2

The R environment

This chapter collects some practical aspects of working with R. It describes issues regarding the structure of the workspace, graphical devices and their parameters, and elementary programming, and includes a fairly extensive, although far from complete, discussion of data entry.

2.1 Session management

2.1.1 The workspace

All variables created in R are stored in a common workspace. To see which variables are defined in the workspace, you can use the function `ls` (*ls*t). It should look as follows if you have run all the examples in the preceding chapter:

```
> ls()
 [1] "bmi"           "d"             "exp.lean"
 [4] "exp.obese"     "fpain"         "height"
 [7] "hh"            "intake.post"   "intake.pre"
[10] "intake.sorted" "l"             "m"
[13] "mylist"        "o"             "oops"
[16] "pain"          "sel"           "v"
[19] "weight"        "x"             "xbar"
[22] "y"
```

P. Dalgaard, *Introductory Statistics with R*,
DOI: 10.1007/978-0-387-79054-1_2,

Remember that you cannot omit the parentheses in `ls()`.

If at some point things begin to look messy, you can delete some of the objects. This is done using `rm` (*rem*ove), so that

```
> rm(height, weight)
```

deletes the variables `height` and `weight`.

The entire workspace can be cleared using `rm(list=ls())` and also via the "Remove all objects" or "Clear Workspace" menu entries in the Windows and Macintosh GUIs. This does not remove variables whose name begins with a dot because they are not listed by `ls()` — you would need `ls(all=T)` for that, but it could be dangerous because such names are used for system purposes.

If you are acquainted with the Unix operating system, for which the S language, which preceded R, was originally written, then you will know that the commands for listing and removing files in Unix are called precisely `ls` and `rm`.

It is possible to save the workspace to a file at any time. If you just write

```
save.image()
```

then it will be saved to a file called `.RData` in your working directory. The Windows version also has this on the File menu. When you exit R, you will be asked whether to save the workspace image; if you accept, the same thing will happen. It is also possible to specify an alternative filename (within quotes). You can also save selected objects with `save`. The `.RData` file is loaded by default when R is started in its directory. Other save files can be loaded into your current workspace using `load`.

2.1.2 *Textual output*

It is important to note that the workspace consists only of R objects, not of any of the output that you have generated during a session. If you want to save your output, use "Save to File" from the File menu in Windows or use standard cut-and-paste facilities. You can also use ESS (Emacs Speaks Statistics), which works on all platforms. It is a "mode" for the Emacs editor where you can run your entire session in an Emacs buffer. You can get ESS and installation instructions for it from CRAN (see Appendix A).

An alternative way of diverting output to a file is to use the `sink` function. This is largely a relic from the days of the 80×25 computer terminal, where cut-and-paste techniques were not available, but it can still be use-

ful at times. In particular, it can be used in batch processing. The way it works is as follows:

```
> sink("myfile")
> ls()
```

No output appears! This is because the output goes into the file `myfile` in the current directory. The system will remain in a state where commands are processed, but the output (apparently) goes into the drain until the normal state of affairs is reestablished by

```
> sink()
```

The current working directory can be obtained by `getwd()` and changed by `setwd(mydir)`, where `mydir` is a character string. The initial working directory is system-dependent; for instance, the Windows GUI sets it to the user's home directory, and command line versions use the directory from which you start R.

2.1.3 *Scripting*

Beyond a certain level of complexity, you will not want to work with R on a line-by-line basis. For instance, if you have entered an 8×8 matrix over eight lines and realize that you made a mistake, you will find yourself using the up-arrow key 64 times to reenter it! In such cases, it is better to work with R *scripts*, collections of lines of R code stored either in a file or in computer memory somehow.

One option is to use the `source` function, which is sort of the opposite of `sink`. It takes the input (i.e., the commands from a file) and runs them. Notice, though, that the entire file is syntax-checked before anything is executed. It is often useful to set `echo=T` in the call so that commands are printed along with the output.

Another option is more interactive in nature. You can work with a script editor window, which allows you to submit one or more lines of the script to a running R, which will then behave as if the same lines had been entered at the prompt. The Windows and Macintosh versions of R have simple scripting windows built-in, and a number of text editors also have features for sending commands to R; popular choices on Windows include TINN-R and WinEdt. This is also available as part of ESS (see the preceding section).

The history of commands entered in a session can be saved and reloaded using the `savehistory` and `loadhistory` commands, which are also mapped to menu entries in Windows. Saved histories can be useful as a

starting point for writing scripts; notice also that the `history()` function will show the last commands entered at the console (up to a maximum of 25 lines by default).

2.1.4 *Getting help*

R can do a lot more than what a typical beginner can be expected to need or even understand. This book is written so that most of the code you are likely to need in relation to the statistical procedures is described in the text, and the compendium in Appendix C is designed to provide a basic overview. However, it is obviously not possible to cover everything.

