INFO I-415
Lab 1

In this lab, we will introduce some simple R commands. The best way to learn a new language is to try out the commands. R can be downloaded from

http://cran.r-project.org/

We recommend that you run R within an integrated development environment (IDE) such as RStudio, which can be freely downloaded from

http://rstudio.com

The RStudio website also provides a cloud-based version of R, which does not require installing any software.

Basic Commands

R uses *functions* to perform operations. To run a function called funcname, we type funcname(input1, input2), where the inputs (or *arguments*) input1 and input2 tell R how to run the function. A function can have any number of inputs. For example, to create a vector of numbers, we use the function c() (for *concatenate*). Any numbers inside the parentheses are joined together. The following command instructs R to join together the numbers 1, 3, 2, and 5, and to save them as a *vector* named x. When we type x, it gives us back the vector.

```
> x <- c(1, 3, 2, 5)
> x
[1] 1 3 2 5
```

Note that the > is not part of the command; rather, it is printed by R to indicate that it is ready for another command to be entered. We can also save things using = rather than <-:

```
> x = c(1, 6, 2)
> x
[1] 1 6 2
> y = c(1, 4, 3)
```

Hitting the *up* arrow multiple times will display the previous commands, which can then be edited. This is useful since one often wishes to repeat a similar command. In addition, typing ?funcname will always cause R to open a new help file window with additional information about the function funcname().

We can tell R to add two sets of numbers together. It will then add the first number from x to the first number from y, and so on. However, x and y should be the same length. We can check their length using the length() function.

```
> length(x)
[1] 3
> length(y)
[1] 3
> x + y
[1]  2 10  5
```

The ls() function allows us to look at a list of all of the objects, such as data and functions, that we have saved so far. The rm() function can be used to delete any that we don't want.

```
> ls()
[1] "x" "y"
> rm(x, y)
```

```
> ls()
character(0)
```

It's also possible to remove all objects at once:

```
> rm(list = ls())
```

The `matrix()` function can be used to create a matrix of numbers. Before we use the `matrix()` function, we can learn more about it:

```
> ?matrix
```

The help file reveals that the `matrix()` function takes a number of inputs, but for now we focus on the first three: the data (the entries in the matrix), the number of rows, and the number of columns. First, we create a simple matrix.

```
> x <- matrix(data = c(1, 2, 3, 4), nrow = 2, ncol = 2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Note that we could just as well omit typing `data=`, `nrow=`, and `ncol=` in the `matrix()` command above: that is, we could just type

```
> x <- matrix(c(1, 2, 3, 4), 2, 2)
```

and this would have the same effect. However, it can sometimes be useful to specify the names of the arguments passed in, since otherwise `R` will assume that the function arguments are passed into the function in the same order that is given in the function's help file. As this example illustrates, by default `R` creates matrices by successively filling in columns. Alternatively, the `byrow = TRUE` option can be used to populate the matrix in order of the rows.

```
> matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

Notice that in the above command we did not assign the matrix to a value such as `x`. In this case the matrix is printed to the screen but is not saved for future calculations. The `sqrt()` function returns the square root of each element of a vector or matrix. The command `x^2` raises each element of `x` to the power `2`; any powers are possible, including fractional or negative powers.

```
> sqrt(x)
     [,1] [,2]
[1,] 1.00 1.73
[2,] 1.41 2.00
> x^2
     [,1] [,2]
[1,]    1    9
[2,]    4   16
```

The `rnorm()` function generates a vector of random normal variables, with first argument `n` the sample size. Each time we call this function, we will get a different answer. Here we create two correlated sets of numbers, `x` and `y`, and use the `cor()` function to compute the correlation between them.

```
> x <- rnorm(50)
> y <- x + rnorm(50, mean = 50, sd = .1)
> cor(x, y)
[1] 0.995
```

