

An Introduction to the R Language

David Harte



GNS Science
PO Box 30 368
Lower Hutt 5040
NEW ZEALAND
www.gns.cri.nz

This Version: 23 June 2014

Postal Address: GNS Science
PO Box 30 368
Lower Hutt 5040
NEW ZEALAND

URL: www.gns.cri.nz

Email: D.Harte@'gns.cri.nz

Contents

1	Introduction to R	1
1.1	Running R Interactively	1
1.1.1	Starting R	1
1.1.2	Quitting R	2
1.2	Some Elementary (Atomic) Data Objects	2
1.2.1	Vectors	2
1.2.2	Matrices	3
1.2.3	Mode of an Object	5
1.3	Functions	6
1.3.1	Help Documentation	6
1.3.2	Writing Functions	7
1.4	Simple Graphs	8
2	Input-Output Methods	11
2.1	Reading Data into R	11
2.1.1	Reading Data from a CSV Text File	11
2.1.2	Reading Data from a Text File	11
2.1.3	Reading Data from Other Formats	13
2.2	Executing an R Program Contained in a Source File	14
2.2.1	Executing an R Program Source File Interactively	14
2.2.2	Executing Jobs in Batch Mode	14
2.2.3	Executing Jobs as a System Script File	15
2.3	Writing Output to a Text File	16
2.3.1	Writing R Objects to a Text File	16
2.3.2	Writing Program Output to a Text File	17
2.4	Writing R Objects to an R Binary Disk File	17
2.4.1	Saving R Objects for Use in a Subsequent Session	17
2.4.2	Retrieving R Objects from a Previous Session	17
2.5	Execution of External Programs From Within R	18
2.5.1	Executing FORTRAN and C/C++ from within R	18
2.5.2	Operating System Commands	18
3	Graphs	19
3.1	Scatter Plots	19
3.2	Modifying Plots	19
3.3	Barplots and Tables of Counts	21
3.4	Variations on Scatter Plots	21
3.5	Boxplots	22

3.6	Histograms	22
3.7	Multiple Plots Per Page	22
3.8	Selecting the Graphics Device	23
3.9	Maps	25
3.10	Scaling Plots	25
4	More Advanced Data Structures	27
4.1	List Objects	27
4.2	Data Frame	28
4.3	Factors	30
4.4	Attributes	32
4.5	Class Attribute, Generic and Method Functions	33
5	Some Standard Statistical Analyses	35
5.1	ANOVA	35
5.2	Generalised Linear Models	35
5.3	Other Models	36
A	Appendix: Software Installation	37
A.1	Installing Base R Software	37
A.2	Installing Contributed Packages	37
B	Appendix: Example Datasets	39
B.1	Large NZ Earthquake Events	39
B.2	Kauri Saplings	40
	References	41

Chapter 1

Introduction to R

Like S-PLUS ([Statistical Sciences Inc., 1992](#)), R is a statistical programming language ([R Development Core Team, 2003](#)) based on the S language (see [Chambers & Hastie, 1991](#)). R originated at the University of Auckland in New Zealand, the original authors being [Ihaka & Gentleman \(1996\)](#). An introduction to R is provided by the [R Development Core Team \(2014a\)](#), and another introduction has been written by [Maindonald & Braun \(2003\)](#).

R refers to data structures, and functions containing programming code, as *objects*. There are four object types in which we are particularly interested: vectors, matrices, lists, and functions. In this chapter we introduce the basic objects: vector, matrix, and function; and introduce simple graphics. In Chapter 4 compound data objects are introduced, i.e. objects that are essentially constructed with multiple basic objects¹.



R is an interpreted language like Python and MatLab, but unlike FORTRAN which must be compiled to form an executable binary. Programs in interpreted languages tend to be easier to write, but those languages that get compiled are more computationally efficient.

In this guide, we have included much programming code. We have not included the R response to these commands. It is intended that the reader has an R session running while viewing this document with a PDF reader. One can then easily highlight the programming statements in the PDF reader, and dump them into the R window, and then observe the response.

1.1 Running R Interactively

1.1.1 Starting R

In MS Windows, R can be started by initiating it from the “Start” menu².

¹R refers to objects like vectors and matrices as *atomic* and objects like lists and data.frames as *recursive*.

²The default R MS Windows set-up will have one large R window with multiple sub-windows. If you want separate windows (like in UNIX), go to “Edit” → “GUI Preferences ...”, and select “SDI” at the top. The font size can be changed here too. Then press “Save” and restart R.

In UNIX like environments, start R by entering

```
R
```

on the xterm or console command line. Your window will look something like the following.

```
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

>
```

1.1.2 Quitting R

When you have completed your session within R, quit by entering `q()`. When you quit R, it will ask whether you want to save any of the R objects that you may have created during the session. You can list them by entering `ls()`³ on the command line. So far we have not created anything, so there should be nothing listed. If you choose to save any objects when quitting R they will be saved in a file called `.RData` in the *current directory*⁴. Next time you start R from within this subdirectory, the objects that have been saved into the file `.RData` will be automatically loaded into the R session. This is discussed further in §2.4.1.

1.2 Some Elementary (Atomic) Data Objects

1.2.1 Vectors

1. Vectors are constructed using the `c` function, which stands for *combine*. For example, at the R command prompt, enter

```
a <- c(1, 2, 3, 4, 5)
b <- c(2, 4, 6, 8, 10)
d <- c(3, 9, 27)
```

³Note the UNIX like names, short and terse. The reflects R's UNIX origins.

⁴In UNIX the current directory is the directory from which R was started. In Windows the current directory defaults to "My Documents". The working directory can also be displayed by running the function `getwd()` and changed by `setwd()`. Alternatively in Windows, go to "File" → "Change dir ...".

The `<-` means assign the value of the object on the right to an object with the given name on the left. Alternatively, one can now use the “=” sign. Thus we have three vectors `a`, `b` and `d`.

To save possible confusion, we prefer not to use `c` as a vector name, since it is the name of a system function `c` (combine). We will look at functions more carefully in §1.3.

2. Basic arithmetic operators (`^`, `*`, `/`, `+`, `-`) act on an element by element basis. Entering `a*b` on the command line gives `c(2, 8, 18, 32, 50)`. Similarly, `2*b` is not ambiguous, and hence will give `c(4, 8, 12, 16, 20)`. Similarly with the other operators: `a-b` gives `c(-1, -2, -3, -4, -5)`.

Now enter `b*d`. This is ambiguous, and will produce a warning message. Often R will attempt to interpret the meaning, and hence the given answer could be quite different to that intended.

