1

WHY R?

1.1 What Is R?

R is a scripting language for statistical data manipulation and analysis. It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T. The name S, obviously standing for statistics, was an allusion to another programming language developed at AT&T with a one-letter name, C. S later was sold to a small firm, which added a GUI interface and named the result S-Plus.

R has become more popular than S/S-Plus, both because it's free and because more people are contributing to it. R is sometimes called "GNU S," to reflect its open source nature. (GNU is a major collection of open source software.)

1.2 Why Use R for Your Statistical Work?

Why use anything else? As the Cantonese say, *yauh peng, yauh leng*—"both inexpensive and beautiful."

Its virtues:

- a public-domain implementation of the widely-regarded S statistical language; R/S is the *de facto* standard among professional statisticians
- comparable, and often superior, in power to commercial products in most senses
- available for Windows, Macs, Linux
- in addition to enabling statistical operations, it's a general programming language, so that you can automate your analyses and create new functions
- object-oriented and functional programming structure
- your data sets are saved between sessions, so you don't have to reload each time
- open-software nature means it's easy to get help from the user community, and lots of new functions get contributed by users, many of whom are prominent statisticians

I should warn you that one typically submits commands to R via text in a terminal window, rather than mouse clicks in a Graphical User Interface (GUI). If you can't live without GUIs, you use one of the free GUIs that have been developed for R, e.g. R Commander or JGR, or for programming, ESS, but most users do not use a GUI. This is not to say that R doesn't do graphics. On the contrary, it produces excellent graphics. But the graphics are for the output, e.g. plots, not for the input.

The terms *object-oriented* and *functional programming* may pique the interests of computer scientists, but they may be foreign to other readers. Yet they are actually quite relevant to anyone who uses R.

The term *object-oriented* can be explained by example, say statistical regression. When you perform a regression analysis with other statistical packages, say SAS or SPSS, you get a mountain of output on the screen. By contrast, if you call the `lm()` regression function in R, the function returns an object containing all the results—estimated coefficients, their standard errors, residuals, etc. You then pick and choose which parts of that object to extract, as you wish.

You will see that this makes programming much easier, partly because there is a uniformity of access. This stems from the fact that R is *polymorphic*, which means that the same function can be applied to different types of objects, with results tailored to the different object types. Such a function is called a *generic function*. (If you are a C++ programmer, you may have seen the same concept in *virtual functions*.) Consider for instance the `plot()` function. If you apply it to a simple list of numbers, you get a simple plot of them, but if you apply it to the output of a regression analysis, you get a set of plots of various aspects of the regression output. This is nice, since it means that you, as a user, have fewer commands to remember! For instance, you know that you can use the `plot()` function on just about any object produced by R.

The fact that R is a programming language rather than a collection of discrete commands means that you can combine several commands, each

one using the output of the last, with the resulting combination being quite powerful and extremely flexible. (Linux users will recognize the similarity to shell pipe commands.)

For example, consider this (compound) command

```
nrow(subset(x03,z==1))
```

First the `subset()` function takes the data frame `x03`, and extracts all those records for which the variable `z` has the value 1. The resulting new frame is then fed into the `nrow()` function, the function that counts the number of rows in a frame. The net effect is to report a count of `z` = 1 in the original frame.

R has many functional programming features, with important advantages:

- Clearer, more compact code.
- Potentially much faster execution speed.
- Less debugging (since you write less code).
- Easier transition to parallel programming.

A common theme in R programming is the avoidance of writing explicit loops. Instead, one exploits R's functional programming and other features, which do the loops internally. They are in some cases much more efficient, which can make a huge timing difference when running R on large data sets.

2

GETTING STARTED

In this chapter you'll get a quick introduction to R—how to invoke it, what it can do and what files it uses. We'll develop just enough of the basics for use in examples in the next few chapters. Details will come later.

You may already have R installed on your machine, as for instance your employer or university may have installed it. If not, see Chapter ?? for easy installation instructions.

