel('Ages')
```

produces the plot shown in Figure 12.2.

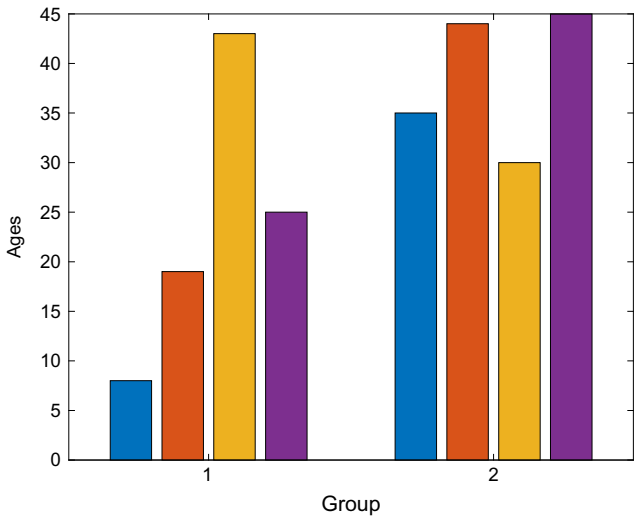


**FIGURE 12.2**
Data from a matrix in a bar chart.

**Note**
The colors used for the bars changed as of R2017b.

```
>> bar(groupages,'stacked')
>> xlabel('Group')
>> ylabel('Ages')
```

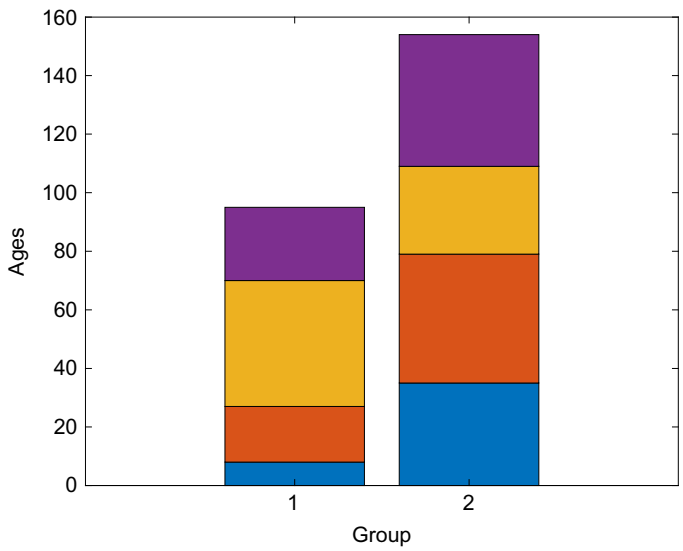


**FIGURE 12.3**
Stacked bar chart of matrix data.

**Note**
MATLAB groups together the values in the first row and then in the second row. It cycles through colors to distinguish the bars. The 'stacked' option will stack rather than group the values, so the "*y*" value represented by the top of the bar is the sum of the values from that row (shown in Figure 12.3).

## PRACTICE 12.1

Create a file that has two lines with *n* numbers in each. Use **load** to read this into a matrix. Then, use **subplot** to show the **barh** and stacked **bar** charts side by side. Put labels 'Groups' for the two groups and 'Values' for the data values on the axes (note the difference between the *x* and *y* labels for these two plot types).

### 12.1.2 Histograms and Pie Charts

A ***histogram*** is a particular type of bar chart that shows the frequency of occurrence of values within a vector. Histograms use what are called ***bins*** to collect values that are in given ranges. MATLAB has a function to create a histogram, **histogram**. Calling the function with the form **histogram(vec)** by default takes the values in the vector *vec* and puts them into bins; the number of bins is determined by the **histogram** function (or **histogram(vec,n)** will put them into *n* bins) and plots this, as shown in Figure 12.4.

```
>> quizzes = [10 8 5 10 10 6 9 7 8 10 1 8];
>> histogram(quizzes)
>> xlabel('Grade')
>> ylabel('#')
>> title('Quiz Grades')
```

In this example, the numbers range from 1 to 10 in the vector, and there are 10 bins in the range from 1 to 10. The heights of the bins represent the number of values

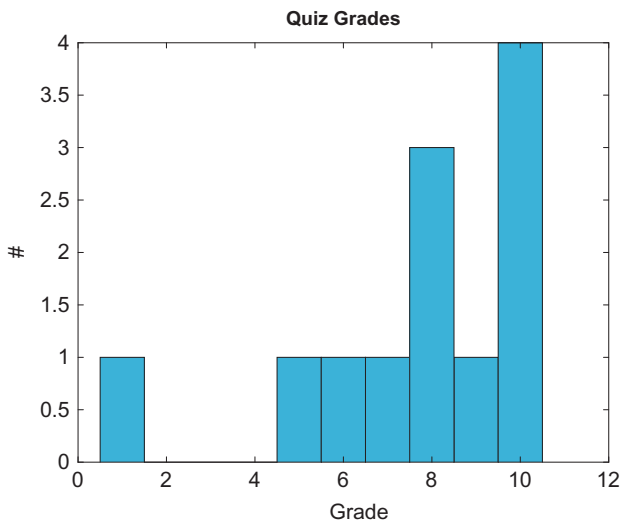


**FIGURE 12.4**
Histogram of data.

that fall within that particular bin. The handle of a **histogram** can also be stored in an object variable; the properties can then be inspected and/or modified.

```
>> hhan = histogram(quizzes)
hhan =
  Histogram with properties:
            Data: [10 8 5 10 10 6 9 7 8 10 1 8]
          Values: [1 0 0 0 1 1 1 3 1 4]
         NumBins: 10
        BinEdges: [0.5000 1.5000 2.5000 3.5000 4.5000
                   5.5000 6.5000 7.5000 8.5000 9.5000 10.5000]
        BinWidth: 1
       BinLimits: [0.5000 10.5000]
   Normalization: 'count'
       FaceColor: 'auto'
       EdgeColor: [0 0 0]
  Show all properties
```

Histograms are used for statistical analyses on data; more statistics will be covered in Chapter 14.

MATLAB has a function, **pie**, that will create a pie chart. Calling the function with the form **pie(vec)** draws a pie chart using the percentage of each element of *vec* of the whole (the sum). It shows these starting from the top of the circle and going around counterclockwise. For example, the first value in the vector [11 14 8 3 1], 11, is 30% of the sum, 14 is 38% of the sum, and so forth, as shown in Figure 12.5.

```
>> pie([11 14 8 3 1])
```

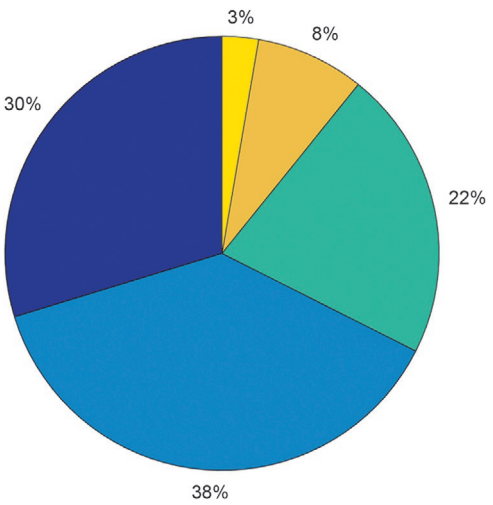


**FIGURE 12.5**
Pie chart showing percentages.

Labels can also be passed to the **pie** function; these labels will appear instead of the percentages (shown in Figure 12.6). The labels can be either a cell array of character vectors or a string array.

```
>> pie([11 14 8 3 1], {'A','B','C','D', 'F'})
```

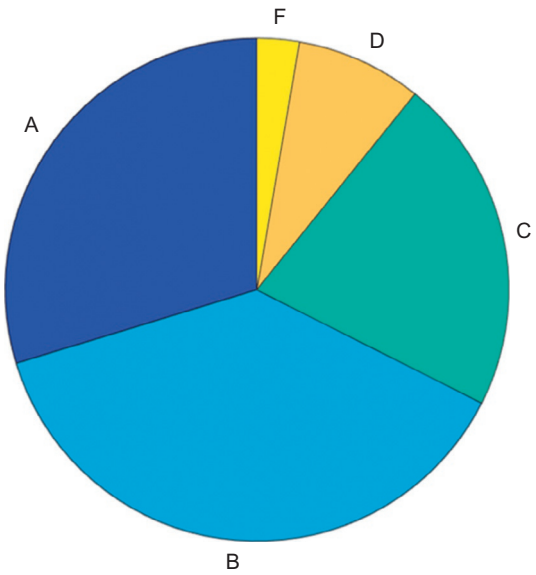


**FIGURE 12.6**
Pie chart with labels from a cell array.

## PRACTICE 12.2

A chemistry professor teaches three classes. These are the course numbers and enrollments:

```
CH 101    111
CH 105     52
CH 555     12
```

Use **subplot** to show this information using **pie** charts: the **pie** chart on the right should show the percentage of students in each course, and on the left the course numbers. Put appropriate titles on them.

### 12.1.3 Log Scales

The **plot** function uses linear scales for both the $x$ and $y$ axes. There are several functions that instead use logarithmic scales for one or both axes: the function **loglog** uses logarithmic scales for both the $x$ and $y$ axes, the function **semilogy** uses a linear scale for the $x$-axis and a logarithmic scale for the $y$-axis, and the

function **semilogx** uses a logarithmic scale for the $x$-axis and a linear scale for the $y$-axis. The following example uses **subplot** to show the difference, for example, between using the **plot** and **semilogy** functions, as seen in Figure 12.7.