R also comes with extensive online help in text form as well as in the form of a series of HTML files that can be read using a Web browser such as Netscape or Internet Explorer. The help pages can be accessed via "help" in the menu bar on Windows and by entering `help.start()` on any platform. You will find that the pages are of a technical nature. Precision and conciseness here take precedence over readability and pedagogy (something one learns to appreciate after exposure to the opposite).

From the command line, you can always enter `help(aggregate)` to get help on the `aggregate` function or use the prefix form `?aggregate`. If the HTML viewer is running, then the help page is shown there. Otherwise it is shown as text either through a pager to the terminal window or in a separate window.

Notice that the HTML version of the help system features a very useful "Search Engine and Keywords" and that the `apropos` function allows you to get a list of command names that contain a given pattern. The function `help.search` is similar but uses fuzzy matching and searches deeper into the help pages, so that it will be able to locate, for example, Kendall's correlation coefficient in `cor.test` if you use `help.search("kendal")`.

Also available with the R distributions is a set of documents in various formats. Of particular interest is "An Introduction to R", originally based on a set of notes for S-PLUS by Bill Venables and David Smith and modified for R by various people. It contains an introduction to the R language and environment in a rather more language-centric fashion than this book. On the Windows platform, you can choose to install PDF documents as part of the installation procedure so that — provided the Adobe Acrobat Reader program is also installed — it can be accessed via the Help menu. An HTML version (without pictures) can be accessed via the browser interface on all platforms.

2.1.5 Packages

An R installation contains one or more libraries of packages. Some of these packages are part of the basic installation. Others can be downloaded from CRAN (see Appendix A), which currently hosts over 1000 packages for various purposes. You can even create your own packages.

A library is generally just a folder on your disk. A system library is created when R is installed. In some installations, users may be prohibited from modifying the system library. It is possible to set up private user libraries; see `help(".Library")` for details.

A package can contain functions written in the R language, dynamically loaded libraries of compiled code (written in C or Fortran mostly), and data sets. It generally implements functionality that most users will probably not need to have loaded all the time. A package is loaded into R using the `library` command, so to load the `survival` package you should enter

```
> library(survival)
```

The loaded packages are not considered part of the user workspace. If you terminate your R session and start a new session with the saved workspace, then you will have to load the packages again. For the same reason, it is rarely necessary to remove a package that you have loaded, but it can be done if desired with

```
> detach("package:survival")
```

(see also Section 2.1.7).

2.1.6 Built-in data

Many packages, both inside and outside the standard R distribution, come with built-in data sets. Such data sets can be rather large, so it is not a good idea to keep them all in computer memory at all times. A mechanism for on-demand loading is required. In many packages, this works via a mechanism called *lazy loading*, which allows the system to "pretend" that the data are in memory, but in fact they are not loaded until they are referenced for the first time.

With this mechanism, data are "just there". For example, if you type "thuesen", the data frame of that name is displayed. Some packages still require explicit calls to the `data` function. Most often, this loads a data frame with the name that its argument specifies; `data(thuesen)` will, for instance, load the `thuesen` data frame.

What `data` does is to go through the data directories associated with each package (see Section 2.1.5) and look for files whose basename matches the given name. Depending on the file extension, several things can then happen. Files with a `.tab` extension are read using `read.table` (Section 2.4), whereas files with a `.R` extension are executed as source files (and could, in general, do anything!), to give two common examples.

If there is a subdirectory of the current directory called `data`, then it is searched as well. This can be quite a handy way of organizing your personal projects.

2.1.7 *attach and detach*

The notation for accessing variables in data frames gets rather heavy if you repeatedly have to write longish commands like

```
plot(thuesen$blood.glucose,thuesen$short.velocity)
```

Fortunately, you can make R look for objects among the variables in a given data frame, for example `thuesen`. You write

```
> attach(thuesen)
```

and then `thuesen`'s data are available without the clumsy `$`-notation:

```
> blood.glucose
 [1] 15.3 10.8  8.1 19.5  7.2  5.3  9.3 11.1  7.5 12.2  6.7  5.2
[13] 19.0 15.1  6.7  8.6  4.2 10.3 12.5 16.1 13.3  4.9  8.8  9.5
```

What happens is that the data frame `thuesen` is placed in the system's *search path*. You can view the search path with `search`:

```
> search()
 [1] ".GlobalEnv"        "thuesen"           "package:ISwR"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

Notice that `thuesen` is placed as no. 2 in the search path. `.GlobalEnv` is the workspace and `package:base` is the system library where all standard functions are defined. `Autoloads` is not described here. `package:stats` and onwards contains the basic statistical routines such as the Wilcoxon test, and the other packages similarly contain various functions and data sets. (The package system is modular, and you can run R with a minimal set of packages for specific uses.) Finally, `package:ISwR` contains the data sets used for this book.

There may be several objects of the same name in different parts of the search path. In that case, R chooses the first one (that is, it searches first in `.GlobalEnv`, then in `thuesen`, and so forth). For this reason, you need to be a little careful with "loose" objects that are defined in the workspace outside a data frame since they will be used before any vectors and factors of the same name in an attached data frame. For the same reason, it is not a good idea to give a data frame the same name as one of the variables inside it. Note also that changing a data frame after attaching it will not affect the variables available since `attach` involves a (virtual) copy operation of the data frame.

It is not possible to attach data frames in front of `.GlobalEnv` or following `package:base`. However, it is possible to attach more than one data frame. New data frames are inserted into position 2 by default, and everything except `.GlobalEnv` moves one step to the right. It is, however, possible to specify that a data frame should be searched before `.GlobalEnv` by using constructions of the form