2.1 How to Run R

R has two modes, *interactive* and *batch*. The former is the typical one used. You type in commands, get results, type further commands and so on. Batch mode is useful for production jobs, in which the same program is run, say once per day, and you want the process to be automated.

2.1.1 Interactive Mode

You start R by typing "R" on the command line in a Linux or Mac terminal window. (The Macintosh operating system is basically Unix, and thus R's behavior is typically Unix-like, thus the same as Linux.) In a Windows, click on the R icon. In any of these cases, you'll get a greeting, and then the R prompt, the > sign. The screen will look something like

```
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*.

For example, let's find the mean absolute value of the N(0,1) distribution, based on a simulated sample of 100 N(0,1) variates:

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

The code generates the 100 random variates, then finds their absolute values, then finds the mean of those values.

The "[1]" here means in this row of output, the first item is item 1 of that output. If there were say, two rows of output with six items per row, the second row would be labeled [7]. Our output in this case consists of only one row, but this notation helps users read voluminous output consisting of many rows. Here is an example:

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

There are 10 values in the output, and the label "[7]" in the second row is telling us that, for instance, the eighth output item is 0.6877001.

You can also store some R commands in a file, say **z.R**. (By convention, R code files have the suffix **.R** or **.r**.) You could then issue the command

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

which would execute the contents of that file.

2.1.2 Running R in Batch Mode

Sometimes it's preferable to automate the process of running R. For example, we may wish to run an R script that generates a graph output file, and not have to bother with manually running R. Here's how it could be done.

Let's put our code into a file **z.R**, with 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, but serve as memos to ourselves, reminding us what we were doing. Here is what the above code does:

- We call the `pdf()` function to inform R that we will want the graph we create to be saved in the file **xh.pdf**.
- We call `rnorm()` ("random normal") to generate 100 N(0,1) random variates.
- We call `hist()` on those variates, to draw a histogram of them.
- We call `dev.off()`, to close the graphical device we were using, in this case the file `xh.pdf`. This is the mechanism that actually causes the file contents to be written to disk.

We could run this code automatically, without entering R's interactive mode, by simply typing an operating system shell command (e.g. with the $ prompt common in Linux systems):

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

You can confirm that this worked by using your PDF viewer to display the saved histogram.

2.2 A First R Example Session (5 Minutes)

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

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

The standard assignment operator in R is `<-`. One can use =, but its use is discouraged, as it does not work in some special situations.

Note that there are no types associated with variables. Here we've pointed `x` to a vector, but later we could assign something of a different type to it.

The "c" stands for "concatenate." Here we are concatenating the numbers 1, 2 and 4. Or more precisely, we are concatenating three one-element vectors consisting of those numbers. This is because any number is considered a one-element vector.

We can also do, for instance,

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

which would set `q` to (1,2,4,1,2,4,8) (yes, including the duplicates).

Since "seeing is believing," we can 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 an expression, while in interactive mode, R will print out the value of that variable or expression. (Programmers of some other languages, such as Python, will find this feature familiar.) For example,

```
> 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 "[ ]." For instance, let's print out the third element of x:

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

As with other languages, that 3 is called the index or subscript. For those familiar with ALGOL-family languages such as C/C++, note that R elements begin at index 1, not 0.

A very important operation on vectors is subsetting. For example,

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

The 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:

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

Note that this is again an example of R's interactive mode feature in which typing an expression results in printing the expression's value. In the first instance above, our expression is "mean(x)," which does have a value—the return value of the function. Thus the value is printed automatically, without our having to, say, call R's print() function.

If we had wanted to save the mean in a variable instead of just printing it to the screen, we could do, say,

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

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

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

As noted earlier, we use # to write comments.

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

These of course are especially useful when writing programs, but they are useful for interactive use too, since R does record your command history (see Section ??). The comments then help you remember what you were doing when you later read that record.

