1

Getting started

Welcome to the world of statistical programming. This book will contain a lot of specific advice about the hows and whys of the subject. We start in this chapter by giving you an idea of what statistical programming is all about. We will also tell you what to expect as you proceed through the rest of the book. The chapter will finish with some instructions about how to download and install R, the software package and language on which we base our programming examples.

1.1 What is statistical programming?

Computer programming involves controlling computers, telling them what calculations to do, what to display, etc. Statistical programming is harder to define. One definition might be that it's the kind of computer programming statisticians do – but statisticians do all sorts of programming. Another would be that it's the kind of programming one does when one is doing statistics – but again, statistics involves a wide variety of computing tasks.

For example, statisticians are concerned with collecting and analyzing data, and some statisticians would be involved in setting up connections between computers and laboratory instruments – but we would not call that statistical programming. Statisticians often oversee data entry from questionnaires, and may set up programs to aid in detecting data entry errors. That *is* statistical programming, but it is quite specialized, and beyond the scope of this book.

Statistical programming involves doing computations to aid in statistical analysis. For example, data must be summarized and displayed. Models must be fit to data, and the results displayed. These tasks can be done in a number of different computer applications: Microsoft Excel, SAS, SPSS, S-PLUS, R, Stata, etc. Using these applications is certainly statistical computing, and usually involves statistical programming, but it is not the focus of this book. In this book our aim is to provide a foundation for an understanding of how these applications work: we describe the calculations they do, and how you could do them yourself.

Since graphs play an important role in statistical analysis, drawing graphics of one, two, or higher dimensional data is an aspect of statistical programming.

An important part of statistical programming is stochastic simulation. Digital computers are naturally very good at exact, reproducible computations, but the real world is full of randomness. In stochastic simulation we program a computer to act as though it is producing random results, even though if we knew enough, the results would be exactly predictable.

Statistical programming is closely related to other forms of numerical programming. It involves optimization and approximation of mathematical functions. There is less emphasis on differential equations than in physics or applied mathematics (though this is slowly changing). We tend to place more of an emphasis on the results and less on the analysis of the algorithms than in computer science.

1.2 Outline of the book

This book is an introduction to statistical programming. We will start with basic programming: how to tell a computer what to do. We do this using the open source R statistical package, so we will teach you R, but we will try not to *just* teach you R. We will emphasize those things that are common to many computing platforms.

Statisticians need to display data. We will show you how to construct statistical graphics. In doing this, we will learn a little bit about human vision, and how it motivates our choice of display.

In our introduction to programming, we will show how to control the flow of execution of a program. For example, we might wish to do repeated calculations as long as the input consists of positive integers, but then stop when an input value hits 0. Programming a computer requires basic logic, and we will touch on Boolean algebra, a formal way to manipulate logical statements. The best programs are thought through carefully *before* being implemented, and we will discuss how to break down complex problems into simple parts. When we are discussing programming, we will spend quite a lot of time discussing how to *get it right*: how to be sure that the computer program is calculating what you want it to calculate.

One distinguishing characteristic of statistical programming is that it is concerned with randomness: random errors in data, and models that include stochastic components. We will discuss methods for simulating random values with specified characteristics, and show how random simulations are useful in a variety of problems.

Many statistical procedures are based on linear models. While discussion of linear regression and other linear models is beyond the scope of this book, we do discuss some of the background linear algebra, and how the computations it involves can be carried out. We also discuss the general problem of numerical optimization: finding the values which make a function as large or as small as possible.

Each chapter has a number of exercises which are at varying degrees of difficulty. Solutions to selected exercises can be found on the web at `www.stats.uwo.ca/faculty/braun/statprog!`.

1.3 The R package

This book uses R, which is an open-source package for statistical computing. "Open source" has a number of different meanings; here the important one is that R is freely available, and its users are free to see how it is written, and to improve it. R is based on the computer language S, developed by John Chambers and others at Bell Laboratories in 1976. In 1993 Robert Gentleman and Ross Ihaka at the University of Auckland wanted to experiment with the language, so they developed an implementation, and named it R. They made it open source in 1995, and hundreds of people around the world have contributed to its development.

S-PLUS is a commercial implementation of the S language. Because both R and S-PLUS are based on the S language, much of what is described in what follows will apply without change to S-PLUS.

1.4 Why use a command line?

The R system is mainly command-driven, with the user typing in text and asking R to execute it. Nowadays most programs use interactive graphical user interfaces (menus, etc.) instead. So why did we choose such an old-fashioned way of doing things?