```
with(thuesen, plot(blood.glucose, short.velocity))
```

In some contexts, R uses a slightly different method when looking for objects. If looking for a variable of a specific type (usually a function), R will skip those of other types. This is what saves you from the worst consequences of accidentally naming a variable (say) `c`, even though there is a system function of the same name.

You can remove a data frame from the search path with `detach`. If no arguments are given, the data frame in position 2 is removed, which is generally what is desired. `.GlobalEnv` and `package:base` cannot be `detach`'ed.

```
> detach()
> search()
 [1] ".GlobalEnv"        "package:ISwR"      "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

2.1.8 `subset`, `transform`, and `within`

You can attach a data frame to avoid the cumbersome indexing of every variable inside of it. However, this is less helpful for selecting subsets of data and for creating new data frames with transformed variables. A couple of functions exist to make these operations easier. They are used as follows:

```
> thue2 <- subset(thuesen,blood.glucose<7)
> thue2
   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
> thue3 <- transform(thuesen,log.gluc=log(blood.glucose))
> thue3
   blood.glucose short.velocity log.gluc
1           15.3           1.76 2.727853
2           10.8           1.34 2.379546
3            8.1           1.27 2.091864
4           19.5           1.47 2.970414
5            7.2           1.27 1.974081
...
22           4.9           1.03 1.589235
23           8.8           1.12 2.174752
24           9.5           1.70 2.251292
```

Notice that the variables used in the expressions for new variables or for subsetting are evaluated with variables taken from the data frame.

`subset` also works on single vectors. This is nearly the same as indexing with a logical vector (such as `short.velocity[blood.glucose<7]`), except that observations with missing values in the selection criterion are excluded.

`subset` also has a `select` argument which can be used to extract variables from the data frame. We shall return to this in Section 10.3.1.

The `transform` function has a couple of drawbacks, the most serious of which is probably that it does not allow chained calculations where some of the new variables depend on the others. The = signs in the syntax are not assignments, but indicate names, which are assigned to the computed vectors in the last step.

An alternative to `transform` is the `within` function, which can be used like this:

```
> thue4 <- within(thuesen,{
+    log.gluc <- log(blood.glucose)
+    m <- mean(log.gluc)
+    centered.log.gluc <- log.gluc - m
+    rm(m)
+ })
> thue4
   blood.glucose short.velocity centered.log.gluc log.gluc
1           15.3           1.76       0.481879807 2.727853
2           10.8           1.34       0.133573113 2.379546
```

```
3            8.1          1.27      -0.154108960 2.091864
4           19.5          1.47       0.724441444 2.970414
5            7.2          1.27      -0.271891996 1.974081
...
22           4.9          1.03      -0.656737817 1.589235
23           8.8          1.12      -0.071221300 2.174752
24           9.5          1.70       0.005318777 2.251292
```

Notice that the second argument is an arbitrary expression (here a *compound* expression, see p. 45). The function is similar to `with`, but instead of just returning the computed value, it collects all new and modified variables into a modified data frame, which is then returned. As shown, variables containing intermediate results can be discarded with `rm`. (It is particularly important to do this if the contents are incompatible with the data frame.)

2.2 The graphics subsystem

In Section 1.1.5, we saw how to generate a simple plot and superimpose a curve on it. It is quite common in statistical graphics for you to want to create a plot that is slightly different from the default: Sometimes you will want to add annotation, sometimes you want the axes to be different — labels instead of numbers, irregular placement of tick marks, etc. All these things can be obtained in R. The methods for doing them may feel slightly unusual at first, but offers a very flexible and powerful approach.

In this section, we look deeper into the structure of a typical plot and give some indication of how you can work with plots to achieve your desired results. Beware, though, that this is a large and complex area and it is not within the scope of this book to cover it completely. In fact, we completely ignore important newer tools in the `grid` and `lattice` packages.

2.2.1 Plot layout

In the graphics model that R uses, there is (for a single plot) a figure region containing a central plotting region surrounded by margins. Coordinates inside the plotting region are specified in data units (the kind generally used to label the axes). Coordinates in the margins are specified in *lines of text* as you move in a direction perpendicular to a side of the plotting region but in data units as you move along the side. This is useful since you generally want to put text in the margins of a plot.