As the last example in this quick introduction to R, let's work with one of R's internal datasets, which it uses for demos. You can get a list of these datasets by typing

```
> data()
```

One of the datasets is `Nile`, containing data on the flow of the Nile River. Let's again find the mean and standard deviation,

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

and also plot a histogram of the data:

```
> hist(Nile)
```

A window pops up with the histogram in it, as seen in Figure ??. This one is bare-bones simple, but R has all kinds of bells and whistles you can use optionally. For instance, you can change the number of bins by specifying the breaks variable; `hist(z,breaks=12)` would draw a histogram of the data `z` with 12 bins. You can make nicer labels, and do many other things to make the graph more informative and eye-appealing. 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 this first 5-minute introduction. We leave by calling the quit function (or optionally by hitting ctrl-d in Linux or a Mac):

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

That last question asks whether we want to save our variables, so that we can resume work later on. If we answer y, then the next time we run R, all those objects will automatically be loaded. This is a very important feature, especially when working with large or numerous datasets; see more in

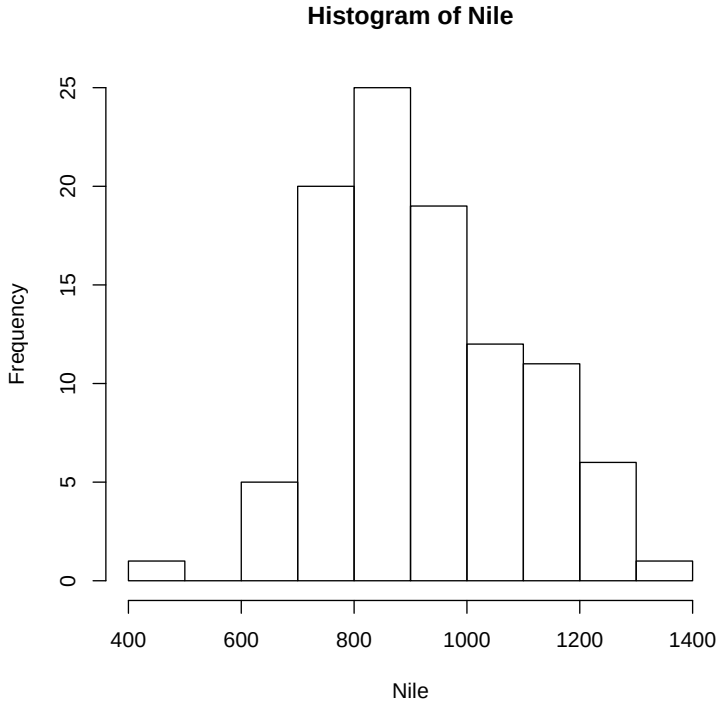

Figure 2-1: Nile data, nonfancy presentation

Section ??. Note that answering y here also results in our command history being saved.

2.3 Functions: a Short Programming Example

As with most programming languages, the heart of R programming consists of writing `functions`, groups of code that take inputs, compute something with them, and then output a result.

In the following example, we first define a function `oddcount()` while in R's interactive mode. (Normally we would compose the function using a text editor, but in this quick-and-dirty example, we enter it line by line in interactive mode.) We then call the function on a couple of test cases.

The goal of the function is to count the number of odd numbers in its argument vector.

```
# 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
```

Since there are three odd numbers in the vector (1,3,5), the call `oddcount(c(1,3,5))` returned the value 3. There are four odd numbers in (1,2,3,7,9), so the call returned 4.

Here is what happened when we defined the function above: We first told R that we would define a function `oddcount()` of one argument `x`. The left brace demarcates the start of the body of the function. We wrote one R statement per line. Since we were still in the body of the function, R reminded us of that by using + as its prompt instead of the usual >. (Actually, this is a line continuation character.) After we finally entered a right brace to end the function body, R resumed the > prompt.

