UNIT 1

INTRODUCTION TO R

How to Run R :

R operates in two modes: *interactive* and *batch*. The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on. On the other hand, batch mode does not require interaction with the user. It's useful for production jobs, such as when a program must be run periodically, say once per day, because you can automate the process.

Interactive Mode

On a Linux or Mac system, start an R session by typing R on the command line in a terminal window. On a Windows machine, start R by clicking the R icon.The result is a greeting and the R prompt, which is the > sign. The screen will look something like this:

```
R version 2.10.0 (2009-10-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
```

You can then execute R commands. The window in which all this appears is called the R *console*. As a quick example, consider a standard normal distribution—that is, with mean 0 and variance 1. If a random variable X has that distribution,
then its values are centered around 0, some negative, some positive, averaging in the end to 0. Now form a new random variable $Y = |X|$. Since we've taken the absolute value, the values of Y will *not* be centered around 0, and the mean of Y will be positive. Let's find the mean of Y. Our approach is based on a simulated example of $N(0,1)$ variates.

```
> mean(abs(rnorm(100)))
[1] 0.7194236
```

This code generates the 100 random variates, finds their absolute values,and then finds the mean of the absolute values.
The [1] you see means that the first item in this line of output is item 1.In this case, our output consists of only one line (and one item), so this is redundant. This notation becomes helpful when you need to read voluminous output that consists of a lot of items spread over many lines. For example, if there were two rows of output with six items per row, the second row would be labeled [7].

```
> rnorm(10)
[1] -0.6427784 -1.0416696 -1.4020476 -0.6718250 -0.9590894 -0.8684650
[7] -0.5974668 0.6877001 1.3577618 -2.2794378
```

Here, there are 10 values in the output, and the label [7] in the second row lets you quickly see that 0.6877001, for instance, is the eighth output item.You can also store R commands in a file. By convention, R code files have the suffix *.R* or *.r*. If you create a code file called *z.R*, you can execute the contents of that file by issuing the following command:

```
> source("z.R")
```

Batch Mode

Sometimes it's convenient to automate R sessions. For example, you may wish to run an R script that generates a graph without needing to bother with manually launching R and executing the script yourself. Here you would run R in batch mode
.As an example, let's put our graph-making code into a file named *z.R* with the following contents:

```
pdf("xh.pdf") # set graphical output file
hist(rnorm(100)) # generate 100 N(0,1) variates and plot their histogram
dev.off() # close the graphical output file
```

The items marked with # are *comments*. They're ignored by the R interpreter.Comments serve as notes to remind us and others what the code is doing, in a human-readable format.
Here's a step-by-step breakdown of what we're doing in the preceding code:
• We call the pdf() function to inform R that we want the graph we create to be saved in the PDF file *xh.pdf*.
• We call rnorm() (for *random normal*) to generate 100 *N*(0,1) random variates.
• We call hist() on those variates to draw a histogram of these values.
• We call dev.off() to close the graphical "device" we are using, which is
the file *xh.pdf* in this case. This is the mechanism that actually causes the file to be written to disk.
We could run this code automatically, without entering R's interactive mode, by invoking R with an operating system shell command (such as at the $ prompt commonly used in Linux systems):

```
$ R CMD BATCH z.R
```

You can confirm that this worked by using your PDF viewer to display the saved histogram. (It will just be a plain-vanilla histogram, but R is capable of producing quite sophisticated variations.)

A First R Session:

Let's make a simple data set (in R parlance, a *vector*) consisting of the numbers 1, 2, and 4, and name it x:

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

The standard assignment operator in R is <-. You can also use =, but this is discouraged, as it does not work in some special situations. Note that there are no fixed types associated with variables. Here, we've assigned a vector to x, but later we might assign something of a different type to it. We'll look at vectors and the other types in Section 1.4.
The c stands for *concatenate*. Here, we are concatenating the numbers 1, 2, and 4. More precisely, we are concatenating three one-element vectors that consist of those numbers. This is because any number is also considered to be a one-element vector.Now we can also do the following:

```
> q <- c(x,x,8)
```

which sets q to (1,2,4,1,2,4,8) (yes, including the duplicates).Now let's confirm that the data is really in x. To print the vector to the screen, simply type its name. If you type any variable name (or, more generally,any expression) while in interactive mode, R will print out the value of that variable (or expression). Programmers familiar with other languages such as Python will find this feature familiar. For our example, enter this:

```
> x
[1] 1 2 4
```