Menu-based interfaces are very convenient when applied to a limited set of commands, from a few to one or two hundred. However, a command-line interface is open ended. As we will show in this book, if you want to program a computer to do something that no one has done before, you can easily do it by breaking down the task into the parts that make it up, and then building up a program to carry it out. This may be possible in some menu-driven interfaces, but it is much easier in a command-driven interface.

Moreover, learning how to use one command line interface will give you skills that carry over to others, and may even give you some insight into how a menu-driven interface is implemented. As statisticians it is our belief that your goal should be understanding, and learning how to program at a command line will give you that at a fundamental level. Learning to use a menu-based program makes you dependent on the particular organization of that program.

There is a fairly rich menu-driven interface to R available in the `Rcmdr` package.[1] After you have worked through this book, if you come upon a statistical task that you don't know how to start, you may find that the menus in `Rcmdr` give you an idea of what methods are available.

[1] A package is a collection of functions and programs that can be used within R.

1.5 Font conventions

This book describes how to do computations in R. As we will see in the next chapter, this requires that the user types input, and R responds with text or graphs as output. To indicate the difference, we have typeset the user input in a slanted typewriter font, and text output in an upright version of the same font. For example,

```
> This was typed by the user
This is a response from R
```

In most cases other than this one and certain exercises, we will show the actual response from R.[2]

There are also situations where the code is purely illustrative and is not meant to be executed. (Many of those are not correct R code at all; others illustrate the syntax of R code in a general way.) In these situations we have typeset the code examples in an upright typewriter font. For example,

```
f( some arguments )
```

1.6 Installation of R

R can be downloaded from `http://cran.r-project.org!`. Most users should download and install a *binary version*. This is a version that has been translated (by *compilers*) into machine language for execution on a particular type of computer with a particular operating system. R is designed to be very *portable*: it will run on Microsoft Windows, Linux, Solaris, Mac OSX, and other operating systems, but different binary versions are required for each. In this book most of what we do would be the same on any system, but when we write system-specific instructions, we will assume that readers are using Microsoft Windows.

Installation on Microsoft Windows is straightforward. A binary version is available for Windows 98 or above from the web page `http://cran.r-project.org/bin/windows/base`.
Download the "setup program," a file with a name like `R-2.5.1-win32.exe`. Clicking on this file will start an almost automatic installation of the R system. Though it is possible to customize the installation, the default responses will lead to a satisfactory installation in most situations, particularly for beginning users.

One of the default settings of the installation procedure is to create an R icon on your computer's desktop.

Once you have installed R, you will be ready to start statistical programming. Let's learn how.

[2] We have used the `Sweave` package so that R itself is computing the output. The computations in the text were done with a pre-release version of R 2.5.0.

2

Introduction to the R language

Having installed the R system, you are now ready to begin to learn the art of statistical programming. The first step is to learn the *syntax* of the language that you will be programming in; you need to know the rules of the language. This chapter will give you an introduction to the syntax of R.

2.1 Starting and quitting R

In Microsoft Windows, the R installer will have created a Start Menu item and an icon for R on your desktop. Double clicking on the R icon starts the program.[1] The first thing that will happen is that R will open the *console*, into which the user can type commands.

The greater-than sign (>) is the prompt symbol. When this appears, you can begin typing commands.

For example, R can be used as a calculator. We can type simple arithmetical expressions at the > prompt:

```
> 5 + 49
```

Upon pressing the Enter key, the result `54` appears, prefixed by the number 1 in square brackets:

```
> 5 + 49
[1] 54
```

The `[1]` indicates that this is the first (and in this case only) result from the command. Other commands return multiple values, and each line of results will be labeled to aid the user in deciphering the output. For example, the sequence of integers from 1 to 20 may be displayed as follows:

```
> options(width=40)
> 1:20
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
[13] 13 14 15 16 17 18 19 20
```

[1] Other systems may install an icon to click, or may require you to type "R" at a command prompt.

The first line starts with the first return value, so is labeled `[1]`; the second line starts with the 13th, so is labeled `[13]`.[2]

Anything that can be computed on a pocket calculator can be computed at the R prompt. Here are some additional examples:

```
> #  "*" is the symbol for multiplication.
> # Everything following a # sign is assumed to be a
> # comment and is ignored by R.
> 3 * 5
[1] 15
> 3 - 8
[1] -5
> 12 / 4
[1] 3
```

To quit your R session, type

```
> q()
```

If you then hit the Enter key, you will be asked whether to save an image of the current workspace, or not, or to cancel. The workspace image contains a record of the computations you've done, and may contain some saved results. Hitting the Cancel option allows you to continue your current R session. We rarely save the current workspace image, but occasionally find it convenient to do so.