As noted in the comment, in R `%%` is the "mod" operator, i.e. for remainder arithmetic. For example, 38 divided by 7 leaves a remainder of 3:

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

By the way, C/C++ programmers might be tempted to write the above loop as

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

Here `length(x)` is the number of elements in `x`. Say it's 25. Then `1:length(x)` means 1:25, which in turn means 1,2,3,...,25. This would work, but one of the major themes of R is to first avoid loops if possible, and if not, to keep loops simple. Our original formulation

```
for (n in x)  {
   if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
}
```

is simpler and cleaner, as we do not have to resort to using the `length()` function.

In general programming language terminology, `x` is referred to as the *formal argument* (or *formal parameter*) of the function `oddcount()`. In the first example call above, c(1,3,5) is referred to as the *actual argument*. These terms allude to the fact that `x` is just a placeholder, while c(1,3,5) is the value ac-

tually used in the computation. In the second example, c(1,2,3,7,9) is the actual argument.

A variable in effect only within a function body is said to be *local* to that function. In oddcount() above, k and n are local variables. They disappear when the function returns, e.g.

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

It's very important to note that function arguments in R are read-only. Suppose we have the call

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

and suppose that the code in a different version of oddcount() were to change x. Then z would not change. (This is because, internally R makes a copy of each argument to an unseen local variable, and changes to an argument are implemented as changes to the unseen local.) This will be discussed in detail in Chapter ??.

Variables created outside functions are *global*, and are available within functions as well:

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

Many programmers dislike using global variables, as they believe it makes code hard to follow. I do not hold that view, and would point out that in fact some situations, e.g. threads programming (Chapter ??), actually *require* use of globals.

In any event, if one feels comfortable using global variables, a global can be written to from within a function, using R's superassignment operator, <<-. This is discussed in Chapter ??.

R also makes frequent use of *default arguments*. In the (partial) function definition

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

y will be initialized to 2 if the programmer does not specify y in the call. Similarly, z will have the default value TRUE. In the call

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

the value 12 will be the actual argument for x, and we are accepting the default value of 2 for y, but we are overriding the default for z, setting it to FALSE.

Note that R allows one to abbreviate TRUE and FALSE to Tand F, which I often do. But this can cause trouble if the programmer has T or F as a variable name, so you may wish to avoid abbreviating the boolean, i.e. logical, values.

2.4 Preview of Some Important R Data Structures

Here we browse through some of the most frequently-used R data structures. This will give you a better overview of R before diving into the details, and will also allow usage of these structures in examples without having "forward references."

2.4.1 Vectors, the R Workhorse

The vector type is really the core of R. It's hard to imagine R code, or even an R interactive session, that doesn't involve vectors.

The elements of a vector must all have the same *mode*, i.e. data type. One can have, say, a vector consisting of three character strings, i.e. three elements of mode character, or three elements of integers (mode integer), but not one integer element and two character string elements.

There is much more to say about vectors, but our examples of vectors in the preceding sessions will suffice for now. So, let's move on to the next type of data, matrices.

2.4.2 Matrices

An R matrix corresponds to the mathematical concept of the same name, i.e. a rectangular array of numbers. Technically, it is a vector, with two attributes added—the numbers of rows and columns.

Here is some sample 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 used the rbind() ("row bind") function to build a matrix from two vectors, storing the result in m. We then typed that latter name, to confirm that we produced the intended matrix. Finally, we computed the matrix product of the vector (5,4) and m. In order to get matrix multiplication of

the type many readers know from linear algebra courses, we used the %*% operator.

Subscripting is done similarly to C/C++ (though again subscripts start at 1 instead of 0):

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

An extremely useful feature is that we can extract submatrices, similarly to our extraction of subvectors from vectors. For example,

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

Details are in Chapter ??.

2.4.3 Lists

An R list is a container whose contents can be items of diverse data types, as opposed to them being, say, all numbers, as in a vector. (C/C++ programmers will note the analogy to a C struct.)

List members (which in C are delimited with periods) are indicated with dollar signs in R. Here's a quick toy example:

```
> x <- list(u=2, v="abc")
> x
$u
[1] 2

$v
[1] "abc"

> x$u
[1] 2
```

Here x$u is the u component in the list x.