A standard x–y plot has an x and a y title label generated from the expressions being plotted. You may, however, override these labels and also

add two further titles, a main title above the plot and a subtitle at the very bottom, in the plot call.

```
> x <- runif(50,0,2)
> y <- runif(50,0,2)
> plot(x, y, main="Main title", sub="subtitle",
+      xlab="x-label", ylab="y-label")
```

Inside the plotting region, you can place points and lines that are either specified in the `plot` call or added later with `points` and `lines`. You can also place a text with

```
> text(0.6,0.6,"text at (0.6,0.6)")
> abline(h=.6,v=.6)
```

Here, the `abline` call is just to show how the text is centered on the point $(0.6, 0.6)$. (Normally, `abline` plots the line $y = a + bx$ when given `a` and `b` as arguments, but it can also be used to draw horizontal and vertical lines as shown.)

The margin coordinates are used by the `mtext` function. They can be demonstrated as follows:

```
> for (side in 1:4) mtext(-1:4,side=side,at=.7,line=-1:4)
> mtext(paste("side",1:4), side=1:4, line=-1,font=2)
```

The `for` loop (see Section 2.3.1) places the numbers -1 to 4 on corresponding lines in each of the four margins at an off-center position of 0.7 measured in user coordinates. The subsequent call places a label on each side, giving the side number. The argument `font=2` means that a boldface font is used. Notice in Figure 2.1 that not all the margins are wide enough to hold all the numbers and that it is possible to use negative line numbers to place text within the plotting region.

2.2.2 *Building a plot from pieces*

High-level plots are composed of elements, each of which can also be drawn separately. The separate drawing commands often allow finer control of the element, so a standard strategy to achieve a given effect is first to draw the plot without that element and add the element subsequently. As an extreme case, the following command will plot absolutely nothing:

```
> plot(x, y, type="n", xlab="", ylab="", axes=F)
```

Here `type="n"` causes the points not to be drawn. `axes=F` suppresses the axes and the box around the plot, and the x and y title labels are set to empty strings.

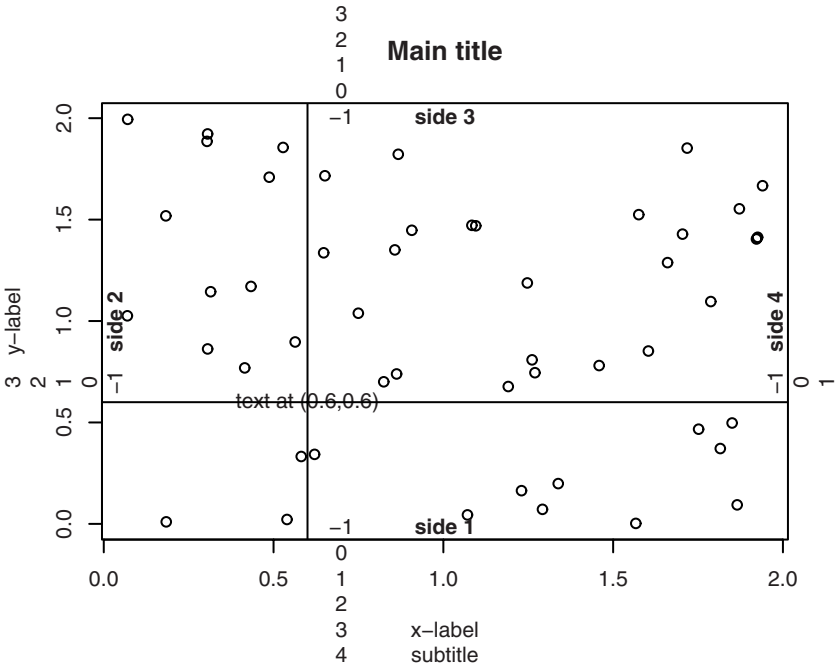

Figure 2.1. The layout of a standard plot.

However, the fact that nothing is plotted does not mean that nothing happened. The command sets up the plotting region and coordinate systems just as if it had actually plotted the data. To add the plot elements, evaluate the following:

```
> points(x,y)
> axis(1)
> axis(2,at=seq(0.2,1.8,0.2))
> box()
> title(main="Main title", sub="subtitle",
+     xlab="x-label", ylab="y-label")
```

Notice how the second `axis` call specifies an alternative set of tick marks (and labels). This is a common technique used to create special axes on a plot and might also be used to create nonequidistant axes as well as axes with nonnumeric labelling.

Plotting with `type="n"` is sometimes a useful technique because it has the side effect of dimensioning the plot area. For instance, to create a plot with different colours for different groups, you could first plot all data with `type="n"`, ensuring that the plot region is large enough, and then

add the points for each group using `points`. (Passing a vector argument for `col` is more expedient in this particular case.)

2.2.3 *Using `par`*

The `par` function allows incredibly fine control over the details of a plot, although it can be quite confusing to the beginner (and even to experienced users at times). The best strategy for learning it may well be simply to try and pick up a few useful tricks at a time and once in a while try to solve a particular problem by poring over the help page.

Some of the parameters, but not all, can also be set via arguments to plotting functions, which also have some arguments that cannot be set by `par`. When a parameter can be set by both methods, the difference is generally that if something is set via `par`, then it stays set subsequently.

