Getting Started with R

At its most basic, R is a free, open-source statistical computing and graphics environment. Anyone can download it via the link http://cran.r-project.org/. Click on whichever operating platform listed – Linux, MacOS X, or Windows – is appropriate for the computer you will be working on, and follow the stated instructions. (Note to Windows users: You must click base to access the download page.) Once installed, click the generated icon to run the program, which should automatically display an R console on your monitor, similar to the one shown here.

As a first example, let us type an easy mathematical expression after the prompt (>) at the start of the command line, such as that shown below, followed by Enter.

```
> 2 + 3
[1] 5
```

The [1] in front simply indicates that there is only one row of values displayed in the output; more on this later. Another way of performing this operation might be:

```
> x = 2 + 3
> x
[1] 5
```

The first line assigns the value of 2 + 3 to the variable x, the second line calls the result of this operation. (Note: The first line could also be written as x <- 2 + 3.) Another approach to the same problem:

```
> y = c(2, 3)
> y
[1] 2 3
```
The first command combines the values 2 and 3 into an ordered pair, as shown. The last command utilizes one of the built-in mathematical functions, of which there are many. To obtain detailed information about its syntax, format, etc., we may simply type `help(sum)` or `?sum`, which will generate a separate help window on this topic, and similarly for any object in R. (More details on this below…)

As a second example, consider

```r
> v = c(5: 40)
> v
> length(v)
[1] 36
```

Hence, the “vector” \( v \) consists of the ordered sequence of 36 integers from 5 to 40, which is output in R in multiple rows. That is, component-wise, we have \( v[1] = 5, v[2] = 6, v[3] = 7, \ldots v[36] = 40 \). The [1] and [26] at the front of each row indicate that \( v[1] = 5 \) and \( v[26] = 30 \), and so serve as pointers. Thus, for example, we have…

```r
> v[10]
[1] 14
```

… however,

```r
> v[-10]
```

gives back the original vector with the tenth component (i.e., 14) deleted, shown below…

```r
```

… while \( v[50] \), i.e., the 50th component, returns \textbf{NA} (“Not Available”).

The treatment of all objects in R as vectors makes it extremely flexible when performing complex calculations. For example, the input

```r
> z = c(3:10)
> z + 5
```

produces the vector

```
[1]  8  9 10 11 12 13 14 15
```
as output. That is, 5 is added to each component of the vector \( z = c(3, 4, 5, 6, 7, 8, 9, 10) \)

Similarly, we may easily multiply or divide a vector by a constant value:

\[
> 2 * z
\]

\[
[1] 6 8 10 12 14 16 18 20
\]

We may also add, subtract, multiply or divide two vectors, component by component, provided they are the same size:

\[
> w = c(6.9, 2.7, 0, -11.3, 5.5, -7.8, 4.1, 3.2)
\]

\[
> w + z
\]

\[
[1] 9.9 6.7 5.0 -5.3 12.5 0.2 13.1 13.2
\]

\[
> w * z
\]

\[
[1] 20.7 10.8 0.0 -67.8 38.5 -62.4 36.9 32.0
\]

\[
> w / z
\]

\[
[1] 2.3000000 0.6750000 0.0000000 -1.8833333 0.7857143 -0.9750000 0.4555556
[8] 0.3200000
\]

However, an attempt to divide by 0 produces an “infinity” component, as seen below.

\[
> z / w
\]

\[
[1] 0.4347826 1.4814815 Inf -0.5309735 1.2727273 -1.0256410 2.1951220 3.1250000
\]

Note that a special case of multiplying together two vectors in \( \mathbb{R} \) is component-wise squaring:

\[
> w^2
\]

\[
[1] 47.61 7.29 0.00 127.69 30.25 60.84 16.81 10.24
\]

Similarly, other powers are possible. For example, either \( z^{(1/2)} \) or \( \text{sqrt}(z) \) yields:

\[
\]