A common usage of lists is to package the return values of elaborate statistical functions. As an example, consider R's basic histogram function hist(), introduced earlier in Section ??. There we call the function on R's built-in Nile River data set:

```
> hist(Nile)
```

This produced a graph, of course, but `hist()` actually does return a value, which we could 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, but consider for instance the `breaks` component. This tells us where the bins in the histogram start and end. The `counts` component is of course the numbers of observations in each bin.

Again, the details are not important here, but the point is that the designers of R decided to package all of the output of `hist()` into a single R list, whose individual components are accessible via the dollar sign.

2.4.4 Data Frames

A typical data set contains data of diverse 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, while a data set of, say, 50 employees with 4 variables per worker, has the "look and feel" of a 50x4 matrix, it does

not qualify as such in R if it mixes types. Instead of a matrix, we use an R data frame.

A data frame is technically a list, with each component of the list being a vector corresponding to a column in our data "matrix." In fact, one can actually create data frames using this idea:

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

Typically, though, data frames are created by reading in a data set from a file.

2.4.5 Classes

Again, R is an object-oriented language. The objects are instances of a programming data type called a `class`.

Classes are a bit more abstract than the data types we've presented so far, so we'll postpone the details until Chapter ??. But as it's impossible to discuss R without at least mentioning the word "class" occasionally, we'll go through a very brief overview here.

R's S3 classes, on which most of R is based, are exceedingly simple: They are simply R lists—but with an extra attribute, which is the class.

For example, we noted in Section ?? that the (nongraphical) output of the `hist()` histogram function is a list, with components such as `breaks`, `counts`. There was also an attribute, the class of the list, "`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"
```

Some readers at this point might be asking, "If a class is just a list, why do we need two concepts, list and class?" The answer lies in something called *generic functions*.

A generic function is actually a placeholder for a family of functions having similar actions but each one appropriate to a specific class. A common such function is `summary()`. An R user who is trying a new statistical function

but who is unsure of how to deal with its output (which can be voluminous), can simply call `summary()` on the output. That function is actually a family of functions, each for a different class. When the user calls `summary()`, R will then search for a summary function appropriate to the class at hand. Similarly, an R user can call `plot()` on an object of a class that is new to him/her, and R will find a plotting function appropriate for that class.

2.4.6 Character Strings

Character strings are actually single-element vectors, of mode `character` rather than `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 above, we create a vector `x` of numbers, thus of mode `numeric`. Then we create vectors of mode `character`, with `y` being a one-element, i.e. one-string, vector, and `z` consisting of three strings.

R has various string-manipulation functions. Many deal with putting strings together or taking them apart, such as:

```
> 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"
```

2.5 Extended Example: Regression Analysis of Exam Grades

For our second introductory example, we walk through a brief statistical regression analysis. There won't be much actual programming in this example, but it will illustrate usage of some of the data types from the last section, including R's S3 objects, and will serve as the basis for several of our programming examples in subsequent chapters.

Here I have a file, **ExamsQuiz.txt** of grades from a class I taught. The first few lines are

```
2    3.3   4
3.3  2     3.7
4    4.3   4
2.3  0     3.3
...
```

The numbers correspond to letter grades on a four-point scale, so that 3.3, for instance, is a B+. Each line contains the data for one student, consisting of the midterm examination grade, final examination grade, and the average quiz grade. One might be interested in seeing how well the midterm and quiz grades predict the student's grade on the final examination.

Let's first read in the file:

```
> examsquiz <- read.table("ExamsQuiz.txt",header=FALSE)
```

Our file had no header line, i.e. no line naming each of the variables, so we specified `header=FALSE`, an example of the default arguments mentioned in Section **??**. Actually, the default value of that argument is `FALSE` anyway, as can be checked by R's online help facility for `read.table()`. Thus we didn't need to specify the `header` argument, but it's clearer if we do.

So, our data is now in `examsquiz`, an R object of class "`data.frame`":