```
>> subplot(1,2,1)
>> plot(logspace(1,10))
>> title('plot')
>> subplot(1,2,2)
>> semilogy(logspace(1,10))
>> title('semilogy')
```

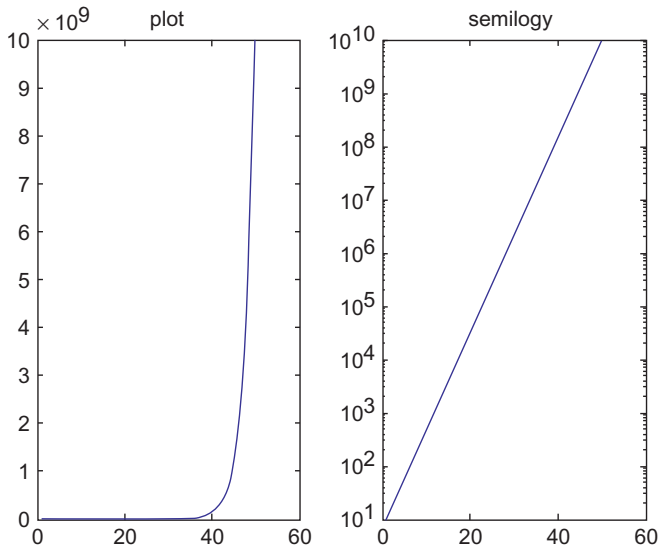


**FIGURE 12.7**
Plot versus semilogy.

## 12.1.4 Animation

There are several ways to ***animate*** a plot. These are visuals, so the results can't really be shown here; it is necessary to type these into MATLAB to see the results.

We'll start by animating a plot of **sin(x)** with the vectors:

```
>> x = -2*pi : 1/100 : 2*pi;
>> y = sin(x);
```

This results in enough points that we'll be able to see the result using the built-in **comet** function, which shows the plot by first showing the point $(x(1),y(1))$, and then moving on to the point $(x(2),y(2))$, and so on, leaving a trail (like a comet!) of all of the previous points.

```
>> comet(x,y)
```

The end result looks similar to the result of **plot(x,y)**, although the color will be different.

Another way of animating is to use the built-in function **movie**, which displays recorded movie frames. The frames are captured in a loop using the built-in function **getframe** and are stored in a matrix. For example, the following script again animates the **sin** function. The **axis** function is used so that MATLAB will use the same set of axes for all frames, and using the **min** and **max** functions on the data vectors $x$ and $y$ will allow us to see all points. It displays the "movie" once in the **for** loop, and then again when the movie function is called.

sinmovie.m

```
% Shows a movie of the sin function
clear

x = -2*pi: 1/5 : 2*pi;
y = sin(x);
n = length(x);

for i = 1:n
    plot(x(i),y(i),'r*')
    axis([min(x)-1 max(x)+1 min(y)-1 max(y)+1])
    M(i) = getframe;
end
movie(M)
```

### 12.1.5 Customizing Plots

There are many ways to customize figures in the Figure Window. Clicking on the Plot Tools icon in the Figure Window itself will bring up the Property Editor and Plot Browser, with many options for modifying the current plot. Additionally, there are ***plot properties*** that can be modified from the defaults in the plot functions. Bringing up the documentation page for the function name will show all of the options for that particular plot function.

For example, the **bar** and **barh** functions by default have a "width" of 0.8 for the bars. When called as **bar(x,y)**, the width of 0.8 is used. If, instead, a third argument is passed, it is the width, for example, **barh(x,y,width)**. The following script uses **subplot** to show variations on the width. A width of 0.6 results in more space between the bars since the bars are not as wide. A width of 1 makes the bars touch each other, and with a width greater than 1, the bars actually overlap. The results are shown in Figure 12.8.

barwidths.m

```
% Subplot to show varying bar widths

year = 2016:2020;
pop = [0.9 1.4 1.7 1.3 1.8];

for i = 1:4
    subplot(1,4,i)
    % width will be 0.6, 0.8, 1, 1.2
    barh(year,pop,0.4+i*.2)
    title(sprintf('Width = %.1f',0.4+i*.2))
    xlabel('Population (mil)')
    ylabel('Year')
end
```

Alternatively, the BarWidth property can be modified.

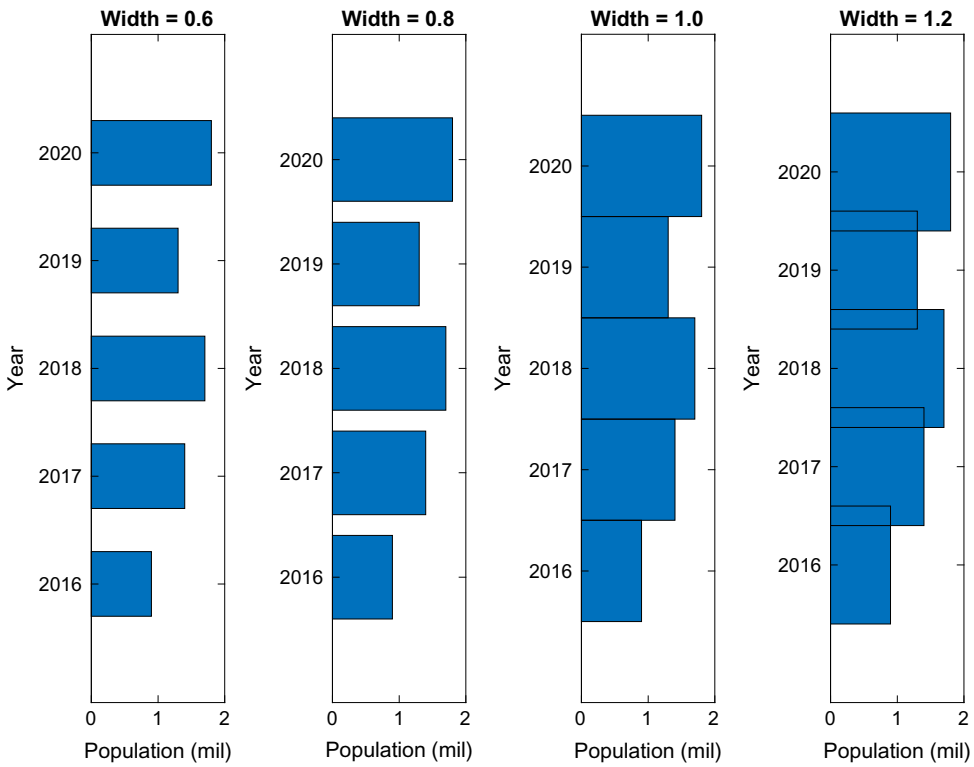


**FIGURE 12.8**
Subplot demonstrates varying widths in a bar chart.

## PRACTICE 12.3

Use **help area** to find out how to change the base level on an **area** chart (from the default of 0).

As another example of customizing plots, pieces of a pie chart can be "exploded" from the rest. In this case, two vectors are passed to the **pie** function: first the data vector, then a **logical** vector; the elements for which the **logical** vector is **true** will be exploded from (separated from) the pie chart. A third argument, a cell array of labels, can also be passed. The result is seen in Figure 12.9.

```
>> gradenums = [11 14 8 3 1];
>> letgrades = {'A','B','C','D','F'};
>> which = gradenums == max(gradenums)
which =
     0     1     0     0     0
>> pie(gradenums,which,letgrades)
>> title(sprintf('Largest Fraction of Grades: %c',...
         letgrades{which}))
```

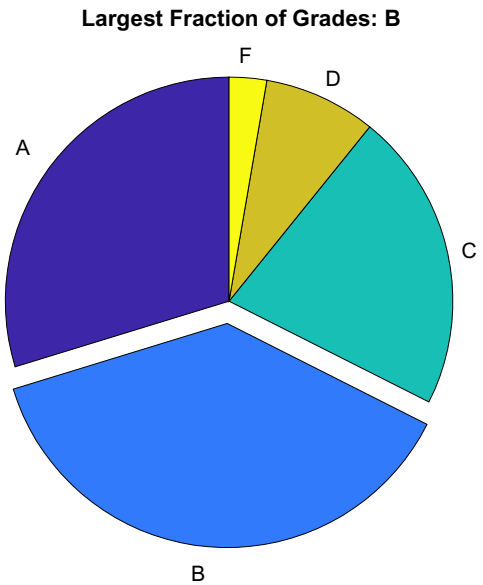


**FIGURE 12.9**
Exploding pie chart.

## 12.2 3D PLOTS

MATLAB has functions that will display 3D plots. Many of these functions have the same name as the corresponding 2D plot function with a '3' at the end. For example, the 3D line plot function is called **plot3**. Other functions include **bar3**, **bar3h**, **pie3**, **comet3**, and **stem3**.

Vectors representing *x*, *y*, and *z* coordinates are passed to the **plot3** and **stem3** functions. These functions show the points in 3D space. Clicking on the rotate 3D icon and then in the plot allows the user to rotate to see the plot from different angles. Also, using the **grid** function makes it easier to visualize, as shown in Figure 12.10. The function **zlabel** is used to label the *z*-axis.

```
>> x = 1:5;
>> y = [0 -2 4 11 3];
>> z = 2:2:10;
>> plot3(x,y,z,'k*')
>> grid
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
>> title('3D Plot')
```

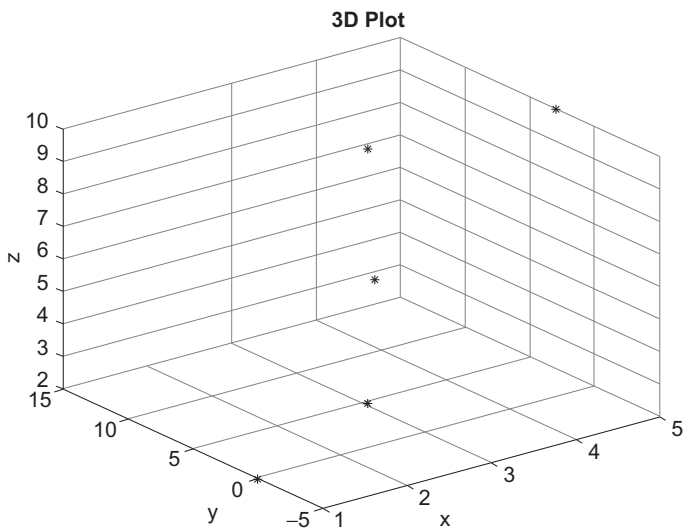


**FIGURE 12.10**
Three-dimensional plot with a grid.

For the **bar3** and **bar3h** functions, *y* and *z* vectors are passed and the function shows 3D bars as shown, for example, for **bar3** in Figure 12.11.