Vectors can also be concatenated in various ways to form new vectors…

\[
> c(z, w)
\]

\[
[1] 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 6.9 2.7 0.0 -11.3 5.5 -7.8 4.1 3.2
\]

\[
> c(w, z)
\]

\[
[1] 6.9 2.7 0.0 -11.3 5.5 -7.8 4.1 3.2 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
\]

… as well as combined via rows or columns to form matrices:
> two.by.eight = rbind(w, z)
> two.by.eight

w  6.9  2.7   0  -11.3  5.5  -7.8  4.1  3.2
z  3.0  4.0   5   6.0  7.0  8.0  9.0 10.0

Thus, for example, an individual matrix entry can be specified via a double index:

> two.by.eight[1, 6]

  w
-7.8

> eight.by.two = cbind(w, z)
> eight.by.two

    w  z
[1,]  6.9  3
[2,]  2.7  4
[3,]  0.0  5
[4,] -11.3  6
[5,]  5.5  7
[6,]  -7.8  8
[7,]   4.1  9
[8,]   3.2 10

This can also be accomplished via the command matrix. But this command is a bit more sophisticated (and much more flexible); in order to use it correctly, type either help(matrix) or ?matrix to generate a help window. Alas, the documentation is often cryptically written in “computer-ese,” and might appear difficult to translate, until you get the hang of it. To illustrate: Under Description, the matrix command is formally defined (as well as a couple of related commands, which we can ignore). Under Usage, we see the rather cryptic-looking statement

matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)

which gives us the proper syntax of the command. Note that the options inside the parentheses have names that are suggestive of their role. Skipping down to the Arguments section, we see that the first one in the parentheses, data, is a vector of data values. NA means “Not Available”
in R, and just refers to the fact that this must be provided by the user. In our example, this is the vector \( c(w, z) \) above. Next is \( \text{nrow} \), the number of rows, and \( \text{ncol} \), the number of columns, respectively. If we stop here, and faithfully type the command syntax exactly as it appears...

\[
> \text{matrix}(\text{data} = c(w, z), \text{nrow} = 8, \text{ncol} = 2)
\]

... then we obtain the following output:

\[
\begin{array}{ccc}
[,1] & [,2] \\
[1,] & 6.9 & 3 \\
[2,] & 2.7 & 4 \\
[3,] & 0.0 & 5 \\
[4,] & -11.3 & 6 \\
[5,] & 5.5 & 7 \\
[6,] & -7.8 & 8 \\
[7,] & 4.1 & 9 \\
[8,] & 3.2 & 10
\end{array}
\]

**Note:** This command may be shortened to \( \text{matrix}(c(w, z), \text{nr} = 8, \text{nc} = 2) \), or the even simpler \( \text{matrix}(c(w, z), 8, 2) \), provided that the order is preserved.

Moving on, the “default” option for how the data vector is arranged is given by \( \text{byrow} = \text{FALSE} \), i.e., our vector is bound by columns. Indeed, the redundant commands

- \( \text{matrix}(c(w, z), 8, 2, \text{byrow} = \text{FALSE}) \)
- \( \text{matrix}(c(w, z), 8, 2, \text{byrow} = \text{F}) \)
- \( \text{matrix}(c(w, z), 8, 2, \text{F}) \)

will all yield exactly the same results as the highlighted command above. (Change \( \text{F} \) to \( \text{T} \) in the last command and see what happens. What changes in the output?) Finally, the \( \text{dimnames} \) option allows us to give labels to our rows and columns. Type \( \text{matrix}(c(w, z), 8, 2, \text{F}, \text{list}(c(1:8), c("w", "z"))) \) and verify this. (If \( \text{byrow} = \text{F} \) is the default, then why is it necessary to include the \( \text{F} \) in the previous line? Retype the command without it and see.)
A square matrix can be generated via the command:

```r
> M = matrix(c(w, z), 4, 4)
> M
[1,]  6.9  5.5  3  7
[2,]  2.7 -7.8  4  8
[3,]  0.0  4.1  5  9
[4,] -11.3  3.2  6 10
```

From this, we may select out specific submatrices, if desired:

```r
> M[2:4, 1:2]
[,1] [,2]
[1,]  2.7 -7.8
[2,]  0.0  4.1
[3,] -11.3  3.2
```

Some elementary practice exercises… Use R commands to generate each of the following.