```
> class(examsquiz)
[1] "data.frame"
```

Just to check that the file was read in correctly, let's take a look at the first few rows:

```
> head(examsquiz)
  V1  V2  V3
1 2.0 3.3 4.0
2 3.3 2.0 3.7
3 4.0 4.3 4.0
4 2.3 0.0 3.3
5 2.3 1.0 3.3
6 3.3 3.7 4.0
```

Lacking a header for the data, R named the columns V1, V2 and V3. Row numbers appear on the left. (Actually, it is better to have a header in our data file, with meaningful names such as `Exam1`. We will usually specify names in later examples.)

Let's try to predict Exam 2 (second column of `examsquiz`) from Exam 1 (first column):

```
lma <- lm(examsquiz[,2] ~ examsquiz[,1])
```

The `lm()` ("linear model") function call here instructs R to fit the prediction equation

$$\text{predicted Exam 2} = \beta_0 + \beta_1 \text{Exam 1} \tag{2.1}$$

using least squares. Note that Exam 1, being stored in column of our data frame, is referred to collectively as `examsquiz[,1]`. Here the lack of the first subscript, i.e. row number, means that we are referring to the entire column, and similarly for Exam 2.

We also could have written

```
lma <- lm(examsquiz$V2 ~ examsquiz$V1)
```

recalling that a data frame is a special cases of a list, with each column being one element of the list.

The results are returned in the object we've named `lma` of class "`lm`". We can see the various components of that object by calling `attributes()`:

```
> attributes(lma)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
```

For instance, the estimated values of the β_i are stored in `lma$coefficients`. As usual, we can print them, by typing the name.

And by the way save some typing by abbreviating, as long as we don't shorten a component's name to the point of confusing it with other components. If a list consists (only) of the components, say, `xyz`, `xywa` and `xbcde`, then the second and third ones can be abbreviated to `xyw` and `xb`, respectively.

Thus we can type

```
> lma$coef
   (Intercept) examsquiz[, 1]
     1.1205209      0.5899803
```

Since `lma$coefficients` is a vector, printing it is simple. But consider what happens when we print the object `lma` itself:

```
> lma

Call:
lm(formula = examsquiz[, 2] ~ examsquiz[, 1])

Coefficients:
   (Intercept)  examsquiz[, 1]
         1.121           0.590
```

How did R know to print only these items, and not the other components of `lma`? The answer is that `print()` is another example of R's generic functions. As explained earlier in Section **??**, `print()`, actually hands off the work to a print function that has been declared to be the one associated with objects of class "`lm`", namely `print.lm()`.

We can get a more detailed printout of the contents of `lma` by calling `summary()`, our earlier example of a generic function. It in this case triggers a call to `summary.lm()` behind the scenes. Thus we get a regression-specific summary, which is:

```
> summary(lma)

Call:
lm(formula = examsquiz[, 2] ~ examsquiz[, 1])

Residuals:
    Min      1Q  Median      3Q     Max
-3.4804 -0.1239  0.3426  0.7261  1.2225

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)      1.1205     0.6375   1.758  0.08709 .
examsquiz[, 1]   0.5900     0.2030   2.907  0.00614 **
...
```

A number of other generic functions are defined for this class. See the online help (Section **??**) for `lm()` for details.

To estimate a prediction equation for Exam 2 from both Exam 1 and the Quiz score, we would use the '+' notation:

```
> lmb <- lm(examsquiz[,2] ~ examsquiz[,1] + examsquiz[,3])
```

2.6 Startup and Shutdown

Like any sophisticated application, R can be customized for your convenience with startup files. In addition, it can save all or part of your session, such as a record of what you did, to an output file.

If there are R commands you would like to have executed at the beginning of every R session, you can place them in a file **.Rprofile** either in your home directory or in the directory from which you are running R. The latter directory is searched for such a file first, which allows you to customize for a particular project.

For example, one might place in **.Rprofile** the line

```
options(editor="/usr/bin/vim")
```

to set one's text editor for R to invoke if we call `edit()`.

In **.Rprofile** on my machine at home, I have a line