```
>> y = 1:6;
>> z = [33 11 5 9 22 30];
>> bar3(y,z)
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
>> title('3D Bar')
```

A matrix can also be passed, for example a $5 \times 5$ **spiral** matrix (which "spirals" the integers 1 to 25 or more generally from 1 to $n^2$ for **spiral(n)**) as shown in Figure 12.12.

```
>> mat = spiral(5)
mat =
    21    22    23    24    25
    20     7     8     9    10
    19     6     1     2    11
    18     5     4     3    12
    17    16    15    14    13
>> bar3(mat)
>> title('3D Spiral')
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
```

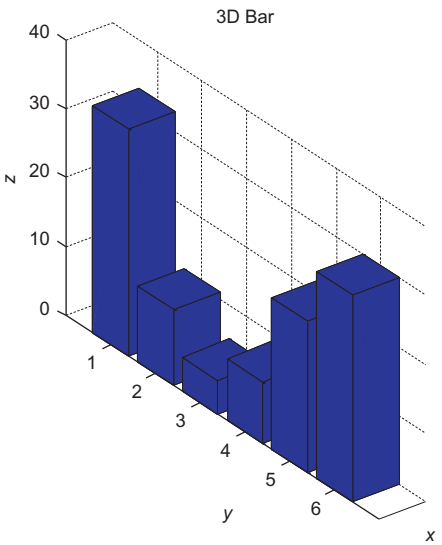


**FIGURE 12.11**
Three-dimensional bar chart.

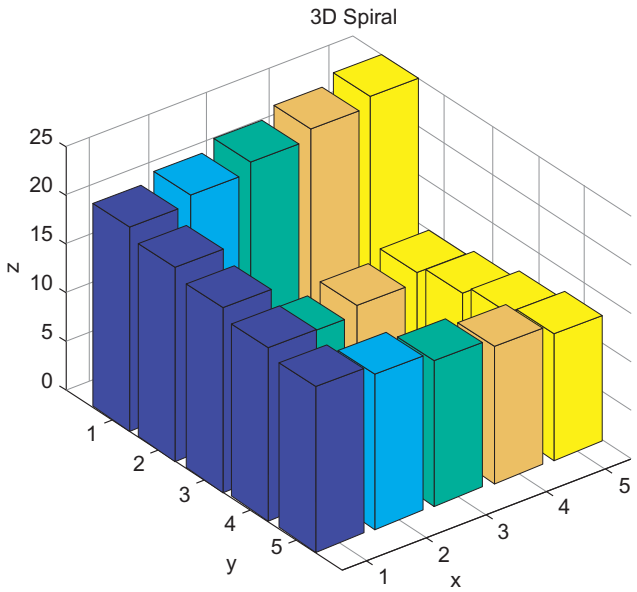


**FIGURE 12.12**
Three-dimensional bar chart of a spiral matrix.

Similarly, the **pie3** function shows data from a vector as a 3D pie, as shown in Figure 12.13.

```
>> pie3([3 10 5 2])
```

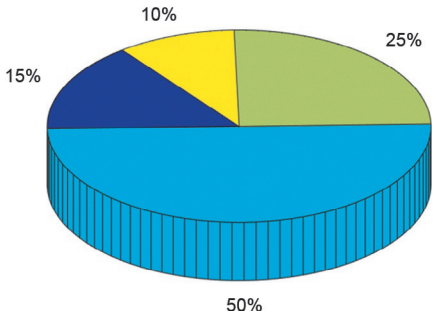


**FIGURE 12.13**
Three-dimensional pie chart.

Displaying the result of an animated plot in three dimensions is interesting. For example, try the following using the **comet3** function:

```
>> t = 0:0.001:12*pi;
>> comet3(cos(t), sin(t), t)
```

Other interesting 3D plot types include **mesh** and **surf**. The **mesh** function draws a wireframe mesh of 3D points, whereas the **surf** function creates a surface plot by using color to display the parametric surfaces defined by the points. MATLAB has several functions that will create the matrices used for the $(x,y,z)$ coordinates for specified shapes (e.g., **sphere** and **cylinder**).

For example, passing an integer $n$ to the **sphere** function creates $n+1 \times n+1$ $x$, $y$, and $z$ matrices, which can then be passed to the **mesh** function (Figure 12.14) or the **surf** function (Figure 12.15).

```
>> [x,y,z] = sphere(15);
>> size(x)
ans =
    16    16
>> mesh(x,y,z)
>> title('Mesh of sphere')
```

Additionally, the **colorbar** function displays a colorbar to the right of the plot, showing the range of colors.

```
>> [x,y,z] = sphere(15);
>> sh = surf(x,y,z);
>> title('Surf of sphere')
>> colorbar
```

**Note**
More options for colors will be described in Chapter 13.

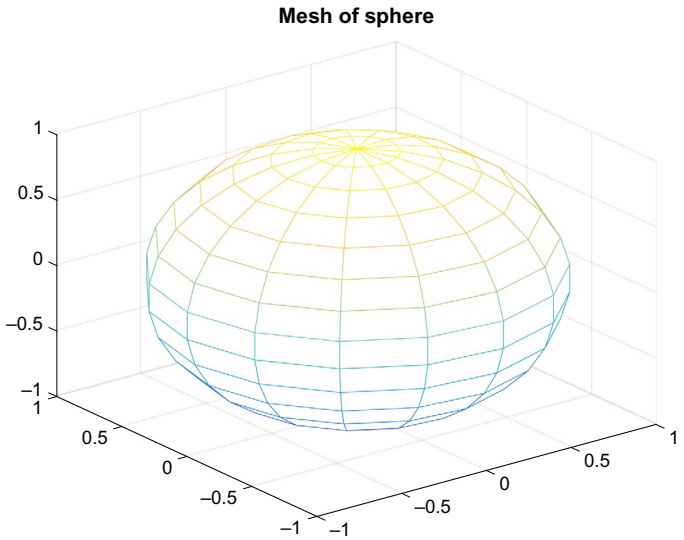


**FIGURE 12.14**
Mesh plot of sphere.

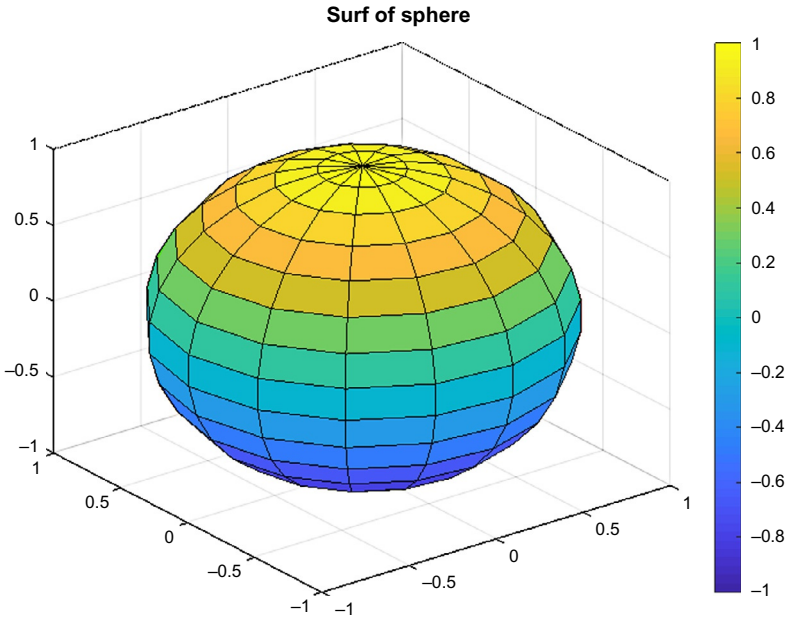


**FIGURE 12.15**
Surf plot of sphere.

One of the properties of the object stored in *sh* is the FaceAlpha property, which is a measure of the transparency. The result of modifying it to 0.5 is shown in Figure 12.16.

```
>> sh
sh =
  Surface with properties:

      EdgeColor: [0 0 0]
      LineStyle: '-'
      FaceColor: 'flat'
  FaceLighting: 'flat'
     FaceAlpha: 0.5000
          XData: [16x16 double]
          YData: [16x16 double]
          ZData: [16x16 double]
          CData: [16x16 double]

  Show all properties
>> sh.FaceAlpha = 0.5;
```

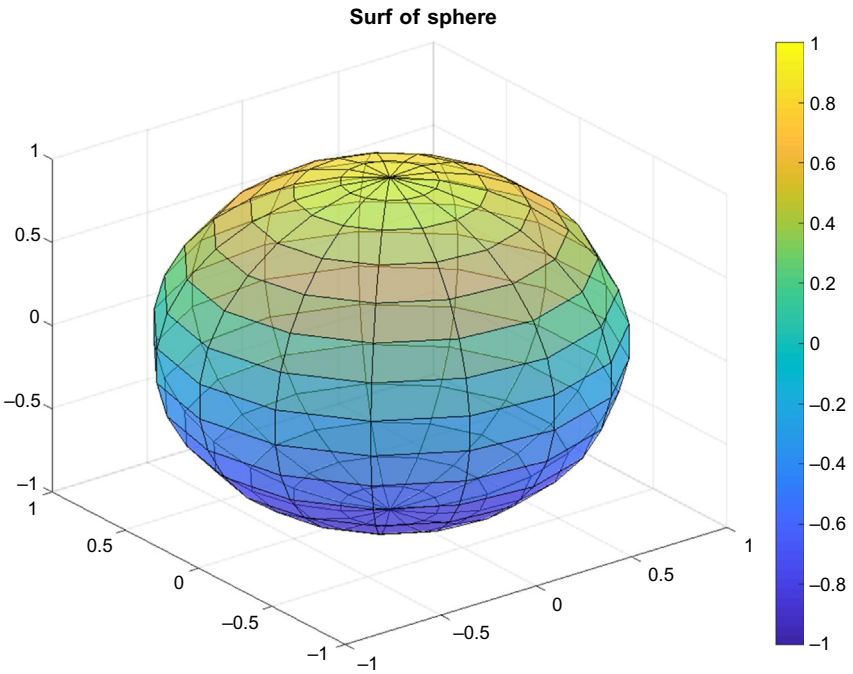


**FIGURE 12.16**
Surf plot of sphere with FaceAlpha modified.

The **meshgrid** function can be used to create $(x,y)$ points for which $z = f(x,y)$; then the $x$, $y$, and $z$ matrices can be passed to **mesh** or **surf**. For example, the following creates a surface plot of the function $\cos(x)+\sin(y)$, as seen in Figure 12.17.

