CHAPTER 5

Data Visualization

In Chapter 4, you learned about important programming constructs like decision making, loops, and user-defined functions. These programming constructs are very useful when you need to include the logic of decision making in your program. You also learned how to work with files of various formats and data from the Internet.

In this chapter, you will learn about data plotting and visualization in detail. In scientific and business applications, visualization is a very important application/step and it is often consumed by business end-users. The following is the list of the topics that you will learn and demonstrate in this chapter:

- Simple plots
- Plotting options
- Errorbars
- More visualizations
- 3D visualizations

After reading this chapter, you will be able to create visualizations with GNU Octave for scientific and business applications.

A. Pajankar and S. Chandu, *GNU Octave by Example*,
https://doi.org/10.1007/978-1-4842-6086-9_5

Simple Plots

You will use a Jupyter notebook for this chapter. It's best to create a separate, new Jupyter notebook for this chapter, as you did for earlier chapters. Now let's see how to draw simple plots. Create data points for the X and Y axes as follows:

```
x = linspace( 1, 100, 1000);
y = x + 3;
```

You use the function `linspace()` in this code to create a matrix of values in `x` from 1 to 1000 with a step of 100 and corresponding values for `y` with the equation. This will be your data, which has pairs `(x, y)` for points. You can draw a simple line graph as follows:

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

The output can be seen in the notebook itself. The output is shown in Figure 5-1.

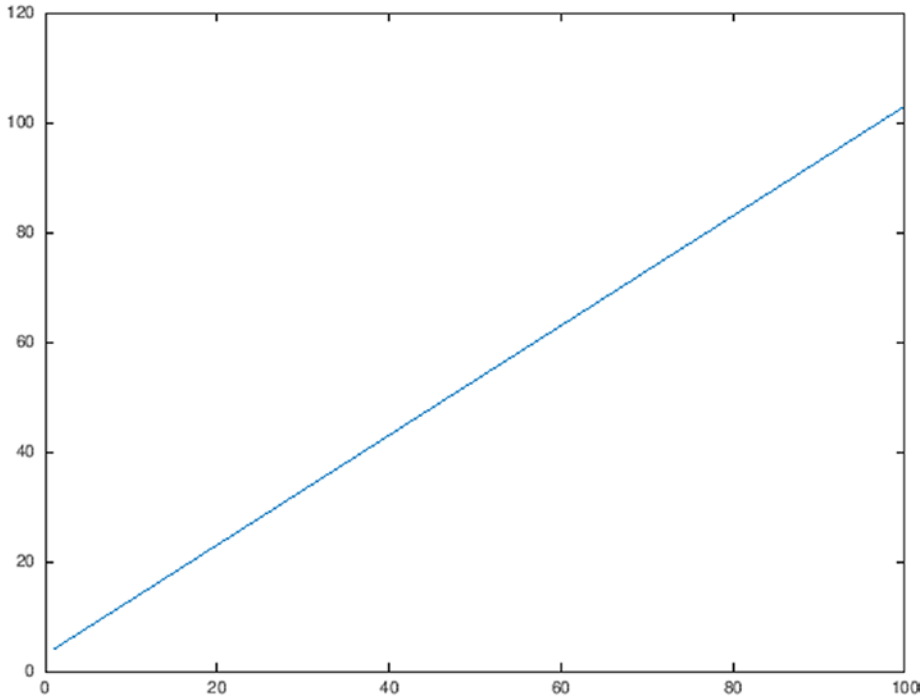

Figure 5-1. *A simple liner plot*

You can use the functionality of gnuplot to show the visualizations in different windows. Gnuplot is a command-line-driven graphics utility that works with many OSes like Windows, Linux, and Mac.

You will use the same data for the demonstration with gnuplot:

```
%plot gnuplot
plot(x, y)
```

In this code, the first line enables the gnuplot for the current session. All of the output from now on will be shown in separate windows. The output is displayed in a separate gnuplot window, as shown in Figure 5-2.

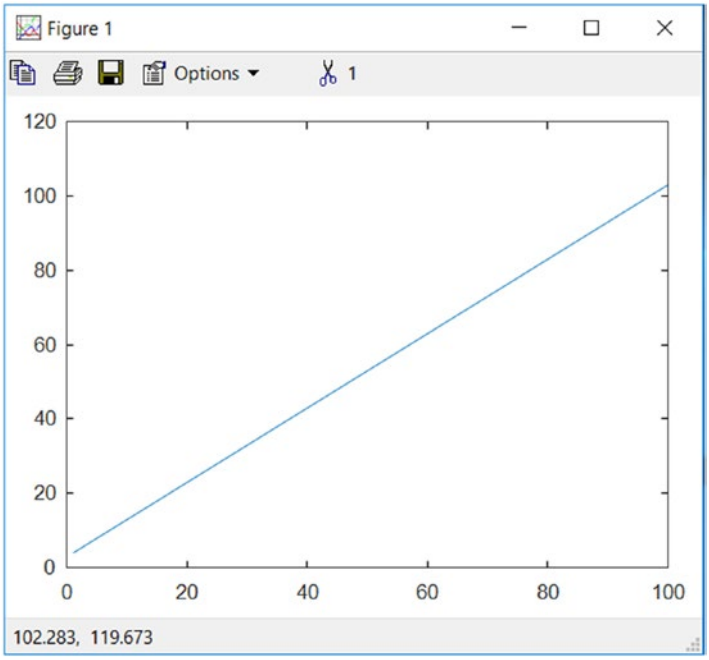

Figure 5-2. *A simple liner plot in a separate gnuplot window*

You can save your output in popular image formats as follows:

```
print("image1.png", "-dpng");
print("image2.jpg", "-djpg");
print("image3.pdf", "-dpdf");
```

You will find these images in the respective formats in the directory where you launched the Jupyter notebook server using the command prompt.

This was an example of a linear graph. Now let's plot the graph of a square function:

```
x = linspace( 1, 10, 10);
y = x.^2;
plot(x, y)
```

The output is shown in Figure 5-3.

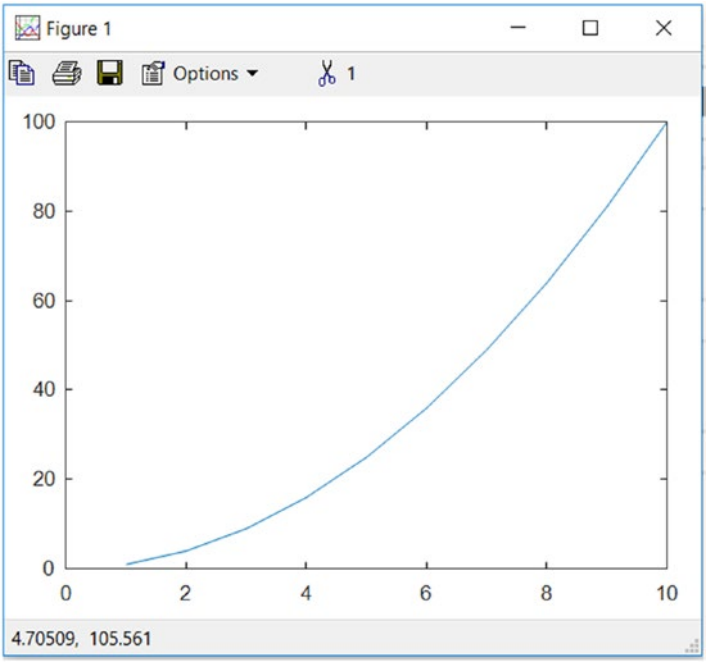

Figure 5-3. *Plot of* $y = x^2$

You can also visualize a logarithmic graph:

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

The output is shown in Figure 5-4.