3. Individual elements of a vector can be indexed using square brackets. There are a couple ways to select the required elements from a vector:
 - (a) To select, for example, the 2nd element from vector `b`, enter `b[2]` on the command line. To select the 2nd twice and 4th elements, enter `b[c(2,2,4)]` on the command line. Notice that the indices are contained within a vector, i.e. `c(2, 2, 4)`.
 - (b) By using negative indicies, one *excludes* those elements, for example `b[-c(2,4)]` excludes the 2nd and 4th elements.
 - (c) Alternatively, one can use a logical (Boolean) vector of the same length as `b`. To select those elements in `b` that are greater than 6, create a logical vector by entering `e <- (b > 6)`. Now enter `print(e)`, and you will have a vector of the same length as `b` with a sequence of `TRUE`'s and `FALSE`'s depending on whether the expression is true for each element in the vector. Those elements can now be selected by entering `b[e]`. Alternatively, you could simply enter `b[b>6]`.
4. Often when data are collected, there are situations where some values are missing. For example, the depths of some historical earthquakes are often missing. In the R language, missing values (in *numeric* objects) are coded as `NA` without quotes. For example, say we have four earthquakes, the first three have depths (km): 31, 150, and 2, but the fourth is missing. These data would be assigned to the variable `depth` as:

```
depth <- c(31, 150, 2, NA)
```

Any arithmetic operations that are performed on a missing value will give a missing value, for example, try `2*depth`.

1.2.2 Matrices

1. One way to construct matrices (there are many ways) is as follows:

```
x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), byrow=FALSE, ncol=2)
print(x)
```

This gives

```
      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Like `c`, `matrix` is also a function. The statement above says to make a matrix with 2 columns, called `x`, and with elements $1, 2, \dots, 10$. The `byrow=FALSE` means load the matrix column by column.

Similarly,

```
y <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), byrow=FALSE, ncol=5)
print(y)
```

gives

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

2. As for vectors, the arithmetic operators ($\wedge, *, /, +, -$) act on matrices *element by element*, for example, `y*y` gives

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    9   25   49   81
[2,]    4   16   36   64  100
```

The same result would be given by entering `y^2` or `y**2`. However, `x*y` will produce an error.

3. Individual elements of the matrix can be selected, for example, entering `x[4, 2]` gives 9. As for vectors, one could also index using a logical matrix of the same dimensions. For example, enter:

```
z <- (x < 6)
```

The matrix `z` is a logical matrix with the same dimensions as `x`, containing either `TRUE`'s or `FALSE`'s depending on whether $x_{ij} < 6$.

4. Matrix multiplication is achieved with the symbol `%%`, hence `x %% y` gives

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   13   27   41   55   69
[2,]   16   34   52   70   88
[3,]   19   41   63   85  107
[4,]   22   48   74  100  126
[5,]   25   55   85  115  145
```

5. Character matrices (or vectors) can also be defined in the same manner, for example:

```
colours <- matrix(c("red", "blue", "green", "cyan", "yellow", "magenta"),
                  byrow=TRUE, nrow=2)
print(colours)
```


This gives:

```
      [,1]      [,2]      [,3]
[1,] "red"   "blue"   "green"
[2,] "cyan" "yellow" "magenta"
```

Names can be added to the rows and columns (see §4.4(5)).

1.2.3 Mode of an Object

In the subsections above, we looked at two types of data structures: vectors and matrices. However, the contents of `a` were numeric, of `e` were logical and of `colours` were character. This is referred to as the *mode*. The mode of an object can be determined by using the `mode` function. Possible modes are: "logical", "numeric", "complex", "character" and "function".

Recall from above that

```
a <- c(1, 2, 3, 4, 5)
b <- c(2, 4, 6, 8, 10)
x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), byrow=FALSE, ncol=2)
e <- (b > 6)
colours <- matrix(c("red", "blue", "green", "cyan", "yellow", "magenta"),
                  byrow=TRUE, nrow=2)
```

1. Entering `mode(a)` or `mode(x)` gives "numeric".
2. Entering `mode(e)` gives "logical".
3. Entering `mode(colours)` gives "character".
4. Recall that `c` is the combine function, thus entering `mode(c)` gives "function".
5. Entering `mode(matrix)` gives "function".
6. Objects with mode "numeric" could be of type (`typeof`) "double" precision or "integer". For example:

```
typeof(a)
typeof(a/3)
is.integer(a)
is.integer(a/3)
is.double(a)
is.double(a/3)
```

We need to be careful here: above, `a` was constructed as `a <- c(1, 2, 3, 4, 5)`. In this situation, `is.integer(a)` is `FALSE`. Now define `a` as

```
a <- 1:5
```

and repeat the above statements. Here we find that `is.integer(a)` is `TRUE`. Probably this is because the colon construct above is only meaningful with respect to integers. Care needs to be taken here, see the note on the documentation page for the function `seq` under 'Value'.

7. Complex numbers can also be represented, e.g.

```

z <- complex(real=1:5, imaginary=6:10)
print(z)
# the real and imaginary parts have mode numeric
mode(Re(z))
mode(Im(z))

```

1.3 Functions

1. A function is a sequence of commands or operations that are stored within an object. We have already used a number of functions, provided by the R system, in our discussion above, e.g. `c` and `matrix`. The functions in R are very similar in nature to FORTRAN functions. The functions may have one or more objects as input. They manipulate the objects in some specified manner, and *one* object is passed out of the function at the end.
2. Function objects have mode equal to `"function"`. By entering `mode(matrix)` on the command line, R informs us that the object `matrix` is a function.
3. We can view the internal commands within a function by entering its name (without brackets) on the command line, for example enter `matrix` on the command line.
4. Attaching brackets to a function name will cause the function to be executed. Unspecified function arguments within the brackets will cause the default values to be used. For example, entering `matrix()` will produce a matrix with one element which is assigned a missing value. Many functions require at least one argument to be set.
5. Trivial functions are `ls` (list objects) and `q` (quit). Recall that one quits the R session by *executing* the quit function, i.e. `q()` (see §1.1.2). Enter `q` without `()` to see the function definition, or `q("no")` to quit without saving the workspace.
6. There are in fact many functions within the R system: for performing mathematical operations, fitting statistical models, and graphical functions. For example, the object `y %*% x` represents a 2×2 matrix. The function `solve(y %*% x)` will invert the matrix, and `eigen(y %*% x)` will calculate the eigen values and vectors of the 2×2 matrix `y %*% x`. Note that the output from the `eigen` function is a list object (see §4.1), containing a vector of eigen values and a matrix of eigen vectors.

1.3.1 Help Documentation

1. Help documentation can be found by entering `help.start()` on the command line. When the web browser window appears, click on Packages, then find the entry for “base” and click on it. This contains documentation for each function in the base R package. Note that there are *required arguments* and *optional arguments*. The optional arguments will always have default settings. Those arguments with no default settings are *required arguments* and must be specified. The structure of the object that is passed out of the function is described under “Value”. “Details” contains information about the algorithm being used. All documentation for R functions is set out in a similar manner.

- Many help pages have a set of examples at the bottom that can be easily run. For example, select the help information for `eigen`. The first example is:

```
eigen(cbind(c(1,-1),c(-1,1)))
```

which will calculate the eigen values and vectors of the 2 by 2 matrix with one's on the main diagonal and negative one's on the minor diagonal. The code can be executed by highlighting it in the web browser, and dumping onto the R command line.

- Documentation for each of the functions referred to in this document can be found in the web browser help window.

1.3.2 Writing Functions

- The R system will never contain all of the functions that you will require, and you will need to be able to write your own. One needs to be careful in selecting function names that are meaningful and have not been used before. For example, enter `test` on the command line. It will probably say that the object does not exist, and hence we can use it as our function name.
- Say we have a vector, we want to multiply each element by 2 and add 1. We want to pass this vector into the function, and pass out the computed vector. This can be done as follows:

```
test <- function(invector){
  outvector <- 2*invector + 1
  return(outvector)
}
```

To execute the function using vector `a` in §1.2.1, enter `test(a)` on the command line. Complicated functions can be written using both logical (Boolean) and looping constructions. If arithmetic operations are applied to a logical vector, the elements will be treated as zero's (`FALSE`) and one's (`TRUE`).

- Note that R is a matrix like language, that is, the statement `2*invector` in the function means that *all* elements of `invector` should be multiplied by 2. If the dimensions of the objects in the expression being evaluated are compatible, and the expression is unambiguous, then it will be evaluated without a warning (recall the warning with `b*d`, or error with `x*y`). In FORTRAN, we would need to loop over each element. This could be done in R too, but is *extremely inefficient*:

```
test1 <- function(invector){
  n <- length(invector)
  outvector <- rep(NA, n)
  for (i in 1:n){
    outvector[i] <- 2*invector[i] + 1
  }
  return(outvector)
}
```

Compare the times taken by each function by running the following script:

```
invector <- seq(1, 1000000, 1)

# execution time with 1st function
```

```

start.time <- Sys.time()
out <- test(invector)
cat("time taken =", Sys.time()-start.time, "\n")

# execution time with 2nd function
start.time <- Sys.time()
out1 <- test1(invector)
cat("time taken =", Sys.time()-start.time, "\n")

```

4. It is possible to call both FORTRAN and C⁺⁺ code from within a function (see §2.5.1).

1.4 Simple Graphs

In this subsection, a very brief outline of graphical methods in R is given. More detail is given in Chapter 3.

1. Graphs are written to a graphics device. R has a number of these (see the topic “Devices” in the help documentation, §1.3.1), and the most appropriate one to use is determined by what one wants to do with the graph. For more details, see §3.8.
2. Having opened an appropriate graphics device, one often wants to change various parameters, for example: the number of graphs on the page, the axis layout, available colours, font types, etc. Various options can be selected by using the `par` function.
3. There are many functions to do various types of graphs. The most common are `plot`, `hist`, `curve`, and `barplot`. For example, say we wanted to plot the cubic function $f(x) = x(x-3)(x+1)$ on the interval $(-1.5, 3.5)$. This can be done by entering:

```

x <- seq(-1.5, 3.5, 0.01)
f <- x*(x-3)*(x+1)
plot(x, f, type="l")

```

If no graphics device is open, then in Linux (or UNIX) R will usually open an X11 window automatically, and in Microsoft Windows a “windows” window.

4. One can start with an initial plot, and overlay additional plots. For example, below we simulate 1000 random numbers from a standard normal distribution, and then overlay the standard normal density function.

```

set.seed(5)
hist(rnorm(1000, mean=0, sd=1), freq=FALSE,
     main="1000 Simulations From N(0,1) Distribution")
z <- seq(-4, 4, 0.01)
points(z, dnorm(z, mean=0, sd=1), type="l", col="blue")
box()

```

The first line is not necessary, but by setting the seed for the random number generator, the same values will be generated each time. Note that R has functions to calculate various aspects of many probability distributions, each following the same naming convention. For example, given a distribution `xyz`, then the functions

`dxyz`, `pxyz`, `qxyz`, and `rxyz` calculate the density function, probability function, quantiles and random numbers, respectively.

5. Maps of coastlines can also be drawn, for example enter:

```
library(maps)
map("usa")
map("nz")
map("world")
```

Higher resolution maps are provided in the [mapdata](#) package, and various projections are provided in the [mapproj](#) package. To install packages, see §A.2.

Chapter 2

Input-Output Methods

2.1 Reading Data into R

2.1.1 Reading Data from a CSV Text File

Appendix B.1 contains earthquake events (1 per row), with the variables separated by a comma. The data also has a header that contains the variable names (no spaces in the names). Assume that the data are in a file called “earthquakes.csv”. This is a fairly standard CSV file layout. Then the data can be read from the *current directory* as follows:

```
events <- read.csv(file="earthquakes.csv", as.is=TRUE)
```

A full or relative directory path can also be specified, including a web address:

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"
events <- read.csv(file=fname, as.is=TRUE)
print(events)
```

The variable names will be those on the first line of the spreadsheet. All subsequent lines are assumed to be data. Note that variable names which are often used in spreadsheets are rather long and descriptive, and would be very clumsy to use in a computer program¹. More succinct names are better. See the help documentation for `read.csv` which also describes other supported formats.

Note the argument `as.is=TRUE`. This is because the data contains a character variable (month). By default, R will turn this variable into a *factor* with levels sorted alphabetically. Essentially, a factor is a coded variable. Given that months have a natural non-alphabetical order, this variable should be an *ordered factor* (more on this later). Hence, above we tell it to leave it as is, and it can be coded as an ordered *factor* later.

2.1.2 Reading Data from a Text File

Listed below are earthquake events with magnitude ≥ 6.7 between 1965 and 1995 in or near New Zealand.

¹Often spreadsheet column names consist of multiple words and hence contain spaces. Variable names in R cannot contain spaces. Invalid characters in the variable name get replaced with “.”.

Latitude	Longitude	Event Name	Depth	Magn	Date	Time
-41.76	172.04	Westport	12	6.7	23 May 1968	17:24:17.4
-34.94	179.30	Kermadec Trench	297	6.8	08 Jan 1970	17:12:36.6
-39.13	175.18	National Park	173	7.0	05 Jan 1973	13:54:27.6
-41.61	173.65	Marlborough	84	6.7	27 May 1992	22:30:36.1
-45.21	166.71	Secretary Island	5	6.7	10 Aug 1993	00:51:51.6
-43.01	171.46	Arthurs Pass	11	6.7	18 Jun 1994	03:25:15.2
-37.65	179.49	East Cape	12	7.0	05 Feb 1995	22:51:02.3

Data are often presented in a messy way, and not nicely delimited with well defined variable names as in the previous example.