The `par` settings allow you to control line width and type, character size and font, colour, style of axis calculation, size of the plot and figure regions, clipping, etc. It is possible to divide a figure into several subfigures by using the `mfrow` and `mfcol` parameters.

For instance, the default margin sizes are just over 5, 4, 4, and 2 lines. You might set `par(mar=c(4,4,2,2)+0.1)` before plotting. This shaves one line off the bottom margin and two lines off the top margin of the plot, which will reduce the amount of unused whitespace when there is no main title or subtitle. If you look carefully, you will in fact notice that Figure 2.1 has a somewhat smaller plotting region than the other plots in this book. This is because the other plots have been made with reduced margins for typesetting reasons.

However, it is quite pointless to describe the graphics parameters completely at this point. Instead, we return to them as they are used for specific plots.

2.2.4 *Combining plots*

Some special considerations arise when you wish to put several elements together in the same plot. Consider overlaying a histogram with a normal density (see Sections 4.2 and 4.4.1 for information on histograms and Section 3.5.1 for density). The following is close, but only nearly good enough (figure not shown).

```
> x <- rnorm(100)
> hist(x,freq=F)
> curve(dnorm(x),add=T)
```

The `freq=F` argument to `hist` ensures that the histogram is in terms of densities rather than absolute counts. The `curve` function graphs an expression (in terms of `x`) and its `add=T` allows it to overplot an existing plot. So things are generally set up correctly, but sometimes the top of the density function gets chopped off. The reason is of course that the height of the normal density played no role in the setting of the *y*-axis for the histogram. It will not help to reverse the order and draw the curve first and add the histogram because then the highest bars might get clipped.

The solution is first to get hold of the magnitude of the *y* values for both plot elements and make the plot big enough to hold both (Figure 2.2):

```
> h <- hist(x, plot=F)
> ylim <- range(0, h$density, dnorm(0))
> hist(x, freq=F, ylim=ylim)
> curve(dnorm(x), add=T)
```

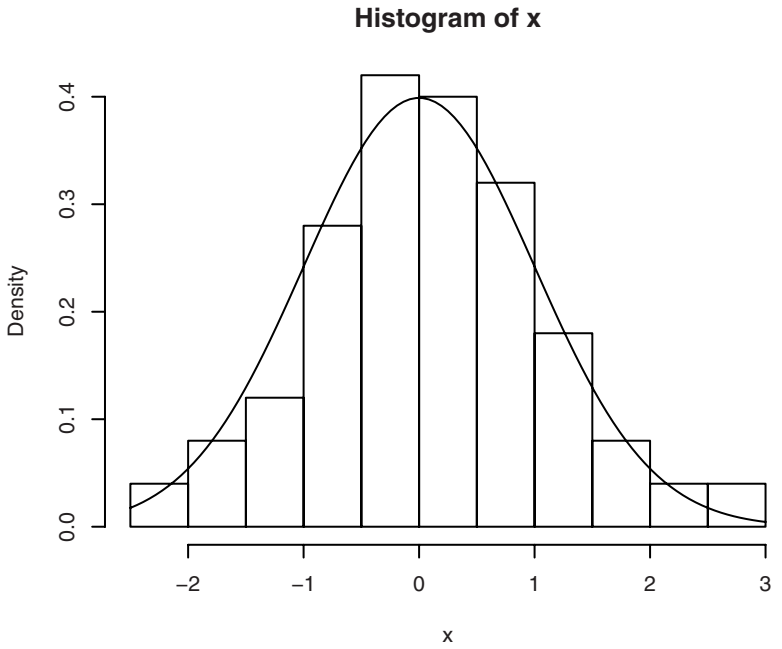

Figure 2.2. Histogram with normal density overlaid.

When called with `plot=F`, `hist` will not plot anything, but it will return a structure containing the bar heights on the density scale. This and the fact that the maximum of `dnorm(x)` is `dnorm(0)` allows us to calculate a range covering both the bars and the normal density. The zero in

the `range` call ensures that the bottom of the bars will be in range, too. The range of y values is then passed to the `hist` function via the `ylim` argument.

2.3 R programming

It is possible to write your own R functions. In fact, this is a major aspect and attraction of working with the system in the long run. This book largely avoids the issue in favour of covering a larger set of basic statistical procedures that can be executed from the command line. However, to give you a feel for what can be done, consider the following function, which wraps the code from the example of Section 2.2.4 so that you can just say `hist.with.normal(rnorm(200))`. It has been slightly extended so that it now uses the empirical mean and standard deviation of the data instead of just 0 and 1.

```
> hist.with.normal <- function(x, xlab=deparse(substitute(x)),...)
+ {
+     h <- hist(x, plot=F, ...)
+     s <- sd(x)
+     m <- mean(x)
+     ylim <- range(0,h$density,dnorm(0,sd=s))
+     hist(x, freq=F, ylim=ylim, xlab=xlab, ...)
+     curve(dnorm(x,m,s), add=T)
+ }
```

Notice the use of a default argument for `xlab`. If `xlab` is not specified, then it is obtained from this expression, which evaluates to a character form of the expression given for `x`; that is, if you pass `rnorm(100)` for `x`, then the x label becomes "rnorm(100)". Notice also the use of a `...` argument, which collects any additional arguments and passes them on to `hist` in the two calls.