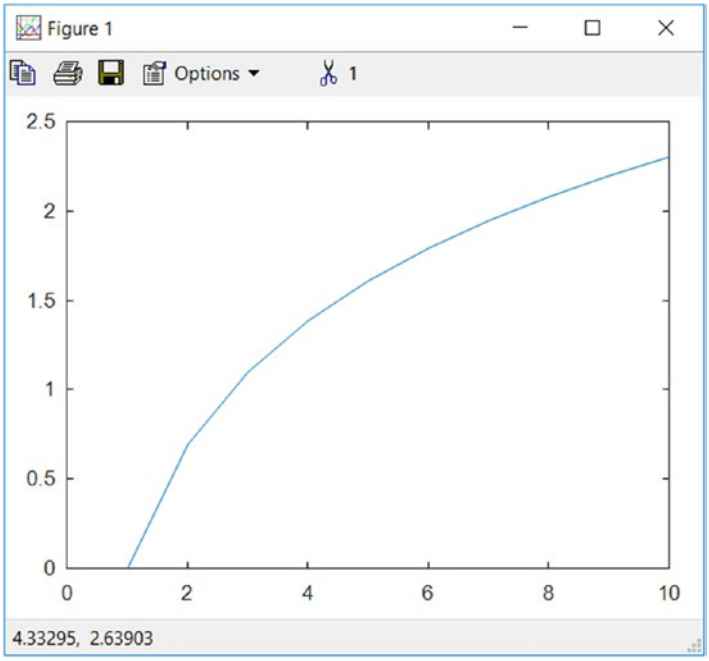

Figure 5-4. *Plot of y = log(x)*

The following is the example when data for both axes is logarithmic:

```
x = logspace( 1, 10, 10);
y = x;
plot(x, y)
```

Like `linspace`, `logspace` creates a matrix of values but with a logarithm step between the beginning and end values. The output is shown in Figure 5-5, and it is a line since both axes are logarithmic.

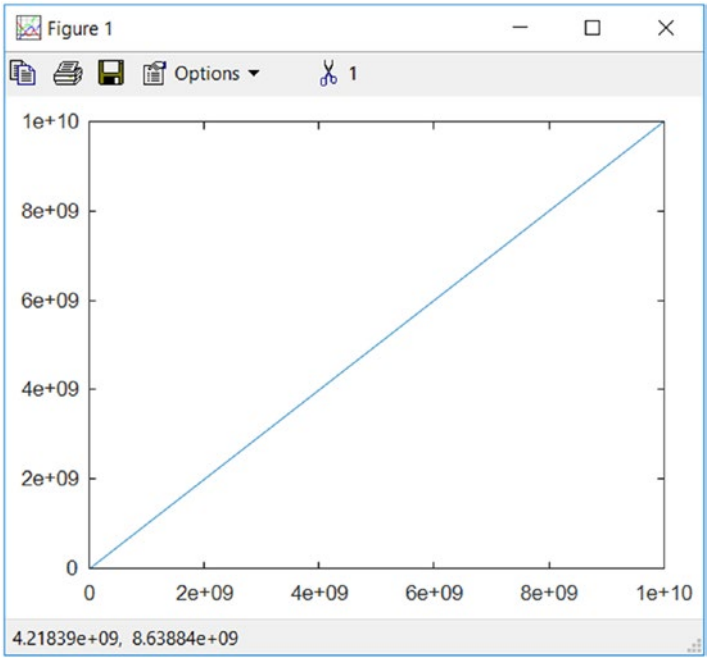

Figure 5-5. *Plot of logarithmic data*

Let's see an example of a sinusoidal:

```
x = -pi:0.01:pi;
n = 3;
y = sin(n*x);
plot(x, y)
```

In this code, you assign values from -pi to pi to the x axis with a step value of 0.01. The variable n is the number of repetitions of the sine wave. The output is shown in Figure 5-6.

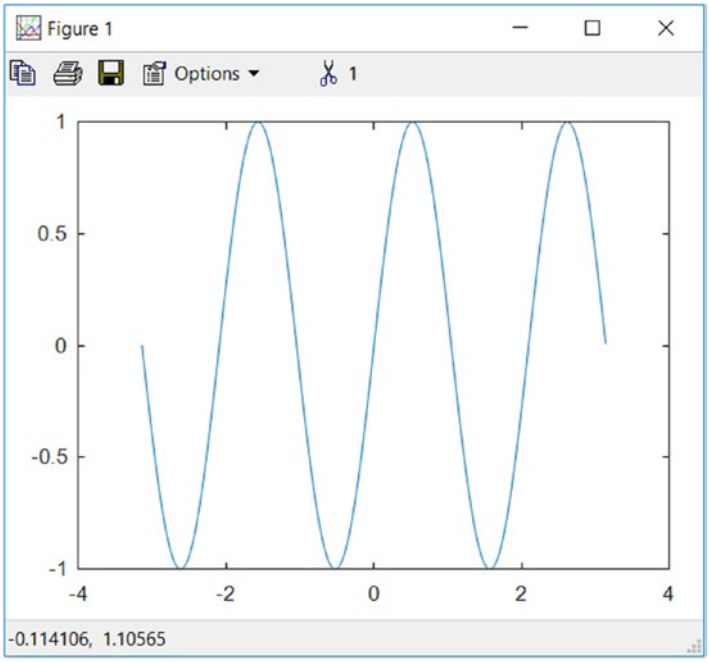

Figure 5-6. *Plot of a sine wave*

Similarly, you can plot other trigonometric functions. As an exercise, try plotting other trigonometric functions or combinations like sin(x) + cos(x).

Plotting Options

Let's see how to label axes and how to add legends. You will also add a title to the figure. Create the data first:

```
t = [0:0.01:1.0];
n = 5;
y1 = sin(2*n*pi*t);
```

Now add labels, legends, and titles as follows:

```
plot(t, y1)
xlabel('Time')
ylabel('Value')
legend('Sin')
title('Sine Plot')
```

The output has a title, a legend, and labels for the axes, as shown in Figure 5-7.

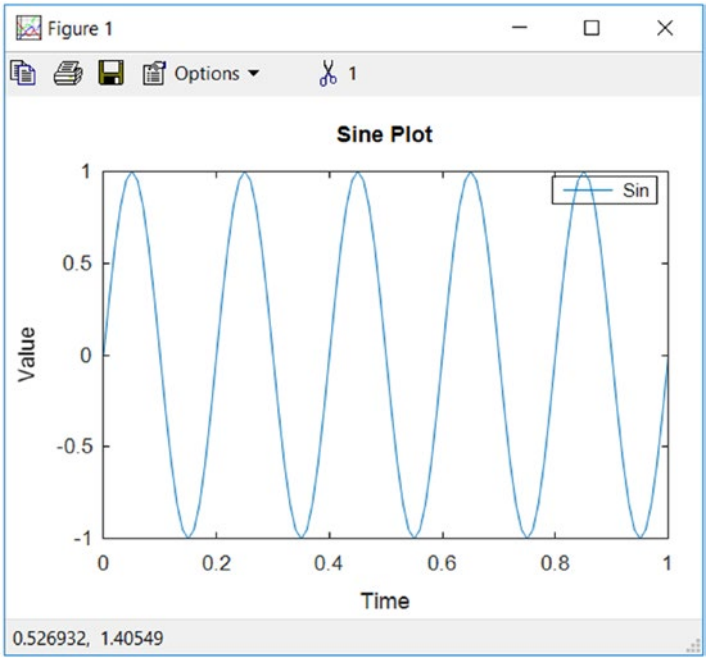

Figure 5-7. *Adding a title, legend, and labels*

You can plot multiple graphs as follows:

```
y2 = cos(2*n*pi*t);
plot(t, y1, 'r-', t, y2, 'b-.')
xlabel('Time')
```

```
ylabel('Value')
legend('Sin', 'Cos')
grid on
title('Sine and Cosine Plot')
```

As seen in the code, in `plot()`, you assign different styles to the graphs: one uses a red color with a - and the other a blue color with - . -, as you can see in Figure 5-8. You use `plot()` to draw multiple graphs in the same output. Also, you turn the grid on and add a title and a legend. You use the functions `xlabel()` and `ylabel()` to add labels to the image. You also use `legend()` to identify the data in the output.