1. Assume that the data are stored in a file called “example1.txt” in the format below. Hence, the file contains no variable names, and the white space denotes the break between variables. The underscore in the event name ensures that the white space rule holds.

```
-41.76 172.04 Westport      12 6.7 23 05 1968 17 24 17.4
-34.94 179.30 Kermadec_Trench 297 6.8 08 01 1970 17 12 36.6
-39.13 175.18 National_Park  173 7.0 05 01 1973 13 54 27.6
-41.61 173.65 Marlborough    84 6.7 27 05 1992 22 30 36.1
-45.21 166.71 Secretary_Island 5 6.7 10 08 1993 00 51 51.6
-43.01 171.46 Arthurs_Pass   11 6.7 18 06 1994 03 25 15.2
-37.65 179.49 East_Cape      12 7.0 05 02 1995 22 51 02.3
```

2. These can be read into a list object by using the `scan` function:

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/example1.txt"
NZ1 <- scan(file=fname, what=list(latitude=0, longitude=0,
    event="", depth=0, magnitude=0, day=0, month=0,
    year=0, hour=0, minute=0, second=0), sep="")
```

If the each white space was replaced by a comma, then the above would have `sep=","`, and then the event name could legitimately have a space.

3. The object `NZ1` will be a list object, try the function `is.list(NZ1)`. When we print the object, i.e. `print(NZ1)`, it prints as a list. Lists are discussed further in §4.1.
4. These data can be presented in a tabular format by turning the object `NZ1` into a data frame:

```
NZ1 <- as.data.frame(NZ1, stringsAsFactors=FALSE)
print(NZ1)
```

Data frames will be discussed further in §4.2. They are a special version of a list where each component is a vector of the same length, and each row represents a particular sample unit.

5. Note that `mode(NZ1$latitude)` is “numeric” and `mode(NZ1$event)` is “character” as required.
6. Now consider the data stored in the file “example2.txt”, which does not contain the underscores, and some values are missing (unfortunately, often indicated by blanks), e.g. the depth for Westport.


```

-41.76 172.04 Westport          6.7 23 05 1968 17 24 17.4
-34.94 179.30 Kermadec Trench 297 6.8 08 01 1970 17 12 36.6
-39.13 175.18 National Park    173 7.0 05 01 1973 13 54 27.6
-41.61 173.65 Marlborough      84 6.7 27 05 1992 22 30 36.1
-45.21 166.71 Secretary Island  5 6.7 10 08 1993 00 51 51.6
-43.01 171.46 Arthurs Pass     11 6.7 18 06 1994 03 25 15.2
-37.65 179.49 East Cape        7.0 05 02 1995 22 51 02.3

```

7. The use of a separator as above will not work here. In this situation, one reads each complete line (record) into one character variable. The use of `sep="\n"` indicates that the end of record denotes the next value. One then picks off the substrings relating to the individual variables, and the numeric variables must be “coerced” from character to numeric:

```

fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/example2.txt"
a <- scan(file=fname, what=character(), sep="\n")
NZ2 <- NULL
NZ2$latitude <- as.numeric(substr(a, 1, 6))
NZ2$longitude <- as.numeric(substr(a, 8, 13))
NZ2$event <- sub(" +$", "", substr(a, 15, 30))
NZ2$depth <- as.numeric(substr(a, 32, 34))
NZ2$magnitude <- as.numeric(substr(a, 36, 38))
NZ2$day <- as.numeric(substr(a, 40, 41))
NZ2$month <- as.numeric(substr(a, 43, 44))
NZ2$year <- as.numeric(substr(a, 46, 49))
NZ2$hour <- as.numeric(substr(a, 51, 52))
NZ2$minute <- as.numeric(substr(a, 54, 55))
NZ2$second <- as.numeric(substr(a, 57, 60))

NZ2 <- as.data.frame(NZ2, stringsAsFactors=FALSE)
print(NZ2)

```

Note the use of `sub` to process `NZ2$event`. The dollar in `" +$"` means the end of the string, and space followed by a plus means any number of spaces, i.e. remove any spaces at the end of the string `substr(a, 15, 30)`. The syntax used in `" +$"` is that of a *regular expression*. A regular expression allows one to express complex structure of character strings.

A similar processing strategy could be used when reading a file that contains multiple record types. Initially read each complete record into a character vector as above. One would cycle through the vector, element by element, initially determining the record type, then using an appropriate sequence of statements for that particular record.

2.1.3 Reading Data from Other Formats

There are a couple of packages on [CRAN](#) (Comprehensive R Archive Network) for reading or writing Excel spreadsheets. See, for example, [excel.link](#), [WriteXLS](#), [XLConnect](#), and [xlsx](#).

[CRAN](#) also contains a number of packages for importing and exporting to XML files.

Note that the quality of the packages on CRAN can be variable, and software may not properly deal with spreadsheets that are badly defined and set out. The safest method is to write the spreadsheet data into a CSV file, and then read this file into R. To install packages, see §A.2.

2.2 Executing an R Program Contained in a Source File

2.2.1 Executing an R Program Source File Interactively

Usually programming commands will be contained within a text file, and this file called from within R. This can be done as follows by using the `source` function:

1. The text file can be written with any text editor, e.g. emacs, gedit, or notepad. Create a file with the name “test.R”
2. Enter the required programming code into the file. For example:

```
a <- c(1, 2, 3, 4, 5)
b <- c(2, 4, 6, 8, 10)
# print the product of a and b
print(a*b)
```

Note that the hash character (#) starts a comment line. This comment remains in effect until the next hard return (hard line feed). Save the file “test.R”.

3. The R source code within the text file can now be executed in by entering

```
source("test.R")
```

on the R command line. R will look for the file in the *current directory*, which can be listed with the command `getwd()`. If the created file is not in the current directory, then the directory path will need to be specified. In MS Windows, it can be changed by selecting “Change dir ...” in the “File” menu². Alternatively, the full/relative directory path or URL could be specified:

```
source("ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/test.R")
```

2.2.2 Executing Jobs in Batch Mode

It is often required to run an R program as part of a sequence of other jobs not being executed in the R language.

UNIX Like Operating System

These jobs may be initiated by commands contained in an executable UNIX file. To execute R source code contained in `infile` from outside R, the appropriate command is

```
R CMD BATCH infile outfile
```

Here the `infile` cannot be a URL as above. The output that would be normally written to the VDU in an interactive session will now be written to `outfile`. For more information, enter

```
R CMD BATCH --help
```

²Alternatively, the function `setwd()` could be used to change the working directory, or the full directory path name could be included in the `source()` function call.

Windows Operating System

MS Windows is more of a “point and click” operating system. Though their latest creation has moved on to “touch and swipe”. R was developed in a UNIX operating system and hence running file scripts containing commands is quite natural. In fact, when running large and complex analyses, where there are many possible options, a script containing the required commands is the only reliable way. Entering the options through a GUI interface each time the analysis is done is a recipe for disaster.

Initially one opens a command line window by going to “Start” → “Run...”, enter `cmd.exe` in the box, and press “OK”. Alternatively, you may find it under “Start” → “All Programs” → “Command Prompt”.

A new window will appear, probably with a black background. The default background colour and fonts can be changed.

Say we have a program script called `test.R`, as below, on the drive H:

```
x <- seq(0, 20, 2)
print(x)
```

The program is run in batch mode by entering the following commands in the command window:

```
H:
"C:\Program Files\R\R-3.1.0\bin\x64\R.exe" CMD BATCH test.R
```

Points to note are as follows.

1. When R starts, the current directory will be H:, hence this is where the output file called `test.Rout` will be written.
2. Note the quotes around `"C:\Program Files\R\R-3.1.0\bin\x64\R.exe"`. This is because the command gets broken with the space in `Program Files` if they are excluded. It is generally not a good idea to use spaces in file and directory names. It is an unfortunate MS Windows innovation.
3. Note also the “backward slash” in the directory path. This was a MS DOS innovation. File paths on the WWW and UNIX use “/”.

2.2.3 Executing Jobs as a System Script File

R can be executed as a shell script. A trivial example is given below.

UNIX Like Operating System

In the UNIX operating system, a file would be created containing the script, for example, as below. Say this is called `script.sh`. The file should have permissions to be executable.

```
#!/usr/bin/Rscript --vanilla

# expects two args: n (+ve integer) & your name
args <- commandArgs(TRUE)

# note that the arguments are in a character vector
```

```

print(args)
print(is.character(args))

# can coerce n to be integer
n <- as.integer(args[1])
nm <- args[2]

for (i in 1:n){
  cat(i, "\n")
  cat("Hello", nm, "\n")
}

# note the difference with print()
for (i in 1:n){
  print(i)
  print(paste("Hello", nm))
}

q()

```

Note the first line in the file: this is important, telling the system that the relevant R program to execute the script is found in `/usr/bin/Rscript`. The `vanilla` option combines the following options:

```
Rscript --no-save --no-restore --no-site-file --no-init-file --no-environ
```

For further information about options in UNIX, see

```
man Rscript
```

or

```
Rscript --help
```

The script would be executed by running `script.sh` at the command line with the desired arguments, for example:

```
./script.sh 4 David
```

Windows Operating System

The standard Windows command line has no concept of a script. While “Rscript.exe” can be run in Windows (see below), it is not that different to the BATCH option previously discussed. The main difference is that output appears directly in the command window.

```

H:
"C:\Program Files\R\R-3.1.0\bin\x64\Rscript.exe" test.R

```

2.3 Writing Output to a Text File

2.3.1 Writing R Objects to a Text File

We need to distinguish between function objects and data objects. In the case of function objects, one may want to put the function code into a text file so that

it can be modified, then included back into R and executed. For example, by entering

```
dump("matrix", file="temp.R")
```

the code for the function `matrix` will be written into the text file “temp.R”. This code could be edited and included back into R by using the `source` function (see §2.2.1).

Data objects that are required for use in programs outside of R will need to be written to a text file. Some possibly useful functions are `sink`, `print`, and `cat`.

2.3.2 Writing Program Output to a Text File

R output can be written to a text file by using the `sink` function.

For example, enter `sink("test.out")` on the R command line. This tells R to direct all output that would have normally gone to the screen to go to the file “test.out”. Now enter `source("test.R")` again (created in §2.2.1 above). To close the file “test.out”, enter `sink()`. Now, there should be a file called “test.out”, which can be viewed by using a text editor.

2.4 Writing R Objects to an R Binary Disk File

2.4.1 Saving R Objects for Use in a Subsequent Session

When quitting from R one is asked whether the “workspace” should be saved (i.e. the objects displayed when one runs `ls()`), see §1.1.2.

One can be more selective about the objects to save, together with the location and name of the “Rda” file (i.e. the R binary file) by using the `save` function. For example,

```
a <- 10
b <- 15
d <- 21

save(a, b, file="temp.Rda")
```

will save the objects `a` and `b` only in an R format in the file “temp.Rda”. To save all current objects, run:

```
save(list=ls(), file="temp.Rda")
```

2.4.2 Retrieving R Objects from a Previous Session

Saved objects can be reloaded into R by using the `load` function. For example, the file “temp.Rda” created in §2.4.1 is loaded by executing the following command:

```
load("temp.Rda")
```

2.5 Execution of External Programs From Within R

2.5.1 Executing FORTRAN and C/C++ from within R

Compiled FORTRAN and C++ code can be executed and linked to internal R objects. This is done by initially compiling the program code, then using `dyn.load` to load the compiled binary into R, then using the functions `.C`, `.Fortran`, `.Call` or `.External` (refer to their documentation under topic `Foreign`) to execute the external subroutine.

2.5.2 Operating System Commands

System commands can be executed using the `system` function. Say the current directory contains a number of CSV files, all in the same format. We want to read them all, and combine them into one dataset.

```
fnames <- system("ls *.csv", intern=TRUE)

y <- NULL
for (nm in fnames){
  x <- read.csv(file=nm, as.is=TRUE)
  x$nm <- sub(".csv", "", nm)
  y <- rbind(y, x)
}
```

The statement `x$nm <- sub(".csv", "", nm)` adds a new variable to each block of data added, containing their origin file name without the `".csv"`.

Chapter 3

Graphs

The code in this chapter uses earthquakes extracted from the NZ Catalogue ([GeoNet](#)), and are events between 1 Jan 1960 and 31 Dec 2000 with a magnitude ≥ 6.5 . They are listed in Appendix [B.1](#). The code below assumes that they are stored in a text file called `earthquakes.csv`. We take the following from [§2.1.1](#).

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"
events <- read.csv(file=fname, as.is=TRUE)
```

3.1 Scatter Plots

Simple scatter plots can be drawn as follows:

```
plot(events$longitude, events$latitude)
```

By default, axis labels are the variable names, and the axis lengths are determined by the range covered by the variables. It has an algorithm to make it look nice or “pretty” so that there is a reasonable amount of white space at the ends, and the tick marks occur at “nice” values.

One can draw lines between the points as follows:

```
plot(events$longitude, events$latitude, type="l")
```

The line will start at the first listed point (in the two vectors), go to the 2nd, then 3rd, and so on. By default (as in the first plot), `type="p"` (i.e. points).