1. \( \sqrt{-1} \) (Note: NaN stands for “Not a Number”)
2. The vector consisting of the decreasing sequence of consecutive integers from 57 to –11.  
   **Hint**: See `seq`
3. The vector consisting of the increasing sequence of odd integers from –11 to 57.  
   **Hint**: See `seq`
4. The vector consisting of five hundred alternating zeros and ones.  
   **Hint**: See `rep`
5. The average of 12.7, 9.4, 6.6, 10.8, 5.3, and 7.2.  
   **Hint**: See `mean`
6. Sort the six preceding numbers in decreasing order.  
   **Hint**: See `sort`
7. A vector of fifty, uniformly distributed, random values between –1 and +1.  
   **Hint**: See `runif`
8. The vector consisting of positive values from the preceding vector.  
   **Hint**: See `subset`
9. The positions of the positive values in the original vector.  
   **Hint**: See `which`
10. The vector of the first 1000 positive integers, without the perfect squares.

One of the nicest features of R is its graphics capabilities. Consider the following simple example.

```r
> my.parabola = function(x) (x^2)
> my.parabola(3.4)
[1] 11.56
```

First we defined a function \( f(x) = x^2 \), then asked R to evaluate \( f(3.4) = (3.4)^2 \). Now type

```r
> plot(my.parabola)
```
to get a generic-looking “no frills” graph in the default interval [0, 1]. However, type `help(par)` or `?par` to open a help window, and under **Graphical Parameters**, we see that all sorts of optional “bells and whistles” are available to fancy it up, such as `cex` and `pch`. For example, graph and interpret each of the options for the command:

```r
> plot(my.parabola, xlim = range(-3, 3), ylim = range(0, 10), col = "blue", lty = 5, lwd = 2, xlab = "X-axis", ylab = "Y-axis", main = "GRAPH OF PARABOLA Y = X^2", col.main = "red")
```

(Most of these are easy enough to figure out by inspection.) Now type

```r
> x1.coords = c(-2, 1)
> y1.coords = c(5, 8)
> lines(x1.coords, y1.coords, col ="green")
```
to add a straight line connecting the points (–2, 5) to (1, 8) – *note the syntax! – and

```r
> x2.coords = -1:3
> y2.coords = abs(cos(x2.coords))
> points(x2.coords, y2.coords, pch = 19, col ="purple")
```
to add the five points (–1, |cos(–1)|), (0, |cos(0)|), (1, |cos(1)|), (2, |cos(2)|), and (3, |cos(3)|) to the graph.

One of the most important objects in the study of probability and statistics is the so-called “standard normal distribution,” more popularly known as the “bell curve,” and is given by the function $\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}$; its values are calculated by the R command `dnorm`.

**Exercise:** Graph this function between –3 and +3. Note that it is symmetric around the center $z = 0$.

The R command `pnorm` calculates the related function $\Phi(z)$, the cumulative area under this curve to the left of any value of $z$. For example, the area under the curve up to $z = +2.3$ is calculated via

```r
> pnorm(2.3)

[1] 0.9892759
```
Likewise, the area under the curve up to $z = -1.4$ is calculated via

```r
> pnorm(-1.4)

[1] 0.08075666
```
and so on…

**Exercise:** What is the value of the area between –1.4 and +2.3?
An important mathematical property of this function is that the total area under the curve is equal to 1.