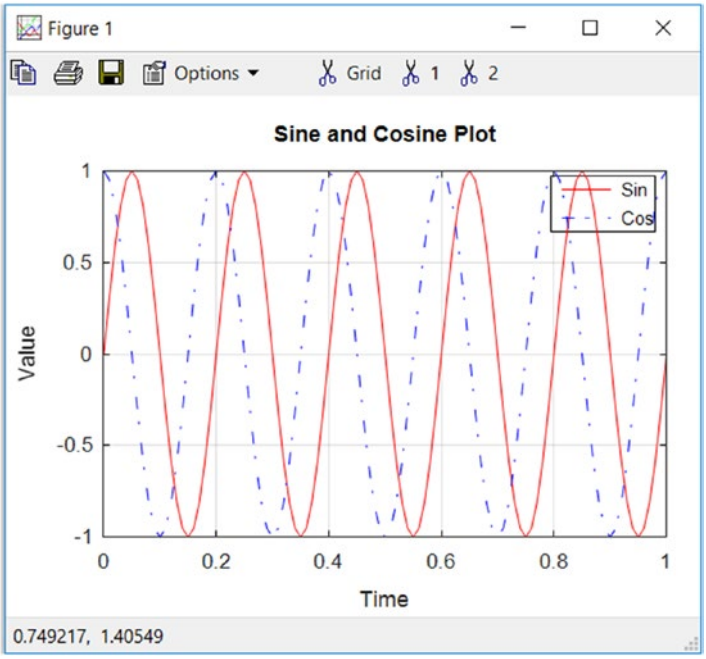

Figure 5-8. *Multiple plots in the output*

Now let's see how to use colors and styles for drawing graphs in detail. There are seven colors and seven marker styles you can use. In the `plot()` function call, after mentioning `x` and `y`, you must mention the color and the style. For example, `k+` refers to the color black and style `+`. Let's see how to use all of the colors and styles. The following is the data:

```
x = [0:1:10];
y1 = x;
y2 = x + 2;
y3 = x + 4;
y4 = x + 6;
y5 = x + 8;
y6 = x + 10;
y7 = x + 12;
```

You can use marker styles and colors as follows:

```
grid on
plot(x, y1, 'k+', x, y2, 'ro', x, y3, 'g*', x, y4,
'b.', x, y5, 'mx', x, y6, 'cs', x, y7, 'wd')
```

The output is shown in Figure 5-9.

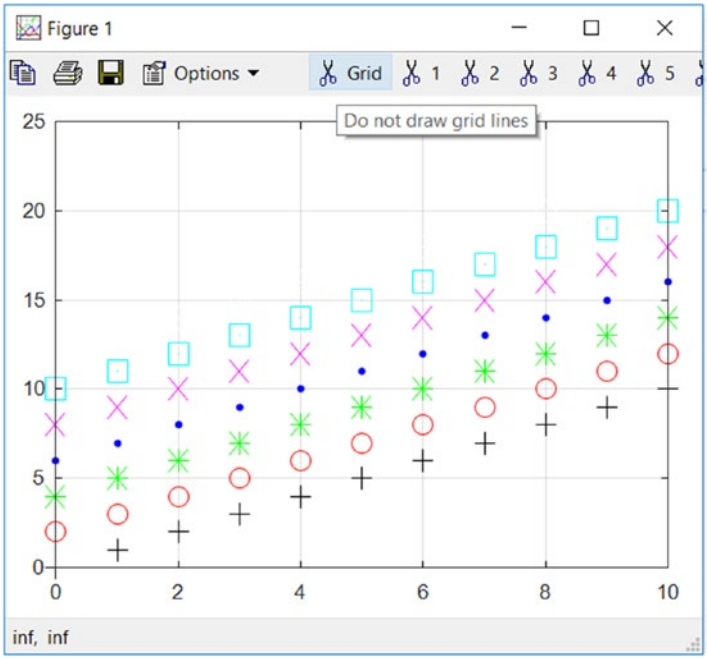

Figure 5-9. *Marker styles and colors*

You also have different line styles as follows:

```
grid on
plot(x, y1, 'k-', x, y2, 'k--', x, y3, 'k-.', x, y4, 'k:')
```

The output is shown in Figure 5-10.

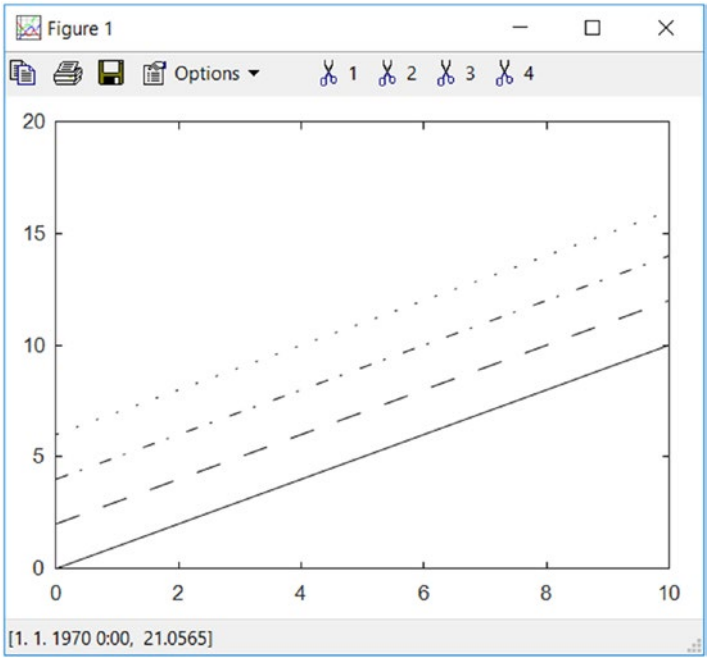

Figure 5-10. *Different line styles*

You can explore the different combinations of styles, markers, and colors yourself by changing the code snippets above.

You can also use subplots to plot multiple graphs separately in the same output window. You must use the `subplot()` function for this. It accepts three arguments. The first two are the dimensions of the output grid where plots are to be displayed. The last argument is the position of the plot in that grid. The following is the data:

```
x = linspace(1, 100, 100);
y1 = x.^ 2.0;
y2 = sin(x);
y3 = log(x);
```

Now use the function `subplot()` as follows:

```
subplot(3, 1, 1), plot(x, y1)
subplot(3, 1, 2), plot(x, y2)
subplot(3, 1, 3), plot(x, y3)
```

This code creates a grid of three rows and a column. In every row of the grid, one plot is displayed (as defined by the final argument of each `subplot()` call). The output is shown in Figure 5-11.

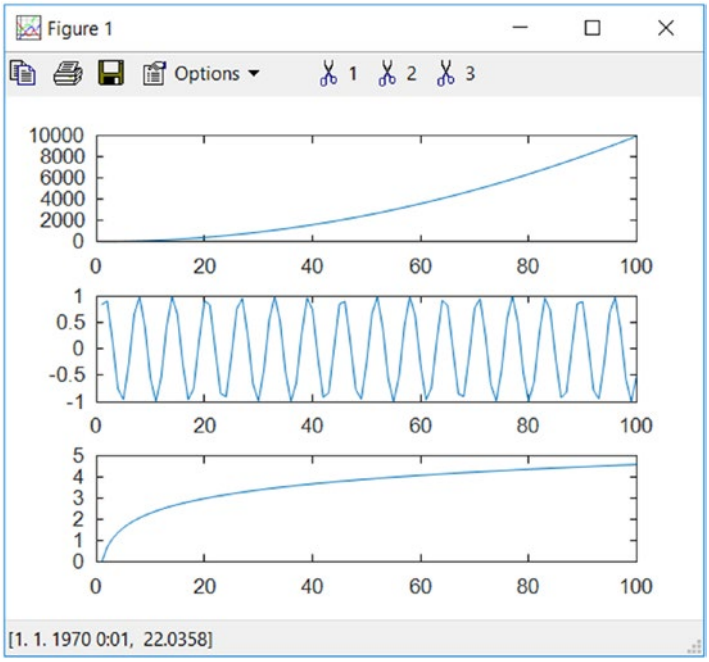

Figure 5-11. *Showing different plots with the function subplot()*

You can even show these plots in different gnuplot windows as follows:

```
close all
figure(1), plot(x, y1)
figure(2), plot(x, y2)
figure(3), plot(x, y3)
```

In this code, the statement `close all` closes and clears all of the previous visualization windows. You use the function `figure()` to create a separate window for visualization. Run the code and see the output.

Errorbars

You can even include the visualization of errors in the output. The following is a simple example:

```
close all
t = -1:0.1:1;
y = sin (pi*t);
lerr = 0.1 .* rand (size (t));
uerr = 0.1 .* rand (size (t));
errorbar (t, y, lerr, uerr);
```

In this example, you use the function `errorbar()` to visualize an error in the y-axis. The variables `lerr` and `uerr` are used to show the lower and upper value of the error for a data point. The output is shown in Figure 5-12.