You can learn more about programming in R by studying the built-in functions, starting with simple ones like `log10` or `weighted.mean`.

2.3.1 Flow control

Until now, we have seen components of the R language that cause evaluation of single expressions. However, R is a true programming language that allows conditional execution and looping constructs as well. Consider, for instance, the following code. (The code implements a version of Newton's method for calculating the square root of `y`.)

```
> y <- 12345
> x <- y/2
> while (abs(x*x-y) > 1e-10) x <- (x + y/x)/2
> x
[1] 111.1081
> x^2
[1] 12345
```

Notice the `while(condition) expression` construction, which says that the expression should be evaluated as long as the condition is `TRUE`. The test occurs at the top of the loop, so the expression might never be evaluated.

A variation of the same algorithm with the test at the bottom of the loop can be written with a `repeat` construction:

```
> x <- y/2
> repeat{
+     x <- (x + y/x)/2
+     if (abs(x*x-y) < 1e-10) break
+ }
> x
[1] 111.1081
```

This also illustrates three other flow control structures: (a) a *compound expression*, several expressions held together between curly braces; (b) an `if` construction for conditional execution; and (c) a `break` expression, which causes the enclosing loop to exit.

Incidentally, the loop could allow for `y` being a vector simply by changing the termination condition to

```
if (all(abs(x*x - y) < 1e-10)) break
```

This would iterate excessively for some elements, but the vectorized arithmetic would likely more than make up for that.

However, the most frequently used looping construct is `for`, which loops over a fixed set of values as in the following example, which plots a set of power curves on the unit interval.

```
> x <- seq(0, 1,.05)
> plot(x, x, ylab="y", type="l")
> for ( j in 2:8 ) lines(x, x^j)
```

Notice the *loop variable* `j`, which in turn takes the values of the given sequence when used in the `lines` call.

2.3.2 Classes and generic functions

Object-oriented programming is about creating coherent systems of data and methods that work upon them. One purpose is to simplify programs by accommodating the fact that you will have conceptually similar methods for different types of data, even though the implementations will have to be different. A prototype example is the print method: It makes sense to print many kinds of data objects, but the print layout will depend on what the data object is. You will generally have a *class* of data objects and a *print method* for that class. There are several object-oriented languages implementing these ideas in different ways.

Most of the basic parts of R use the same object system as S version 3. An alternative object system similar to that of S version 4 has been developed in recent years. The new system has several advantages over the old one, but we shall restrict attention to the latter. The S3 object system is a simple system in which an object has a `class` attribute, which is simply a character vector. One example of this is that all the return values of the classical tests such as `t.test` have class `"htest"`, indicating that they are the result of a hypothesis test. When these objects are printed, it is done by `print.htest`, which creates the nice layout (see Chapter 5 for examples). However, from a programmatic viewpoint, these objects are just lists, and you can, for instance, extract the *p*-value by writing

```
> t.test(bmi, mu=22.5)$p.value
[1] 0.7442183
```

The function `print` is a *generic function*, one that acts differently depending on its argument. These generally look like this:

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

What `UseMethod("print")` means is that R should pass control to a function named according to the object class (`print.htest` for objects of class `"htest"`, etc.) or, if this is not found, to `print.default`. To see all the methods available for `print`, type `methods(print)` (there are 138 of them in R 2.6.2, so the output is not shown here).

2.4 Data entry

Data sets do not have to be very large before it becomes impractical to type them in with `c(...)`. Most of the examples in this book use data sets in-

cluded in the ISwR package, made available to you by library(ISwR). However, as soon as you wish to apply the methods to your own data, you will have to deal with data file formats and the specification thereof.

In this section we discuss how to read data files and how to use the data editor module in R. The text has some bias toward Windows systems, mainly because of some special issues that need to be mentioned for that platform.

2.4.1 *Reading from a text file*

The most convenient way of reading data into R is via the function called read.table. It requires that data be in "ASCII format"; that is, a "flat file" as created with Windows' NotePad or any plain-text editor. The result of read.table is a data frame, and it expects to find data in a corresponding layout where each line in the file contains all data from one subject (or rat or ...) in a specific order, separated by blanks or, optionally, some other separator. The first line of the file can contain a header giving the names of the variables, a practice that is highly recommended.

Table 11.6 in Altman (1991) contains an example on ventricular circumferential shortening velocity versus fasting blood glucose by Thuesen et al. We used those data to illustrate subsetting and use them again in the chapter on correlation and regression. They are among the built-in data sets in the ISwR package and available as the data frame thuesen, but the point here is to show how to read them from a plain-text file.

