Namir's R 102 Plotting Tutorial

by

Namir Shammas

Dedication

This tutorial is dedicated to memory of my uncles and father—A Shammas generation that has passed on.

Table of Contents

"A picture is worth a thousand words."

Napoleon Bonaparte

"Quotes often have a level of uncertainty about their sources. After all, quotes are nothing but 'open source' nuggets of wisdom"

The Author

Introduction

This tutorial complements *Namir's R 101 Tutorial* and focuses on plotting graphs, pie charts, and histograms. The aim of the tutorial is to cover enough of common plotting functions in R, but by no means covering ALL plotting functions and ALL graphics features.

My hope is that this tutorial gets your feet wet, so to speak. I encourage you to further experiment with the different graphics functions to learn more about how to customize the graphs they produce. The bottom line is that learning any system must involve working and tinkering with that system. Cursory glances at R documentation contribute very little to learning R proficiently.

Before You Start

The tutorial contains several user-defined functions presented to facilitate certain tasks. The tutorial text asks you to save the code for these functions in .r script files and then load them using the source() function. To make matters easy, set your working directory to easily retrieve these functions. Use the setwd() function to specify the working path you will use in following this tutorial. When you save a script file from the R editor window, make sure that the target .r file resides in your working directory. Thus, the calls to function source() automatically retrieve the script files from your working directory, and need not specify the path in the function calls.

Simple Plotting

R offers the function plot() to plot data. This function is perhaps the most popular graphics function in R. The function plot() (and like it, function matplot(), which I present later) establishes a canvas for graphics upon which other graphing functions can add to. In fact you can call function plot() for the sole purpose of preparing the graphics canvas and hold back the function plot() from drawing any lines, curves, or points!

The simplest form of this function is plot(x, y) which plots the data in the paired-value numeric vectors x and y.

To illustrate the function call of plot(x, y), execute the following commands to assign data in the vectors x and y, and then invoke the plot() function:

```
> x = seq(-1,4,0.01)
> y = exp(x)-3*x^2
> plot(x, y)
```

R creates the following plot:

Figure 1. Plot of points for y = exp(x) - 3*x^2

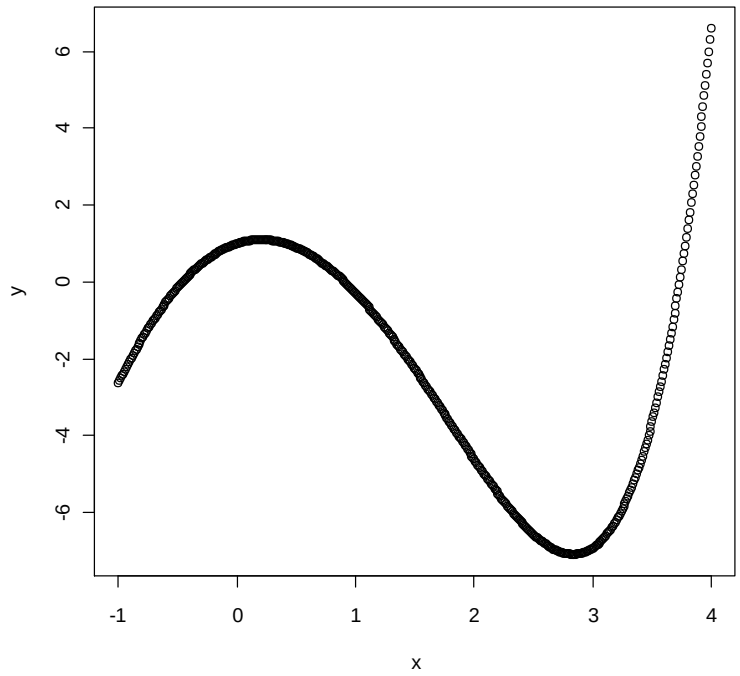


Figure 1 plots the points in the paired vectors x and y, draws the X and Y axes, labels the axes, and displays the values on each axis. As you see, the basic call to plot(x, y) performs a good number of tasks.

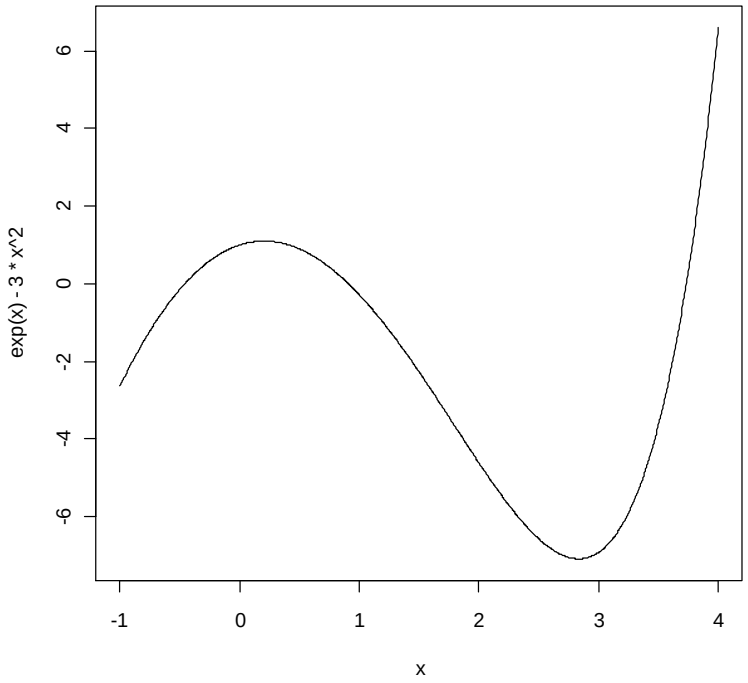
Drawing Curves

The function plot() limits you to drawing one set of observations. This is in contrast with Matlab's own plot() function that allows you to plot multiple sets of observation. So how do you plot additional curves once you called function plot()? The function curve() comes to the rescue. The function curve(fx, from, to, n, col='black") plots the function **fx** using **n** values of x that fall in the range of (**from**, **to**). The argument for parameter fx must be an expression of variable x. The parameter **col** specifies the color of the curves. The default color is black. You can call function curve() several times, after you first call function plot(), to draw additional curves.

As an example, you can use the curve() function to redraw the mathematical function of Figure 1, by typing the following command:

```
> curve(exp(x)-3*x^2, from = -1, to = 4, n = 1001)
```


Figure 2. Using the curve function.



The arguments for the curve() function are:

1. The expression of the plotted function
2. The lower x range limit of -1
3. The upper x range limit of 4
4. One thousand and one points to plot. This arguments sets the values of x to be the sequence -1.000, -0.999, -0.998, …, 3.998, 3.999, and 4.000.

Figure 2 shows the plot produced. Instead of the label y, the function curve() displays the name of plotted mathematical function to the left of the Y axis. The figure shows a smooth curve and properly formatted axes.

☞ It is interesting to point that the function plot() can also plot equations! When you type the following commands, you also obtain a graph like the one in Figure 2:

```
> plot(exp(x)-3*x^2 ~ x, type = "l")
```

A single parameter replaces the first two typical parameters x and y. The new first parameter represents a declaration that uses the tilde character to separate the expression to be plotted from

the independent variable used in that expression. The tilde tells function plot() to draw what appears on the left of the tilde as Y vs. what appears on the right of the tilde as X. Also note that in the above command, the plotted expression in function plot() must use the actual variable name. This is in contrast with function curve() which employs the generic variable name x. So you could type the above command as follows:

```
> x.var = seq(-1,4,0.01)
> plot(exp(x.var)-3*x.var^2 ~ x.var, type = "l")
```

And obtain the same curve in Figure 2. The labels on the axes will, of course, reflect the name of variable x.var.

You can also use expressions on both sides of the tilde in the first parameter of function plot(). For example you can execute the following command:

```
> x = 1:10
> plot(x^2 ~ log10(x), type = "l")
```

The above call to function plot() plots the square of x vs. the common logarithm of x. I have tinkered with the kind of expressions that can be placed after the tilde. The conclusion I drew is that the function plot() is a little bit picky with the kind of post-tilde expressions you give it. It will accept some and reject others.

I just wanted you to know that you can specify equations with the function plot() and not just variables that contain data. I will be mostly using the plot(x, y,…) form in this tutorial as well as other tutorials that I will develop. While the function curve() allows you to specify the range of values using arguments, the function plot() requires that you predefine the ranges in the variables used.

Plotting with Bells and Whistles

Fine-Tuning the Output of Function plot()

In addition to taking arguments for the x and y data, the function plot() can take additional arguments.

The **type** parameter specifies the type of plot to draw. Values can be one of the following:

- The character "p" for points
- The character "l" for lines
- The character "b" for both
- The character "c" for the lines part alone of "b"
- The character "o" for both over-plotted
- The character "h" for 'histogram' like (or 'high-density') vertical lines

- The character "s" for stair steps
- The character "n" for no plotting. Using this option draws the axes to fit the data.

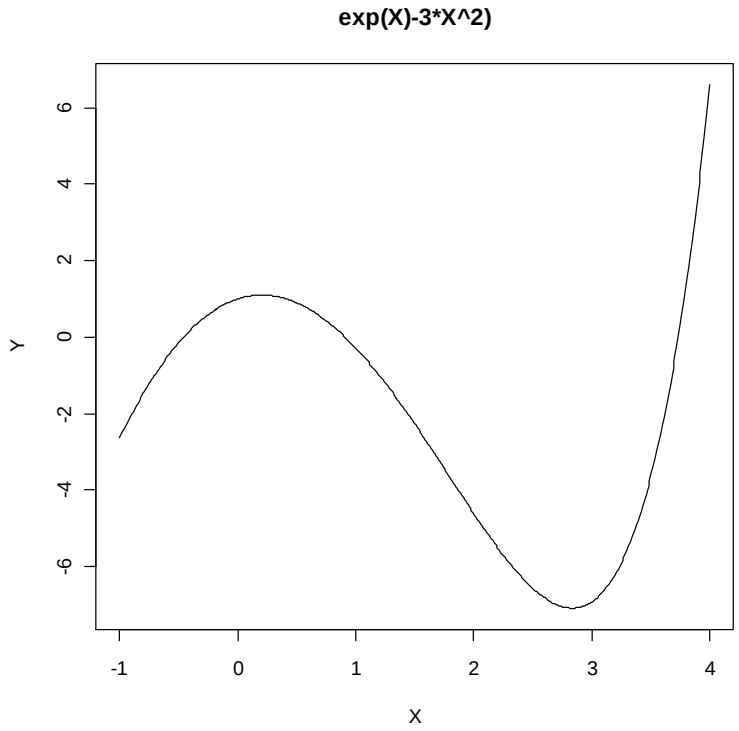
The parameter **main** specifies the main plot title. The parameter **sub** specifies the subtitle. The parameters **xlab** and **ylab** specify labels for the x and y axes, respectively. The parameter **asp** specifies the y/x aspect ratio. The parameter **col** specifies the color of the data points or curves. The parameter **xlim** is a two-element vector that specifies the range for the X axis. The parameter **ylim** is a two-element vector that specifies the range for the Y axis. When you don't give arguments to the parameters xlim and ylim, the function plot() calculates the *default axes ranges* for the X and Y axes based on the range of the values in vectors x and y, respectively. Use the parameters xlim and ylim to declare axes range that are wider than the *default ranges* that are based on the values passed by the parameters x and y. This approach is helpful when you plan to additional draw points and lines that lie outside the default ranges

 To suppress the labeling of the Y axis and/or the X axis, you can include the arguments yaxt="n" and xaxt="n" in the call to function plot(). You then use the function axis() to label these axes.

Let's re-plot Figure 1 using more arguments for function plot(). Execute the following command to obtain Figure 3:

```
> plot(x, y, type="l", xlab="X", ylab="Y", main="exp(X)-3*X^2)")
```


Figure 3. Plot using more arguments with function plot()



The call to function plot() specifies the line type, labels for the X and Y axes, and a main title.

Drawing Special Lines

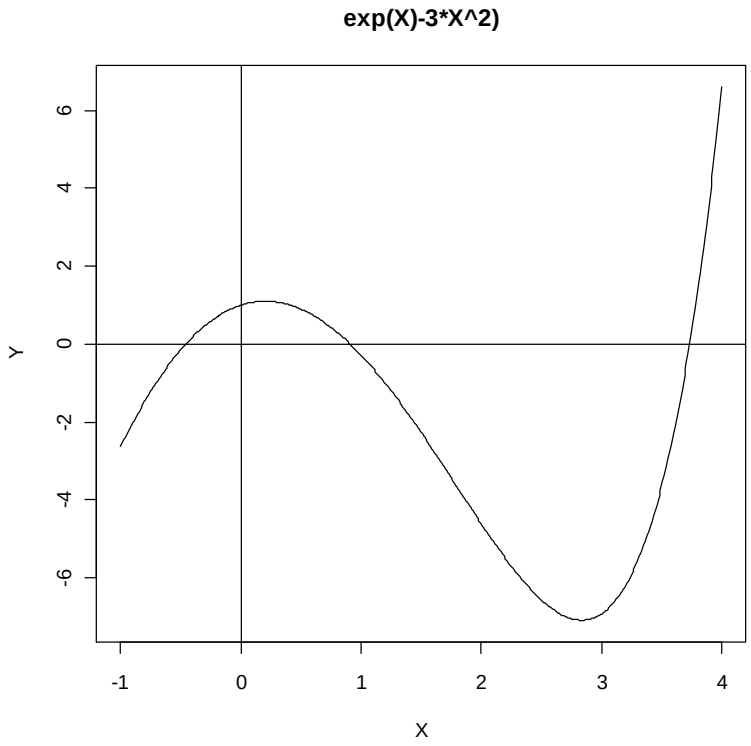
The lines in Figures 1, 2, and 3 look good, but lack vertical and horizontal lines that pass through the origin (0,0). R offers the function abline() to draw vertical and horizontal lines. The form abline(v=x) draws a vertical line at x, and the form abline(h=y) draws a horizontal line at y. You can call function abline() several times, after you first call function plot(), to draw additional lines.