3.2 Modifying Plots

Generally one will want to modify the default characteristics. Here we tell it to redo the plot, initially with with no axes, but specify the axis limits. We make the points bigger (`cex=4`). We then add the x axis (`axis(1)`) and put the y axis on the right hand side (`axis(4)`). Finally a title is added.

```
plot(events$longitude, events$latitude, xlab="", ylab="", axes=FALSE,
      cex=4, xlim=c(155, 190), ylim=c(-55, -20))
axis(1)
axis(4)
box()
title(main="Large NZ Earthquake Events", line=2.7, cex.main=2)
```

Many options can be set with the `par()` function. For example, if we really want the y axis on the right hand side, then we probably want a wider margin on the right and less on the left, hence:

```
print(par())$mar)
par(mar=c(5.1, 2.1, 4.1, 4.1))
```

We can add reference lines to the plot:

```
abline(v=seq(160, 185, 5), lty=3, col="blue")
abline(h=seq(-50, -25, 5), lty=3, col="blue", lwd=0.3)
```

Note that the vertical ones look rather heavy, so the horizontal ones have been made lighter. Note that `lwd` is an option found in the documentation under `par()`.

Lets start again, including all of the above changes, plus a few more. Check the options `las` and `tcl` in the documentation for `par()`:

```
par(mar=c(5.1, 2.1, 4.1, 4.1))
plot(events$longitude, events$latitude, xlab="", ylab="", axes=FALSE,
      cex=4, xlim=c(155, 190), ylim=c(-55, -20), xaxs="i", yaxs="i")
axis(1, tcl=0.5)
axis(4, las=2)
box()
title(main="Large NZ Earthquake Events", line=2.7, cex.main=2)
title(xlab="Longitude")
abline(v=seq(160, 185, 5), lty=3, col="blue", lwd=0.3)
abline(h=seq(-50, -25, 5), lty=3, col="blue", lwd=0.3)
```

Text (like place names) can be added as follows:

```
text(172.04, -41.76, "Westport", adj=c(1.2, 0.4), srt=-45, cex=2,
     col="red")
```

Now assume that we want to scale the size of the points to be proportional to the event magnitude, and the colour of the point to be related to its depth. This can be done as follows:

```
par(mar=c(5.1, 2.1, 4.1, 4.1))
plot(events$longitude, events$latitude, xlab="", ylab="", axes=FALSE,
      cex=4, xlim=c(155, 190), ylim=c(-55, -20), xaxs="i", yaxs="i",
      type="n")
axis(1)
axis(4)
box()
title(main="Large NZ Earthquake Events", line=2.7, cex.main=2)
abline(v=seq(160, 185, 5), lty=3, col="blue", lwd=0.3)
abline(h=seq(-50, -25, 5), lty=3, col="blue", lwd=0.3)
depthlevel <- 1 + (events$depth >= 50) +
                (events$depth >= 100) +
                (events$depth >= 150) +
                (events$depth >= 200)
colours = c("red", "yellow", "green", "cyan", "blue")
points(events$longitude, events$latitude,
        cex=3*(events$magnitude-6.4),
        col=colours[depthlevel],
        lwd=0.5+2*(events$magnitude-6.4))
title(sub=expression(paste("Depth (km): ", 0 <= {red < {50 <=
{yellow < {100 <= {green < {150 <= {cyan < {200 <= {blue <
infinity}}}}}}}})), line=3)
```


The `points` function call above needs further explanation: note that both `events$longitude` and `events$latitude` are the x and y variables, respectively, being plotted. The number of events being plotted, say n , could be calculated as `n <- length(events$longitude)`. Also note though, that `length(events$magnitude)` and `colours[depthlevel]` are also of length n . This means that the optional arguments `cex`, `col` and `lwd` all have lengths of n . Hence, when the plot function is executed, it will take the latitude and longitude for the first data point, and use the first value from each of `cex`, `col` and `lwd` to display this point (event). Similarly, the i th point (event) ($i = 1, \dots, n$) will use the i value.

When plotting some data (like geographical locations or a map) the aspect ratio (plot height divided by plot width) is important. In the above example, the aspect ratio will be determined so as to maximise the use of the given graphics device. The aspect ratio can be specified, see §3.9.

The above shows how mathematical symbols (including Greek characters) can be included into graphics, see help documentation for topic `plotmath` for a complete description. For a demonstration, enter `demo(plotmath)` at the R prompt.

Many different colours can be specified in many different formats. On the help page, under “Search Engine”, tick “keywords” and search for color.

3.3 Barplots and Tables of Counts

Counts of event numbers over discrete categories (e.g. months) can be tabulated or plotted as a barplot. Notice how the months are ordered alphabetically:

```
table(events$month)
table(events$year, events$month)
barplot(table(events$month))
```

Technically, we would refer to the month variable as an *ordered factor*. By defining it as such, the months (i.e. levels) will be ordered correctly. While the years are ordered correctly, it has only included those where an event occurred. By defining it formally as an *ordered factor* as well, we can specify all possible years:

```
events$month <- factor(events$month, levels=month.abb, ordered=TRUE)
events$year <- factor(events$year, levels=seq(1960, 2000, 1),
                      ordered=TRUE)
table(events$year, events$month)
barplot(table(events$month))
```

3.4 Variations on Scatter Plots

We may want to plot each earthquake event at its occurrence time as a vertical line representing the magnitude. Firstly, we calculate the time as the number of days since 1 Jan 1960, then divide by 365.25 to represent the number of years since 1 Jan 1960. Dates are tricky!

```
event_date <- as.Date(paste(events$day, events$month, events$year, sep=""),
                     "%d%b%Y")
num_days <- julian(event_date, origin=as.Date("1960-01-01")) +
             events$hour/24 + events$minute/(24*60) +
             events$second/(24*60*60)
num_years <- num_days/365.25
plot(num_years, events$magnitude, type="h", ylim=c(6.5, 8.0),
```

```
xlab="Years Since 1 Jan 1960", ylab="Event Magnitude",
col="gray50", lwd=2)
```

The sequence of inter-event times can be plotted as follows:

```
n <- length(num_days)
diff_days <- num_days - c(0, num_days[-n])
plot(seq(1, n), diff_days, type="l", col="blue",
      xlab="Event Number", ylab="Inter Event Time (days)")
```

3.5 Boxplots

Boxplots can be used to compare the distribution of one variable (e.g. event depth) in each level of a factor variable (e.g. month). Obviously, there is no reason to expect the distributions to be different here!!!

```
boxplot(events$depth ~ events$month)
```

Note the strange syntax (i.e. `events$depth ~ events$month`). This is a “model” *formula* (more about this later). These are also used to specify regression and ANOVA type models.