**Exercise:** Calculate `pnorm(5)`. Calculate `pnorm(0)`. Why are these values not surprising?

**Exercise:** What is the value of the area to the right of $-1.4$? to the right of $+2.3$?

Finally, the R command `qnorm` calculates the inverse function of $\Phi(z)$, i.e., it outputs the value of $z$ that corresponds to any input left-cumulative area from 0 to 1. (In other words, it’s the “backward” operation of `pnorm`.) Referring back to the preceding examples, we thus have the following:

```
> qnorm(0.9892759)
[1] 2.300000
```

i.e., the value of $z$ that corresponds to a left-cumulative area of 0.9892759 is $z = 2.3$. Likewise,

```
> qnorm(0.08075666)
[1] -1.4
```

and so on…

**Exercise:** Calculate `qnorm(0)`. Calculate `qnorm(0.5)`. Calculate `qnorm(1)`.

These basic examples using R only scratch its surface. One of its greatest strengths of course lays in its statistical data analysis capabilities. Among the many built-in R packages of great use are `mean`, `var`, `sd`, `dbinom`, `pbinom`, `qbinom`, `pnorm`, `qnorm`, `pt`, `qt`, `pchisq`, `qchisq`, `pf`, `qf`, `t.test`, `chisq.test`, `lsfit`, and others. To learn more, numerous basic and advanced reference materials can be found online and in books. Two particularly nice beginner’s guides are the book *Introductory Statistics with R*, by Peter Dalgaard (Springer, 2008), and a primer written by one of my colleagues here in the Dept. of Statistics, Prof. Bret Larget: [http://www.stat.wisc.edu/~larget/r.html](http://www.stat.wisc.edu/~larget/r.html). Happy computing!

---

**~ An Important Note on the Use of R in Homework Assignments ~**

Some of the problems that will be assigned are necessarily computationally intensive, and you will really not want to work them out by hand. For your convenience, the folder [http://www.stat.wisc.edu/~ifischer/Intro_Stat/Lecture_Notes/Rcode/](http://www.stat.wisc.edu/~ifischer/Intro_Stat/Lecture_Notes/Rcode/) contains computer code for selected problems, which can be copied and pasted directly into the R console. These HW problems will have a direct R link to this folder. (Other R problems without the link are straightforward enough to figure out.) However, it is NOT acceptable simply to run the code, and turn in the output without reading it. Part of such homework problems will ask specific questions about the output, and you must know where to find the answers, and indicate them.

**Note:** You do not have to turn in the problems in this document, but please do try them!
Optional: Basic Programming in R

Although you will not have to write your own computer code for this course, you may eventually encounter situations when you do (... plus, it may make the posted code for the homework problems easier to understand). This section provides a very elementary guide to some of the fundamental programming steps in \textit{R}, assuming no previous programming experience.

Example: For any positive whole number \( n \), the symbol \( n! \) is read “\( n \) factorial,” and denotes the value of the product \( 1 \times 2 \times 3 \times \ldots \times n \). (Each term in the product is called a “factor.”) For instance, the expression \( 5! \) (read “5 factorial”) = \( 1 \times 2 \times 3 \times 4 \times 5 = 120 \). In \textit{R}, this could easily be computed via the built-in function \texttt{factorial(5)} or \texttt{prod(1:5)} – try them – but if we did not know that, we could still write a little program that accomplishes the same thing. Consider the following sequence of statements, which you can just copy and paste “as is”: (Incidentally, any statements that are preceded by the “pound sign” (\#) are automatically interpreted as comments, and ignored by \textit{R}).

\begin{verbatim}
 n = 5   # We could enter any integer value of n \( \geq 0 \) here.
 product = 1   # This tells R that the first factor is \( 1 \).
 if (n > 1) {
   for (k in 2:n) {product = product * k}       # recursive loop
 }
\end{verbatim}