To add the desired lines through the origin, type in the following commands:

```
> abline(v=0)
> abline(h=0)
```

R adds the two lines and yields the plot you see in Figure 4.

Figure 4. Plot with axes through the origin.



Here is a user-defined function, plot.ext() which draws the values in vectors x and y and then draws the Y axis and/or the X axis if the values for y and/or for x cover both positive and negative values. The declaration for function plot.ext() is:

```
plot.ext = function(x, y, type="l", xlab="X", ylab="Y", main="Y vs. X")
{
  x.min = min(x)
  x.max = max(x)
  y.min = min(y)
  y.max = max(y)

  plot(x, y, type=type, xlab=xlab, ylab=ylab, main=main)
  if (x.min <= 0 & x.max >= 0) abline(v=0)
  if (y.min <= 0 & y.max >= 0) abline(h=0)
}
```

The function has the parameters **x**, **y**, **type**, **xlab**, **ylab**, and **main** that parallel the parameters in function plot(). If you execute the following command you get a graph like the one in Figure 4:

```
> plot(x, y, type="l", xlab="X", ylab="Y", main="exp(X)-3*X^2)")
```

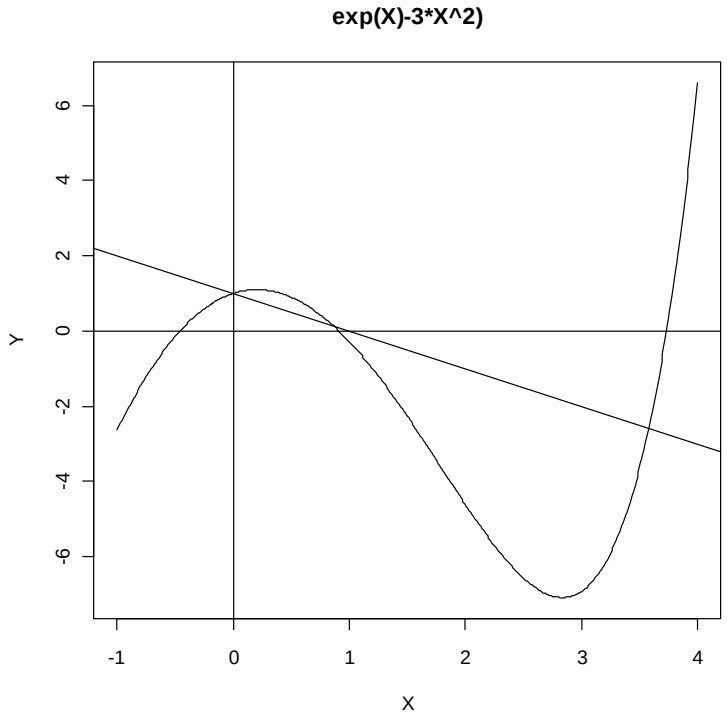
To draw multiple vertical or horizontal lines, in one swoop, using the abline() function, supply a vector of values instead of a single value. For example calling abline(v=c(1,2,3,4)) draws vertical

lines at x = 1, 2, 3, and 4. Another example is calling abline(h=seq(1,2,0.1)) draws eleven horizontal lines at y = 1.0, 1.1, 1.2, ... ,1.8, 1.9, and 2.0.

The abline(intercept, slope) function can also draw a straight line with a given values for the intercept and slope. For example, you can add a line to Figure 4 that has an intercept of 1 and slope of -1 (that is, you are drawing the equation y = 1 –x). Type in the following command to add that line to the current plot in order to obtain Figure 5:

```
> abline(1,-1)
```

Figure 5. Plot with a line of y = 1 - x,



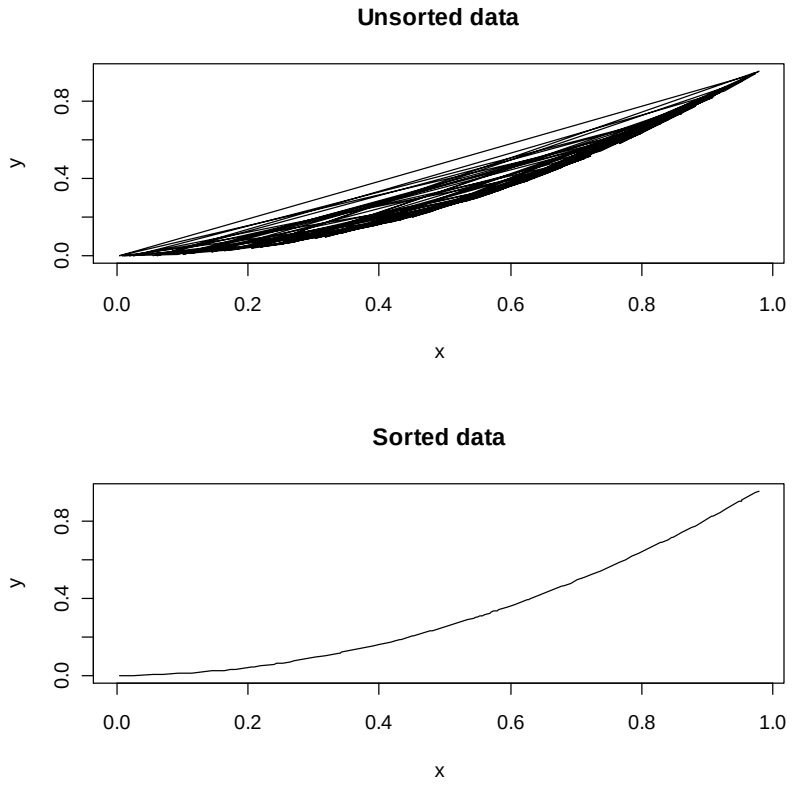
Before we move on to another graphing function, I want to mention that the abline() function accepts a **col** parameter to specify the color of the line(s) you draw. The default value for this parameter is "black".

To draw smooth lines and prevent them from zigzaging between data points, make sure that the arguments for parameter x and y are sorted according to the values in vector x. Use the function order() to obtain the ranks from the argument for vector x and apply these ranks to both vectors x and y. Here is an example that shows how to order the data and get a smooth curve. To

make the example drive the point home, execute the following statements that draw two graphs--one before ordering the data and the other one after:

```
# generate random values in a vector
x = runif(100, 0, 1)
# calculate y as a function of x
y = x^2
par(mfrow=c(2,1))
# plot the curve for the unsorted data
plot(x, y, type="l", main="Unsorted data")
# get the sort order
sort.order=order(x)
# sort the vector x
x = x[sort.order]
# sort the vector y
y = y[sort.order]
# plot the curve for the sorted data
plot(x, y, type="l", main="Sorted data")
par(mfrow=c(1,1))
```


The key to the above example is generating random data. Most examples in this tutorial create data that are ordered with respect to the independent variable. The next two figures show what a difference ordering the data makes! This feature works with the functions plot() and lines().



Drawing Lines

R offers the function lines(x, y, type="l", col="black", lty="solid") to draw additional lines on a graph created by an earlier call to function plot(). The parameter **type** specifies the type of line or points. The parameter **col** designates the color of the line. The parameter **lty** specifies the type of line ("solid" or "dotted").

Here is an example. First create the vector x as having values from 0 to 1 in 0.01 increments. Then using the function plot() graph x and x^2. Notice that the call to function plot() specified the Y axis label to be "x^n for n=0.5,1,2, and 3" to override the default of "x^2" which would be incorrect, since we are plotting multiple curves that correspond to different powers of x. After calling function plot(), use the function lines() to add the following three lines:

- x^3 vs. x
- x vs. x
- √x vs. x

To perform the above tasks in R, type in the following commands:

```
> x=seq(0,1,0.01)
> plot(x,x^2,type="l", ylab="x^n for n = 0.5,1,2, and 3")
> lines(x,x^3)
> lines(x,x)
> lines(x,sqrt(x))
```

Figure 6. Plots for x^3, x^2, x, and x^0.5 vs x.

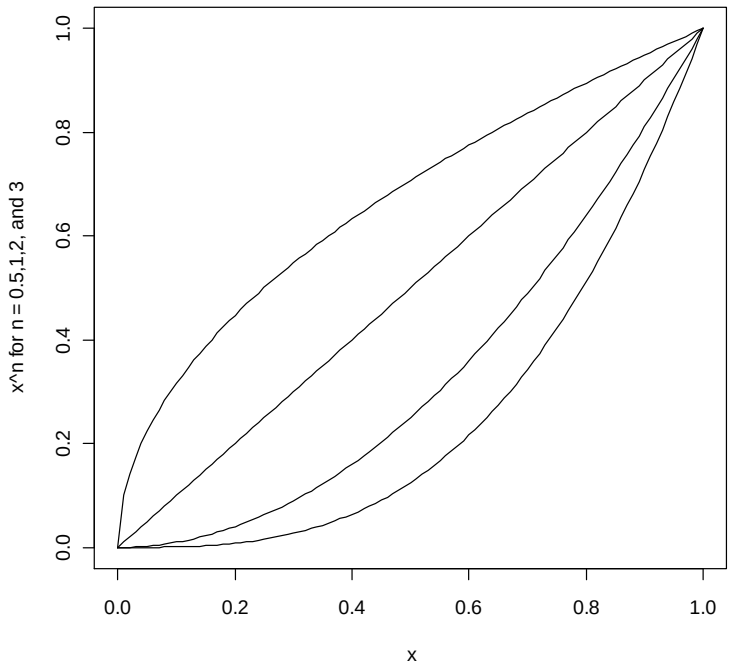


Figure 6 shows the resulting curves. Since you are plotting values between 0 and 1, the lower curves correspond to values for higher powers of x, and vice versa.

Drawing Points

R supplies the function points(x, y = NULL, type = "p", col="black") to draw additional distinct points on a graph. You draw the first set of points by calling function plot() and specifying the "p" value to the type parameter. Then you call function points() to draw additional sets of points. You can call function points() several times, after you first call function plot(), to draw additional points.

Let's redraw Figure 6, using points instead of lines. Execute the following commands that start by assigning a new vector to x with values ranging from 0 to 1 in increments of 0.1. This vector will show points that are not so close to each other. After assigning values to vector x, call

function plot() to plot the values for x vs. x and then call function points() to plot x^3 vs. x, x^2 vs. x, and x^0.5 vs. x:

```
> x=seq(0, 1, 0.1)
> plot(x, x, type="p", ylab="x^n for n = 0.5,1,2, and 3")
> points(x, x^3)
> points(x, x^2)
> points(x, x^0.5)
```

Figure 7. Using the function points() to plot points.

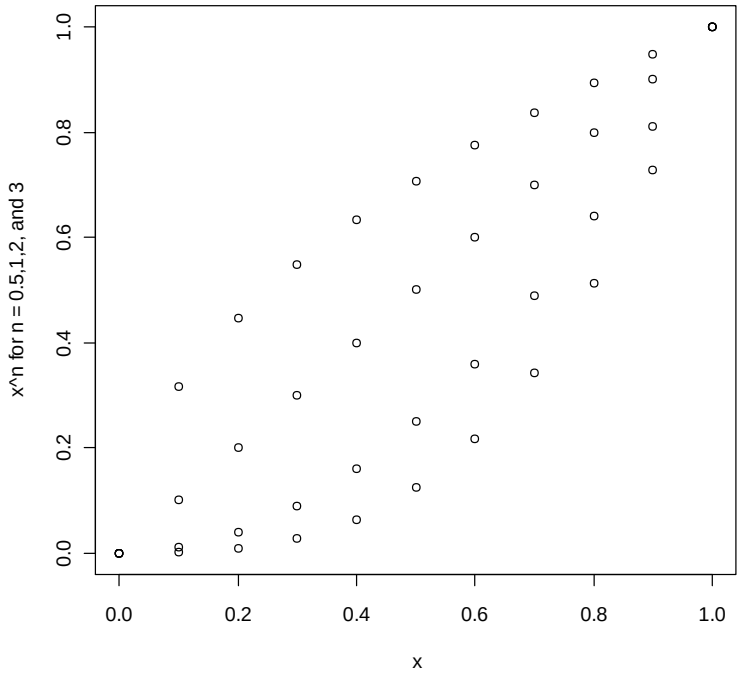


Figure 7 shows the output of plotting data for four curves. The figure is correct but lacks visual clarity. To improve the plot, execute the following calls to function lines() to draw the lines between the points:

```
> lines(x, x)
> lines(x, x^3)
> lines(x, x^2)
> lines(x, x^0.5)
```


Figure 8. Graph with lines and points.