Yep, sure enough, x consists of the numbers 1, 2, and 4. Individual elements of a vector are accessed via [ ]. Here's how we can print out the third element of x:

```
> x[3]
[1] 4
```

As in other languages, the selector (here, 3) is called the *index* or *subscript*. Those familiar with ALGOL-family languages, such as C and C++, should note that elements of R vectors are indexed starting from 1, not 0. *Subsetting* is a very important operation on vectors. Here's an example:

```
> x <- c(1,2,4)
> x[2:3]
```

[1The expression x[2:3] refers to the subvector of x consisting of elements 2 through 3, which are 2 and 4 here.
We can easily find the mean and standard deviation of our data set, as follows:

```
> mean(x)
[1] 2.333333
> sd(x)
[1] 1.527525
```

This again demonstrates typing an expression at the prompt in order to print it. In the first line, our expression is the function call mean(x). The return value from that call is printed automatically, without requiring a call to R's print() function.
If we want to save the computed mean in a variable instead of just printing it to the screen, we could execute this code:

```
> y <- mean(x)
```

Again, let's confirm that y really does contain the mean of x:

```
> y
[1] 2.333333
```

As noted earlier, we use # to write comments, like this:

```
> y # print out y
[1] 2.333333
```

Comments are especially valuable for documenting program code, but they are useful in interactive sessions, too, since R records the command history (as discussed in Section 1.6). If you save your session and resume it later, the comments can help you remember what you were doing.Finally, let's do something with one of R's internal data sets (these are
used for demos). You can get a list of these data sets by typing the following:

```
> data()
```

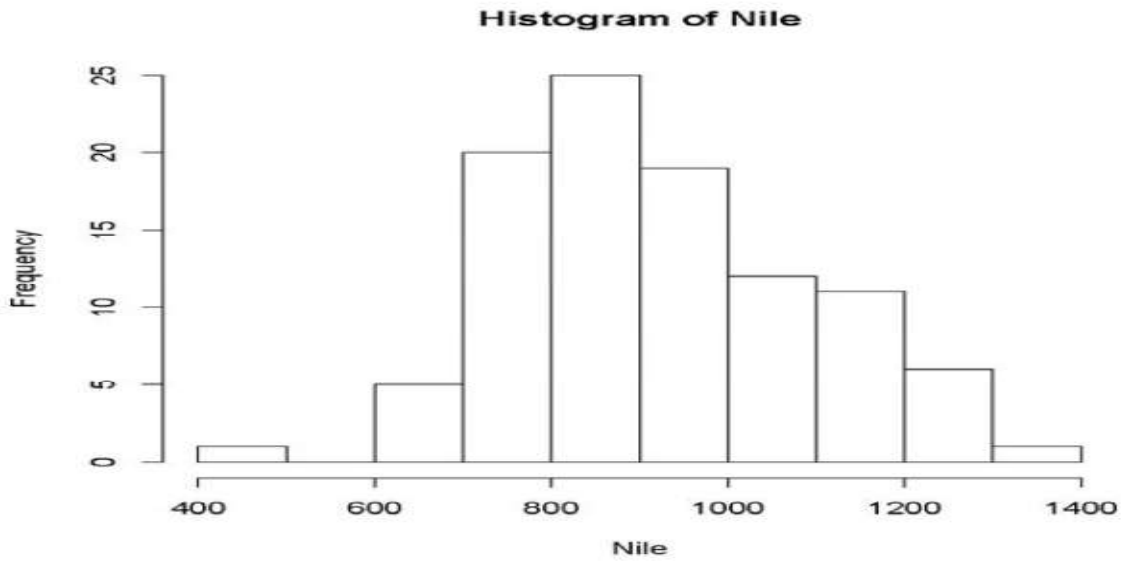
One of the data sets is called Nile and contains data on the flow of the Nile River. Let's find the mean and standard deviation of this data set:

```
> mean(Nile)
[1] 919.35
> sd(Nile)
[1] 169.2275] 2 4
```

We can also plot a histogram of the data:

```
> hist(Nile)
```

A window pops up with the histogram in it, as shown in Figure 1-1. This graph is bare-bones simple, but R has all kinds of optional bells and whistles for plotting. For instance, you can change the number of bins by specifying the breaks variable. The call hist(z,breaks=12) would draw a histogram of the data set z with 12 bins. You can also create nicer labels, make use of color, and make many other changes to create a more informative and eyeappealing graph. When you become more familiar with R, you'll be able to construct complex, rich color graphics of striking beauty.