Assume that the data are contained in the file thuesen.txt, which looks as follows:

```
blood.glucose   short.velocity
15.3            1.76
10.8            1.34
8.1             1.27
19.5            1.47
7.2             1.27
5.3             1.49
9.3             1.31
11.1            1.09
7.5             1.18
12.2            1.22
6.7             1.25
5.2             1.19
19.0            1.95
15.1            1.28
6.7             1.52
8.6             NA
4.2             1.12
```

```
10.3            1.37
12.5            1.19
16.1            1.05
13.3            1.32
4.9             1.03
8.8             1.12
9.5             1.70
```

To enter the data into the file, you could start up Windows' NotePad or any other plain-text editor, such as those discussed in Section 2.1.3. Unix/Linux users should just use a standard editor, such as `emacs` or `vi`. If you must, you can even use a word processing program with a little care.

You should simply type in the data as shown. Notice that the columns are separated by an arbitrary number of blanks and that `NA` represents a missing value.

At the end, you should save the data to a text file. Notice that word processors require special actions in order to save as text. Their normal save format is difficult to read from other programs.

Assuming further that the file is in the `ISwR` folder on the `N:` drive, the data can be read using

```
> thuesen2 <- read.table("N:/ISwR/thuesen.txt",header=T)
```

Notice `header=T` specifying that the first line is a header containing the names of variables contained in the file. Also note that you use forward slashes (`/`), not backslashes (`\`), in the filename, even on a Windows system.

The reason for avoiding backslashes in Windows filenames is that the symbol is used as an escape character (see Section 1.2.4) and therefore needs to be doubled. You could have used `N:\\ISwR\\thuesen.txt`.

The result is a data frame, which is assigned to the variable `thuesen2` and looks as follows:

```
> thuesen2
   blood.glucose short.velocity
1           15.3           1.76
2           10.8           1.34
3            8.1           1.27
4           19.5           1.47
5            7.2           1.27
6            5.3           1.49
7            9.3           1.31
8           11.1           1.09
9            7.5           1.18
10          12.2           1.22
```

```
11           6.7           1.25
12           5.2           1.19
13          19.0           1.95
14          15.1           1.28
15           6.7           1.52
16           8.6             NA
17           4.2           1.12
18          10.3           1.37
19          12.5           1.19
20          16.1           1.05
21          13.3           1.32
22           4.9           1.03
23           8.8           1.12
24           9.5           1.70
```

To read in factor variables (see Section 1.2.8), the easiest way may be to encode them using a textual representation. The `read.table` function autodetects whether a vector is text or numeric and converts it to a factor in the former case (but makes no attempt to recognize numerically coded factors). For instance, the `secretin` built-in data set is read from a file that begins like this:

```
   gluc person time repl time20plus time.comb
1    92      A  pre    a        pre       pre
2    93      A  pre    b        pre       pre
3    84      A   20    a        20+        20
4    88      A   20    b        20+        20
5    88      A   30    a        20+       30+
6    90      A   30    b        20+       30+
7    86      A   60    a        20+       30+
8    89      A   60    b        20+       30+
9    87      A   90    a        20+       30+
10   90      A   90    b        20+       30+
11   85      B  pre    a        pre       pre
12   85      B  pre    b        pre       pre
13   74      B   20    a        20+        20
....
```

This file can be read directly by `read.table` with no arguments other than the filename. It will recognize the case where the first line is one item shorter than the rest and will interpret that layout to imply that the first line contains a header and the first value on all subsequent lines is a row label — that is, exactly the layout generated when printing a data frame.

Reading factors like this may be convenient, but there is a drawback: The level order is alphabetic, so for instance

```
> levels(secretin$time)
[1] "20"  "30"  "60"  "90"  "pre"
```

If this is not what you want, then you may have to manipulate the factor levels; see Section 10.1.2.

A technical note: The files referenced above are contained in the `ISwR` package in the subdirectory (folder) `rawdata`. Exactly where the file is located on your system will depend on where the `ISwR` package was installed. You can find this out as follows:

```
> system.file("rawdata", "thuesen.txt", package="ISwR")
[1] "/home/pd/Rlibrary/ISwR/rawdata/thuesen.txt"
```

2.4.2 *Further details on `read.table`*

The `read.table` function is a very flexible tool that is controlled by many options. We shall not attempt a full description here but just give some indication of what it can do.

File format details

We have already seen the use of `header=T`. A couple of other options control the detailed format of the input file:

Field separator. This can be specified using `sep`. Notice that when this is used, as opposed to the default use of whitespace, there must be exactly one separator between data fields. Two consecutive separators will imply that there is a missing value in between. Conversely, it is necessary to use specific codes to represent missing values in the default format and also to use some form of quoting for strings that contain embedded spaces.

NA strings. You can specify which strings represent missing values via `na.strings`. There can be several different strings, although not different strings for different columns. For print files from the SAS program, you would use `na.strings="."`.