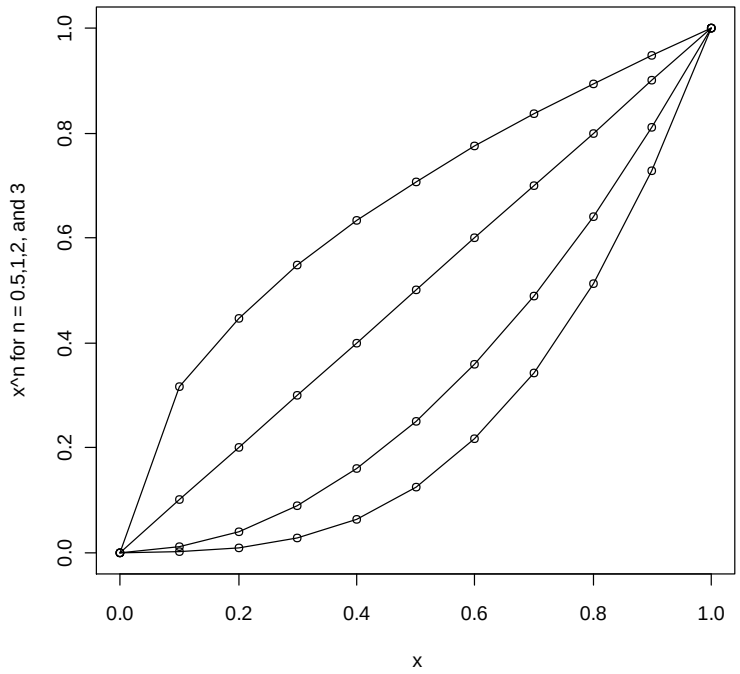


Figure 8 shows a visual improvement over the previous figure!

Graphing Data in Matrices

R offers the functions matplot(), matpoints(), and matlines() to allow you to draw multiple sets of lines, curves, and points using values stored in a matrix. The declarations for the matrix graphing functions are:

```
matplot(x, y, type = "p", lty = 1:5, lwd = 1, lend = par("lend"),
        pch = NULL,
        col = 1:6, cex = NULL, bg = NA,
        xlab = NULL, ylab = NULL, xlim = NULL, ylim = NULL,
        ..., add = FALSE, verbose = getOption("verbose"))

matpoints(x, y, type = "p", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)

matlines (x, y, type = "l", lty = 1:5, lwd = 1, pch = NULL,
          col = 1:6, ...)
```

The parameter **x** is a vector and the parameter **y** is a matrix. This matrix contains several sets of values that the functions uses by drawing the values in each column as a separate set of lines or points.

Here is an illustration for working with function matplot(). Execute the following commands to obtain a graph that closely resembles the one in Figure 6. There is one difference though. Function matplot() draws each line with a different color and line type:

```
> x=seq(0,1,.01)
> y1=sqrt(x)
> y2=x
> y3=x^2
> y4=x^3
> mat.y=matrix(c(y1,y2,y3,y4),ncol=4)
> matplot(x, mat.y, type="l", ylab="x^n for n = 0.5,1,2, and 3")
```

Using the function matplot() allows you to draw in one swoop several curves. The last command replaces several calls to function plot() and lines(). The above commands create several vectors and then use them to assemble a matrix mat.y. The call to function matplot() specifies the first two arguments as the vector x and matrix mat.y.

The functions matpoints() and matlines() work like their vector counterparts functions points() and lines(), except they draw their y coordinate data from matrices instead of vectors.

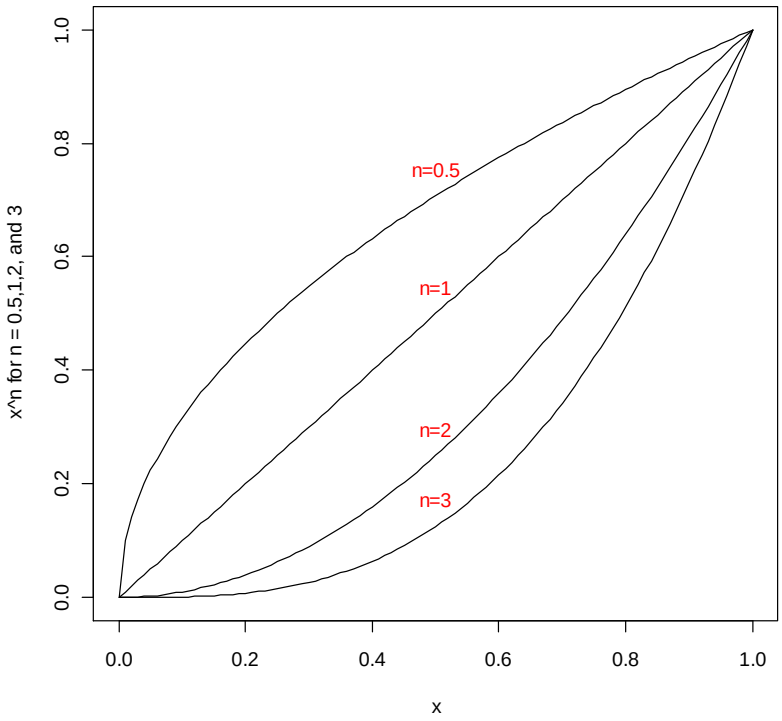
Placing Titles and Text on Graphics

R offers the function text(x, y, message, col="black") to place text inside a plot. The values for parameters **x** and **y** specify the location on the graph where the text of parameter **message** is to appear.

As an example let's plot the four curves in Figure 8 and add four short messages, one on each curve to identify that curve. The color of these messages is red. The calls to function text() should place the labels at x=0.5 and with a vertical shift of 0.5. For example for the curve x^2 vs. x, the argument for x is 0.5 and for y is 0.5^2 + 0.5. Executing the following commands generates the nice plot in Figure 9:

```
> x=seq(0,1,0.01)
> plot(x,x,type="l", ylab="x^n for n = 0.5,1,2, and 3")
> lines(x,x^3)
> lines(x,x^2)
> lines(x,x^0.5)
> text(0.5, 0.55, "n=1", col="red")
> text(0.5, 0.5^3+0.05, "n=3", col="red")
> text(0.5, 0.5^2+0.05, "n=2", col="red")
> text(0.5, 0.5^0.5+0.05, "n=0.5", col="red")
```


Figure 9. The plot of four curves with clarifying labels.



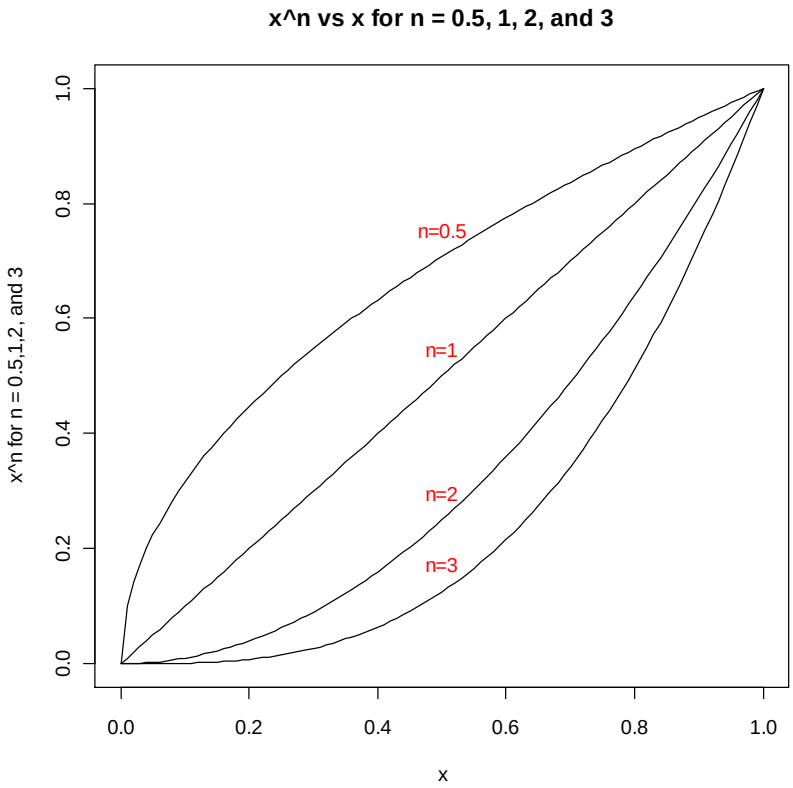
Keep in mind that using the function text() typically requires several attempts to adjust the coordinates so the graph comes out just right.

In addition to function text(), R offers the function title(title.text) to add a title to a graph. You can do the same thing using the argument main in function plot().

To add a title to Figure 9, type the following command to get the graph in Figure 10:

```
> title("x^n vs x for n = 0.5, 1, 2, and 3")
```


Figure 10. Graph with title.



Drawing Rectangles

The R function rect() allows you to draw a colored rectangle and use colored lines to fill it. The declaration for the function rect() is:

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"))
```

The rectangle is defined by the lower left corner coordinate (**xleft**, **ybottom**) and the upper right corner coordinate (**xright**, **ytop**). The parameter **density** defines the number of lines used to fill the rectangle. The parameter **angle** indicates the angle of the lines that fill the rectangle. The parameters **col** and **border** specify the color for the filling lines and the rectangle's border, respectively. The parameter **lty** indicates the line type used to draw and fill the rectangle. You can use arguments line "solid" or "dotted". The parameter **lwd** specifies the width of the lines used to draw and fill the rectangle. You can call function rect() several times, after you first call function plot(), to draw additional rectangles.

Using the rect() function and does some leg work you can write your own functions that draws histograms with different rectangular parts having different colors and fill styles. You can create your own version of function hist() (R's histogram-drawing function) on steroids.

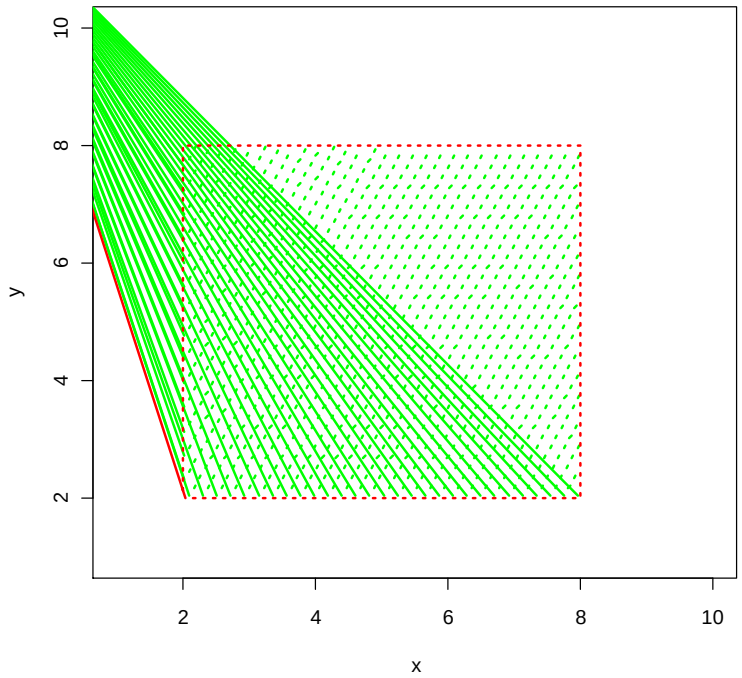
Let's use of function rect(). Define a vector x as having numbers between 1 to 10, in increments of 1. Define the vector y as equal to the vector x. Use the plot() to create the graph but not skip plotting the y vs. x data. You do this by assigning "n" to the parameter type when you call function plot(). Then call function rect() to draw a colored rectangle that has the following specifications:

- The rectangle has the lower left corner of (2,2).
- The rectangle has the upper right corner of (8,8).
- The lines that fill the rectangle have a density of 10.
- The lines that fill the rectangle have a 60 degrees angle.
- The rectangle has a red border and green filling lines.
- The rectangle has a dotted border and dotted fill lines.

Execute the following commands to obtain the rectangle in Figure 11:

```
> x=1:10
> y=x
> plot(x,y,type="n")
> rect(2, 2 , 8, 8, density=10, angle=60, col="green", border="red", lwd=2,
lty="dotted")
```

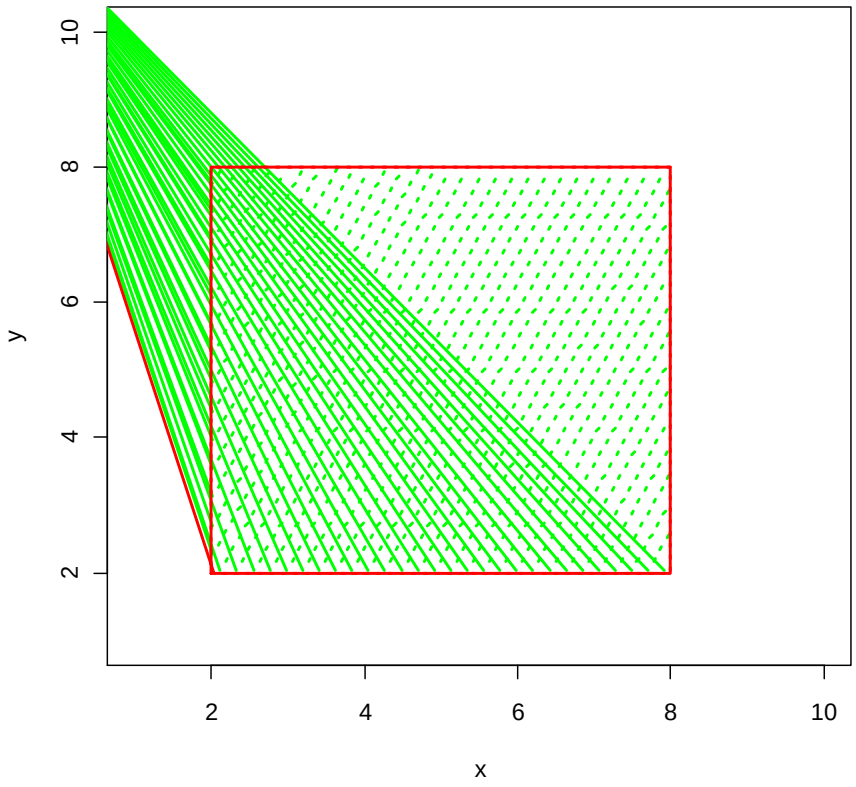
Figure 11. A colored rectangle.



To draw a solid border and dotted file lines, you have to call the function rect() twice. The first call draws the dotted fill lines and dotted border. The second function call draws only a solid border (no fill). Here is an example of R code that generates the rectangle in Figure 12:

```
> x=1:10
> y=x
> plot(x,y,type="n")
> rect(2, 2 , 8, 8, density=10, angle=60, col="green", border="red", lwd=2,
lty="dotted")
> rect(2, 2 , 8, 8, density=0, border="red", lwd=2, lty="solid")
```

Figure 12 A colored rectangle with solid border and dotted fill lines.



Drawing Polygons

The function polygon() allows you to draw a colored polygon and use colored lines to fill it. The declaration for the function polygon() is:

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"))
```


The polygon is defined by the vectors **x** and **y** which store the x and y coordinates which are the vertices of the polygon. The parameter **density** defines the number of lines used to fill the polygon. The parameter **angle** indicates the angle of the lines that fill the polygon. The parameters **col** and **border** specify the color for the filling lines and the polygon border, respectively. The parameter **lty** indicates the line type used to draw and fill the polygon. You can use arguments line "solid" or "dotted". You can call function polygon() several times, after you first call function plot(), to draw additional polygons.