```
.libPaths("/home/nm/R")
```

which automatically adds to my R search path a directory in which I keep my auxiliary packages. In that file I also have quite a few lines of startup material needed running Rmpi, a parallel processing package for R.

Like most programs, R has the notion of your *current working directory*. Upon startup, this will be the directory from which you started R, if you're using Linux or a Mac. For Windows, it will probably be your **Documents** folder. In any case, you can always check your current directory by typing

```
> getwd()
```

You can change your working directory by calling `setwd()` with the desired directory as a quoted argument.

As you proceed through an interactive R session, R will record the commands you submit. And as you long as you answer yes to the question "Save workspace image?" put to you when you quit the session, R will save all the objects you created in that session, and restore them in your next session. You thus do not have to recreate the objects again from scratch if you wish to continue work from before. The saved workspace file is named **.Rdata**, and is located either in the directory from which you invoked this R session (Linux) or in the R installation directory (Windows).

If you wish to have a speedier startup/shutdown you can skip loading all those files, and save your session at the end, by running R with the `vanilla` option:

```
R --vanilla
```

There are various intermediate options. Type

Other information on startup files is available by querying R's online help facility:

```
> ?Startup
```

2.7 Getting Help

There is a plethora of resources one can draw upon to learn more about R. These include several facilities within R itself, and of course on the Web.

2.7.1 R's Internal Help Facilities

Much work has gone into making R self-documenting. The next few subsections will present some of R's help facilities.

2.7.1.1 The help() Function

For online help, invoke `help()`. For example, to get information on the `seq()` function, type

```
> help(seq)
```

The shortcut to `help()` is a question mark, such as

```
> ?seq
```

Special characters and some reserved words must be quoted. For instance, type

```
> ?"<"
```

to get help on the < operator, and

```
> ?"for"
```

to see what the manual has to say about `for` loops.

2.7.1.2 The example() Function

Each of the help entries comes with examples. One really nice feature is that the `example()` function will actually run thus examples for you. For instance:

```
> example(seq)

seq> seq(0, 1, length.out=11)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

seq> seq(stats::rnorm(20))
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

seq> seq(1, 9, by = 2) # match
```

```
[1] 1 3 5 7 9

seq> seq(1, 9, by = pi)# stay below
[1] 1.000000 4.141593 7.283185

seq> seq(1, 6, by = 3)
[1] 1 4

seq> seq(1.575, 5.125, by=0.05)
 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075 2.125
[13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625 2.675 2.725
[25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225 3.275 3.325
[37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725 3.775 3.825 3.875 3.925
[49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.325 4.375 4.425 4.475 4.525
[61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.925 4.975 5.025 5.075 5.125

seq> seq(17) # same as 1:17
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

Imagine how useful this is for graphics! To get a quick and very nice example, the reader is urged to run the following **RIGHT NOW**:

```
> example(persp)
```

A series of example graphs for the `persp` ("perspective") function will be displayed. Hit the Enter key in the R console when you want to go to the next one. Note by the way that the code for each example is shown in the console, so you can experiment, tweaking the arguments. One of the graphs displayed is shown in Figure ??.

You should also look into `demo()`, e.g. calling `demo(persp)`.

2.7.1.3 If You Don't Know Quite What You're Looking for

You can use the function `help.search()` to do a "Google"-style search through R's documentation in order to determine which function will play a desired role. For instance, say you need a function to generate random variates from multivariate normal distributions. To determine what function, if any, does this, you could type

```
> help.search("multivariate normal")
```

getting a response which contains this excerpt:

```
mvrnorm(MASS)           Simulate from a Multivariate Normal
                        Distribution
```

This tells us that the function `mvrnorm()` will do the job, and it is in the package `MASS`.