By default, `rnorm()` creates standard normal random variables with a mean of 0 and a standard deviation of 1. However, the mean and standard deviation can be altered using the `mean` and `sd` arguments, as illustrated above. Sometimes we want our code to reproduce the exact same set of random numbers; we can use the `set.seed()` function to do this. The `set.seed()` function takes an (arbitrary) integer argument.

```
> set.seed(1303)
> rnorm(50)
[1] -1.1440  1.3421  2.1854  0.5364  0.0632  0.5022 -0.0004
. . .
```

We use `set.seed()` throughout the labs whenever we perform calculations involving random quantities. In general this should allow the user to reproduce our results. However, as new versions of `R` become available, small discrepancies may arise between this book and the output from `R`.

The `mean()` and `var()` functions can be used to compute the mean and variance of a vector of numbers. Applying `sqrt()` to the output of `var()` will give the standard deviation. Or we can simply use the `sd()` function.

```
> set.seed(3)
> y <- rnorm(100)
> mean(y)
[1] 0.0110
> var(y)
[1] 0.7329
> sqrt(var(y))
[1] 0.8561
> sd(y)
[1] 0.8561
```

Graphics

The `plot()` function is the primary way to plot data in `R`. For instance, `plot(x, y)` produces a scatterplot of the numbers in `x` versus the numbers in `y`. There are many additional options that can be passed in to the `plot()` function. For example, passing in the argument `xlab` will result in a label on the x-axis. To find out more information about the `plot()` function, type `?plot`.

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, y)
> plot(x, y, xlab = "this is the x-axis",
    ylab = "this is the y-axis",
    main = "Plot of X vs Y")
```

We will often want to save the output of an `R` plot. The command that we use to do this will depend on the file type that we would like to create. For instance, to create a pdf, we use the `pdf()` function, and to create a jpeg, we use the `jpeg()` function.

```
> pdf("Figure.pdf")
> plot(x, y, col = "green")
> dev.off()
null device
          1
```

The function `dev.off()` indicates to `R` that we are done creating the plot. Alternatively, we can simply copy the plot window and paste it into an appropriate file type, such as a Word document.

The function `seq()` can be used to create a sequence of numbers. For instance, `seq(a, b)` makes a vector of integers between `a` and `b`. There are many other options: for instance, `seq(0, 1, length = 10)` makes a sequence of `10` numbers that are equally spaced between `0` and `1`. Typing `3:11` is a shorthand for `seq(3, 11)` for integer arguments.

```
> x <- seq(1, 10)
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> x <- seq(-pi, pi, length = 50)
```

We will now create some more sophisticated plots. The `contour()` function produces a *contour plot* in order to represent three-dimensional data; it is like a topographical map. It takes three arguments:

1. A vector of the `x` values (the first dimension),
2. A vector of the `y` values (the second dimension), and
3. A matrix whose elements correspond to the `z` value (the third dimension) for each pair of (`x`, `y`) coordinates.

As with the `plot()` function, there are many other inputs that can be used to fine-tune the output of the `contour()` function. To learn more about these, take a look at the help file by typing `?contour`.

```
> y <- x
> f <- outer(x, y, function(x, y) cos(y) / (1 + x^2))
> contour(x, y, f)
> contour(x, y, f, nlevels = 45, add = T)
```

```
> fa <- (f - t(f)) / 2
> contour(x, y, fa, nlevels = 15)
```

The `image()` function works the same way as `contour()`, except that it produces a color-coded plot whose colors depend on the `z` value. This is known as a *heatmap*, and is sometimes used to plot temperature in weather forecasts. Alternatively, `persp()` can be used to produce a three-dimensional plot. The arguments `theta` and `phi` control the angles at which the plot is viewed.

```
> image(x, y, fa)
> persp(x, y, fa)
> persp(x, y, fa, theta = 30)
> persp(x, y, fa, theta = 30, phi = 20)
> persp(x, y, fa, theta = 30, phi = 70)
> persp(x, y, fa, theta = 30, phi = 40)
```