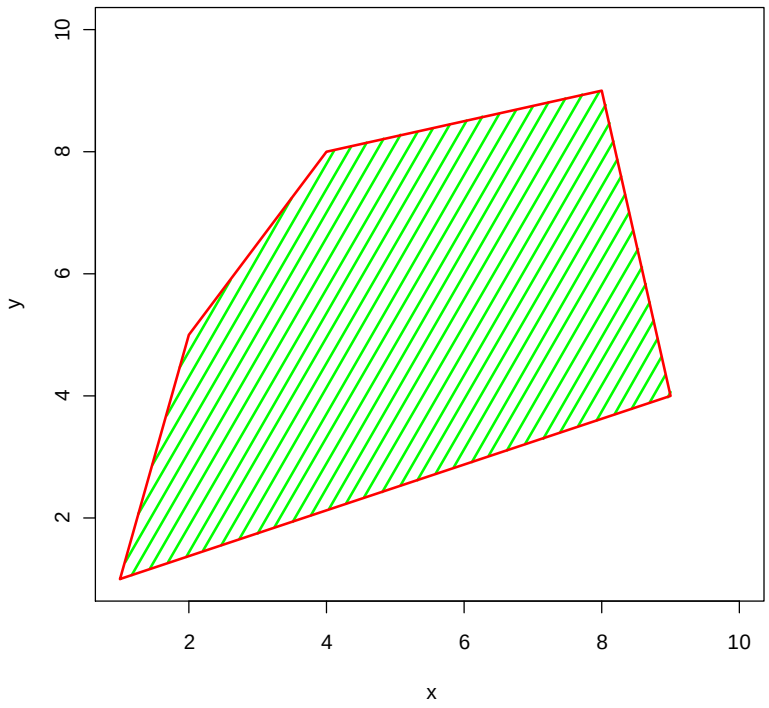
Let's draw a solid and closed polygon with the following specifications:

- The polygon has the vector x as the values 1,2,4,8,9, and 1.
- The polygon has the vector y as the values 1,5,8,9,4, and 1.
- The lines that fill the polygon have a density of 10.
- The lines that fill the polygon have a 60 degrees angle.
- The polygon has a red border and green filling lines.
- The polygon has a solid border and solid fill lines.

Execute the following commands to obtain the polygon that appears in Figure 13:

```
> x=1:10
> y=x
> plot(x,y,type="n")
> xp=c(1,2,4,8,9,1)
> yp=c(1,5,8,9,4,1)
> polygon(xp,yp, density=10, angle=60, col="green", border="red", lwd=2,
lty="solid")
```


Figure 13. A colored and filled polygon.



Adding Axes

Normally, the plot() functions labels the bottom side and the left side of a graph. You can use the axis(side) function to duplicate these labels to the right side and top side of the graph. The sides of a graph are numbered 1 (bottom side), 2, (left side), 3, (top side), and 4 (right side). The function axis() has additional parameters, but we will focus only on the first parameter-the side number

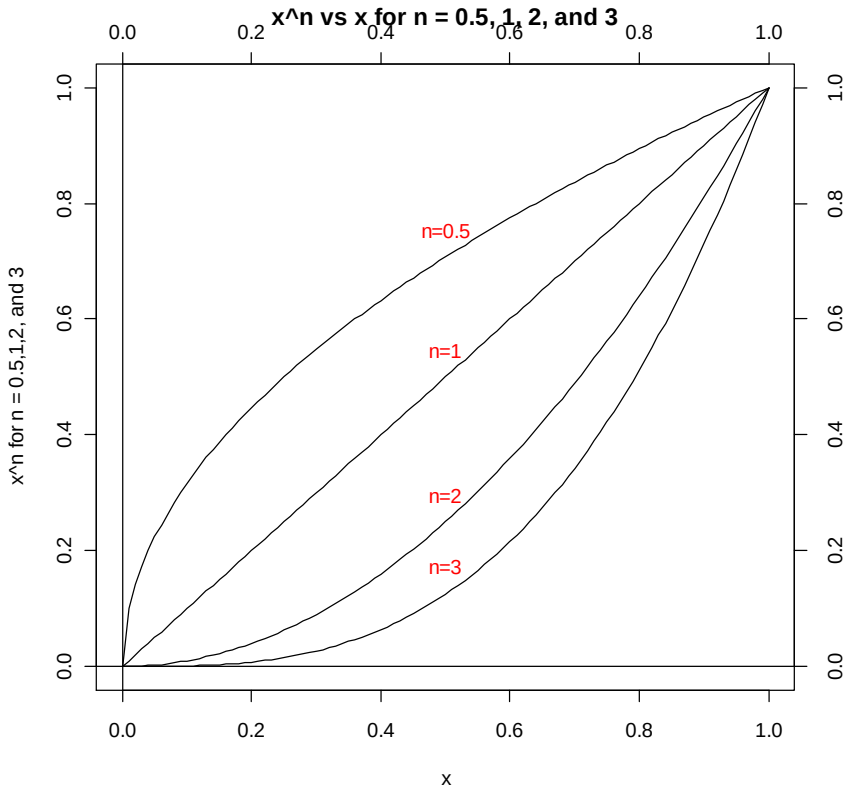
To illustrate using the function axis(), let's enhance Figure 10 by adding labels to the right and top sides of the graph. Also, let's use the function abline() to draw the vertical and horizontal lines that pass through the origin (0,0). Execute the following commands to obtain Figure 14:

```
> x=seq(0,1,0.01)
> plot(x,x,type="l", ylab="x^n for n = 0.5,1,2, and 3")
> lines(x,x^3)
> lines(x,x^2)
> lines(x,x^0.5)
> text(0.5, 0.55, "n=1")
> text(0.5, 0.5^3+0.05, "n=3")
> text(0.5, 0.5^2+0.05, "n=2")
```



```
> text(0.5, 0.5^0.5+0.05, "n=0.5")
> title("x^n vs x for n = 0.5, 1, 2, and 3")
> axis(3)
> axis(4)
> abline(v=0)
> abline(h=0)
```

Figure 14. Graph with double-sided labeled axes.



Adding a Grid

R offers the function grid() to add a grid inside the plot. The declaration for the function grid() is:

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",
     lwd = par("lwd"), equilogs = TRUE)
```

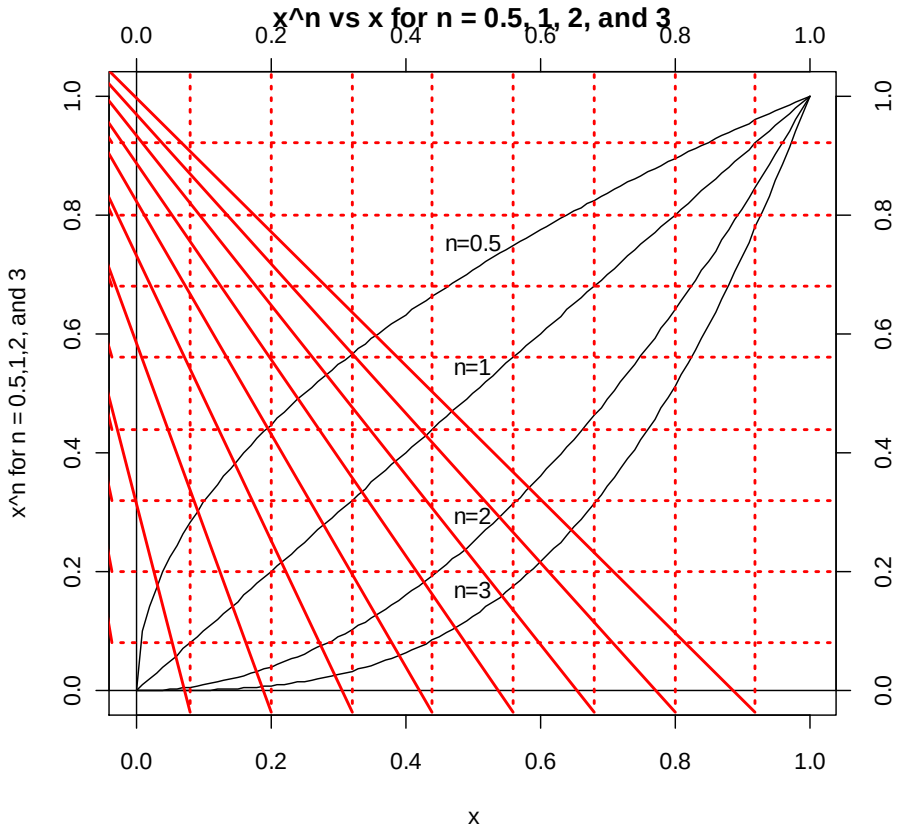
The parameters **nx** and **ny** specify the number of cells in the grid, in the x and y dimensions, respectively. The par **col** selects the color for the grid. The default value for this parameter is "lighgray". The parameter **lty** designates the line type. The default value is "dotted". The parameter **lwd** represents the line width. The function uses Boolean parameter **equilogs** only when log coordinates and alignment with the axis tick marks are active. Setting parameter equilogs to FALSE in that case yields non-equidistant tick aligned grid lines.

Here is an example for using the function grid(). With the graph in Figure 14 in view (if not recreate it by typing in the set of commands in the last subsection) type in the following command to add a red grid:

```
> grid(9, 9, lwd=2, col="red")
```

The call to function grid specifies 9 cells for the x and y dimensions, a line width of 2, and red color for the grid lines. The above command yields the graph in Figure 15.

Figure 15. Graph with a grid.



Locating the Coordinates on a Graph

R offers the parameter-less function locator() that returns the vectors of x and y coordinates for one or more points on a graph. When you invoke function locator(), the runtime system switches into a learning mode. The mouse cursor becomes a plus icon. You can move the mouse over the graph and click over the points whose coordinates you seek. You can click as many times as you like (of course going ballistic with this feature only asks for trouble). When you are done selecting points on the graphs, click the right mouse button. The runtime system responds by displaying a small popup menu with the options Stop and Continue. If you want to continue

selecting points on the graph, click the Continue option. Otherwise select the Stop option. Once you do, the function locator() returns a list structure for the coordinates of the points you clicked over. Here is a sample output for a session I had using function locator() while the graph in Figure 15 was still active:

```
> locator()
$x
[1] 0.4964277 0.4964277 0.4964277 0.4964277

$y
[1] 0.1309557 0.2519150 0.5009487 0.7049192
```

The function returns a list with the vectors tagged x and y as list elements. These elements contain the x and y coordinates for the points on the graph that you clicked on. The points I clicked on were (0.4964277, 0.1309557), (0.4964277, 0.2519150), (0.4964277, 0.5009487), and (0.4964277, 0.7049192)

Plotting Histograms

The function hist() allows you to draw histograms for a vector of data. The function can perform calculations that automatically generate the histograms and display their frequencies. The declaration of the hist() function is:

```
hist(x, breaks = "Sturges",
     freq = NULL, probability = !freq,
     include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL,
     main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL,
     xlab = xname, ylab,
     axes = TRUE, plot = TRUE, ...)
```

The parameter **x** represents the vector of data to plot in a histogram. The parameter breaks is a vector that defines the histogram cells. The parameter **freq** indicates whether to display the count (when NULL or TRUE) or the fractional density when set to FALSE. The parameter **probability** is an alias for !freq and is used for compatibility with the S programming language. The parameter **density** defines the number of lines used to fill the rectangle. The Boolean parameter **include.lowest** sets, when the parameter is TRUE, x[i] equal to the breaks value will be included in the first (or last, for right = FALSE) bar. The function ignores this parameter (with a warning from the function) unless parameter breaks is a vector. The Boolean parameter **right** makes histograms cells to be right-closed (left open) intervals, when the parameter is set to TRUE.

The parameter **density** defines the number of lines used to fill the histogram. The parameter **angle** indicates the angle of the lines that fill the histogram. The parameter **col** and **border**

specify the color for the filling lines and the histogram's border, respectively. The parameter **main** specifies the main title for the histogram. The parameters **xlim** and **ylim** define the limits of the x and y values, respectively. The parameters **xlab** and **ylab** specify the labels for the X axis and Y axis, respectively. The Boolean parameter **axes** is a flag that tells the function to draw the axes if the histogram is drawn. The Boolean parameter **plot** is a flag that plots the histogram when set to TRUE, and otherwise returns a list of break values and histogram counts.

To illustrate using the hist() function, first create a vector x containing 1000 normally-distributed random numbers with a mean of 0 and standard deviation of 2. Then invoke the hist() function with the following arguments:

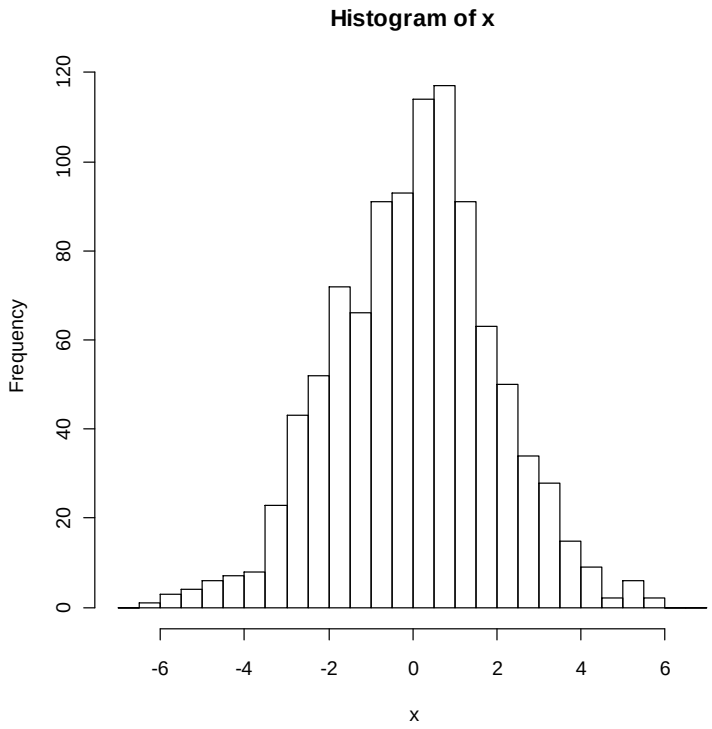
- The argument for the breaks parameter is seq(-7, 7, 0.5).
- The argument for the parameter freq is TRUE.

When you execute the following commands, you obtain the histogram in Figure 16:

```
> x = rnorm(1000,0,2)
> hist(x,seq(-7,7,0.5),freq=TRUE)
```

Keep in mind that the histogram you get will most likely differ from the one you see in Figure 15. In fact, each time you call function rnom() you will get a different vector of random numbers.

Figure 16. Histogram for normally-disturbed random numbers.

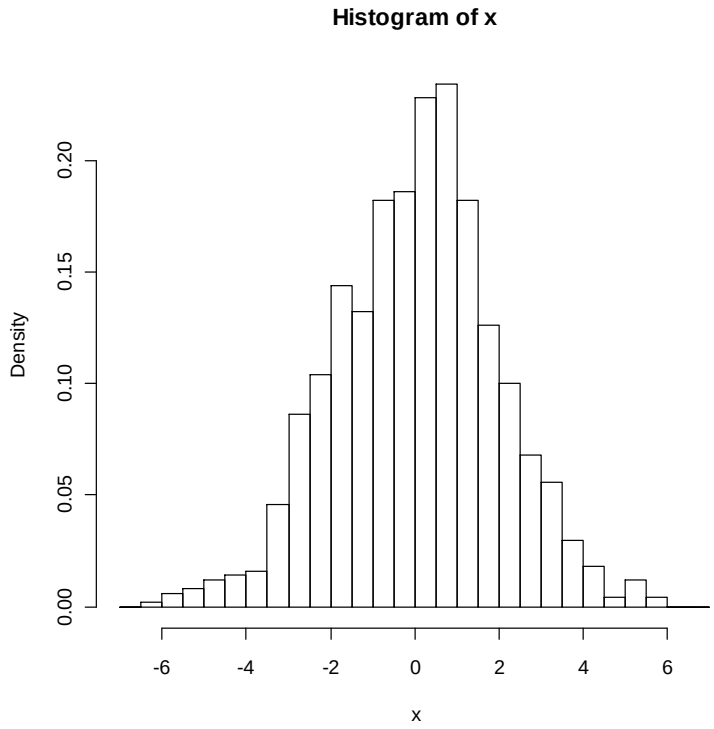


Now execute the last command, setting FALSE the argument to parameter freq in order to obtain the histogram in

Figure 17:

```
> hist(x,seq(-7,7,0.5),freq=FALSE)
```

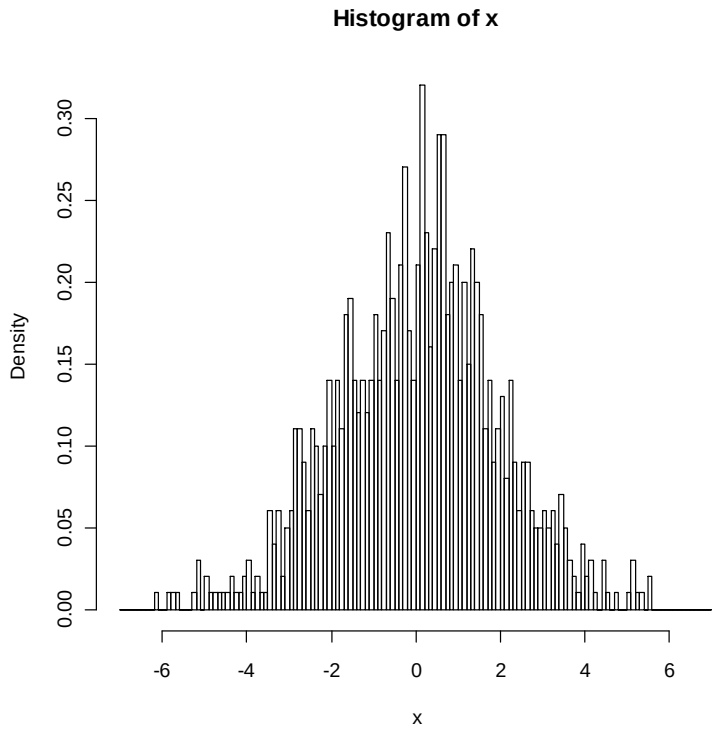

Figure 17. Histogram for normally-disturbed random numbers.



Now execute the last command again, replacing the value 0.5 with 0.1 to obtain the histogram in Figure 18. The new histogram has five times as many cells as the one in Figure 17:

```
> hist(x,seq(-7,7,0.1),freq=FALSE)
```


Figure 18. Histograms with more and smaller cells.



Plotting Pie Charts

The function pie() allows you to draw a pie chart based on the value in an vector. the declaration for the function pie() is:

```
pie(x, labels = names(x), edges = 200, radius = 0.8,
    clockwise = FALSE, init.angle = if(clockwise) 90 else 0,
    density = NULL, angle = 45, col = NULL, border = NULL,
    lty = NULL, main = NULL, ...)
```

The parameter **x** is the vector that contains the data to be plotted in a pie chart. The parameter **labels** is a vector of strings used to label each slice of the pie chart. The parameter **edges** specifies the number of polygon edged used to approximate the pie chart. The parameter **radius** specified the radius of the pie chart which is drawn centered in an invisible square box whose sides range from -1 to 1. When the strings labeling the various slices are long, the function may employ a smaller radius to draw the pie chart. The Boolean parameter **clockwise** indicates if the function draws the slices in a clockwise (when TRUE) or counter clockwise (when FALSE). The parameter **init.angle** specifies the starting angle (in degrees) for drawing the slices. The default is 0 which draws the first slice at middle right direction (think of it as 3 o'clock on a watch). The arguments of 30, 60, and 90 cause the starting angle to be at the bottom (like 6 o'clock), middle

left side (like 9 o'clock), and the top (like 12 o'clock), respectively. The parameter **density** defines the number of lines used to fill the pie chart. The parameter **angle** indicates the angle of the lines that fill the pie chart. The parameters **col** and **border** specify the color for the filling lines and the pie chart's border, respectively. The parameter **lty** indicates the line type used to draw and fill the pie chart. You can use arguments line "solid" or "dotted". The parameter **main** specifies the pie chart title.

Let's first draw a simple pie chart and rely on the default values of the parameters. Create the vector x using the following command:

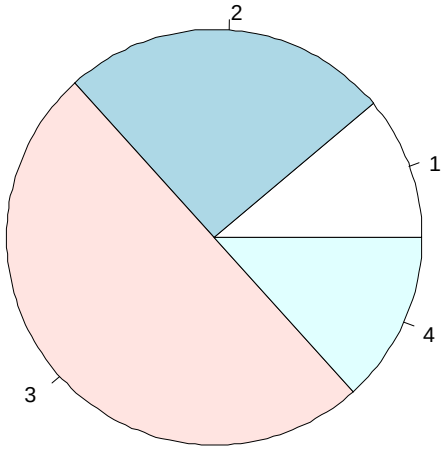
```
> x=c(10, 23, 45, 12)
```

Now use the data in vector x to draw a pie chart:

> pie(x)

The above command creates the colored pie chart in Figure 19.

Figure 19. Simple pie chart.

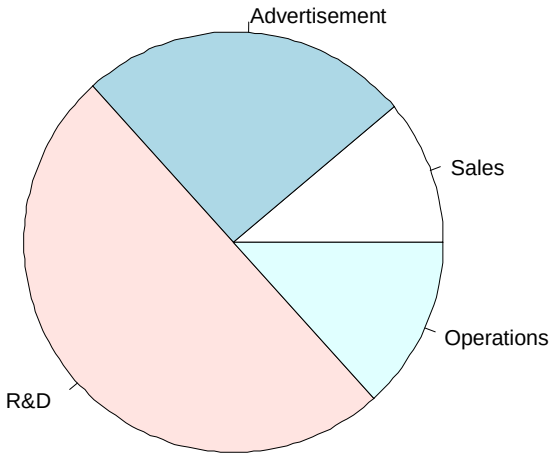


Notice that the pie chart labels the slices using the indices of vector x.

Let's redraw Figure 18 this time we add labels for the slices and obtain the pie chart in Figure 20:

```
> pie(x, c("Sales", "Advertisement", "R&D", "Operations"))
```

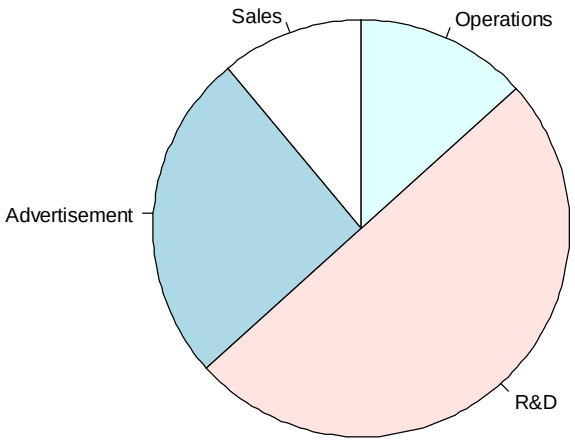

Figure 20. Pie chart with labeled slices.



Finally, let's redraw the pie chart in Figure 19 by assigning 90 degrees (think of it as 12 o'clock on a watch) to the initial.angle parameter to obtain the rotated pie chart in Figure 21:

```
> pie(x, c("Sales", "Advertisement", "R&D", "Operations"), init.angle=90)
```


Figure 21. The rotated pie chart.



Linear Regression Plot

Using the functions plot(), abline(), and text(), you can write a custom function that performs linear regression calculations on vectors of x and y, obtain the regression intercept and slope, plot the original data and the regression line, and return the regression intercept and slope. The next function reg,plot() performs all of these tasks:

```
reg.plot <- function(x, y, xt, yt, decimals)
{
  # get the number of data points
  n = length(x)
  # get the mean and variances
  meanx = mean(x)
  meany = mean(y)
  sxy = sum(x*y) - meanx * meany * n
  sxx = sum(x*x) - meanx^2 * n
  # calculate the regression intercept and slope
  slope = sxy / sxx
  intercept = meany - slope * meanx
  # plot the observations
  plot(x, y, type="p", main="Linear Regression Plot", col="red")
  # plot the regression line
  abline(intercept, slope)
```