Note what happens if you omit the parentheses `()` when attempting to quit:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

This has happened because `q` is a *function* that is used to tell R to quit. Typing `q` by itself tells R to show us the (not very pleasant-looking) contents of the function `q`. By typing `q()`, we are telling R to *call* the function `q`. The action of this function is to quit R. *Everything* that R does is done through calls to functions, though sometimes those calls are hidden (as when we click on menus), or very basic (as when we call the multiplication function to multiply 3 times 5).

2.1.1 Recording your work

Rather than saving the workspace, we prefer to keep a record of the commands we entered, so that we can reproduce the workspace at a later date. In Windows, the easiest way to do this is to enter commands in R's script editor, available from the `File` menu. Commands are executed by highlighting them and hitting `Ctrl-R` (which stands for "run"). At the end of a session, save the final script for a permanent record of your work. In other systems a text editor and some form of cut and paste serve the same purpose.

[2] The position of the line break shown here depends on the optional setting `options(width=40)`. Other choices of line widths would break in different places.

2.2 Basic features of R

2.2.1 Calculating with R

At its most basic level, R can be viewed as a fancy calculator. We saw in the previous section that it can be used to do scalar arithmetic. The basic operations are + (add), - (subtract), * (multiply), and / (divide).

It can also be used to compute powers with the ^ operator. For example,

```
> 3^4
[1] 81
```

Modular arithmetic is also available. For example, we can compute the remainder after division of 31 by 7, i.e. 31 (mod 7):

```
> 31 %% 7
[1] 3
```

and the integer part of a fraction as

```
> 31 %/% 7
[1] 4
```

We can confirm that 31 is the sum of its remainder plus seven times the integer part of the fraction:

```
> 7 * 4 + 3
[1] 31
```

2.2.2 Named storage

R has a workspace known as the *global environment* that can be used to store the results of calculations, and many other types of objects. For a first example, suppose we would like to store the result of the calculation `1.0025^30` for future use. (This number arises out of a compound interest calculation based on an interest rate of 0.25% per year and a 30-year period.) We will assign this value to an object called `interest.30`. To this, we type

```
> interest.30 <- 1.0025^30
>
```

We tell R to make the assignment using an arrow that points to the left, created with the less-than sign (<) and the hyphen (-). R also supports using the equals sign (=) in place of the arrow in most circumstances, but we recommend using the arrow, as it makes clear that we are requesting an *action* (i.e. an assignment), rather than stating a *relation* (i.e. that `interest.30` is equal to `1.0025^30`) or making a permanent definition. Note that when we hit Enter, nothing appears on the screen except a new prompt: R has done what we asked, and is waiting for us to ask for something else.

We can see the results of this assignment by typing the name of our new object at the prompt:

```
> interest.30
[1] 1.077783
```

Think of this as just another calculation: R is calculating the result of the expression `interest.30`, and printing it. We can also use `interest.30` in further calculations if we wish. For example, we can calculate the bank balance after 30 years at 0.25% annual interest, if we start with an initial balance of $3000:

```
> initial.balance <- 3000
> final.balance <- initial.balance * interest.30
> final.balance
[1] 3233.35
```

Example 2.1
An individual wishes to take out a loan, today, of P at a monthly interest rate i. The loan is to be paid back in n monthly installments of size R, beginning one month from now. The problem is to calculate R.

Equating the present value P to the future (discounted) value of the n monthly payments R, we have

$$P = R(1+i)^{-1} + R(1+i)^{-2} + \cdots R(1+i)^{-n}$$

or

$$P = R\sum_{j=1}^{n}(1+i)^{-j}.$$

Summing this geometric series and simplifying, we obtain

$$P = R\left(\frac{1-(1+i)^{-n}}{i}\right).$$

This is the formula for the present value of an annuity. We can find R, given P, n and i as

$$R = P\frac{i}{1-(1+i)^{-n}}.$$

In R, we define variables as follows: `principal` to hold the value of P, and `intRate` to hold the interest rate, and `n` to hold the number of payments. We will assign the resulting payment value to an object called `payment`.

Of course, we need some numerical values to work with, so we will suppose that the loan amount is $1500, the interest rate is 1% and the number of payments is 10. The required code is then

```
> intRate <- 0.01
> n <- 10
> principal <- 1500
> payment <- principal * intRate / (1 - (1 + intRate)^(-n))
```

```
> payment
[1] 158.3731
```

For this particular loan, the monthly payments are $158.37.

2.2.3 Functions

Most of the work in R is done through *functions*. For example, we saw that to quit R we type `q()`. This tells R to *call* the function named `q`. The parentheses surround the *argument list*, which in this case contains nothing: we just want R to quit, and do not need to tell it how.