The last statement, called a \textit{recursive loop}, may look weird, but is key. The “\( k \)” in the parentheses is an index that runs through all of the factors from 2 to \( n \), one at a time. Every time it does, the operation in curly braces is performed, where the “new” product is calculated via multiplying the previous “old” product by the new factor \( k \). Initially, the value of \texttt{product} is here set to 1. The next line is a \textit{conditional statement}. If \( n \) is greater than 1, then \( k = 2 \) in the first pass through the loop, and \texttt{product} = \( 1 \times 2 \). In the second run through the loop, the factor increases to \( k = 3 \), and so \texttt{product} = \( 1 \times 2 \times 3 \). In the next run, \( k = 4 \), and \texttt{product} = \( 1 \times 2 \times 3 \times 4 \), and finally, \texttt{product} = \( 1 \times 2 \times 3 \times 4 \times 5 \) when \( k = 5 \), the last factor in this example. All one needs to do now to view the answer is type \texttt{product} at the prompt, hit \texttt{Enter}, and \textit{R} should output the correct value \texttt{120}.

Now you try… (Solutions are posted at this link. Note that alternate solutions are possible.)

\begin{flushleft}
\textbf{Exercise 1:} Write a program to arrange a group of twenty people into a list of ten random pairs. Use the following general “pseudo-code” as a guide.
\end{flushleft}

\begin{flushleft}
\textbf{Step 1.} Label the \textbf{group} of integers \{1, 2, 3, 4, \ldots, 20\}, and set \texttt{mylist} = \texttt{NULL}.
\end{flushleft}

\begin{flushleft}
\textbf{Step 2.} Run through the following “for” loop ten times:
\begin{itemize}
  \item Randomly select one \textbf{pair} of integers from the \textbf{group}, without replacement. \hspace{1cm} (\textit{Hint:} See \texttt{sample})
  \item Adjoin this \textbf{pair} of integers to \texttt{mylist} as a new row. \hspace{1cm} (\textit{Hint:} See \texttt{rbind})
  \item Remove this \textbf{pair} of integers from the remaining integers in the \textbf{group}. \hspace{1cm} (\textit{Hint:} See \texttt{which})
\end{itemize}
\end{flushleft}

\begin{flushleft}
\textbf{Step 3.} Output \texttt{mylist}.
\end{flushleft}
**Exercise 2:** Redo the previous exercise a more efficient way, i.e., without using a for loop. 
(*Hint:* Using sample and matrix, this can be done in just two lines.)

**Exercise 3:** Write an R program to first create a general random sample of ten uniformly distributed values between –50 and +50 (see runif), and sort them from lowest to highest. Imagine that these values represent the endpoints of nine adjacent intervals on the real line. Compute the midpoints of these nine intervals. Output the ten sorted values, and the midpoints.

**Exercise 4:** Redo the previous exercise a more efficient way, i.e., without using a for loop. 
(*Hint:* This may be a bit more challenging than it looks; you have to think outside the box.)

**Exercise 5:** The Sieve of Eratosthenes is an ancient algorithm for generating prime numbers.*

Start with the integers \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, \ldots\}. Keep the first integer (2), and delete every second integer following it, i.e., 4, 6, 8, 10, 12, 14, 16, 18, 20, etc.: \{2, 3, 5, 7, 9, 11, 13, 15, \ldots\}. Next proceed to the next undeleted integer (3), and delete every third integer following it, i.e., 6 (already deleted), 9, 12 (already deleted), 15, etc.: \{2, 3, 5, 7, 9, 11, 13, 15, \ldots\}. Then proceed to the next undeleted integer (5), and delete every fifth integer following it, i.e., 10 (already deleted), 15 (already deleted), 20 (already deleted), 25, etc.), and so on… The integers that remain – \{2, 3, 5, 7, 11, 13, \ldots\} – are the primes.

Write a short R program that duplicates this procedure, to generate all the prime numbers up to (or including) any given integer \(n\). Use the following general “pseudo-code” as a guide. (Again, there may be multiple ways of doing this, using other commands.)