Well, that's the end of our first, five-minute introduction to R. Quit R by calling the q() function (or alternatively by pressing CTRL-D in Linux or CMD-D on a Mac):

```
> q()
Save workspace image? [y/n/c]: n
```

That last prompt asks whether you want to save your variables so that you can resume work later. If you answer y, then all those objects will be loaded automatically the next time you run R. This is a very important feature, especially when working with large or numerous data sets. Answering y here also saves the session's command history. We'll talk more about saving your workspace.

Introduction to Functions :

As in most programming languages, the heart of R programming consists of writing *functions*. A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

As a simple introduction, let's define a function named oddcount(), whose purpose is to count the odd numbers in a vector of integers. Normally, we would compose the function code using a text editor and save it in a file, but in this quick-and-dirty example, we'll enter it line by line in R's interactive mode. We'll then call the function on a couple of test cases.

```
# counts the number of odd integers in x
> oddcount <- function(x) {
+ k <- 0 # assign 0 to k
+ for (n in x) {
+ if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
+ }
```

```
+ return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

First, we told R that we wanted to define a function named oddcount with one argument, x. The left brace demarcates the start of the body of the function. We wrote one R statement per line.

Until the body of the function is finished, R reminds you that you're still in the definition by using + as its prompt, instead of the usual >. (Actually,+ is a line-continuation character, not a prompt for a new input.) R resumes the > prompt after you finally enter a right brace to conclude the function body.

After defining the function, we evaluated two calls to oddcount(). Since there are three odd numbers in the vector (1,3,5), the call oddcount(c(1,3,5)) returns the value 3. There are four odd numbers in (1,2,3,7,9), so the second call returns 4.

Notice that the modulo operator for remainder arithmetic is %% in R, as indicated by the comment. For example, 38 divided by 7 leaves a remainder of 3:

```
> 38 %% 7
[1] 3
```

For instance, let's see what happens with the following code:

```
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

First, it sets n to x[1], and then it tests that value for being odd or even. If the value is odd, which is the case here, the count variable k is incremented.Then n is set to x[2], tested for being odd or even, and so on.
By the way, C/C++ programmers might be tempted to write the preceding loop like this:

```
for (i in 1:length(x)) {
if (x[i] %% 2 == 1) k <- k+1
}
```

Here, length(x) is the number of elements in x. Suppose there are 25 elements. Then 1:length(x) means 1:25, which in turn means 1,2,3,...,25. This code would also work (unless x were to have length 0), but one of the major themes of R programming is to avoid loops if possible; if not, keep loops simple. Look again at our original formulation:

```
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

It's simpler and cleaner, as we do not need to resort to using the length() function and array indexing.

At the end of the code, we use the return statement:
return(k)
This has the function return the computed value of k to the code that called it. However, simply writing the following also works:

K

R functions will return the last value computed if there is no explicit return() call. However, this approach must be used with care,
In programming language terminology, x is the *formal argument* (or *formal parameter*) of the function oddcount(). In the first function call in the preceding example, c(1,3,5) is referred to as the *actual argument*. These terms allude to the fact that x in the function definition is just a placeholder,whereas c(1,3,5) is the value actually used in the computation. Similarly, in
the second function call, c(1,2,3,7,9) is the actual argument.

Variable Scope

A variable that is visible only within a function body is said to be *local* to that function. In oddcount(), k and n are local variables. They disappear after the function returns:

```
> oddcount(c(1,2,3,7,9))
[1] 4
> n
Error: object 'n' not found
```

It's very important to note that the formal parameters in an R function are local variables. Suppose we make the following function call:

```
> z <- c(2,6,7)
> oddcount(z)
```

Now suppose that the code of oddcount() changes x. Then z would *not* change. After the call to oddcount(), z would have the same value as before. To evaluate a function call, R copies each actual argument to the corresponding local parameter variable, and changes to that variable are not visible outside the function.

Variables created outside functions are *global* and are available within functions as well. Here's an example
:

```
> f <- function(x) return(x+y)
> y <- 3
> f(5)
[1] 8
```

Here y is a global variable.
A global variable can be written to from within a function by using R's *superassignment operator*, <<-.

Default Arguments

R also makes frequent use of *default arguments*. Consider a function definition like this:

```
> g <- function(x,y=2,z=T) { ... }
```

Here y will be initialized to 2 if the programmer does not specify y in the call.Similarly, z will have the default value TRUE.
Now consider this call:

```
> g(12,z=FALSE)
```

Here, the value 12 is the actual argument for x, and we accept the default value of 2 for y, but we override the default for z, setting its value to FALSE.The preceding example also demonstrates that, like many programming languages, R has a *Boolean* type; that is, it has the logical values TRUE and FALSE.

Preview of Some Important R Data Structures

R has a variety of data structures. Here, we will sketch some of the most frequently used structures to give you an overview of R before we dive into the details. This way, you can at least get started with some meaningful examples,
even if the full story behind them must wait.

Vectors, the R Workhorse

The vector type is really the heart of R. It's hard to imagine R code, or even an interactive R session, that doesn't involve vectors. The elements of a vector must all have the same *mode*, or data type. You can have a vector consisting of three character strings (of mode character) or three integer elements (of mode integer), but not a vector with one integer
element and two character string elements..

Scalars

Scalars, or individual numbers, do not really exist in R. As mentioned earlier, what appear to be individual numbers are actually one-element vectors.

Consider the following:

```
> x <- 8
> x
[1] 8
```

Recall that the [1] here signifies that the following row of numbers begins with element 1 of a vector—in this case, x[1]. So you can see that R was indeed treating x as a vector, albeit a vector with just one element.

Character Strings

Character strings are actually single-element vectors of mode character,(rather than mode numeric):

```
> x <- c(5,12,13)
> x
[1] 5 12 13
> length(x)
[1] 3
> mode(x)
[1] "numeric"
> y <- "abc"
> y
[1] "abc"
> length(y)
[1] 1
> mode(y)
[1] "character"
> z <- c("abc","29 88")
> length(z)
[1] 2
> mode(z)
[1] "character"
```

In the first example, we create a vector x of numbers, thus of mode numeric. Then we create two vectors of mode character: y is a one-element (that is,one-string) vector, and z consists of two strings.R has various string-manipulation functions. Many deal with putting strings together or taking them apart, such as the two shown here:

```
> u <- paste("abc","de","f") # concatenate the strings
> u
[1] "abc de f"
> v <- strsplit(u," ") # split the string according to blanks
> v
[[1]]
[1] "abc" "de" "f"
```

Matrices

An R matrix corresponds to the mathematical concept of the same name: a rectangular array of numbers. Technically, a matrix is a vector, but with two additional attributes: the number of rows and the number of columns. Here is some sample matrix code:

```
> m <- rbind(c(1,4),c(2,2))
> m
[,1] [,2]
[1,] 1 4
[2,] 2 2
> m %*% c(1,1)
[,1]
[1,] 5
[2,] 4
```

First, we use the rbind() (for *row bind*) function to build a matrix from two vectors that will serve as its rows, storing the result in m. (A corresponding function, cbind(), combines several columns into a matrix.) Then entering the variable name alone, which we know will print the variable, confirms that the intended matrix was produced. Finally, we compute the matrix product of the vector (1,1) and m. The matrix-multiplication operator, which you may know from linear algebra courses, is %*% in R.

Matrices are indexed using double subscripting, much as in C/C++, although subscripts start at 1 instead of 0.