```
>> [x, y] = meshgrid(-2*pi: 0.1: 2*pi);
>> z = cos(x) + sin(y);
>> surf(x,y,z)
>> title('cos(x) + sin(y)')
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
```

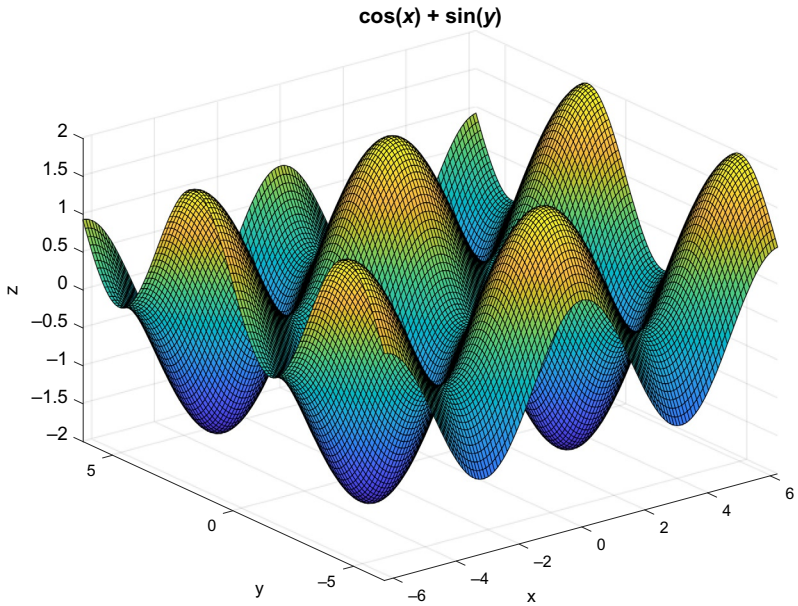


**FIGURE 12.17**
Use of **meshgrid** for *f*(*x*,*y*) points.

## 12.3 CORE GRAPHICS OBJECTS

***Core objects*** in MATLAB are the very basic graphics primitives. A description can be found under the MATLAB help. In the documentation, search for Graphics and Graphics Objects. The core objects include:

- **line**
- **text**
- **rectangle**

- **patch**
- **image**

These are all built-in functions; **help** can be used to find out how each function is used. The first four of these core objects will be described in this section; images will be described in Section 13.1.

A **line** is a core graphics object, which is what is used by the **plot** function. The following is an example of creating a line object, setting some properties, and saving the handle in a variable *hl*:

```
>> x = -2*pi: 1/5 : 2*pi;
>> y = sin(x);
>> hl = line(x,y,'LineWidth', 6, 'Color', [0.5 0.5 0.5])
hl =
  Line with properties:

              Color: [0.5000 0.5000 0.5000]
          LineStyle: '-'
          LineWidth: 6
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x63 double]
              YData: [1x63 double]
              ZData: [1x0 double]

  Show all properties
>>
```

As seen in Figure 12.18, this draws a reasonably thick gray line for the **sin** function. As before, the handle will be valid as long as the Figure Window is not closed.

As another example, the following uses the **line** function to draw a circle. First, a white Figure Window is created. The x and y data points are generated, and then the **line** function is used, specifying a dotted red line with a line width of 4. The **axis** function is used to make the axes square, so the result looks like a circle, but then removes the axes from the Figure Window (using **axis square** and **axis off**, respectively). The result is shown in Figure 12.19.

```
>> figure('Color',[1 1 1])
>> pts = 0:0.1:2*pi;
>> xcir = cos(pts);
>> ycir = sin(pts);
>> line(xcir, ycir, 'LineStyle',':', ...
       'LineWidth',4,'Color','r')
>> axis square
>> axis off
```

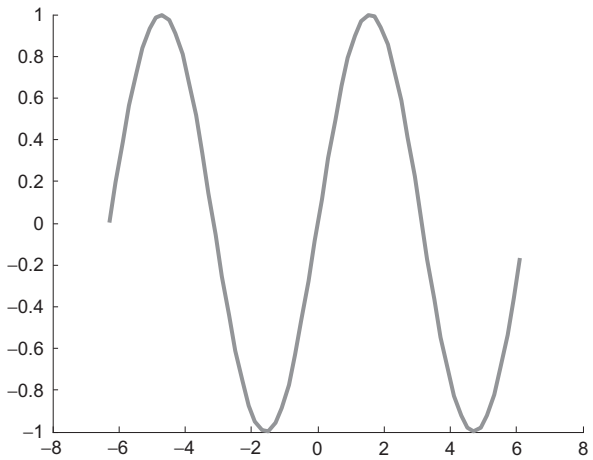


**FIGURE 12.18**
A **line** object with modified line width and color.

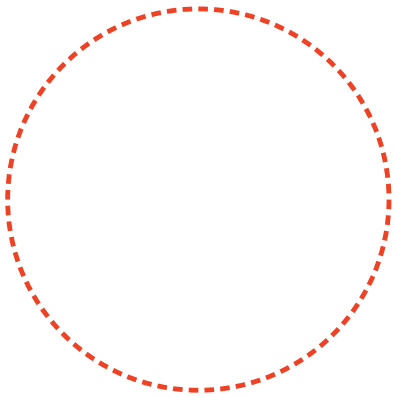

**FIGURE 12.19**
Use of line to draw a circle.

The **text** graphics function allows text to be printed in a Figure Window, including special characters that are printed using `\specchar`, where "specchar" is the actual name of the special character. The format of a call to the **text** function is

```
text(x,y,'text string')
```

where *x* and *y* are the coordinates on the graph of the lower left corner of the ***text box*** in which the text string appears. The special characters include letters of the Greek alphabet, arrows, and characters frequently used in equations. For example, Figure 12.20 displays the Greek symbol for **pi** and a right arrow within the text box.

```
>> x = -4:0.2:4;
>> y = sin(x);
>> hp = line(x,y,'LineWidth',3);
>> thand = text(2,0,'Sin(\pi)\rightarrow')
thand =

  Text (Sin(\pi)\rightarrow) with properties:

                    String: 'Sin(\pi)\rightarrow'
                  FontSize: 10
                FontWeight: 'normal'
                  FontName: 'Helvetica'
                     Color: [0 0 0]
    HorizontalAlignment: 'left'
                  Position: [2 0 0]
                     Units: 'data'

  Show all properties
```

Some of the other properties are shown here.

```
>> thand.Extent
ans =
    2.0000  -0.0380   0.6636   0.0730
>> thand.EdgeColor
ans =
    'none'
>> thand.BackgroundColor
ans =
    'none'
```

Although the Position specified was (2,0), the Extent is the actual extent of the text box, which cannot be seen as the BackgroundColor and EdgeColor are not specified. These can be modified. For example, the following produces the result seen in Figure 12.21.

```
>> thand.BackgroundColor = [0.8 0.8 0.8];
>> thand.EdgeColor = [1 0 0];
```

The **gtext** function allows you to move your mouse to a particular location in a Figure Window, indicating where text should be displayed. As the mouse is moved into the Figure Window, cross hairs indicate a location; clicking on the mouse will display the text in a box with the lower left corner at that location. The **gtext** function uses the **text** function in conjunction with **ginput**, which allows you to click the mouse at various locations within the Figure Window and store the $x$ and $y$ coordinates of these points.

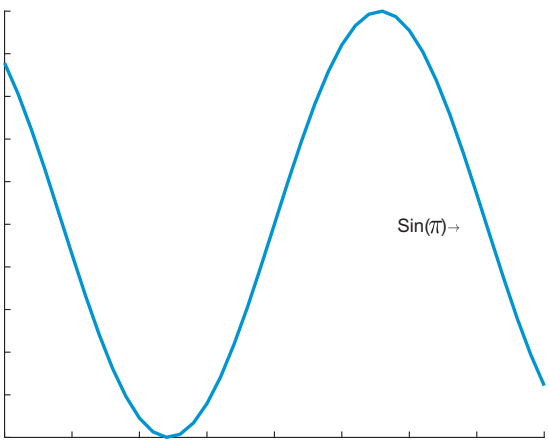


**FIGURE 12.20**
A **line** object with a text box.

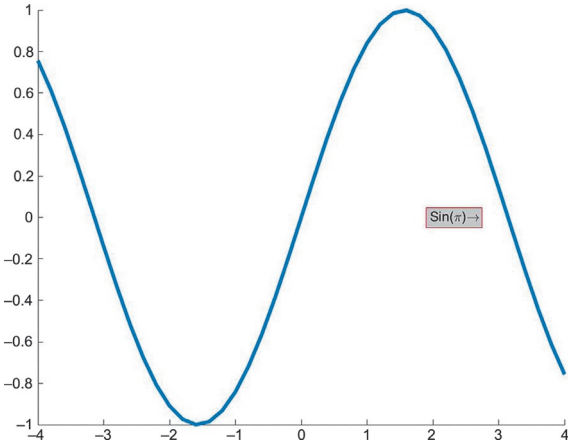


**FIGURE 12.21**
Text box with a modified edge color and background color.

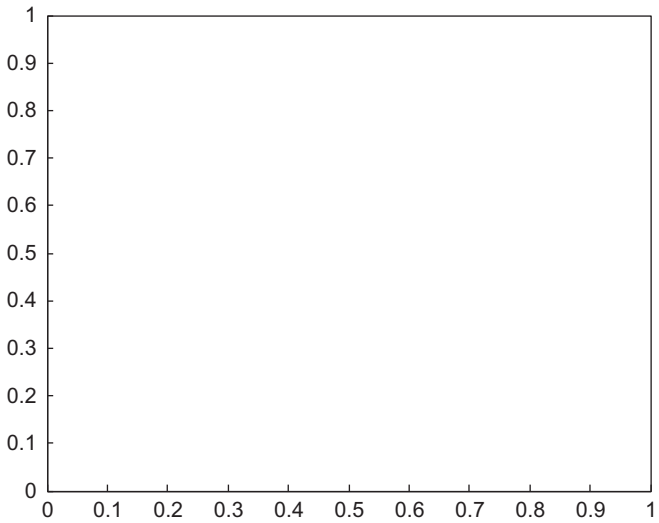


**FIGURE 12.22**
A **rectangle** object.

Another core graphics object is **rectangle**, which can have curvature added to it (!!). Just calling the function **rectangle** without any arguments brings up a Figure Window (shown in Figure 12.22), which, at first glance, doesn't seem to have anything in it:

```
>> recthand = rectangle
recthand =
  Rectangle with properties:

    FaceColor: 'none'
    EdgeColor: [0 0 0]
    LineWidth: 0.5000
    LineStyle: '-'
    Curvature: [0 0]
     Position: [0 0 1 1]

Show all properties
>>
```

The Position of a rectangle is [$x$ $y$ $w$ $h$], where $x$ and $y$ are the coordinates of the lower left point, $w$ is the width, and $h$ is the height. The default rectangle has a Position of [0 0 1 1]. The default Curvature is [0 0], which means no curvature. The values range from [0 0] (no curvature) to [1 1] (ellipse). A more interesting rectangle object is seen in Figure 12.23.

Note that properties can be set when calling the **rectangle** function, and also subsequently using the dot operator, as follows:

```
>> rh = rectangle('Position', [0.2, 0.2, 0.5, 0.8],...
     'Curvature',[0.5, 0.5]);
>> axis([0 1.2 0 1.2])
>> rh.LineWidth = 3;
>> rh.LineStyle = ':';
```

This creates a curved rectangle and uses dotted lines.

The **patch** function is used to create a patch graphics object, which is made from 2D polygons. A simple patch in 2D space, a triangle, is defined by specifying the coordinates of three points as shown in Figure 12.24; in this case, the color red is specified for the polygon.

```
>> x = [0 1 0.5];
>> y = [0 0 1];
>> hp = patch(x,y,'r')
hp =
  Patch with properties:

  FaceColor: [1 0 0]
  FaceAlpha: 1
  EdgeColor: [0 0 0]
  LineStyle: '-'
      Faces: [1 2 3]
   Vertices: [3x2 double]

  Show all properties
```

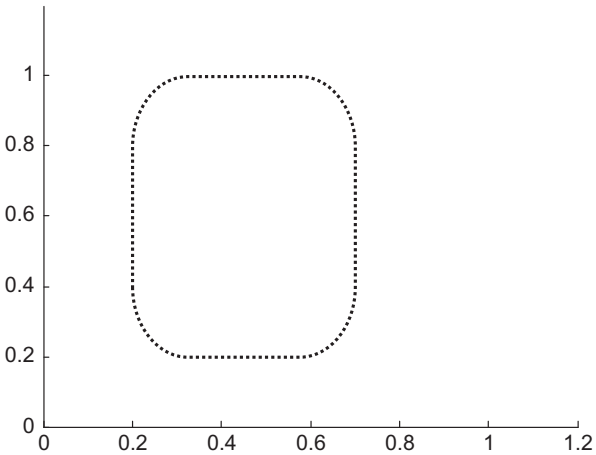


**FIGURE 12.23**
Rectangle object with curvature.

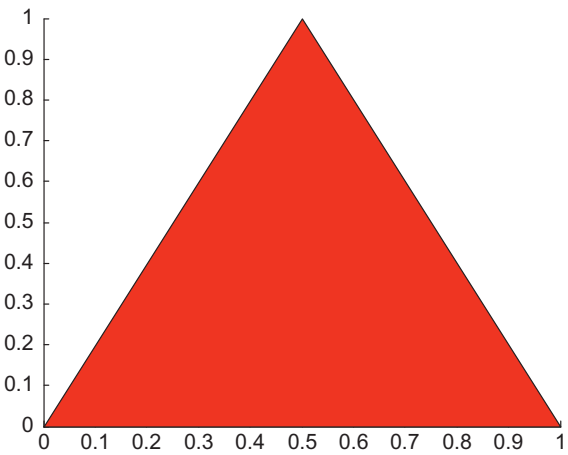


**FIGURE 12.24**
Simple patch.

The Vertices property stores the three points given by the *x* and *y* data vectors.

```
>> hp.Vertices
ans =
        0         0
   1.0000         0
   0.5000    1.0000
```

The Faces property tells how the vertices are connected to create the patch. The vertices are numbered; the first point (0,0) is vertex 1, the point (1,0) is vertex 2, and the last point (0.5, 1) is vertex 3. The Faces property specifies connecting vertex 1 to 2 to 3 (and then by default back to 1). Another method of creating a patch is to specifically use the Vertices and Faces properties.

One way of calling **patch** is `patch(fv)`, where *fv* is a structure variable with field names that are names of properties, and the values are the property values, e.g., fields called *Vertices* and *Faces*. For example, consider a patch object that consists of three connected triangles and has five vertices given by the coordinates:

```
(1)   (0, 0)
(2)   (2, 0)
(3)   (1, 2)
(4)   (1, -2)
(5)   (3, 1)
```

The order in which the points are given is important, as the faces describe how the vertices are linked. To create these vertices in MATLAB and define faces that connect them, we use a structure variable and then pass it to the **patch** function; the result is shown in Figure 12.25.

```
mypatch.Vertices = [...
     0 0
     2 0
     1 2
     1 -2
     3 1];
mypatch.Faces = [
     1 2 3
     2 3 5
     1 2 4];

patchhan = patch(mypatch, 'FaceColor', 'r',...
     'EdgeColor','k');
```

The face color is set to red and the edge color to black.

To vary the colors of the faces of the polygons, the FaceColor property is set to 'flat', which means that every face has a separate color. The *mycolors* variable

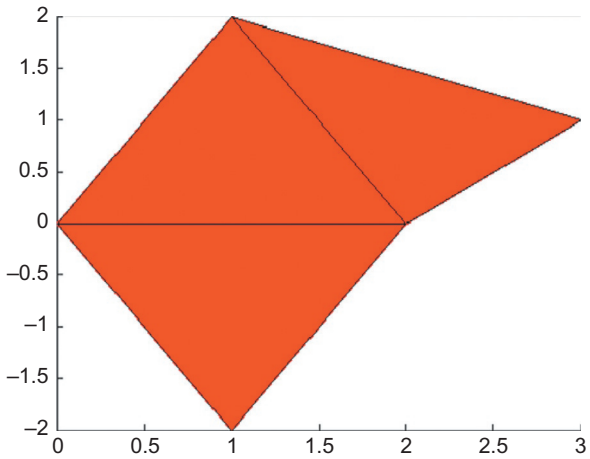


**FIGURE 12.25**
Patch object.

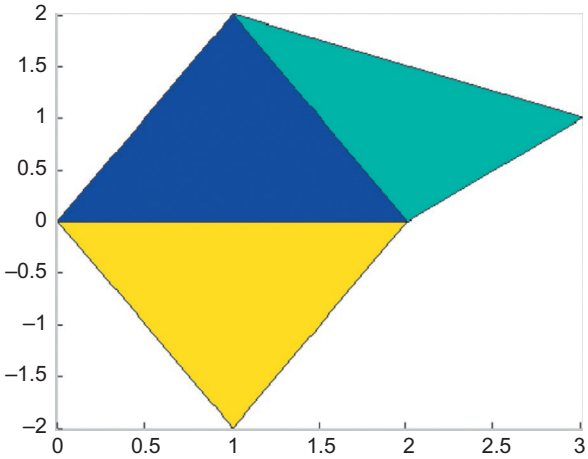


**FIGURE 12.26**
Varying patch colors.

stores three colors in the rows of the matrix by specifying the red, green, and blue components for each; the first is blue, the second is cyan (a combination of green and blue), and the third is yellow (a combination of red and green). The property FaceVertexCData specifies the color data for the vertices, as seen in Figure 12.26.

```
>> mycolors = [0 0 1; 0 1 1; 1 1 0];
>> patchhan = patch(mypatch, 'FaceVertexCData', ...
                mycolors, 'FaceColor','flat');
```

Patches can also be defined in 3D space. For example:

```
polyhedron.Vertices = [...
0 0 0
1 0 0
0 1 0
0.5 0.5 1];

polyhedron.Faces = [...
1 2 3
1 2 4
1 3 4
2 3 4];

pobj = patch(polyhedron, ...
'FaceColor',[0.8, 0.8, 0.8],...
'EdgeColor','black');
```

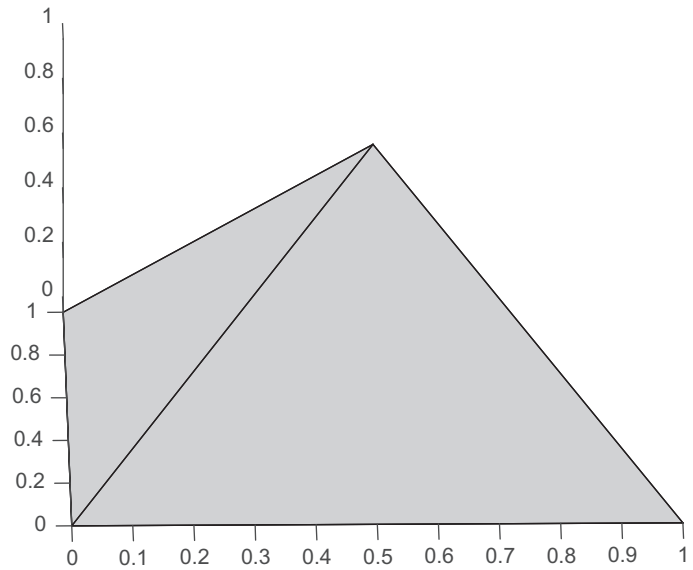


**FIGURE 12.27**
Rotated **patch** object.

The Figure Window initially shows only two faces. Using the rotate icon, the figure can be rotated so that the other edges can be seen as shown in Figure 12.27.

## 12.4 PLOT APPLICATIONS

In this section, we will show some examples that integrate plots and many of the other concepts covered to this point in the book. For example, we will have a function that receives an *x* vector, a function handle of a function used to create the *y* vector, and a cell array of plot types as character vectors and will generate the plots, and we will also show examples of reading data from a file and plotting them.

### 12.4.1 Plotting From a Function

The following function generates a Figure Window (seen in Figure 12.28) that shows different types of plots for the same data. The data are passed as input arguments (as an *x* vector and the handle of a function to create the *y* vector) to the function, as is a cell array with the plot type names. The function generates the Figure Window using the cell array with the plot type names. It creates a function handle for each using the **str2func** function.

plotxywithcell.m