**Step 1.** For any admissible integer value “\(n = \) ” input by the user, let integers denote the initial sequence \{2, 3, 4, \ldots, n\}.

**Step 2.** For any fixed \(k\) in integers, calculate which of them are to be deleted. 
(*Hint:* These would correspond to multiples of \(k\), i.e., a sequence up to \(n\) in steps of \(k\), excluding \(k\) itself.)

**Step 3.** Set these integers to NA (“Not Available”).

**Step 4.** When completed, locate where the remaining integers are; these are the primes. 
(*Hint:* See is.na and !is.na)

---

* Recall that a prime number is an integer greater than 1, that is only divisible by 1 and itself. The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots For example, 2 is prime because it is only divisible by 1 and itself. But it is the only even prime number; all other even numbers are divisible by 2, by definition. Hence all other prime numbers must be odd, but not all odd numbers are prime. For instance, 9 is not prime because it is divisible by 3; likewise, 15 is not prime because it is divisible by both 3 and 5, etc. There is no “simple” mathematical formula for generating prime numbers.
**Exercise 6:** At the Arcode Hotel, there are one hundred rooms numbered 1 to 100, all currently occupied. Initially, all the rooms have their doors closed. The occupant in room #1 exits, and opens the closed door to every room in the hotel (starting with his/her own). The occupant in room #2 exits, and closes the now open door to every second room in the hotel (starting with his/her own), leaving the others alone. The occupant in room #3 exits, and now finding some doors open and some closed, visits every third room in the hotel (starting with his/her own), and closes the door if it is open, opens it if it closed, leaving the others alone. And so on for each \( k = 1, 2, \ldots, 100 \): the occupant in room \( #k \) exits, visits every \( k \)th room in the hotel (starting with his/her own), and closes the door if it is open, opens it if it closed, leaving the others alone. 

**Question:** At the end of this process, which doors are left open? The answer may surprise you. **Extra kudos if you can figure out the reason!**

Use the following general “pseudo-code” as a guide.

**Step 1.** Designate the initial sequence of one-hundred doors as being “closed.” (*Hint:* Use the “binary” coding “0 = open” and “1 = closed,” and see rep.)

**Step 2a.** In general, for \( k = 1, 2, \ldots, 100 \), occupant \( k \) visits rooms numbered with multiples of \( k \), that is, a sequence in steps of \( k \), starting with his/her own room, and...

**Step 2b.** The doors are changed from “open” to “closed,” and from “closed” to “open.”

**Step 3.** Output which of the doors remain “open.”

---

**Exercise 7:** “Monte Carlo” simulation method to estimate the value of \( \pi \).

Recall from basic geometry that the area of a circle is \( \pi r^2 \); hence if the radius \( r = 1 \), then the area is the constant \( \pi \) itself. Imagine that, in order to estimate its value, we randomly throw a large number of darts in a uniform distribution at a \( 1 \times 1 \) square target, and calculate the proportion that lands inside the circle. In principle, that proportion should be very close to \( \pi /4 \), the area of the quarter-circle. Using runif, write a short R program that generates \( n = 10,000 \) random values for \( x \) and \( y \), each between 0 and 1, then calculates four times the proportion of points \( (x, y) \) that satisfy the inequality \( x^2 + y^2 < 1 \). Also, plot a graph of your simulation, similar to the one shown.*

As a final step, find the mean of \( N = 500 \) such simulations (be patient; it may take a little time), and compare this with the true value of \( \pi = 3.14159... \)

* As previously mentioned, the plot command has many options, such as pch, which specifies the “point character” drawn. Open circles are the default; specifying pch = 19 gives filled circles, as shown. The “character expansion” option cex changes the size of the points; specifying cex = .1 shrinks the points to one-tenth of their default size, and is a good choice to use in this problem. See help(par) for more options and details on parameters used in plotting.