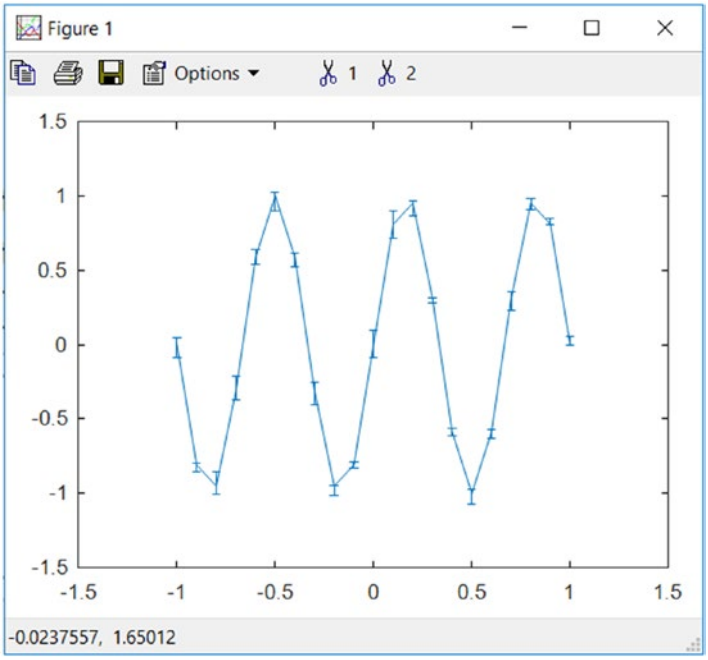

Figure 5-12. *Errorbars for the y-axis data*

Similarly, you can create errorbars for the data of the x-axis as follows:

```
errorbar (t, y, lerr, uerr, ">");
```

Note that in this code you pass an extra argument, ">", that denotes error values are for the data on the x-axis. See Figure 5-13.

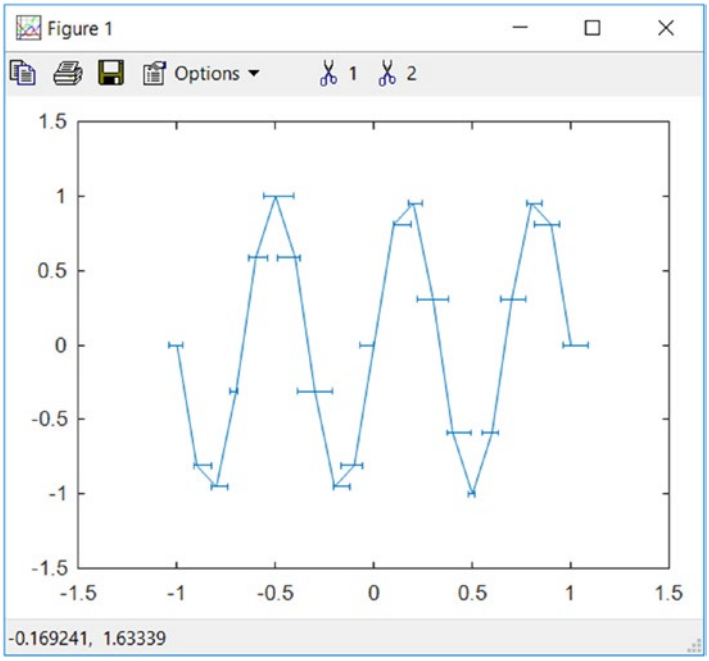

Figure 5-13. *Errorbars for the x-axis data*

Similarly, you can use "~" for error values on the y-axis. Let's see an example of how you can plot the errorbars for data for both the axes in a single visualization:

```
close all
x = 0:0.05:1;
n = 2;
err = rand (size(x))/10;
y1 = sin (n*x*pi);
y2 = cos (n*x*pi);
errorbar (x, y1, err, "~", x, y2, err, ">");
```

The output is shown in Figure 5-14.

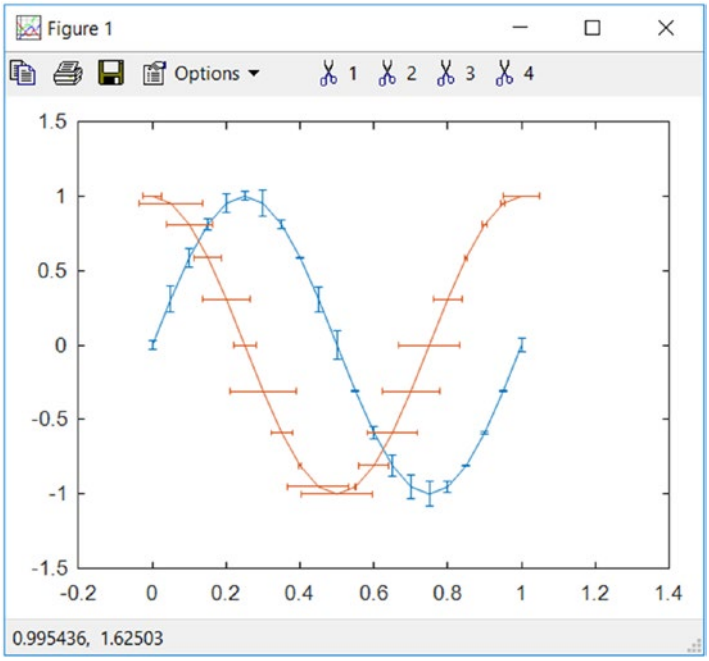

Figure 5-14. *Errorbars for the data of both the axes*

You can even create boxes in place of the errorbars with the following code:

```
errorbar (x, y1, err, err, "#r", x, y2, err, err, "#~");
```

In this function call, `err` stands for the error vector and `r` stands for the red color. You pass the same error vector for both axes. As you must have guessed, # is used to create errorboxes. It produces the output shown in Figure 5-15.

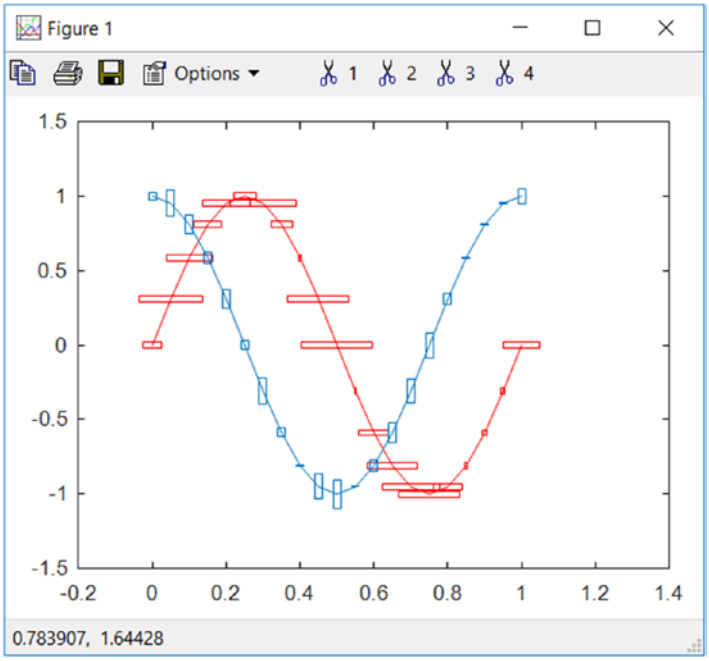

Figure 5-15. *Errorboxes*

This is how you can show the data related to the error. In all of the examples above, the data for error was simulated. But in real-life projects, you will have data from real devices as input. You can store the error margin in arrays and visualize them, as you have seen in previous examples.

More Visualizations

The graphs we have demonstrated until now use lines and curves for plotting functions. In this section, you will see how to use other types of visualizations to represent the data.

Scatter Graphs

Scatter graphs use discrete points rather than continuous curves to represent data. The following is an example of the use of function `scatter()`:

```
close all
x = linspace(1, 100, 100);
y1 = x.^ 2.0;
grid on
scatter(x, y1)
```

The output is shown in Figure 5-16.

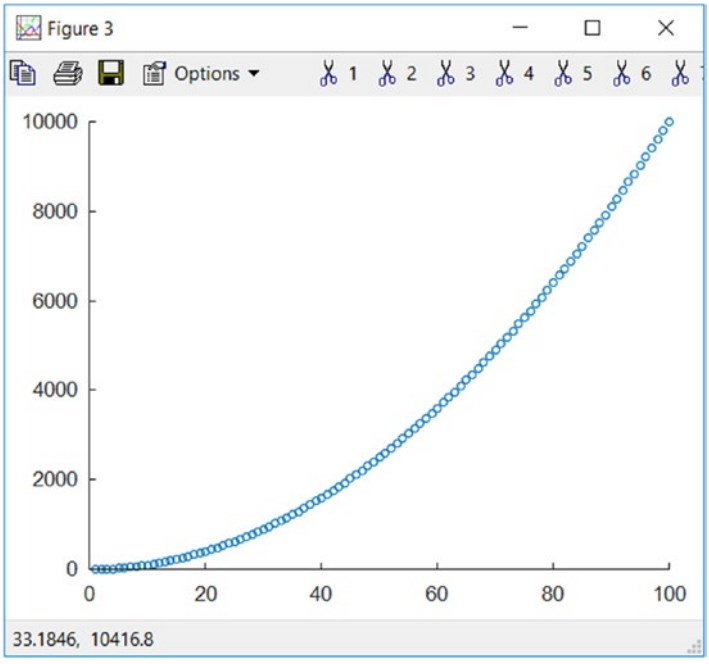

Figure 5-16. *A simple scatter plot*

You can customize the size of circles and the color as follows:

```
close all
scatter(x, y1, s = 10, filled='r')
```

The output is shown in Figure 5-17.