We also saw that `q` is defined as

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

This shows that `q` is a function that has three *arguments*: `save`, `status`, and `runLast`. Each of those has a *default value*: `"default"`, `0`, and `TRUE`, respectively. What happens when we execute `q()` is that R calls the `q` function with the arguments set to their default values.

If we want to change the default values, we specify them when we call the function. Arguments are identified in the call by their position, or by specifying the name explicitly. For example, both

```
q("no")
q(save = "no")
```

tell R to call `q` with the first argument set to `"no"`, i.e. to quit without saving the workspace. If we had given two arguments without names, they would apply to `save` and `status`. If we want to accept the defaults of the early parameters but change later ones, we give the name when calling the function, e.g.

```
q(runLast = FALSE)
```

or use commas to mark the missing arguments, e.g.

```
q( , , FALSE)
```

It is a good idea to use named arguments when calling a function which has many arguments or when using uncommon arguments, because it reduces the risk of specifying the wrong argument, and makes your code easier to read.

2.2.4 Exact or approximate?

One important distinction in computing is between exact and approximate results. Most of what we do in this book is aimed at approximate methods. It is possible in a computer to represent any rational number exactly, but it is more common to use approximate representations: usually *floating point representations*. These are a binary (base-two) variation on scientific

notation. For example, we might write a number to four significant digits in scientific notation as 6.926×10^{-4}. This representation of a number could represent any true value between 0.000 692 55 and 0.000 692 65. Standard floating point representations on computers are similar, except that a power of 2 would be used rather than a power of 10, and the fraction would be written in binary notation. The number above would be written as $1.011_2 \times 2^{-11}$ if four binary digit precision was used. The subscript 2 in the mantissa 1.011_2 indicates that this number is shown in base 2; that is, it represents $1\times2^0+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}$, or 1.375 in decimal notation.

However, 6.926×10^{-4} and $1.011_2\times2^{-11}$ are not identical. Four binary digits give less precision than four decimal digits: a range of values from approximately 0.000 641 to 0.000 702 would all get the same representation to four binary digit precision. In fact, 6.926×10^{-4} *cannot* be represented exactly in binary notation in a finite number of digits. The problem is similar to trying to represent 1/3 as a decimal: 0.3333 is a close approximation, but is not exact. The standard precision in R is 53 binary digits, which is equivalent to about 15 or 16 decimal digits.

To illustrate, consider the fractions 5/4 and 4/5. In decimal notation these can be represented exactly as 1.25 and 0.8 respectively. In binary notation 5/4 is $1 + 1/4 = 1.01_2$. How do we determine the binary representation of 4/5? It is between 0 and 1, so we'd expect something of the form $0.b_1b_2b_3\cdots$, where each b_i represents a "bit," i.e. a 0 or 1 digit. Multiplying by 2 moves the all bits left by one, i.e. $2\times4/5 = 1.6 = b_1.b_2b_3\cdots$. Thus $b_1 = 1$, and $0.6 = 0.b_2b_3\cdots$.

We can now multiply by 2 again to find $2\times0.6 = 1.2 = b_2.b_3\cdots$, so $b_2 = 1$. Repeating twice more yields $b_3 = b_4 = 0$. (Try it!)

At this point we'll have the number 0.8 again, so the sequence of 4 bits will repeat indefinitely: in base 2, 4/5 is $0.110\,011\,001\,100\cdots$. Since R only stores 53 bits, it won't be able to store 0.8 exactly. Some rounding error will occur in the storage.

We can observe the rounding error with the following experiment. With exact arithmetic, $(5/4)\times(4/5) = 1$, so $(5/4)\times(n\times4/5)$ should be exactly n for any value of n. But if we try this calculation in R, we find

```
> n <- 1:10
> 1.25 * (n * 0.8) - n

[1] 0.000000e+00 0.000000e+00 4.440892e-16 0.000000e+00 0.000000e+00
[6] 8.881784e-16 8.881784e-16 0.000000e+00 0.000000e+00 0.000000e+00
```

i.e. it is equal for some values, but not equal for $n = 3$, 6, or 7. The errors are very small, but nonzero.

Rounding error tends to accumulate in most calculations, so usually a long series of calculations will result in larger errors than a short one. Some operations are particularly prone to rounding error: for example, subtraction of two nearly equal numbers, or (equivalently) addition of two numbers with nearly the same magnitude but opposite signs. Since the leading bits in the binary expansions of nearly equal numbers will match, they will cancel in subtraction, and the result will depend on what is stored in the later bits.