```
function plotxywithcell(x, fnhan, rca)
% plotxywithcell receives an x vector, the handle
% of a function (used to create a y vector), and
% a cell array with plot type names; it creates
% a subplot to show all of these plot types
% Format: plotxywithcell(x,fn handle, cell array)

lenrca = length(rca);
y = fnhan(x);
for i = 1:lenrca
    subplot(1,lenrca,i)
    funh = str2func(rca{i});
    funh(x,y)
    title(upper(rca{i}))
    xlabel('x')
    ylabel(func2str(fnhan))
end
end
```

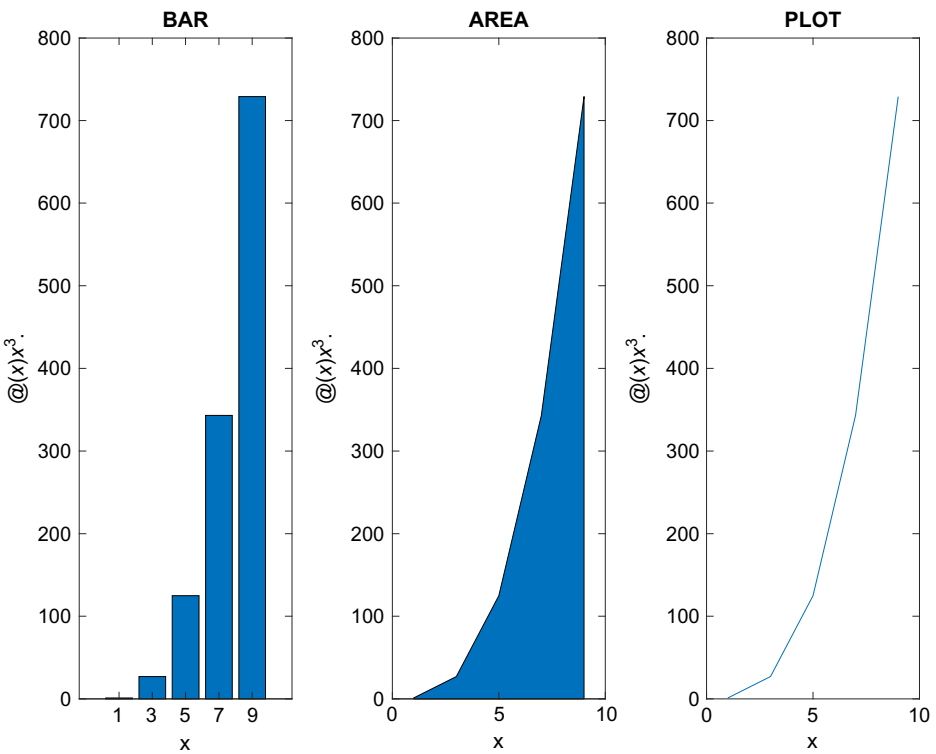


**FIGURE 12.28**
Subplot showing different file types with their names as titles.

For example, the function could be called as follows:

```
>> anfn = @ (x) x .^3;
>> x = 1:2:9;
>> rca = {'bar', 'area', 'plot'};
>> plotxywithcell(x, anfn, rca)
```

The function is general and works for any number of plot types stored in the cell array.

### 12.4.2 Plotting File Data

It is often necessary to read data from a file and plot them. Normally, this entails knowing the format of the file. For example, let us assume that a company has two divisions, A and B. Assume that the file "ab16.dat" contains four lines, with the sales figures (in millions) for the two divisions for each quarter of the year 2016. For example, the file might look like this (and the format will be exactly like this):

```
A5.2B6.4
A3.2B5.5
A4.4B4.3
A4.5B2.2
```

The following script reads in the data and plots the data as bar charts in one Figure Window. The script prints an error message if the file open is not successful or if the file close was not successful. The **axis** command is used to force the *x*-axis to range from 0 to 3 and the *y*-axis from 0 to 8, which will result in the axes shown here. The numbers 1 and 2 would show on the *x*-axis rather than the division labels A and B by default. The **xticklabels** function (introduced in R2016b) changes the XTickLabel property of the axes to use the character vectors in the cell array as labels on the tick marks on the *x*-axis.

plotdivab.m

```
% Reads sales figures for 2 divisions of a company one
% line at a time as strings, and plots the data

fid = fopen('ab16.dat');
if fid == -1
    disp('File open not successful')
else
    for i = 1:4
        % Every line is of the form A#B#; this separates
        % the characters and converts the #'s to actual
        % numbers
        aline = fgetl(fid);
        aline = aline(2:length(aline));
        [compa, rest] = strtok(aline,'B');
```

```
        compa = str2double(compa);
        compb = rest(2:length(rest));
        compb = str2double(compb);

        % Data from every line is in a separate subplot
        subplot(1,4,i)
        bar([compa,compb])
        xticklabels({'A', 'B'})
        axis([0 3 0 8])
        ylabel('Sales (millions)')
        title(sprintf('Quarter %d',i))
    end
    closeresult = fclose(fid);
    if closeresult ~= 0
        disp('File close not successful')
    end
end
```

Running this produces the subplot shown in Figure 12.29.

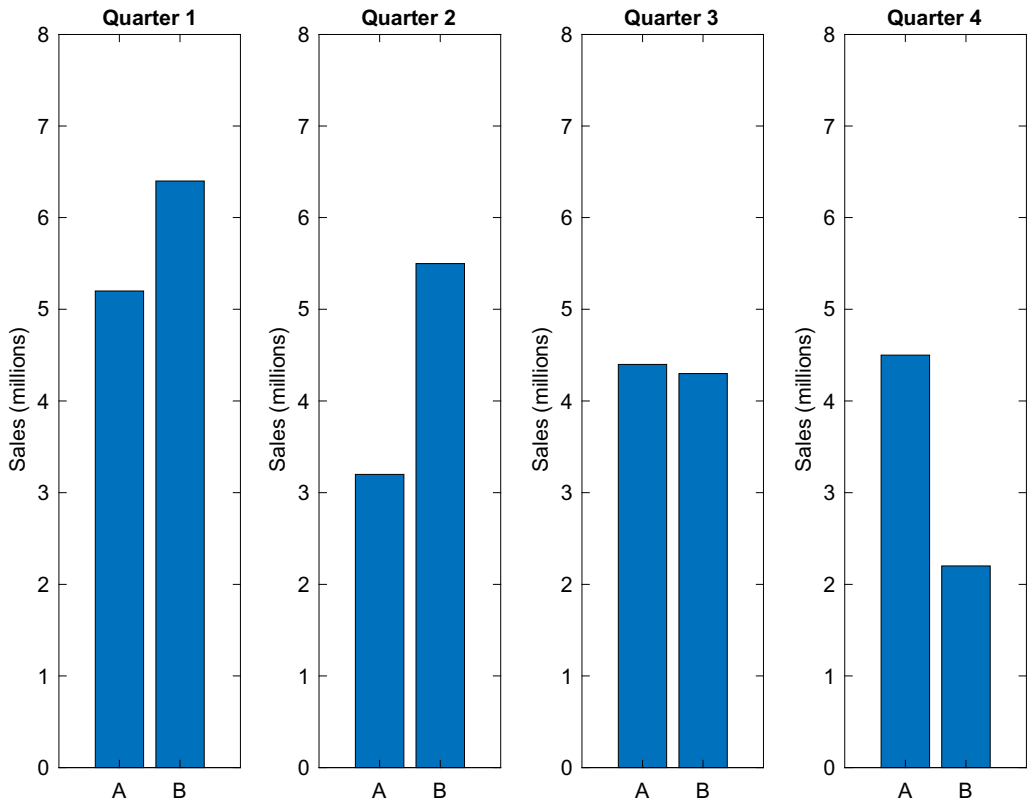


**FIGURE 12.29**
Subplot with customized *x*-axis tick labels.

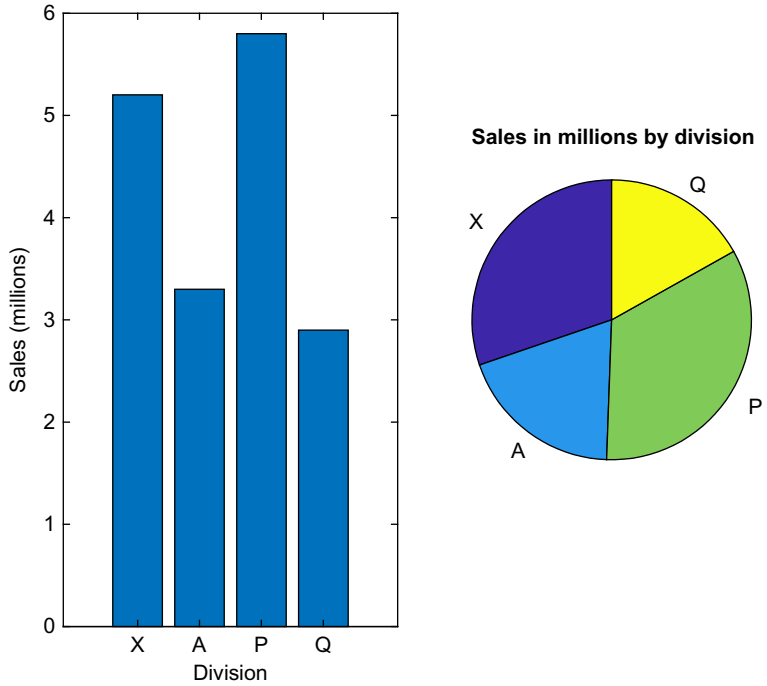


**FIGURE 12.30**
Bar and pie charts with labels from file data.

As another example, a data file called "compsales.dat" stores sales figures (in millions) for divisions in a company. Each line in the file stores the sales number, followed by an abbreviation of the division name, in this format:

```
5.2 X
3.3 A
5.8 P
2.9 Q
```

The script that follows uses the **textscan** function to read this information into a cell array, and then uses **subplot** to produce a Figure Window that displays the information in a **bar** chart and in a **pie** chart (shown in Figure 12.30).

compsalesbarpie.m

```
% Reads sales figures and plots as a bar chart and a pie chart

fid = fopen('compsales.dat');

if fid == -1
```

```
    disp('File open not successful')
else
    % Use textscan to read the numbers and division codes
    % into separate elements in a cell array
    filecell = textscan(fid,'%f %s');
    % plot the bar chart with the division codes on the x ticks
    subplot(1,2,1)
    bar(filecell{1})
    xlabel('Division')
    ylabel('Sales (millions)')
    xticklabels(filecell{2})
    % plot the pie chart with the division codes as labels
    subplot(1,2,2)
    pie(filecell{1}, filecell{2})
    title('Sales in millions by division')

    closeresult = fclose(fid);
    if closeresult ~= 0
       disp('File close not successful')
    end
end
```

## 12.5 SAVING AND PRINTING PLOTS

Once any plot has been created in a Figure Window, there are several options for saving it, printing it, and copying and pasting it into a report. When the Figure Window is open, choosing Edit and then Copy Figure will copy the Figure Window so that it can then be pasted into a word processor. Choosing File and then Save As allows you to save in different formats, including common image types, such as .jpg, .tif, and .png. Another option is to save it as a .fig file, which is a Figure file type used in MATLAB. If the plot was not created programmatically, or the plot properties have been modified using the plot tools icon, choosing File and then Generate Code will generate a script that will recreate the plot.

Choosing File and then Print allows you to print the file on a connected printer. The **print** command can also be used in MATLAB programs. The line

```
print
```

in a script will print the current Figure Window using default formats. Options can also be specified (see the Documentation page on **print** for the options). Also, by specifying a file name, the plot is saved to a file rather than printed.

For example, the following would save a plot as a .tif file with 400 dots per inch in a file named 'plot.tif':

```
print -dtiff -r400 plot.tif
```

### Explore Other Interesting Features

There are many built-in plot functions in MATLAB, and many ways to customize plots. Use the Help facility to find them. Here are some specific suggestions for functions to investigate.

Investigate the **peaks** function, and the use of the resulting matrix as a test for various plot functions.

Investigate how to show confidence intervals for functions using the **errorbar** function.

Find out how to set limits on axes using **xlim**, **ylim**, and **zlim**.

The **plotyy** function allows $y$ axes on both the left and the right of the graph. Find out how to use it, and how to put different labels on the two $y$ axes.

Investigate how to use the **gtext** and **ginput** functions.

Investigate the 3D functions **meshc** and **surfc**, which put contour plots under the mesh and/or surface plots.

Investigate using the **datetick** function to use dates to label tick lines. Note that there are many options!

Investigate the use of **pie** charts with categorical arrays. ■

## SUMMARY

## COMMON PITFALLS

- Closing a Figure Window prematurely—the properties can only be set if the Figure Window is still open!

## PROGRAMMING STYLE GUIDELINES

- Always label plots
- Take care to choose the type of plot in order to highlight the most relevant information

| MATLAB Functions and Commands | | |
|---|---|---|
| barh | plot3 | cylinder |
| area | bar3 | colorbar |
| stem | bar3h | line |
| histogram | pie3 | rectangle |
| pie | comet3 | text |
| loglog | stem3 | patch |
| semilogy | zlabel | image |
| semilogx | spiral | gtext |
| comet | mesh | ginput |
| movie | surf | xticklabels |
| getframe | sphere | print |

## Exercises

1. Create a data file that contains 10 numbers. Write a script that will load the vector from the file, and use **subplot** to do an **area** plot and a **stem** plot with these data in the same Figure Window (Note: a loop is not needed). Prompt the user for a title for each plot.
2. Write a script that will read *x* and *y* data points from a file and will create an **area** plot with those points. The format of every line in the file is the letter '*x*', a space, the *x* value, space, the letter '*y*', space, and the *y* value. You must assume that the data file is in exactly that format, but you may not assume that the number of lines in the file is known. The number of points will be in the plot title. The script loops until the end of file is reached, using **fgetl** to read each line as a character vector.
3. Do a quick survey of your friends to find ice cream flavor preferences. Display this information using a **pie** chart.
4. Create a matrix variable. Display the numbers from the matrix using a subplot as both a **bar** chart and a stacked bar.
5. The number of faculty members in each department at a certain College of Engineering is:

```
ME 22
BM 45
CE 23
EE 33
```

   Experiment with at least 3 different plot types to graphically depict this information. Make sure that you have appropriate titles, labels, and legends on your plots. Which type(s) work best, and why?
6. Create a simple pie chart:

```
>> v = [11 33 5];
>> ph = pie(v)
```

Notice that the result is a graphics array, consisting of 3 patch primitives and 3 text primitives. So, ph is an array that can be indexed. Use the properties to change the face color of one of the patches, e.g., ph(1).

7. Experiment with the **comet** function: try the example given when **help comet** is entered and then animate your own function using **comet**.
8. Experiment with the **comet3** function: try the example given when **help comet3** is entered and then animate your own function using **comet3**.
9. Experiment with the **scatter** and **scatter3** functions.
10. Use the **cylinder** function to create *x*, *y*, and *z* matrices and pass them to the **surf** function to get a surface plot. Experiment with different arguments to **cylinder**.
11. Experiment with **contour** plots.
12. Generate an ellipsoid using the **ellipsoid** function and then plot using **surf**.
13. The electricity generated by wind turbines annually in kilowatt-hours per year is given in a file. The amount of electricity is determined by, among other factors, the diameter of the turbine blade (in feet) and the wind velocity in mph. The file stores on each line the blade diameter, wind velocity, and the approximate electricity generated for the year. For example,

```
5 5 406
5 10 3250
5 15 10970
5 20 26000
10 5 1625
10 10 13000
10 15 43875
10 20 104005
```

Create a file in this format and determine how to graphically display this data.

14. Create an *x* vector, and then two different vectors (*y* and *z*) based on *x*. Plot them with a legend. Use **help legend** to find out how to position the legend itself on the graph, and experiment with different locations.
15. Create an *x* vector that has 30 linearly spaced points in the range from $-2\pi$ to $2\pi$, and then *y* as **sin(x)**. Do a **stem** plot of these points, and store the handle in a variable. Change the face color of the marker.
16. When an object with an initial temperature *T* is placed in a substance that has a temperature *S*, according to Newton's law of cooling, in *t* minutes it will reach a temperature $T_t$ using the formula $T_t = S + (T - S)\, e^{(-kt)}$ where k is a constant value that depends on properties of the object. For an initial temperature of 100 and $k = 0.6$, graphically display the resulting temperatures from 1 to 10 minutes for two different surrounding temperatures: 50 and 20. Use the **plot** function to plot two different lines for these surrounding temperatures and store the handle in a variable. Note that two function handles are actually returned and stored in a vector. Change the line width of one of the lines.

17. Write a script that will draw the line $y = x$ between $x = 2$ and $x = 5$, with a random line width between 1 and 10.
18. Write a script that will plot the data points from *y* and *z* data vectors, and store the handles of the two plots in variables *yhand* and *zhand*. Set the line widths to 3 and 4, respectively. Set the colors and markers to random values (create character vectors containing possible values and pick a random index).
19. Write a function "plotexvar" that will plot data points represented by *x* and *y* vectors which are passed as input arguments. If a third argument is passed, it is a line width for the plot, and if a fourth argument is also passed, it is a color. The plot title will include the total number of arguments passed to the function. Here is an example of calling the function; the resulting plot is shown in Figure 12.31.

```
>> x=-pi:pi/50:2*pi;
>> y = sin(x);
>> plotexvar(x,y,12,'r')
```

20. A file *houseafford.dat* stores on its three lines years, median incomes and median home prices for a city. The dollar amounts are in thousands. For example, it might look like this:

```
2004 2005 2006 2007 2008 2009 2010 2011
72 74 74 77 80 83 89 93
250 270 300 310 350 390 410 380
```

    Create a file in this format, and then **load** the information into a matrix. Create a horizontal stacked bar chart to display the information, with an appropriate title. Note: use the 'XData' property to put the years on the axis as shown in Figure 12.32.
21. Write a function that will plot **cos(x)** for *x* values ranging from –pi to pi in steps of 0.1, using black *'s. It will do this three times across in one Figure Window, with varying line widths (Note: even if individual points are plotted rather than a solid line, the line width property will change the size of these points.). If no arguments are passed to the function, the line widths will be 1, 2, and 3. If, on the other hand, an argument is passed to the function, it is a multiplier for these values (e.g., if 3 is passed, the line widths will be 3, 6, and 9). The line widths will be printed in the titles on the plots.
22. Create a graph, and then use the **text** function to put some text on it, including some \specchar commands to increase the font size and to print some Greek letters and symbols.
23. Create a **rectangle** object and use the **axis** function to change the axes so that you can see the rectangle easily. Change the Position, Curvature, EdgeColor, LineStyle, and LineWidth. Experiment with different values for the Curvature.

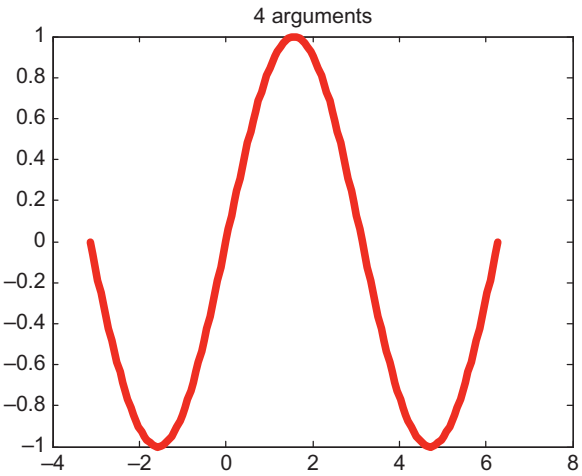


**FIGURE 12.31**
Sin with color and line width.

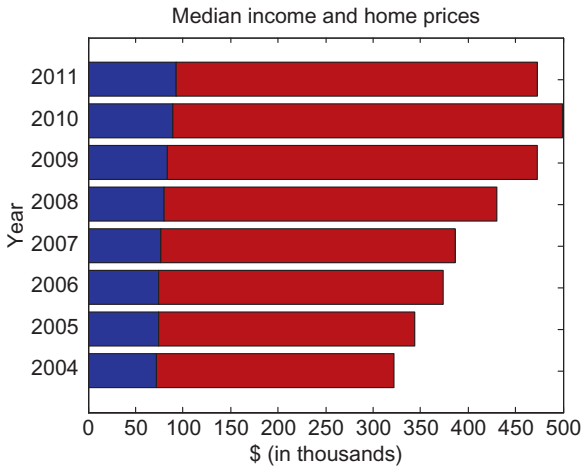


**FIGURE 12.32**
Horizontal stacked bar chart of median incomes and home prices.

24. Write a script that will display rectangles with varying curvatures and line widths, as shown in Figure 12.33. The script will, in a loop, create a 2 by 2 subplot showing rectangles. In all, both the *x* and *y* axes will go from 0 to 1.4. Also, in all, the lower left corner of the rectangle will be at (0.2, 0.2), and the length and width will both be 1. The line width, i, is displayed in the title of each plot. The curvature will be [0.2, 0.2] in the first plot, then [0.4, 0.4], [0.6,0.6], and finally [0.8,0.8].
25. Write a script that will start with a rounded rectangle. Change both the *x* and *y* axes from the default to go from 0 to 3. In a **for** loop, change the position vector by adding 0.1 to all elements 10 times (this will change the location and size of the rectangle each time). Create a movie consisting of the resulting rectangles. The final result should look like the plot shown in Figure 12.34.
26. A hockey rink looks like a rectangle with curvature. Draw a hockey rink, as in Figure 12.35.
27. Write a script that will create a two-dimensional **patch** object with just three vertices and one face connecting them. The *x* and *y* coordinates of the three vertices will be random real numbers in the range from 0 to 1. The lines used for the edges should be black with a width of 3, and the face should be gray. The axes (both *x* and *y*) should go from 0 to 1. For example, depending on what the random numbers are, the Figure Window might look like Figure 12.36.
28. Using the **patch** function, create a black box with unit dimensions (so, there will be eight vertices and six faces). Set the edge color to white so that when you rotate the figure, you can see the edges.

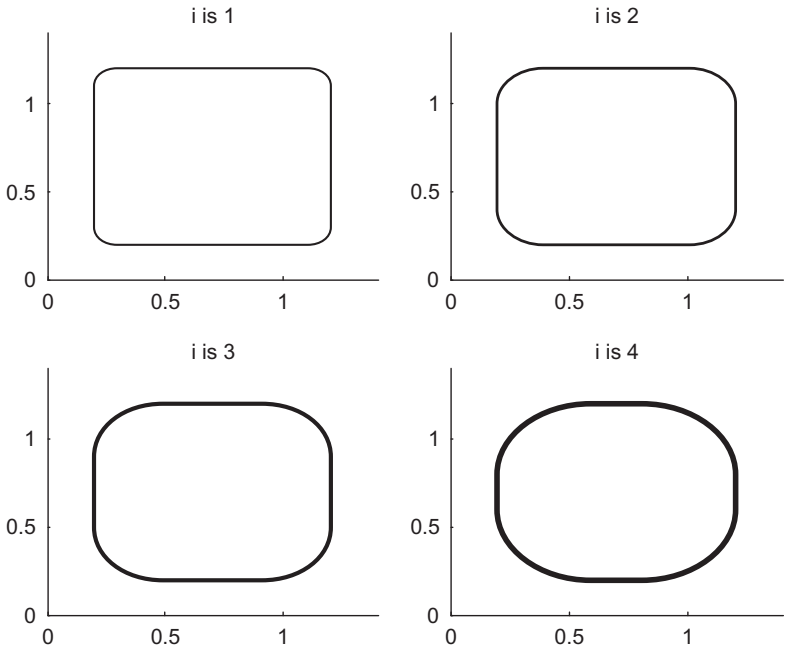


**FIGURE 12.33**
Varying rectangle curvature.

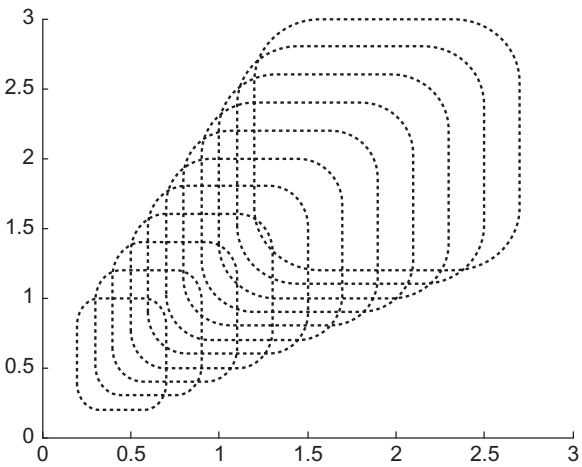


**FIGURE 12.34**
Curved rectangles produced in a loop.

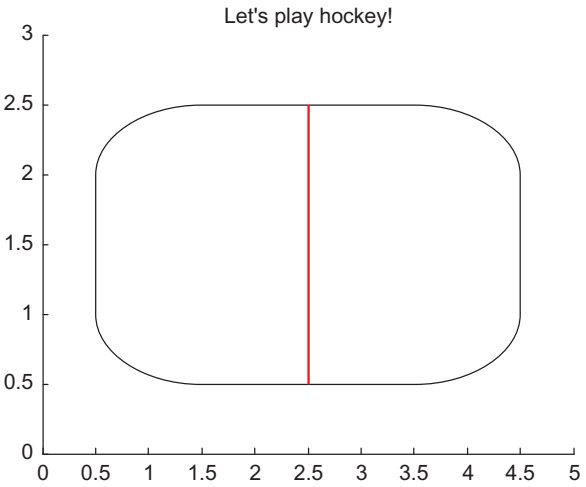


**FIGURE 12.35**
Hockey rink.

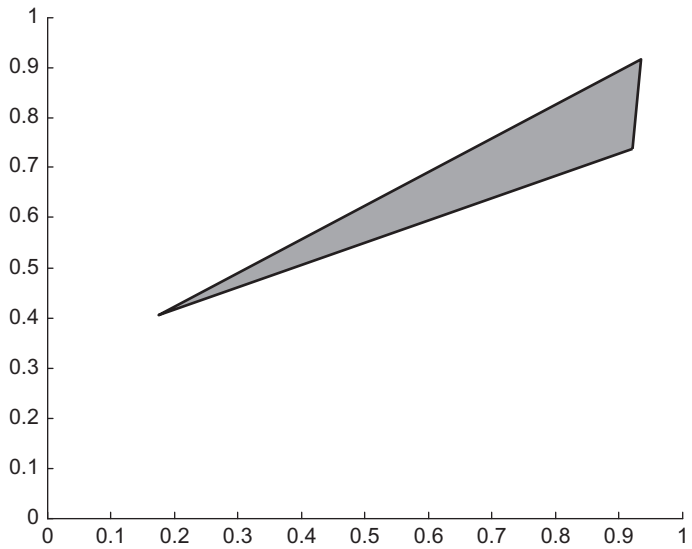


**FIGURE 12.36**
Patch object with black edge.

29. Write a function *plotline* that will receive *x* and *y* vectors of data points and will use the **line** primitive to display a line using these points. If only the *x* and *y* vectors are passed to the function, it will use a line width of 5; otherwise, if a third argument is passed, it is the line width.
30. Write a script that will create the Figure shown in Figure 12.37. Note: the line width is 4.

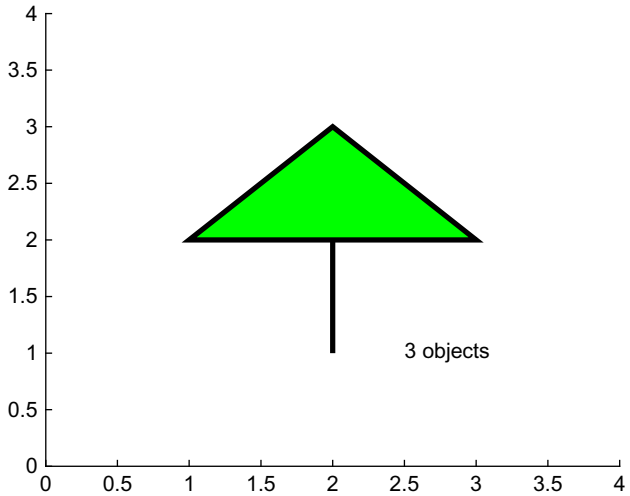


**FIGURE 12.37**
Graphics objects to create.

31. There is a built-in spreadsheet in MATLAB that stores information on tsunamis. Read this in to a table and create a geographic bubble chart using the following code:

```
>> t = readtable('tsunamis.xlsx');
>> gb = geobubble(t.Latitude, t.Longitude)
```

    Investigate the properties of this bubble chart. Note: the **geobubble** function was introduced in R2017b.
32. Investigate the **polyshape** function, introduced in R2017b.
33. The **nsidedpoly** function creates a polygon (a polyshape object) with *n* sides. Create an 8-sided polyshape object and plot it; investigate its properties.
34. The **xticks** function (introduced in R2016b) can be used to create custom tick marks. Create a plot of sin(*x*) where *x* ranges from at least −7 to 7, and then change the *x* ticks with xticks(−2*pi: pi: 2*pi).