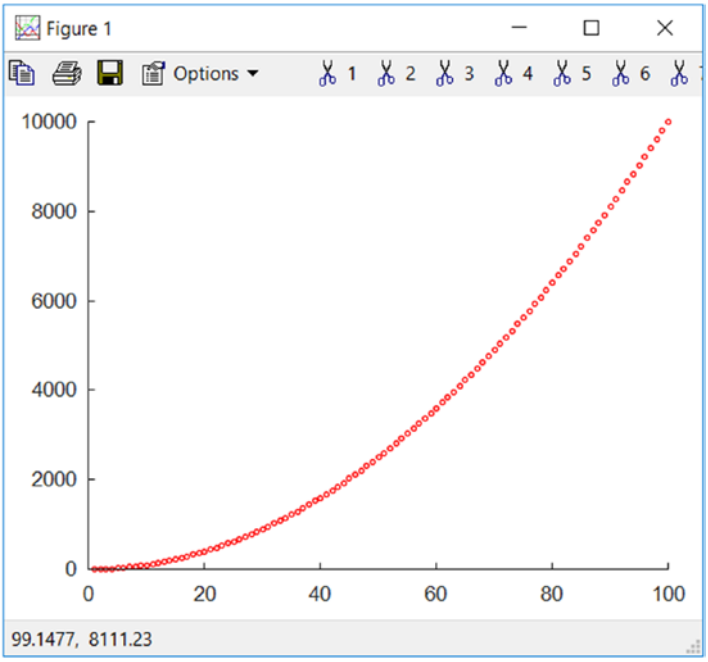

Figure 5-17. *A customized scatter plot*

Histograms

A histogram is a visual reorientation of the distribution of frequency of occurrence of elements in a dataset. In mathematics and statistics, you study frequency distribution tables. A histogram is the visualization of those tables. Write some simple code for a histogram as follows:

```
clear all
close all
a = randn(1000, 1);
hist(a)
```

In this code, you create a matrix of dimensions 1000 X 1 filled with random values from a normal distribution using the function `randn()`. The function `hist()` creates a histogram with 10 bins by default, as shown in Figure 5-18.

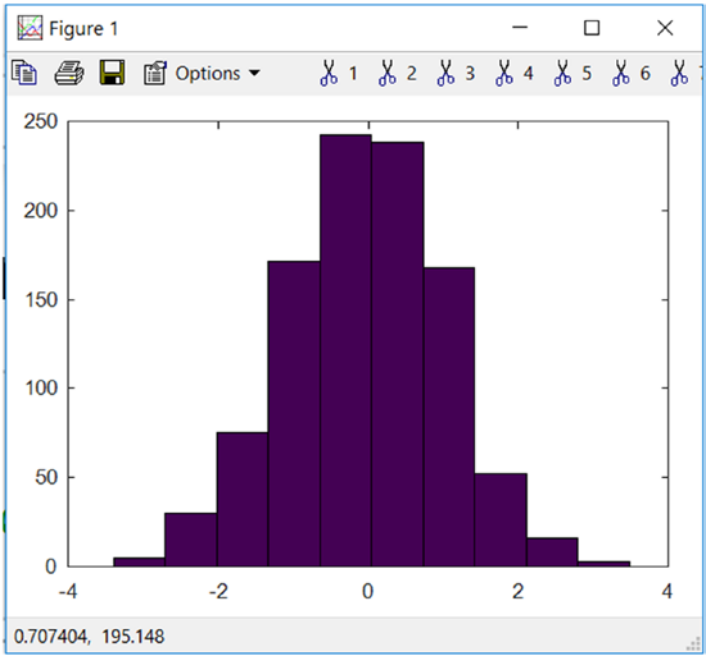

Figure 5-18. *Histogram with default 10 bins*

You can adjust the number of bins in the histogram as follows:

```
hist(a, nbins=100)
```

This code creates a histogram with 100 bins, as shown in Figure 5-19.

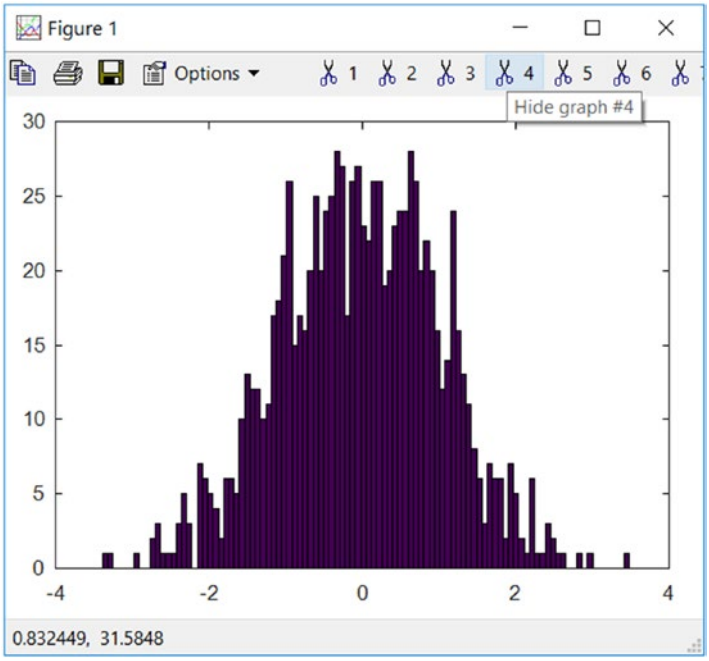

Figure 5-19. *Histogram with 100 bins*

Contours

You can represent data in the form of contours. A contour is a closed shape joining all of the points in an image that have the same value. The most prominent example of the usage of a contour is a topographic map with contour lines.

Here's an example of a contour:

```
x = [1 2 3 4 5 4 3 2 1];
y = x;
z = x' * y;
contour(z)
axis([1 9 1 9])
```

The function `contour()` draws contour visualizations. The function `axis()` is used to set the limits of the values of the axes. In the example, the limits of the x-axis are 1 to 9 and they are the same for the y-axis, as shown in Figure 5-20.

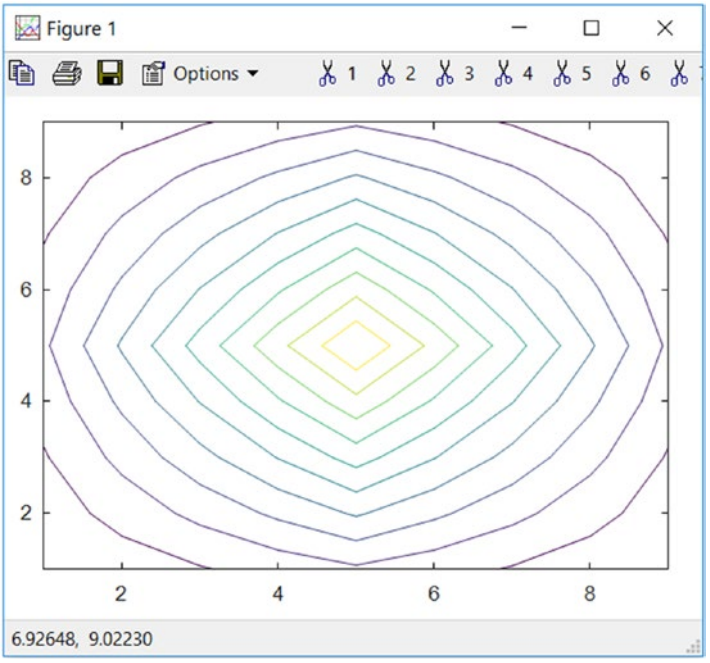

Figure 5-20. *Representation of contours*

You will revisit the concept of contour while demonstrating 3D visualizations, when you will learn and demonstrate 3D versions of contours. You will also demonstrate them with 2D contours.