```
> m[1,2]
[1] 4
> m[2,2]
[1] 2
```

An extremely useful feature of R is that you can extract submatrices from a matrix, much as you extract subvectors from vectors. Here's an example:

```
> m[1,] # row 1
[1] 1 4
> m[,2] # column 2
[1] 4 2
```

Lists

Like an R vector, an R list is a container for values, but its contents can be items of different data types. (C/C++ programmers will note the analogy to a C struct.) List elements are accessed using two-part names, which are indicated with the dollar sign $ in R. Here's a quick example:

```
> x <- list(u=2, v="abc")
> x
```

```
$u
[1] 2
$v
[1] "abc"
> x$u
[1] 2
```

The expression x$u refers to the u component in the list x. The latter contains one other component, denoted by v.
A common use of lists is to combine multiple values into a single package that can be returned by a function. This is especially useful for statistical functions, which can have elaborate results. As an example, consider R's basic histogram function, hist(), introduced in Section 1.2. We called the function on R's built-in Nile River data set:

```
> hist(Nile)
```

This produced a graph, but hist() also returns a value, which we can save:

```
> hn <- hist(Nile)
```

What's in hn? Let's take a look:

```
> print(hn)
$breaks

[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

$counts

[1] 1 0 5 20 25 19 12 11 6 1

$intensities

[1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
[6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$density

[1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
[6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$mids

[1] 450 550 650 750 850 950 1050 1150 1250 1350

$xname

[1] "Nile"

$equidist
[1] TRUE

attr(,"class")
```

[1] "histogram"
Don't try to understand all of that right away. For now, the point is that, besides making a graph, hist() returns a list with a number of components. Here, these components describe the characteristics of the histogram. For instance, the breaks component tells us where the bins in the histogram start and end, and the counts component is the numbers of observations in each bin.

The designers of R decided to package all of the information returned by hist() into an R list, which can be accessed and manipulated by further R commands via the dollar sign. Remember that we could also print hn simply by typing its name:

```
> hn
```

But a more compact alternative for printing lists like this is str():

```
> str(hn)
List of 7
$ breaks : num [1:11] 400 500 600 700 800 900 1000 1100 1200 1300 ...
$ counts : int [1:10] 1 0 5 20 25 19 12 11 6 1
$ intensities: num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ density : num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ mids : num [1:10] 450 550 650 750 850 950 1050 1150 1250 1350
$ xname : chr "Nile"
$ equidist : logi TRUE
- attr(*, "class")= chr "histogram"
```

Here str stands for *structure*. This function shows the internal structure of any R object, not just lists.

Data Frames :

A typical data set contains data of different modes. In an employee data set, for example, we might have character string data, such as employee names, and numeric data, such as salaries. So, although a data set of (say) 50 employees with 4 variables per worker has the look and feel of a 50-by-4 matrix, it does not qualify as such in R, because it mixes types.

Instead of a matrix, we use a *data frame*. A data frame in R is a list, with each component of the list being a vector corresponding to a column in our "matrix" of data. Indeed, you can create data frames in just this way:

```
> d <- data.frame(list(kids=c("Jack","Jill"),ages=c(12,10)))
> d
kids ages
12 Jill 10
> d$ages
[1] 12 10
```

Typically, though, data frames are created by reading in a data set from
a file or database. Jack 12

Classes :

R is an object-oriented language. *Objects* are instances of *classes*. Classes are a bit more abstract than the data types you've met so far. Here, we'll look briefly at the concept using R's S3 classes. (The name stems from their use in the old S language, version 3, which was the inspiration for R.) Most of R is based on these classes, and they are exceedingly simple. Their instances are simply R lists but with an extra attribute: the class name.For example, we noted earlier that the (nongraphical) output of the hist() histogram function is a list with various components, such as break and count components. There was also an *attribute*, which specified the class of the list, namely histogram.

```
> print(hn)
$breaks
[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400
```

$counts
[1] 1 0 5 20 25 19 12 11 6 1
...
...
attr(,"class")
[1] "histogram"
A generic function stands for a family of functions, all serving a similar purpose but each appropriate to a specific class. A commonly used generic function is summary(). An R user who wants to use a statistical function, like hist(), but is unsure of how to deal with its output (which can be voluminous), can simply call summary() on the output, which is not just a list but an instance of an S3 class.
The plot() function is another generic function. You can use plot() on just about any R object. R will find an appropriate plotting function based on the object's class. Classes are used to organize objects. Together with generic functions, they allow flexible code to be developed for handling a variety of different but related tasks. Chapter 9 covers classes in depth.