```
  text(xt, yt, paste("Y = (", round(intercept, decimals),
                     ") + (", round(slope, decimals), ") x"))
  # return the results
  return (c(intercept, slope))
}
```

The function reg.plot() has the following parameters:

- The vectors **x** and **y** that provide the data for the linear regression.
- The parameters **xt** and **yt** provide the location on the graph where the plot shows the linear regression model.
- The parameter **decimals** is the number of decimals used to round the linear regression intercept and slope when displayed on the graph.

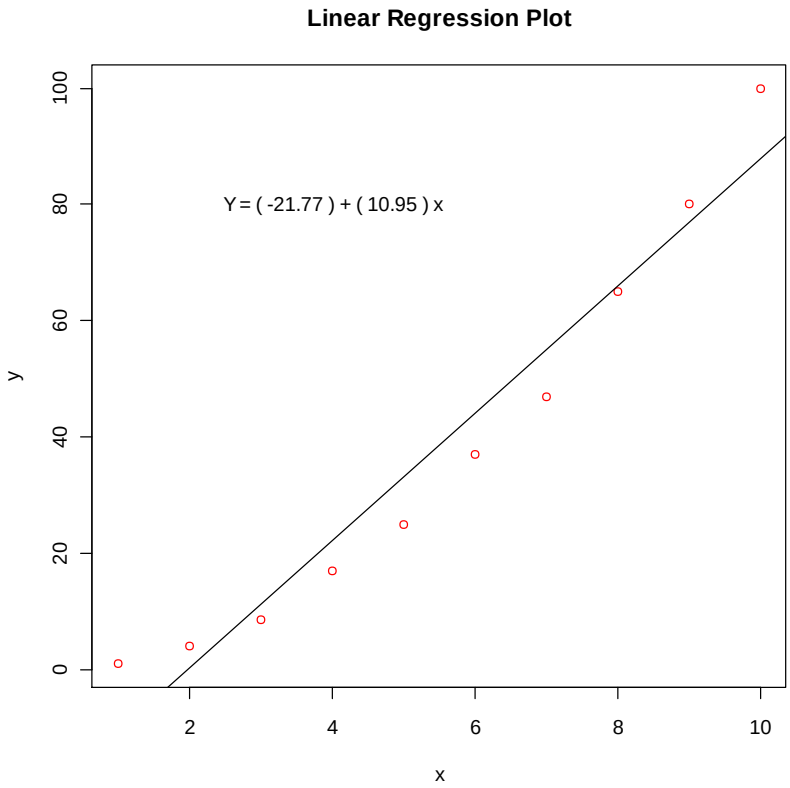
Save the above function in file reg.plot.r and then load it, before you use it in the following commands that perform the following tasks:

- Assign values to vectors x and y.
- Call function reg.plot() to perform the regression. Using arguments make the function display the fitted equation at point (4,80) and also round the regression slope and intercept to 2 decimals when displayed on the graph:

```
> source("reg.plot.r")
> x = c(1,2,3,4,5,6,7,8,9,10)
> y = c(1,4.1,8.5,17,25,37,47,65,80,100)
> reg.plot(x, y, 4, 80, 2)
[1] -21.76667  10.95030
```

The function creates the graph in Figure 22. The function draws the observed data in red. It also displays the best fit linear model on the graph at (4,80) with the values of the regression intercept and slope rounded to 2 decimals.

Figure 22. Linear regression plot.



Displaying Multiple Plots

R permits you to display multiple plots on the graph output window. Of course, the more plots you draw, the smaller each plot gets. The approach R uses is to consider the plots as fitting in a visual matrix. The simplest case is to have two plots in a single row, or in a single column. When dealing with three or four plots, you arrange them in two rows and two columns, one row and three columns, or three rows and one column. In the case of five or six plots you can arrange them in two rows and three columns, or three rows and two columns.

R offers the par() function as a way to set and reset the number of plots. You have two choices:

```
par(mfrow = c(nrows, ncols))
par(mfcol = c(nrows, ncols))
```

In the first form, you tell R that you want to have multiple plots in nrows rows and ncols columns, filled sequentially by row. In the second form, you tell R that you want to have multiple plots in nrows rows and ncols columns, filled sequentially by column. To restore the graphing of a single plot you can use either commands as follows:

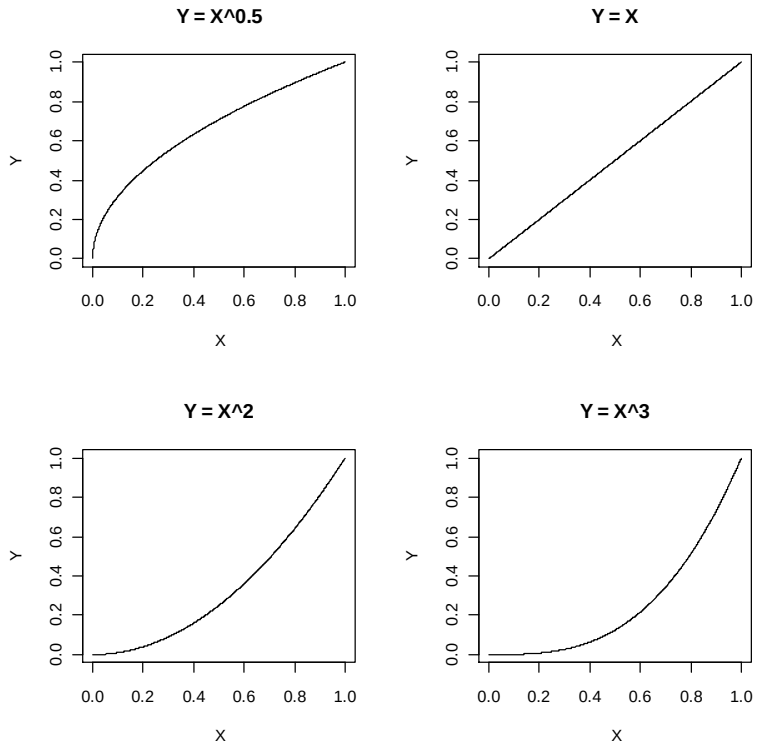

```
par(mfrow = c(1,1))
par(mfcol, c(1,1))
```

Let's illustrate the ability to display multiple plots. The following commands create a vector x and then plot x^0.5 vs. x, x vs. x, x^2 vs. x, and x^3 vs. x:

```
> x=seq(0,1,0.001)
> par(mfrow = c(2,2))
> plot(x, x^0.5, type="l", ylab = "Y", xlab="X", main="Y = X^0.5")
> plot(x, x, type="l", ylab = "Y", xlab="X", main="Y = X")
> plot(x, x^2, type="l", ylab = "Y", xlab="X", main="Y = X^2")
> plot(x, x^3, type="l", ylab = "Y", xlab="X", main="Y = X^3")
> par(mfrow = c(1,1))
```

Notice that the last command restores the single plot mode. Figure 23 shows the multiple plots. Observe the sequence in which the plots appear—by row.

Figure 23. Multiple plots



While we are on the subjects of multiple plots, it is worthwhile mentioning a special call to function par() that makes the R runtime system prompt you to click the mouse when viewing a graph, before viewing the next graph. After all, even with displaying multiple graphs in one window, there are but so many graphs you can show at one time. The following command turns on the user-prompt mode:


```
> par(ask=TRUE)
```

When the R interpreter has plotted a graph and encounters commands to plot another graph, the runtime system displays the following message:

```
Waiting to confirm page change...
```

When you see the above message, move the mouse over the graph window and click the mouse button to view the next graph. Use the function call par(ask=FALSE) to turn off the user-prompt mode. Using the par() function allows you to take your time in viewing graphs and not agonize with multiple graphs that whiz by in the blink of an eye.

Log-Log and Semi-Log Graphs

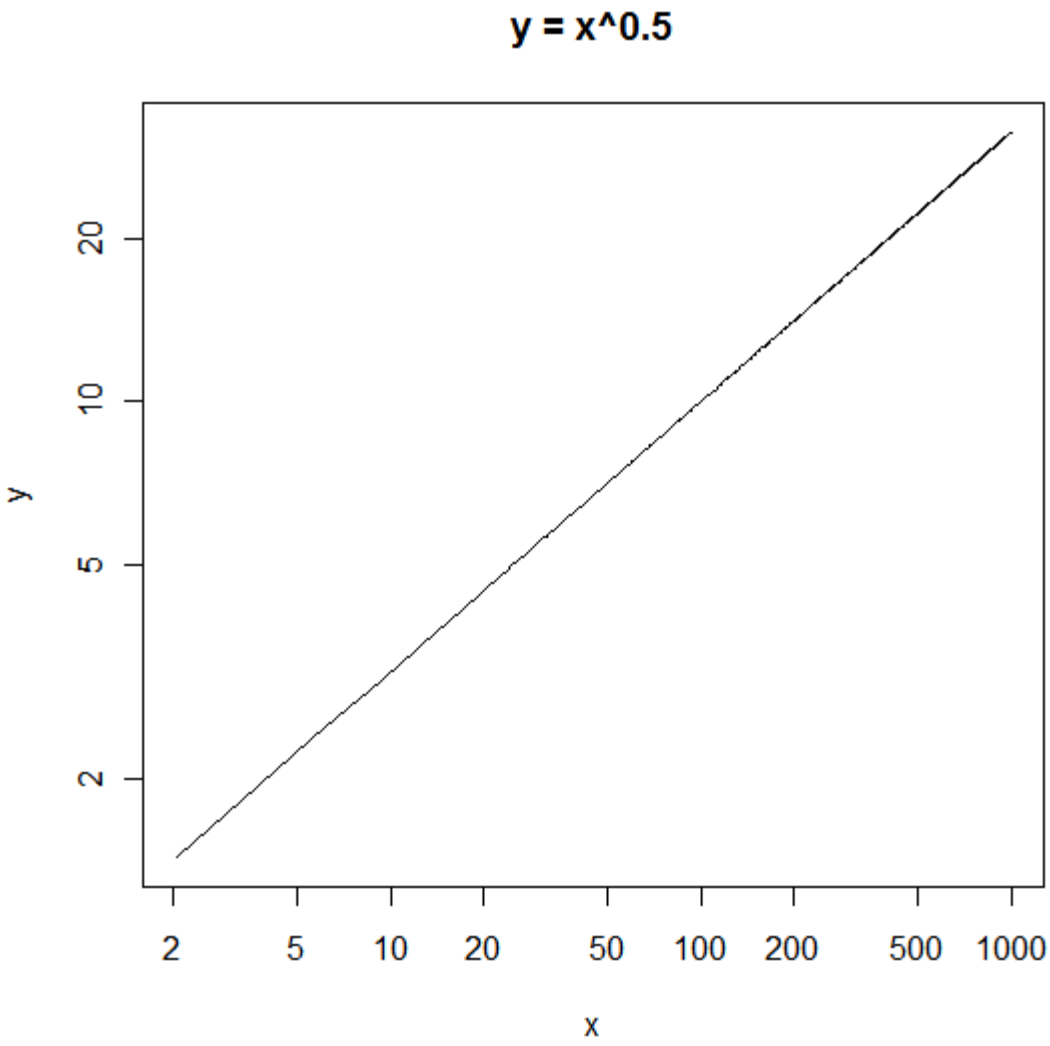
The function plot() offers the **log** parameter that allows you to flag plotting the X and/or Y axes using logarithmic scale. Using the argument log="x" plots the X axis using logarithmic scale. Using the argument log="y" plots the Y axis using logarithmic scale. Using the argument log="xy" plots the both X and Y axes using logarithmic scale.

Here is an example. Type in the following commands to create a log-log plot using random square root values in the range of 1 to 1000:

```
> x=runif(1000,1,1000)
> x=sort(x)
> y=x^0.5
> plot(x, y, type="l", log="xy", main="y = x^0.5")
```

The last command creates the graph in Figure 24.

Figure 24. Log-log plot.



You can create visually improved versions of the logarithmic plots using the log10 of the x and/or y data to obtain semi-log and log-log graphs. In addition, you may need to use abline() function to draw vertical and horizontal lines for the various decades in the x and y axes.

Here is the code for the function that makes log-log plots:

```
plot.loglog <- function(x, y, type="p", xlab="Log(X)", ylab="Log(Y)",
                        main="Log-Log Plot", col="red")
{
  # plot the log-log graph
  plot(log10(x), log10(y), type=type, xlab=xlab, ylab=ylab,
       main=main, col=col)
  factor = 1:10
```



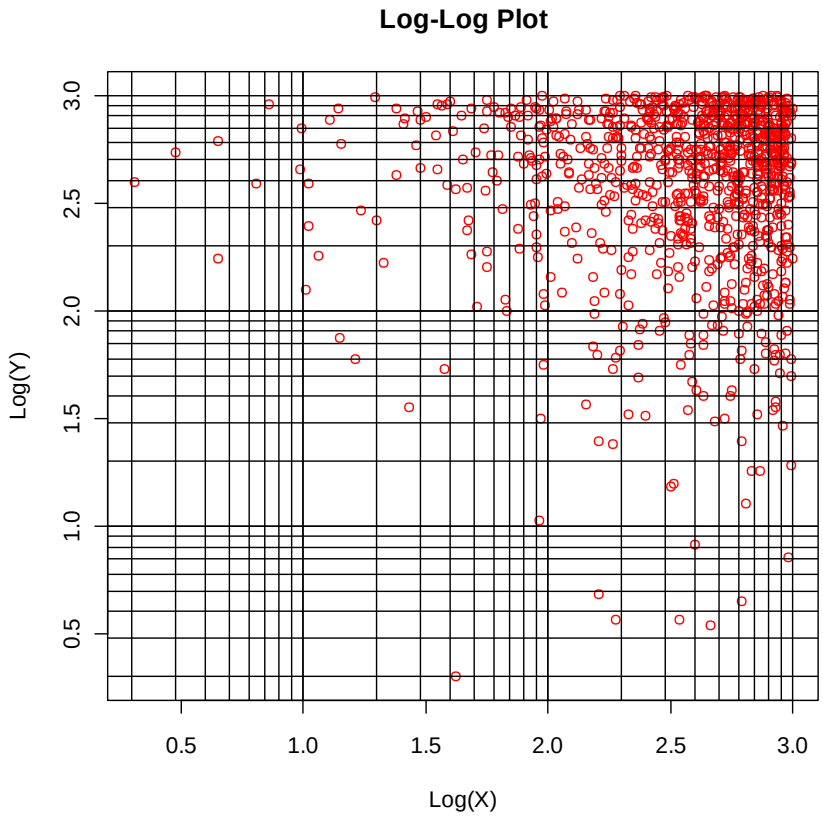
```
  # draw lines on the Y axis
  i1 = as.integer(log10(min(y)))
  i2 = as.integer(log10(max(y)))
  if (i2 > i1) {
    for (i in i1:i2) {
      abline(h=log10(factor*10^i))
    }
  }
  # draw lines on the X axis
  i1 = as.integer(log10(min(x)))
  i2 = as.integer(log10(max(x)))
  if (i2 > i1) {
    for (i in i1:i2) {
      abline(v=log10(factor*10^i))
   }
  }
}
```

Save the function in file plot.loglog.r. Load the function in the workspace and execute the following commands to plot 1000 uniformly distributed random points (x, y) that have both the x and y values in the range of 1 to 1000:

```
> source("plot.loglog.r")
> x=runif(1000, 1, 1000)
> y=runif(1000, 1, 1000)
> plot.loglog(x, y)
```

Figure 25 shows the log-log plot generated. The random data points appear in red, the default color. You can specify the color by argument when calling function plot.loglog().

Figure 25. A sample log-log plot.

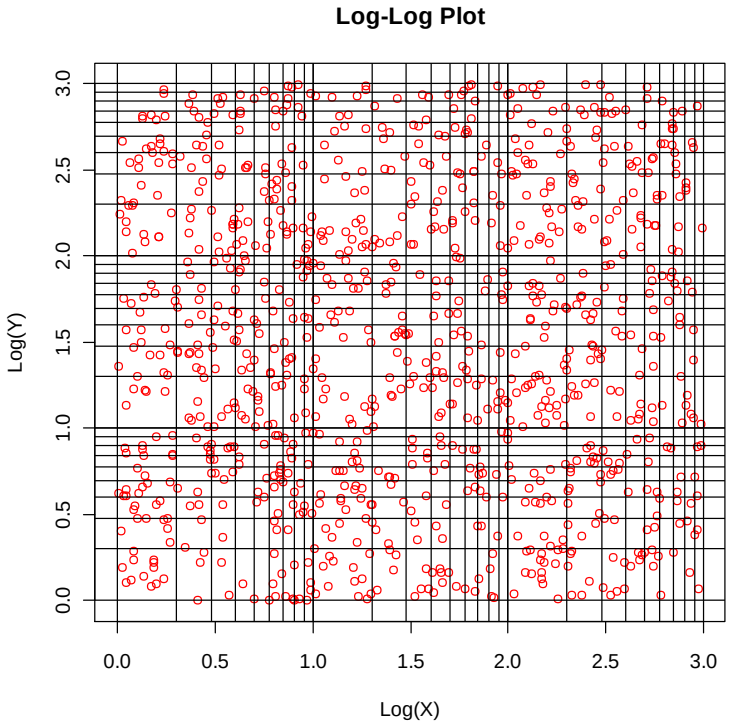


To obtain a more uniform spread of the data on the log-log plot, execute the following commands:

```
> x=runif(1000, log10(1), log10(1000))
> y=runif(1000, log10(1), log10(1000))
> plot.loglog(10^x, 10^y)
```

The calls to functions runif() create uniform random number on the logarithmic scale. Figure 26 shows the results with the data points spread more uniformly.

Figure 26. More uniformly spread data points.



Here is the code for a function that plots log(y) vs. x:

```
plot.logy <- function(x,y,type="p",xlab="X",ylab="Log(Y)",
                      main="Semilog Plot", plot.vect=c(), col="red")
{
  # plot the log(y) vs. x graph
  plot(x, log10(y), type=type, xlab=xlab, ylab=ylab, main=main, col=col)
  factor = 1:10
  # draw lines on the Y axis
  i1 = as.integer(log10(min(y)))
  i2 = as.integer(log10(max(y)))
  if (i2 > i1) {
    for (i in i1:i2) {
      abline(h=log10(factor*10^i))
    }
  }
  if (length(plot.vect) > 0) {
    abline(v=plot.vect)
  }
}
```

The function plot.logy() has the vector parameter **plot.vect** which specifies the values of X where the function draws vertical lines. By default this vector is empty and the function draws no vertical lines. The parameter **col** specifies the color of the plotted data points. The default color is red.

Next you find the code for a function that plots y vs. log(x):

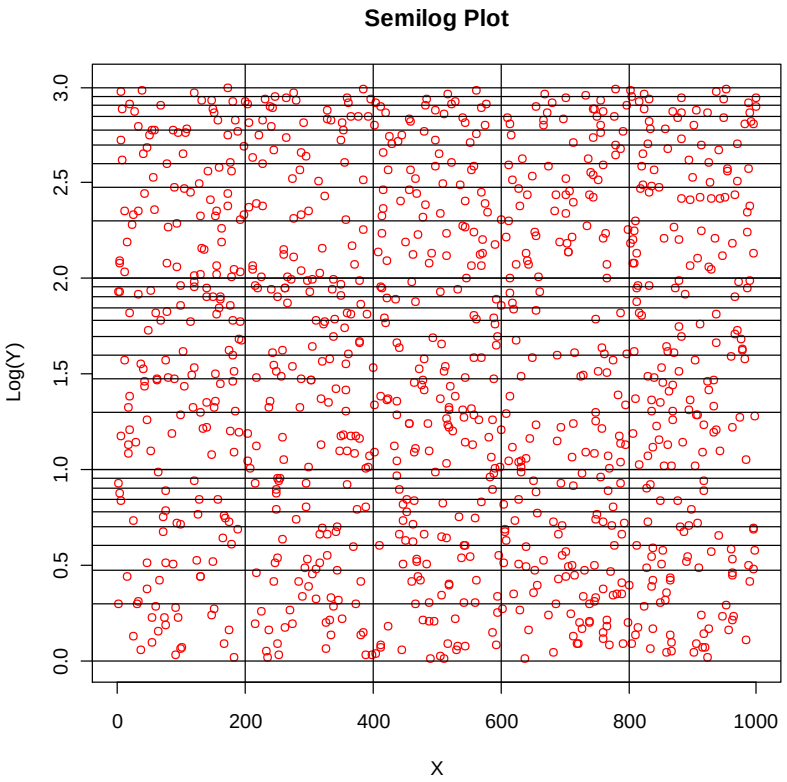
```
plot.logx <- function(x,y,type="p",xlab="Log(X)",ylab="Y",
                      main="Semilog Plot", plot.vect=c(), col="red")
{
  # plot the y vs. log(x) graph
  plot(log10(x), y, type=type, xlab=xlab, ylab=ylab, main=main, col=col)
  factor = 1:10
  # draw lines on the X axis
  i1 = as.integer(log10(min(x)))
  i2 = as.integer(log10(max(x)))
  if (i2 > i1) {
    for (i in i1:i2) {
      abline(v=log10(factor*10^i))
    }
  }
  if (length(plot.vect) > 0) {
    abline(h=plot.vect)
  }
}
```

The function plot.logx() has the vector parameter **plot.vect** which specifies the values of Y where the function draws horizontal lines. By default this vector is empty and the function draws no horizontal lines. The parameter **col** specifies the color of the plotted data points. The default color is red.

Here is an example for using the plot.logy() function to generate Figure 27:

```
> source("plot.logy.r")
> x=runif(1000, 1, 1000)
> y=runif(1000, log10(1), log10(1000))
> plot.logy(x, 10^y, plot.vect = seq(200,800,200))
```


Figure 27. A sample semi-log plot.



The above command calls function plot.logy() to make a semi-log plot of the random data in vectors x and 10^y. The call also supplies the vector (200, 400, 600, 800) to the parameter plot.vect. The function calls plots the data in red, draws the Y axis in a logarithmic scale, and draws the X axis in a linear scale. The function call also draws vertical lines at X = 200, 400, 600, and 800.

Drawing 3D Surfaces and Contours

R offers the functions persp() and contour() to draw three-dimensional surfaces and their contours, respectively. Drawing surfaces and contours in R shares many preparatory steps and calculations. That is why this section groups both kinds of graphs.

The functions persp() and contour() may look complicated and the online examples for using these functions seem to add a bit to the confusion and makes the process more difficult. Fortunately, this section zooms in on presenting a logical and easy progression for creating the data for the plots and then creating these plots.

Plotting Three-Dimensional Surfaces

The declaration for the function persp() is:

```
persp(x = seq(0, 1, length.out = nrow(z)),
      y = seq(0, 1, length.out = ncol(z)),
      z, xlim = range(x), ylim = range(y),
      zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL,
      main = NULL, sub = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1,
      scale = TRUE, expand = 1,
      col = "white", border = NULL, ltheta = -135, lphi = 0,
      shade = NA, box = TRUE, axes = TRUE, nticks = 5,
      ticktype = "simple", ...)
```

The parameters **x** and **y** provide the locations of grid lines at which the values in parameter z are measured. The parameter **z** is the matrix containing the values for the surface. The parameters **xlim**, **ylim**, and **zlim** specify the limits for the values of x, y, and z, respectively. The parameters **xlab**, **ylab**, and **zlab** define the labels for the x, y, and z axes, respectively. The parameters theta and phi represent the viewing angles. The angle theta gives the azimuthal direction while angle phi gives the colatitude. The parameter **r** is the distance of the viewer from the center of the plotting box. The parameter **d** is a value that may be utilized to alter the strength of the perspective transformation. Values that are greater than 1 will decrease the perspective effect, and by contrast, values less and 1 will enhance it. The parameter **scale** controls the coordinates transformation. The function transforms the x, y and z coordinates to the interval [0,1] before it displays the points defining the surface.. When parameter scale is TRUE, the function transforms the x, y and z coordinates separately. Otherwise, when scale is FALSE the function scales the coordinates such that it retains the aspect ratios. The parameter **expand** represents an expansion factor which the function applies to the z coordinates. Typical values are fractions between 0 and 1 serve to shrink the plotting box in the z direction. The parameter **col** specifies the color(s) of the surface. The parameter **border** is the color of the line drawn around the surface facets. The angular parameters **ltheta** and **lphi** cause the function to shade the surface as if it was illuminated from the direction specified by azimuth ltheta and colatitude lphi angles. The parameter **shade** represents the shade at a surface facet, calculated using the expression ((1+d)/2)^shade, where d is the dot product of a unit vector normal to the facet and a unit vector in the direction of a light source. Values of shade close to 1 generate shading similar to a point light source model and values close to 0 yield no shading. Values in the range 0.5 to 0.75 offer an approximation to daylight illumination. The Boolean parameter **box** tells the function whether or not to display the bounding box for the surface. The Boolean parameter **axes** tells the function to add ticks and labels to the box. The parameter **ticktype** is a string that can be either "simple" or "detailed". The option "simple" draws just an arrow parallel to the axis to indicate direction of increase. The option "detailed" draws normal ticks as per 2D plots. The parameter **nticks** represents the

approximate number of tick marks to draw on the axes. The setting for this parameter has no effect if the argument for parameter ticktype is "simple".

Let's use the following equation as the surface model to plot (and then to draw the contours for):

$$z = 24 + 6y - 3x - 7y^3 - 6x^3$$

The example will focus on drawing the surface for the values of x and y in the range of -1 to 1. The steps involved in creating the vectors x and y generates arrays of 11 values in the range of -1 to 1 using the following commands:

```
> x = seq(-1, 1, 0.1)
> y = seq(-1, 1, 0.1)
```

Next, we define the function fz(x,y) that implements the equation for the surface model:

```
> fz = function(x,y) { return (24 + 6*y - 3*x - 7*y^3 - 6*x^3) }
```

Now we need to calculate the values of the dependent variable z using a mesh or grid of the values for variables x and y, each in the range of -1 to 1. The R function that comes to the rescue is the function outer(). This function has the declaration of:

outer(x, y, function.for.z)

Using the function outer to create the matrix z, we execute the following command:

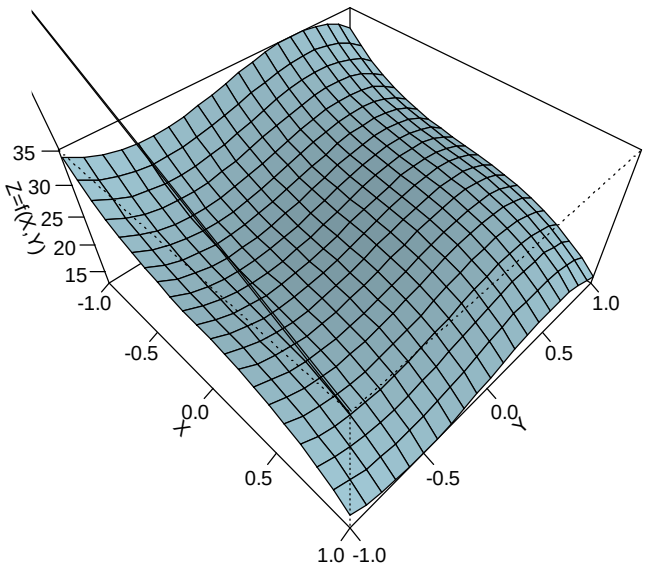
```
> z = outer(x, y, fz)
```

Now we are ready to draw the surface by executing the following command:

```
> persp(x, y, z, theta = 45, phi = 45, expand = 0.5,
          col = "lightblue", ltheta = 120, shade = 0.5,
          ticktype = "detailed", xlab = "X", ylab = "Y", zlab = "Z=f(X,Y)")
```

The above command generates the surface plot in Figure 28.

Figure 28. A surface plot.

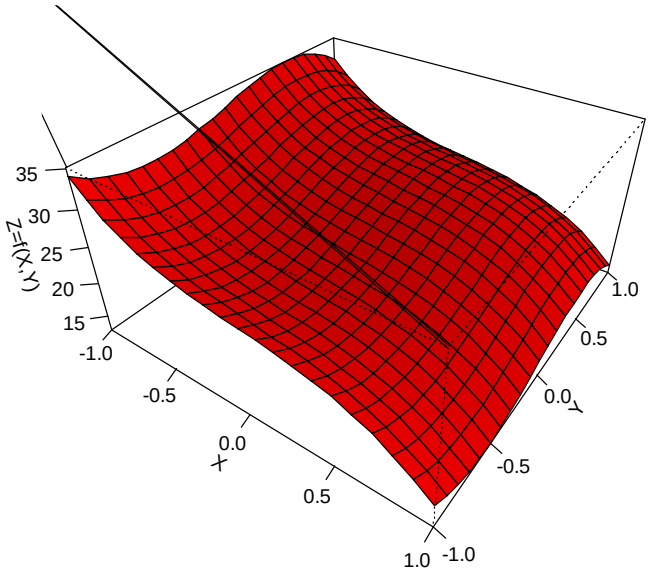


Now call the presp() function again to draw the same surface with a red color and different viewing angles:

```
> persp(x, y, z, theta = 35, phi = 35, expand = 0.5,
          col = "red", ltheta = 120, shade = 0.5,
          ticktype = "detailed", xlab = "X", ylab = "Y", zlab = "Z=f(X,Y)")
```

The above command generates Figure 29.

Figure 29. A red-colored surafce plot.



Drawing Contours

Now, let's turn our attention to drawing contours. The contour() function allows you to draw contours that permit you to peer into certain levels of surface functions. The declaration for the function contour is:

```
contour(x = seq(0, 1, length.out = nrow(z)),
        y = seq(0, 1, length.out = ncol(z)),
        z,
        nlevels = 10, levels = pretty(zlim, nlevels),
        labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont, axes = TRUE, frame.plot = axes,
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

The parameters **x** and **y** provide the locations of grid lines at which the values in parameter z are measured. The parameter **z** is the matrix containing the values for the surface dissected by the contours. The parameter **nlevels** specifies the number of contour levels desired, only when the parameter levels is not supplied. The parameter **levels** represents the numeric vector of levels at which to draw contour lines. The parameter **labels** is a string vector providing the labels for the

contour lines. The default value of NULL causes the function to utilize the numeric values in levels as labels. The parameters **xlim**, **ylim**, and **zlim** specify the ranges for the x, y, and z axes, respectively. The Boolean parameter **drawlabels** specifies whether or not the function labels the contours. The Boolean parameters **axes** and **frame.plot** indicate if the function draws axes and a frame for the contour. The parameter **col** specifies the color for the contour's lines. The parameter **lty** specifies the type of lines draws. The parameter **lwd** indicates the width for the lines drawn. The Boolean parameter **add** tells the function whether or not to add to the current contour plot.

Now let's draw the contours for the equation:

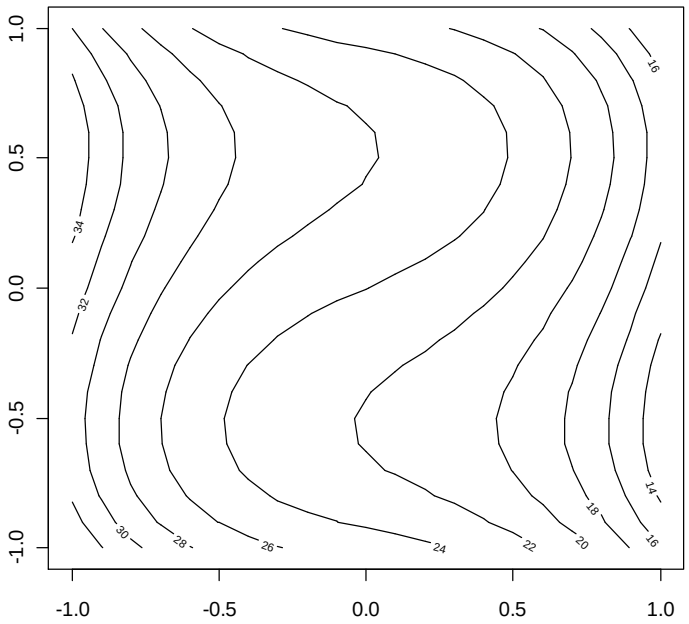
$$z = 24 + 6y - 3x - 7y^3 - 6x^3$$

Armed with the values of vectors, x, y, and z, which we already created for the surface plot, we simply invoke the function contour() as follows:

```
> contour(x, y, z, nlevels=10)
```

The above command creates the contours in Figure 30.

Figure 30. The contour for function z=f(x,y).

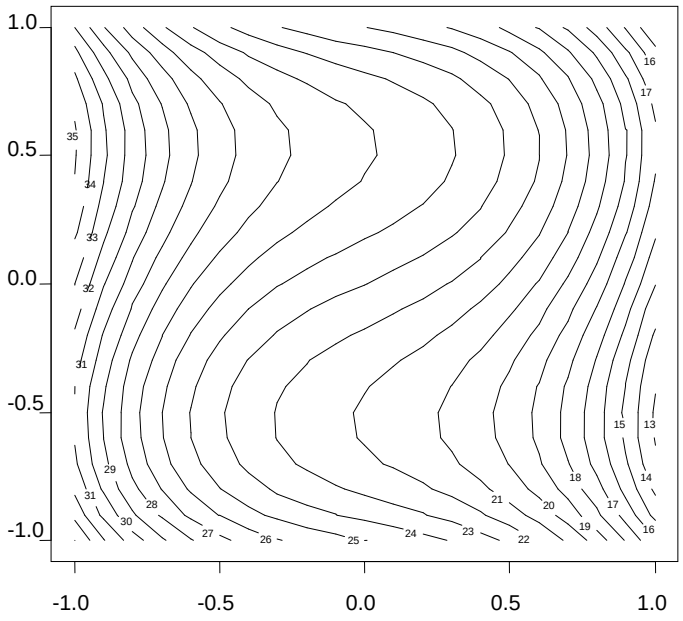


Drawing the same contour with twice as many levels, we write the following command:


```
> contour(x, y, z, nlevels=20)
```

And we obtain Figure 31 that shows more contour lines.

Figure 31. Contour with twice as many levels.



Dumping Graphs to Files

R provides a set of functions that allow you to draw a plot and then send it to a file instead of showing it in a graphing window.

The next table lists the functions that write a graph to a file using a particular file format.

Function	***File Format***
pdf(file)	PDF
png(file)	PNG
jpeg(file)	JPEG
bmp(file)	BMP
tiff(file)	TIFF
postscriptfile)	postscript
win.metafile(file)	Windows metafile
pictex(file)	TeX and LaTeX

The functions listed in the above table have several parameters (in addition to the parameter file). I encourage you to use the online help system and view the full lists of parameters for these functions, since I briefly mention these functions in this tutorial. R requires that you follow the next steps in using the graph dumping functions:

1. Invoke a graph dumping function.
2. Create a graph using the function plot() as well as other graphing functions, as needed.
3. Use the dev.off() function to end the graph dump.

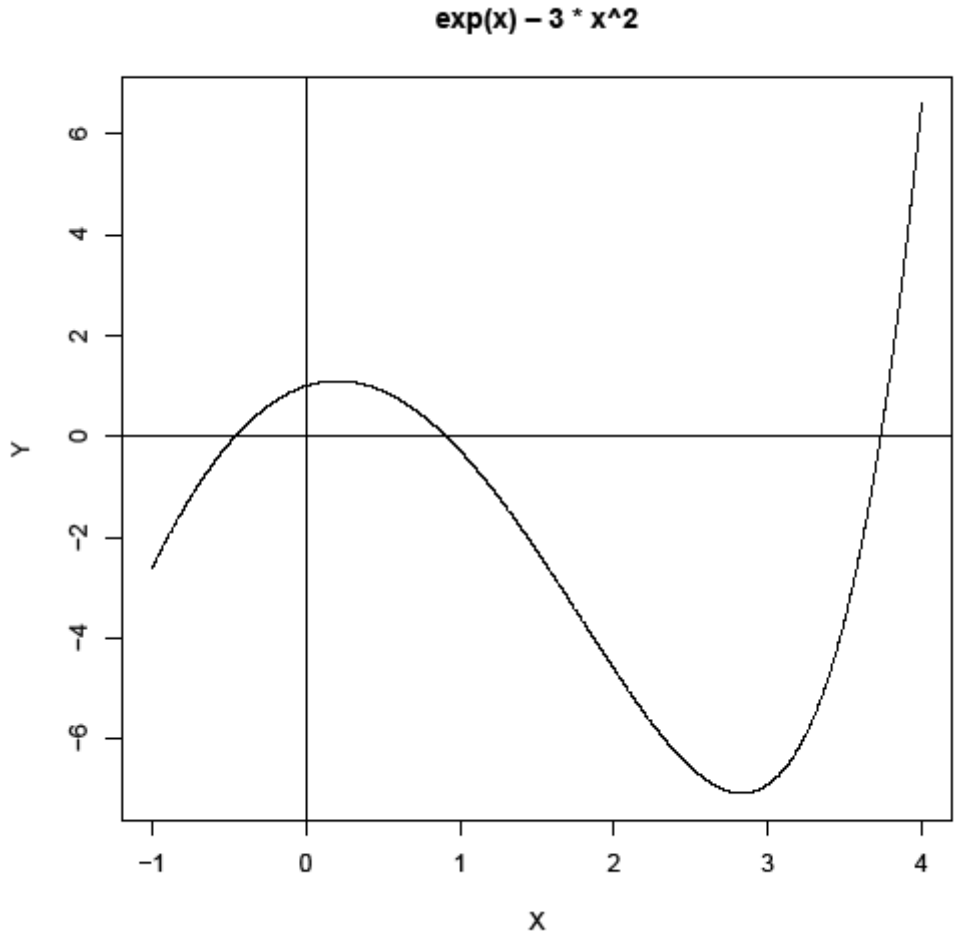
The function pdf() has a parameter that allows you to dump multiple graphs in the same pdf file. This is another reason to learn a bit more about these functions.

Here is an example for dumping a graph:

```
> pdf("C:/graph.pdf")
> x = seq(-1,4,0.001)
> y = exp(x) - 3 * x^2
> plot(x, y, type="l", xlab="X", ylab="Y", main="exp(x) - 3 * x^2")
> abline(v=0)
> abline(h=0)
> dev.off()
```

If you open the file C:\graph.pdf you get the graph in Figure 32.

Figure 32. A graph dumped to a file.



The ability to dump multiple graphs in a single pdf file comes in handy in drawing a series of three-dimensional surfaces with multiple values for the viewing angles theta and phi. Here is the code for the function that performs such a task:

```
multi.3d.plot = function(file, x, y, fz, expand=0.5, col="lightblue",
                         theta.vect = seq(0,90,10), phi.vect= seq(0,90,10),
                         ltheta=120, shade=0.5, ticktype = "detailed",
                         xlab = "X", ylab = "Y", zlab = "Z=f(X,Y)")
{
  z = outer(x, y, fz)
  pdf(file)
  for (theta in theta.vect) {
    for (phi in phi.vect) {
      s = paste(zlab, "with theta =", as.character(theta),
                      "and phi=", as.character(phi))
      persp(x, y, z, theta = theta, phi = phi, expand = expand,
            col = col, ltheta = ltheta, shade = shade,
            ticktype = ticktype , xlab = xlab, ylab = ylab, zlab = zlab)
```



```
      title(s)
    }
  }
  dev.off()
}
```

The parameter **file** specifies the name of the pdf file that will contain the multiple graphs. The parameters **x** and **y** specify the vectors for the x and y values, respectively. The parameter **fz** is the function used to calculate the z values using values of x and y. The parameter **theta.vect** specifies the vector of angle theta values. The default vector is a sequence that varies from 0 to 90 in increments of 10. The parameter **phi.vect** specifies the vector of angle phi values. The default vector is a sequence that varies from 0 to 90 in increments of 10.

The remaining parameters are the same ones you already saw in the function persp(). The function uses two for loops to iterate over the values for angles theta and phi. Each loop iterates by default between 0 and 90 degrees, in increments of 10 degrees. The function generates, by default, 100 graphs in the targeted pdf file! Each graph has a title that includes the values for angles theta and phi. When you open the pdf file and scroll through the figures, you get a slow motion animation of the surface moving as the angles of views change. You can specify arguments for parameters theta.vect and/or phi.vect to alter the angle range and angle increments. Reducing the angle range and angle increment reduces the number of graphs, and vice versa.

You can save the code for the function multi.3d.plot() and then load it by using the source() function. You can instead, select the lines of code for the function and then press the CTRL+R keys to make the R workspace learn the function.

To test the function multi.3d.plot(), execute the following commands to plot several 3D graphs for the function $z = 24 + 6y – 3x –7y^3 – 6x^3$:

```
> x = seq(-1, 1, 0.1)
> y = seq(-1, 1, 0.1)
> fz = function(x,y) { return (24 + 6*y - 3 * x - 7*y^3 - 6 * x^3) }
> multi.3d.plot("C:/graphs.pdf", x, y, fz)
```

Open the file C:\graphs.pdf and scroll through the set of 100 graphs that the function generates!