Polar Graph

The polar coordinate system uses the distance from origin (r) and the angle from a fixed line (θ) to determine the position of a point in the plane. The following formula converts polar coordinates into XY coordinates:

$$x = r \times \cos(\theta)$$

$$y = r \times \sin(\theta)$$

You can draw a simple polar graph as follows:

```
theta = 0:0.1:2*pi;
rho = linspace(0.1, 0.1, 63);
polar(theta, rho)
```

The function `polar()` accepts values of `theta` and `r` as arguments and draws a polar graph, as shown in Figure 5-21.

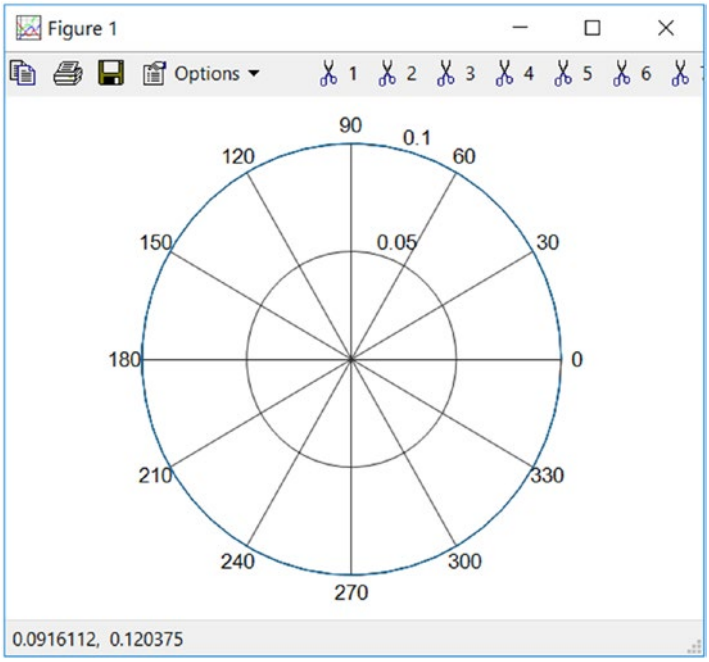

Figure 5-21. *A simple polar graph*

You can combine multiple graphs as follows:

```
theta = 0:0.02:2*pi;
rho1 = 0.4 + 1.1.^theta ;
rho2 = 3 * sin ( theta ) ;
rho3 = 5 * (1 - cos( theta )) ;
rho4 = 4 * cos (8 * theta ) ;
r = [ rho1 ; rho2 ; rho3 ; rho4 ] ;
polar ( theta , r , '.' ) ;
```

The output is shown in Figure 5-22.

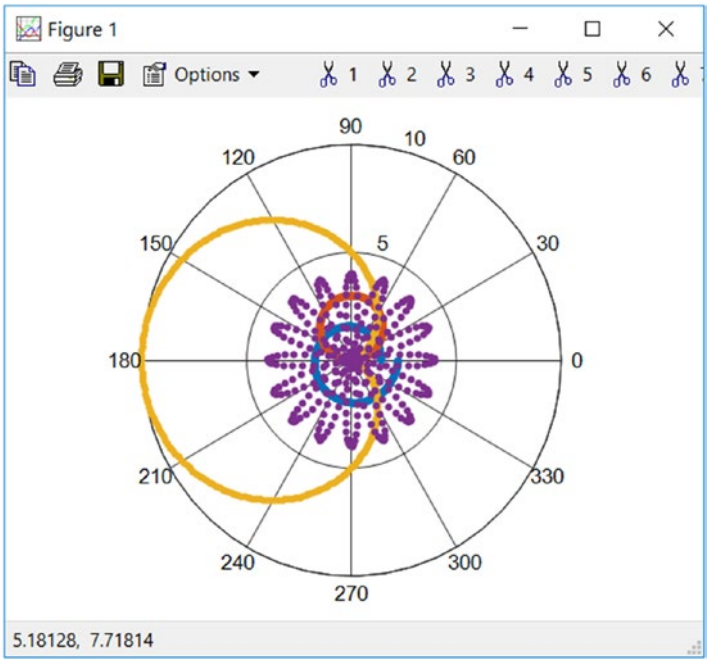

Figure 5-22. *Multiple polar graphs*

Pie Charts

You can create pie charts with Octave. These charts are mostly used in business-related visualizations. Here's how to create a simple pie chart:

```
a = [2, 3 ,5];
pie(a)
```

The function `pie()` divides the pie shape according to the proportion of the weight of the members of the arguments you pass to it. The output is shown in Figure 5-23.

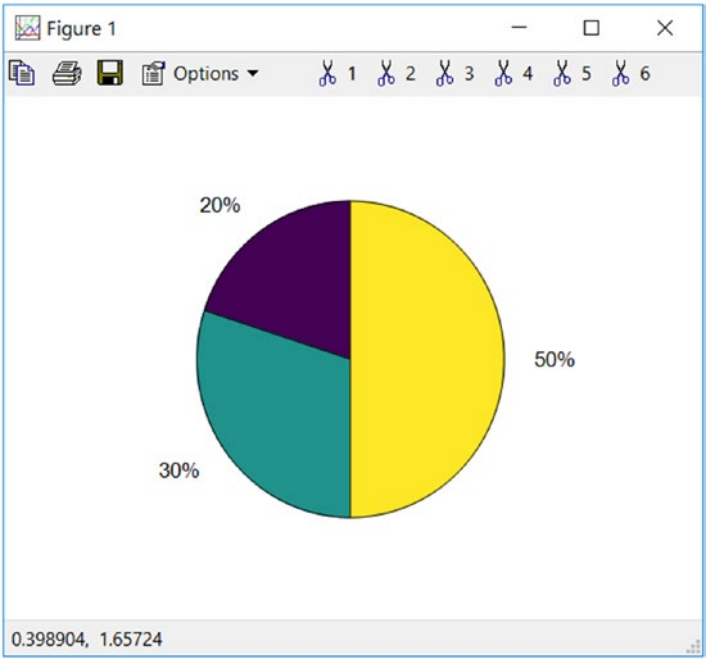

Figure 5-23. *A simple pie chart*

You can also have an exploded pie chart. You need to pass the explosion vector as the second argument to the function `pie()`:

```
e = [1, 0, 1];
pie(a, e)
```

In the explosion vector `e` in this code, `1` stand for enabling an explosion and `0` stands for not enabling it. The output is shown in Figure 5-24.

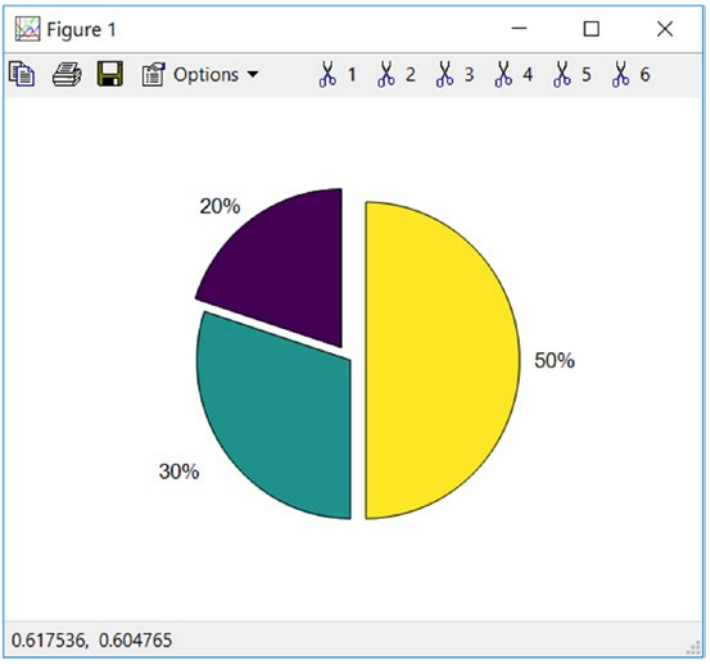

Figure 5-24. *An exploded pie chart*

Visualizing Data as Images

You can visualize your data as images using Octave. Images are represented as numbers in Octave. You will study image processing in detail in a dedicated chapter. For now, you will learn how to visualize

arrays as images. You can use the function `imagesc()` to visualize arrays as images. Let's demonstrate this with the following code:

```
a = randn(50, 50);
imagesc(a)
```

The output is shown in Figure 5-25.