Again, there is a question mark-based shortcut:

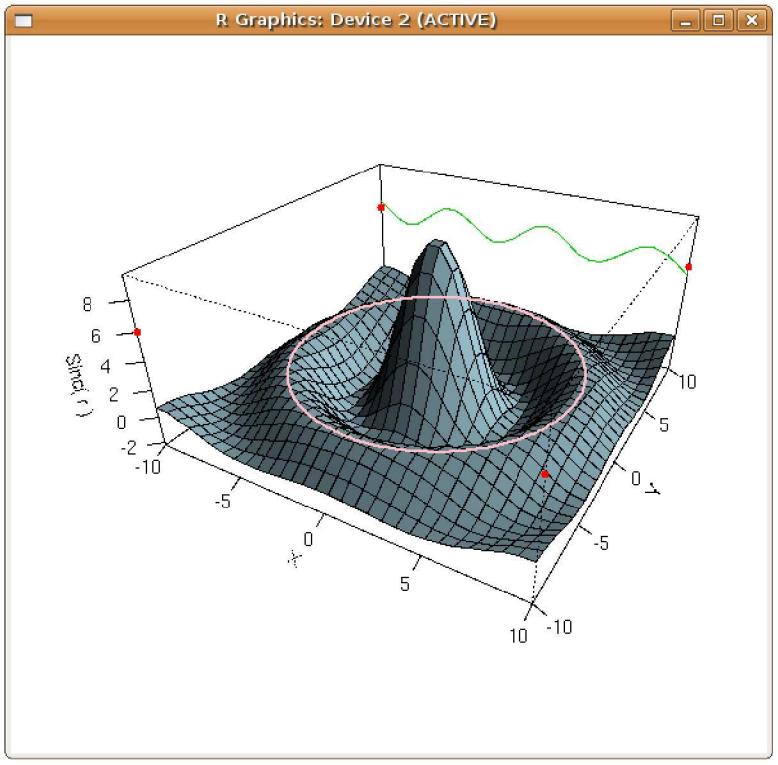

Figure 2-2: One of the `persp()` *examples*

```
> ??"multivariate normal"
```

2.7.1.4 Help for Nonfunctions

R's internal help files cover more than just pages for specific functions.

For example, the function `mvrnorm()` in the package `MASS` was mentioned above. We can not only get information on the function via

```
> ?mvrnorm
```

but we can also learn about the entire package by typing

```
> help(package=MASS)
```

Help is available on general topics too. For instance,

```
> ?files
```

gives us information on a number of file-manipulation functions, such as `file.create()`. Here are some others:

```
Arithmetic
Comparison
Control
Dates
Extract
Math
Memory
NA
NULL
NumericaConstants
Paren
Quotes
Startup
Syntax
```

You may find browsing through these even without a specific need to be educational.

2.7.1.5 Help for Batch Mode

Recall from Section ?? that R has many batch commands. To obtain help on the batch command *cmd*, type

```
R CMD cmd --help
```

For example:

```
R CMD INSTALL --help
```

to learn all the options associated with INSTALL (which installs new R packages; see Chapter ??.

2.7.2 Help on the Internet

There are many excellent resources on R on the Internet. Here are a few:

- The R Project's own manuals are available at the R home page, *http://cran.r-project.org*. Click on "Manuals."
- Various R search engines are listed on the R home page; *http://www.r-project.org*. Click on "Search."
- The sos package offers highly sophisticated searching of R materials. Again, see Chapter ?? on how to install R packages.
- I use the RSeek search engine quite often, *http://www.rseek.org*.
- You can post your R questions to **r-help**, the R listserve. You can obtain information on this and other R listserves at *http://www.r-project.org/mail.html*. There are various interfaces one can use; I like GMANE.

- Given its single-letter name, R is difficult to search for when using general search engines such as Google. But there are definitely tricks one can employ.
  One approach is to use Google's `filetype` criterion. To search for R scripts, i.e. those with a **.R** suffix, that pertain to, say, permutations, enter

```
filetype:R permutations -rebol
```

  That last term asks Google to exclude pages with the word "rebol," as the REBOL programming language uses the same suffix.