Quotes and comments. By default, R-style quotes can be used to delimit character strings, and parts of files following the comment character `#` are ignored. These features can be modified or removed via the `quote` and `comment.char` arguments.

Unequal field count. It is normally considered an error if not all lines contain the same number of values (the first line can be one item short, as described above for the `secretin` data). The `fill` and `flush` arguments can be used in case lines vary in length.

Delimited file types

Applications such as spreadsheets and databases produce text files in formats that require multiple options to be adjusted. For such purposes, there exist "precooked" variants of `read.table`. Two of these are intended to handle CSV files and are called `read.csv` and `read.csv2`. The former assumes that fields are separated by a comma, and the latter assumes that they are separated by semicolons but use a comma as the decimal point (this format is often generated in European locales). Both formats have `header=T` as the default. Further variants are `read.delim` and `read.delim2` for reading delimited files (by default, Tab-delimited files).

Conversion of input

It can be desirable to override the default conversion mechanisms in `read.table`. By default, nonnumeric input is converted to factors, but it does not always make sense. For instance, names and addresses typically should not be converted. This can be modified either for all columns using `stringsAsFactors` or on a per-item basis using `as.is`.

Automatic conversion is often convenient, but it is inefficient in terms of computer time and storage; in order to read a numeric column, `read.table` first reads it as character data, checks whether all elements can be converted to numeric, and only then performs the conversion. The `colClasses` argument allows you to bypass the mechanism by explicitly specifying which columns are of which class (the standard classes `"character"`, `"numeric"`, etc., get special treatment). You can also skip unwanted columns by specifying `"NULL"` as the class.

2.4.3 The data editor

R lets you edit data frames using a spreadsheet-like interface. The interface is a bit rough but quite useful for small data sets.

To edit a data frame, you can use the `edit` function:

```
> aq <- edit(airquality)
```

This brings up a spreadsheet-like editor with a column for each variable in the data frame. The `airquality` data set is built into R; see `help(airquality)` for its contents. Inside the editor, you can move around with the mouse or the cursor keys and edit the current cell by typing in data. The type of variable can be switched between real (numeric) and character (factor) by clicking on the column header, and the name of

the variable can be changed similarly. Note that there is (as of R 2.6.2) no way to delete rows and columns and that new data can be entered only at the end.

When you close the data editor, the edited data frame is assigned to `aq`. The original `airquality` is left intact. Alternatively, if you do not mind overwriting the original data frame, you can use

```
> fix(aq)
```

This is equivalent to `aq <- edit(aq)`.

To enter data into a blank data frame, use

```
> dd <- data.frame()
> fix(dd)
```

An alternative would be `dd <- edit(data.frame())`, which works fine except that beginners tend to reexecute the command when they need to edit `dd`, which of course destroys all data. It is necessary in either case to start with an empty data frame since by default `edit` expects you to want to edit a user-defined function and would bring up a text editor if you started it as `edit()`.

2.4.4 Interfacing to other programs

Sometimes you will want to move data between R and other statistical packages or spreadsheets. A simple fallback approach is to request that the package in question export data as a text file of some sort and use `read.table`, `read.csv`, `read.csv2`, `read.delim`, or `read.delim2`, as previously described.

The `foreign` package is one of the packages labelled "recommended" and should therefore be available with binary distributions of R. It contains routines to read files in several formats, including those from SPSS (`.sav` format), SAS (export libraries), Epi-Info (`.rec`), Stata, Systat, Minitab, and some S-PLUS version 3 dump files.

Unix/Linux users sometimes find themselves with data sets written on Windows machines. The `foreign` package will work there as well for those formats that it supports. Notice that ordinary SAS data sets are not among the supported formats. These have to be converted to export libraries on the originating system. Data that have been entered into Microsoft Excel spreadsheets are most conveniently extracted using a compatible application such as OOo (OpenOffice.org).

An expedient technique is to read from the system clipboard. Say, highlight a rectangular region in a spreadsheet, press Ctrl-C (if on Windows), and inside R use

```
read.table("clipboard", header=T)
```

This does require a little caution, though. It may result in loss of accuracy since you only transfer the data as they appear on the screen. This is mostly a concern if you have data to many significant digits.

For data stored in databases, there exist a number of interface packages on CRAN. Of particular interest on Windows and with some Unix databases is the RODBC package because you can set up ODBC ("Open Database Connectivity") connections to data stored by common applications, including Excel and Access. Some Unix databases (e.g., PostgreSQL) also allow ODBC connections.

For up-to-date information on these matters, consult the "R Data Import/Export" manual that comes with the system.

2.5 Exercises

2.1 Describe how to insert a value between two elements of a vector at a given position by using the `append` function (use the help system to find out). Without `append`, how would you do it?

2.2 Write the built-in data set `thuesen` to a Tab-separated text file with `write.table`. View it with a text editor (depending on your system). Change the `NA` value to `.` (period), and read the changed file back into R with a suitable command. Also try importing the data into other applications of your choice and exporting them to a new file after editing. You may have to remove row names to make this work.