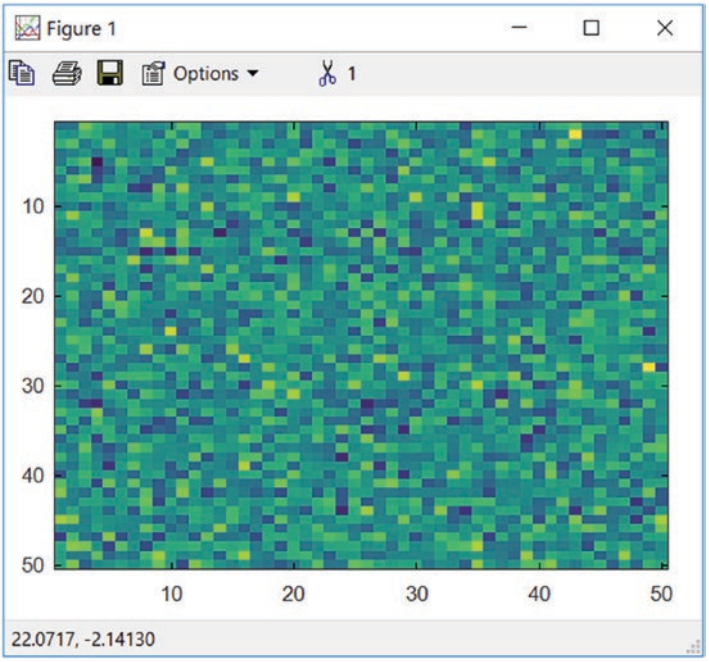

Figure 5-25. *Visualizing an array as an image*

This output is rendered with the default colormap with a jet map of 64 values. A colormap is a set of colors used to represent data. There are many colormaps supported by GNU Octave and you can find the list at https://octave.sourceforge.io/octave/function/colormap.html.

The following code visualizes the same data with another colormap:

```
imagesc(a), colorbar, colormap cool;
```

The function `magic(n)` returns a n×n magic square. Let's visualize it with the viridis colormap:

```
imagesc(magic(6)), colormap viridis;
```

Run both of the lines in separate cells in the Jupyter notebook or the Octave interactive prompt and see the output for yourself.

3D Visualizations

Until now, all the visualizations we demonstrated were 2D visualizations. Now you will learn and demonstrate 3D visualization. Let's use the function `meshgrid()`. You will use this function to create data points for a 3D visualization. Its usage is as follows:

```
y = x = [-3:1:3];
[x1, y1] = meshgrid(x, y)
```

In this code, you define the range of the variables `x` and `y`. Then you pass them to the function `meshgrid()`, `which` returns a grid of points as follows:

```
x1 =

  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
  -3  -2  -1   0   1   2   3
```

```
y1 =

  -3  -3  -3  -3  -3  -3  -3
  -2  -2  -2  -2  -2  -2  -2
  -1  -1  -1  -1  -1  -1  -1
   0   0   0   0   0   0   0
   1   1   1   1   1   1   1
   2   2   2   2   2   2   2
   3   3   3   3   3   3   3
```

For our demonstration, you'll need a bigger grid, as follows:

```
y = x = [-10:1:10];
[x1, y1] = meshgrid(x, y)
```

Let's compute another variable, z, and then use the function `mesh()` to visualize (x1, y1, z) as follows:

```
z  = x1.^2 + y1.^2;
mesh(x1, y1, z)
```

The output is shown in Figure 5-26.

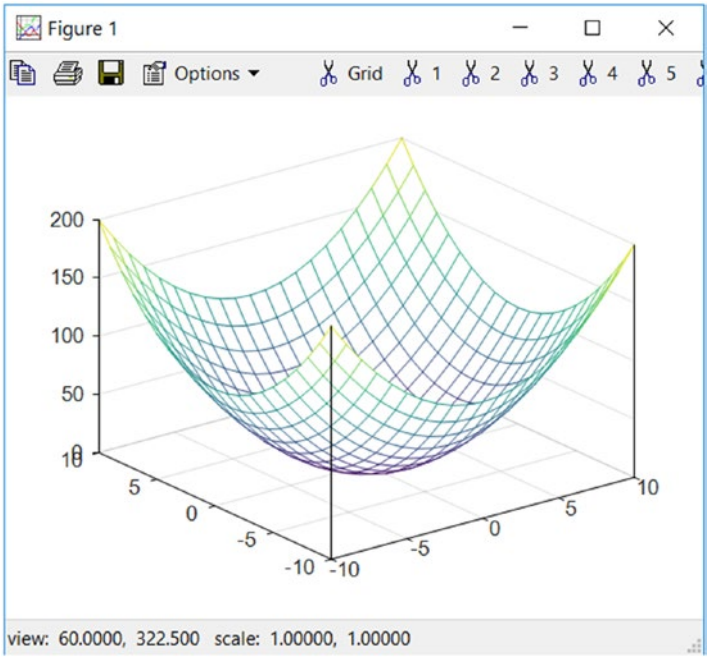

Figure 5-26. *Visualizing mesh*

As you can see, the function `mesh()` plots a 3D wireframe. You can change the orientation of the image by dragging it with a mouse pointer.

Similarly, the function `meshc()` plots mesh with underlying contour lines. Run the following function call in a new cell and see the output:

```
clf;
meshc(x1, y1, z)
```

In this code, you use the command `clf` to clear the earlier figure. The output of the code is shown in Figure 5-27.

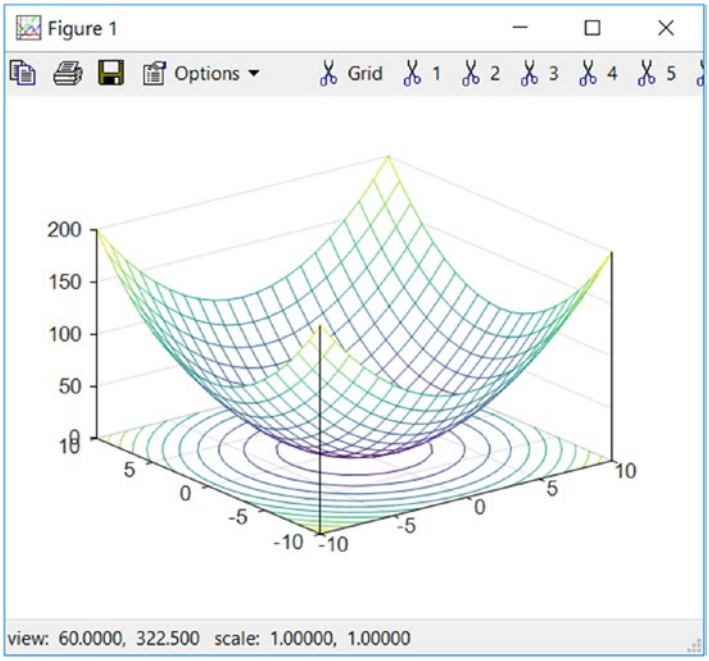

Figure 5-27. *Visualizing mesh with underlying contours*

The function `meshz()` draws a 3D mesh with the surrounding curtain as follows:

```
clf
meshz(x1, y1, z)
```

The output is shown in Figure 5-28.

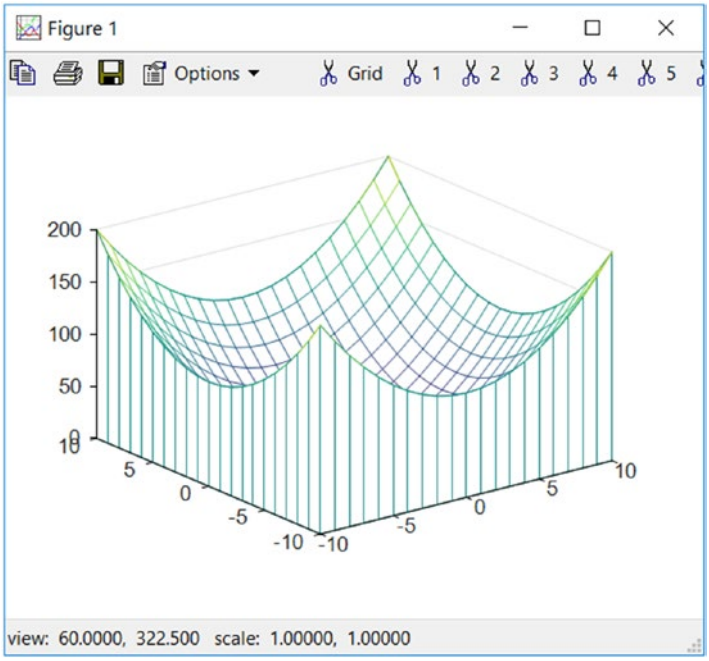

Figure 5-28. *Visualizing mesh with surrounding curtain*

Similar to wireframe mesh, there are functions to draw surfaces. The functions `surf()` and `surface()` draw surfaces using given data points. The following are examples of calls for these functions:

```
surf(x1, y1, z)
surface(x1, y1, z)
```

The output is shown in Figure 5-29.