3.6 Histograms

Here we plot a histogram of the event magnitudes. If the magnitudes satisfy the *Gutenberg-Richter law*, then they have an exponential distribution. Hence, we estimate the parameter of the exponential distribution (`lambda`), and then overlay the exponential probability density (scaled to have the same area) with this parameter value.

```
barwidth <- 0.125
hist(events$magnitude, main="", xlab="Magnitude",
      breaks=seq(6.5, 8, barwidth))
lambda <- 1/mean(events$magnitude-6.5)
magn <- seq(6.5, 8, length.out=1000)
area_scale <- length(events$magnitude)*barwidth
points(magn, area_scale*dexp(magn-6.5, rate=lambda),
      type="l", col="blue", lwd=2)
```

3.7 Multiple Plots Per Page

The option below, `par(mfrow=c(2,2))`, says we want a grid of plots on the page, with 2 rows and 2 columns. The plots will be added row by row. If we want them to be added column by column, we would use `par(mfcol=c(2,2))`. See Figure 3.1 for specifying plot margins.

```
par(mfrow=c(2,2))
print(par()$oma)
print(par()$mar)
hist(events$magnitude)
barplot(table(events$month))
plot(events$longitude, events$latitude)
```

The code for producing Figure 3.1 is as follows:

```
pdf("margins1.pdf", height=10, width=10)

par(mfrow=c(2,2), oma=c(3,3,3,3), mar=c(3,3,3,3))

for (i in 1:4){
  plot(1:5, 1:5, type="p", xlab="", ylab="", axes=F)
  mtext("Side 1 Margin Width = par()$mar[1]", side=1, line=1)
  mtext("Side 2 Margin Width = par()$mar[2]", side=2, line=1)
  mtext("Side 3 Margin Width = par()$mar[3]", side=3, line=1)
  mtext("Side 4 Margin Width = par()$mar[4]", side=4, line=1)
  box(col="blue", lwd=3)
  box(which="figure", col="red", lwd=3)
}

box(which="outer", col="green", lwd=4)
mtext("Side 1 Outer Margin Width = par()$oma[1]", side=1, line=1, outer=TRUE)
mtext("Side 2 Outer Margin Width = par()$oma[2]", side=2, line=1, outer=TRUE)
mtext("Side 3 Outer Margin Width = par()$oma[3]", side=3, line=1, outer=TRUE)
mtext("Side 4 Outer Margin Width = par()$oma[4]", side=4, line=1, outer=TRUE)

dev.off()
```

The page can also be divided in an irregular way. Below we initially divide it into 1 row and 2 columns, now called screen 1 and screen 2, respectively. Then we tell it to divide screen 2 into 2 rows and 1 column, which will be called screens 3 and 4, respectively. We then add some plots.

```
split.screen(c(1,2))
split.screen(c(2,1), screen=2)
screen(1)
plot(events$longitude, events$latitude)
screen(3)
barplot(table(events$month))
screen(4)
hist(events$magnitude)
```

3.8 Selecting the Graphics Device

So far, our graphs have been plotted onto a device on the VDU. They can also be plotted to a number of different file types. For example:

```
pdf("temp.pdf")
plot(events$longitude, events$latitude)
dev.off()
```

Note the `dev.off()` statement. This tells it to close the pdf file. Until this is done, the file remains open, and any further graphical statements will cause additions to be made to that file. The statement `graphics.off()` will close all currently open graphics devices.

Probably the `pdf` and `postscript` types are the most appropriate where publication quality is required. Other types, like `png` or `jpeg` may be more appropriate for inclusion onto web pages. For more possibilities, enter `help.start()` on the R command line, then on the help page in the web browser, click on *Search Engine and Keywords*; in the Keywords section, select *device* under *Graphics*.

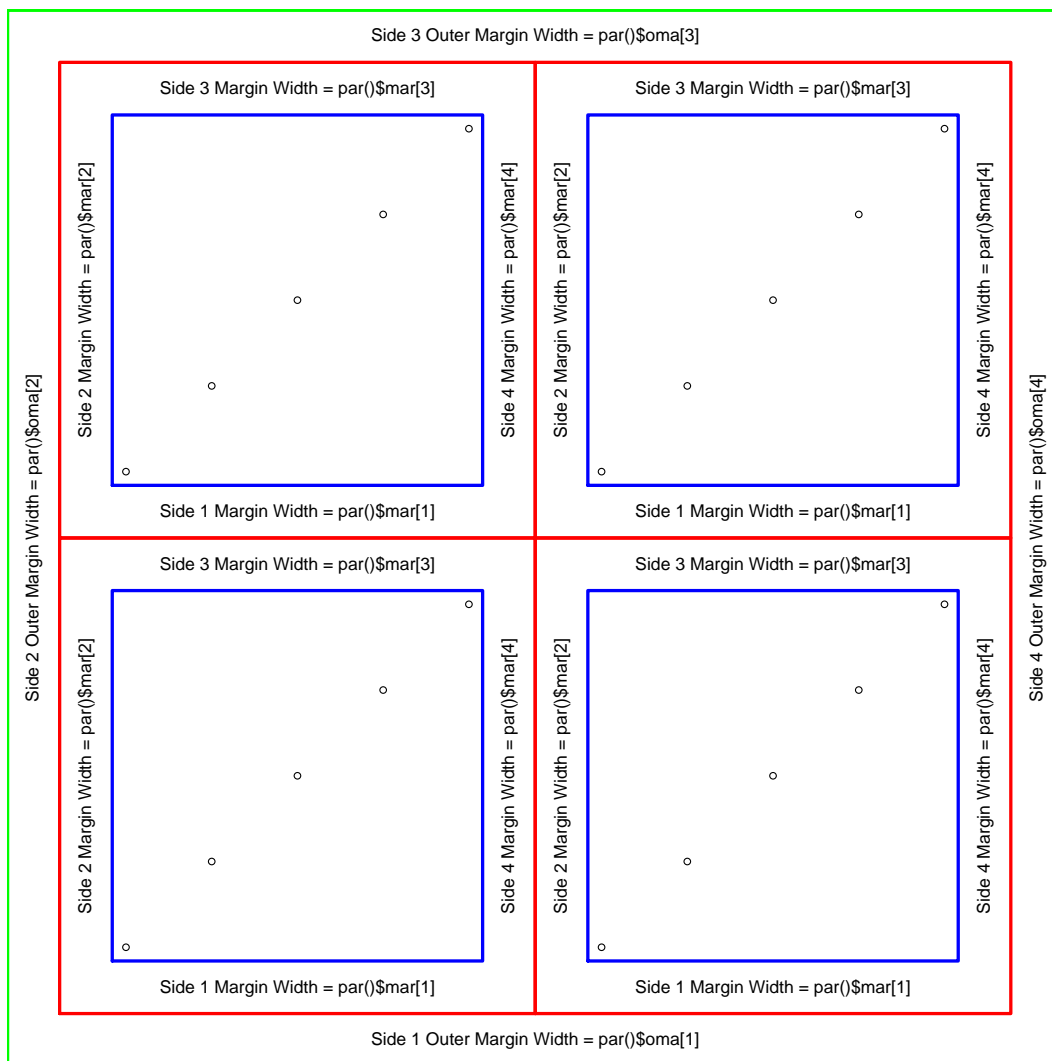


Figure 3.1: Margins on a plot: the area between the green and red lines is the outer margin and is specified by using `par` with argument `oma`. It can be used for overall titles and footnotes. The area between the red and blue lines is the inner margin, and specified by `mar`, and is used for titles and axis labels specific to an individual plot. The axes of the individual plots will align with the blue lines.