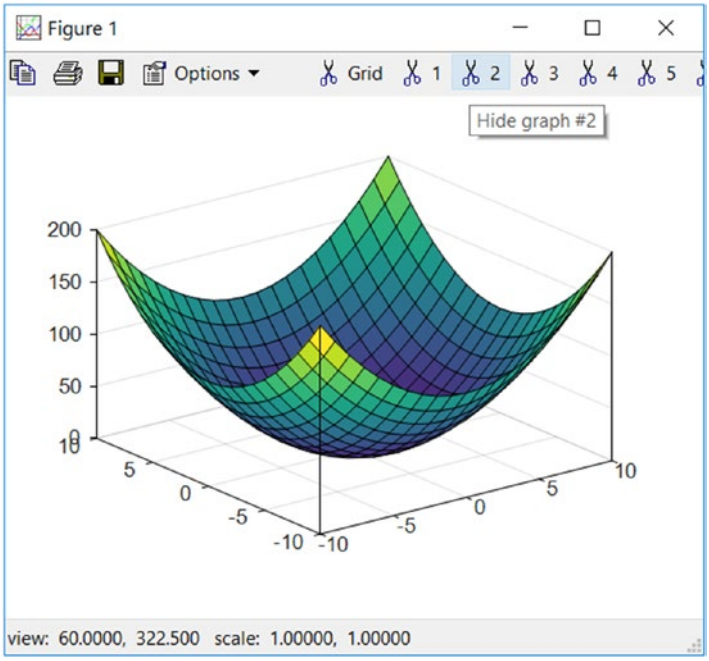

Figure 5-29. *Visualizing surface*

The function `surfc()` draws a surface with associated contours and `surfl()` draws a surface with lighting:

```
surfc(x1, y1, z)
surfl(x1, y1, z)
```

Run the above code in separate cells after the `clf` command and see the output.

You can even visualize 3D plots with the function `plot3()` as follows:

```
clf;
z = [0:0.01:3];
n = 3;
theta = n * pi * z;
plot3 (cos (theta), sin (theta), z);
```

The output is a spring-shaped figure, as shown in Figure 5-30.

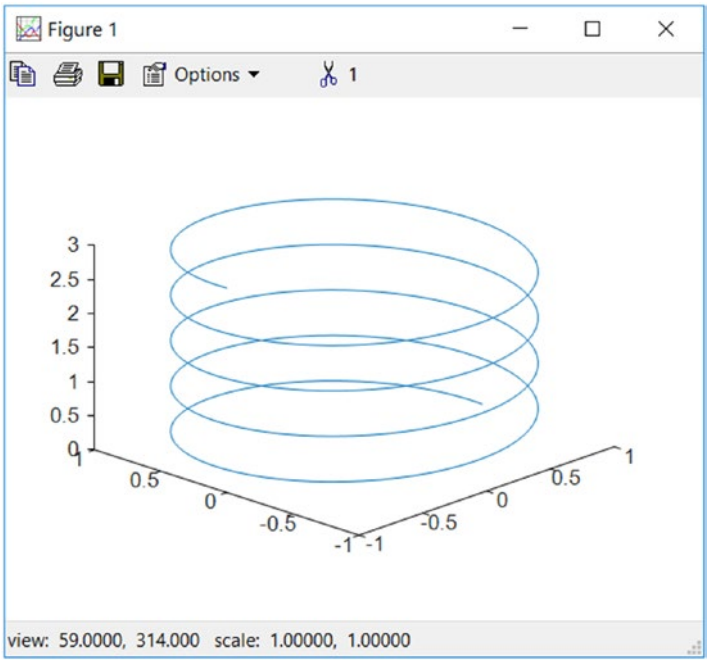

Figure 5-30. *A spring shape*

You already saw how to visualize a 2D contour, but let's revisit it before demonstrating a 3D version of a contour. The following is the data:

```
y = x = [-3:0.1:3];
[X, Y] = meshgrid(x, y);
Z = X.^3 - Y.^3;
```

A regular 2D contour looks as follows:

```
clf
contour(X, Y, Z);
```

The output is shown in Figure 5-31.

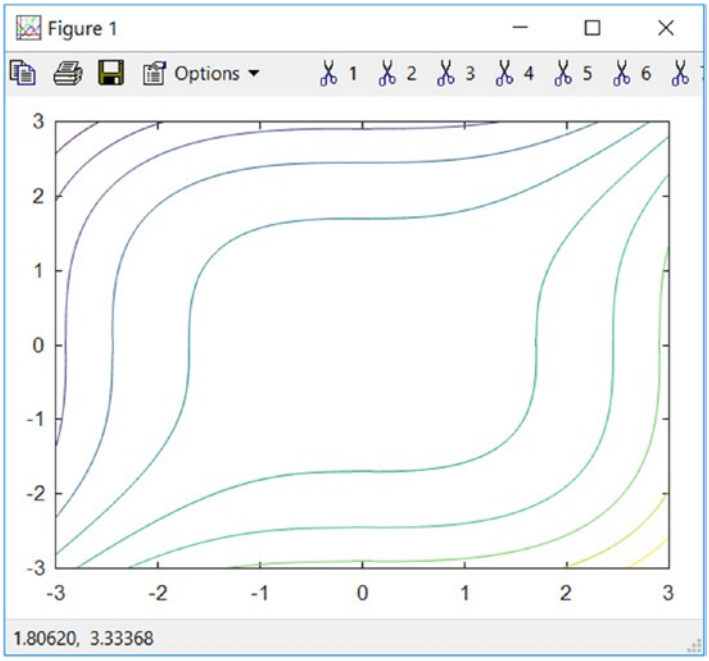

Figure 5-31. *2D contour*

You can draw a 3D contour as follows:

```
clf
contour3(X, Y, Z);
```

The output is shown in Figure 5-32.

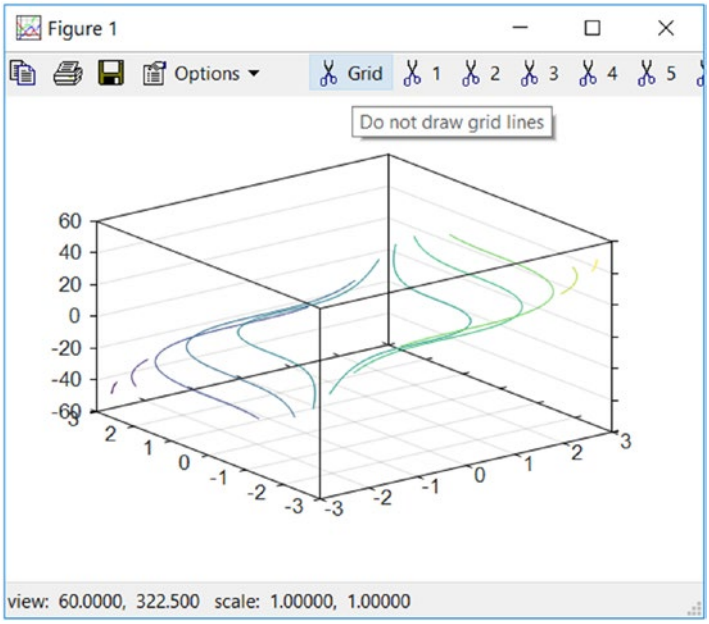

Figure 5-32. *3D contour*

As an exercise, explore the functions `contourc()` and `contourf()` with the same data.

Summary

In this chapter, you learned and demonstrated the ways to visualize multidimensional data with 2D and 3D visualizations in Octave in detail. Now you should be comfortable with the graphical representation of data for scientific and business applications, where data visualization is an important part of the data processing pipeline or architecture.

The next chapter will focus on the topic of data analytics. You will learn and demonstrate various concepts in that area in detail with GNU Octave.