3.9 Maps

When placing graphs into files, it generally becomes very important that the x and y axes are correctly scaled, e.g. when producing maps. Here we return to the earlier example of an epicentral plot, and add a map. The `map` function automatically scales the plotting area to give a reasonable aspect ratio based on a rectangular projection.

```
library(maps)
par(mar=c(5.1, 2.1, 4.1, 4.1))
map(database="nz", xlim=c(155, 190), ylim=c(-55, -20), myborder=0)
axis(1)
axis(4)
box()

title(main="Large NZ Earthquake Events", line=2.7, cex.main=2)
abline(v=seq(160, 185, 5), lty=3, col="blue", lwd=0.3)
abline(h=seq(-50, -25, 5), lty=3, col="blue", lwd=0.3)
depthlevel <- 1 + (events$depth >= 50) +
               (events$depth >= 100) +
               (events$depth >= 150) +
               (events$depth >= 200)
colours = c("red", "yellow", "green", "cyan", "blue")
points(events$longitude, events$latitude,
       cex=3*(events$magnitude-6.4),
       col=colours[depthlevel],
       lwd=0.5+2*(events$magnitude-6.4))
title(sub=expression(paste("Depth (km): ", 0 <= {red < {50 <=
{yellow < {100 <= {green < {150 <= {cyan < {200 <= {blue <
infinity}}}}}}}})), line=3)
```

Note the statement `library(maps)`. So far, all of the programming statements that we have used are in the base R distribution. R is open source software, and there are many contributed packages. The “maps” package is one such package, see Appendix A.2 for further details of where these packages can be found and about their installation.

The [maps](#) package includes a New Zealand map, but this is relatively low resolution. Higher resolution maps are provided in the [mapdata](#) package.

3.10 Scaling Plots

Generally one wants the plot to be of a particular size, and also a particular aspect ratio. The size of the overall plot can be defined by arguments (often `height` and `width`) to the function that defines the graphics device. By default, the plotting routines will try to “fill” the white space nicely in the file. If one wants varying amounts of white space around the plot, this can be achieved by either specifying the size of the margins or the size of the “plot” area (i.e. the area *within* the axes). They can be defined by calling the function `par()` after defining the graphics device, see options `mar`, `pin` and `usr`.

For example, the following defines a pdf file called “temp.pdf”, which is 6 by 6 inches. The width of the plotting area within the axes is 4 inches and the height is 2 inches.

```
pdf("temp.pdf", width=6, height=6)
par(pin=c(4, 2))
plot(0, 1)
dev.off()
```


Chapter 4

More Advanced Data Structures

4.1 List Objects

1. More complicated data structures can be constructed as *list* objects. For example, say we wanted to put vector `a`, matrix `x` and the matrix `colours` into one object called `data` (these objects were created in §1.2.1 and §1.2.2). This is done as follows:

```
a <- c(1, 2, 3, 4, 5)
x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), byrow=FALSE, ncol=2)
colours <- matrix(c("red", "blue", "green", "cyan", "yellow", "magenta"),
                  byrow=TRUE, nrow=2)

data <- list(a, x, colours)
```

2. Now enter `print(data)` on the command line, to get

```
[[1]]:
[1] 1 2 3 4 5

[[2]]:
      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

[[3]]:
      [,1]      [,2]      [,3]
[1,] "red"   "blue"   "green"
[2,] "cyan"  "yellow" "magenta"
```

Notice that the matrix `x` is referred to as `[[2]]` within the list object `data`. Enter `data[[2]]` on the command line to get `x`. The element in the 5th row and 2nd column of `x` can be extracted as `data[[2]][5,2]`.

3. If we want to retain their original names (or allocate new names), enter

```
data <- list(a=a, x=x, colours=colours)
```

Now `x` can be retrieved from the object `data` by entering either `data[[2]]` or `data$x`.

4. Enter `mode(data)` to see that R recognises the object `data` as a list object (§1.2.3). Enter `names(data)` to see the variable names within the list object called `data`.
5. Each part of the list will have its own mode (§1.2.3), eg. enter `mode(data$colours)` and `mode(data$x)`.

4.2 Data Frame

Many datasets are table-like objects, where the rows represent observations, and the columns represent variables. A data frame has these characteristics but with much more flexibility¹. It is basically a list, but where *all* of the variables vectors and *must* be of the same length. Hence it can be treated and displayed like a table or a matrix, where each row represents a particular sample unit (e.g. earthquake, person, event, etc). Further, the storage modes (and other attributes) of the individual variables can be different.

1. Consider the earthquake data in §B.1. We will initially read this here using the `scan` function as follows:

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"

events <- scan(fname, what=list(latitude=0, longitude=0,
                                depth=0, magnitude=0, day=0, month="",
                                year=0, hour=0, minute=0, second=0),
               sep="," , skip=1)
```

Note that the `scan` function does not utilise the first record as variable names, hence we tell it to skip one row at the beginning, and define the variable names explicitly as part of the `scan` function call.

```
print(events)
print(mode(events))
print(is.list(events))
print(is.data.frame(events))
```

Notice that each variable is thought of as a separate piece of data. But here, the *n*th element in each vector represents the *n*th event, and hence it is more natural to print in a tabular or matrix like format. This can be achieved by turning the `events` object into a data frame:

```
events <- as.data.frame(events, stringsAsFactors=FALSE)
print(events)
print(mode(events))
print(is.list(events))
print(is.data.frame(events))
```

Now note that the mode is still “list”, but it is a “data.frame”, i.e. a special type of list where all components are vectors of the same length; and the *n*th element in each vector represents the *n*th sample unit.

2. Note that the `stringsAsFactors=FALSE` option above says not to turn character variables into factors (months in this case); enter `mode(events$month)`.

¹A data frame in R is structurally very similar to a table in an SQL database. In both, the columns (variables) can have different storage modes and attributes.

3. The `read.csv` function originally used to read these data at the start of Chapter 3 created an object called `events` as a data frame immediately. This is because a CSV file is generally a dump from a spreadsheet which has similar characteristics to a data frame. Check this as follows:

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"
events <- read.csv(file=fname, as.is=TRUE)
print(events)
print(mode(events))
print(is.list(events))
print(is.data.frame(events))
print(is.matrix(events))
```

Notice also that `read.csv` utilises the variable names in the first record of the CSV file. Also note that the `as.is=TRUE` option above says not to turn character variables into factors (months in this case); enter `mode(events$month)`. Note that while a data frame is matrix-like, it is not a matrix.

4. There are various functions, some equivalent, telling us about the events data frame:

```
print(dimnames(events))
# dimnames returns a list
print(is.list(dimnames(events)))
print(nrow(events))
print(ncol(events))
print(names(events))
print(colnames(events))
print(rownames(events))
# their values can be assigned to other variables
n <- ncol(events)
m <- nrow(events)
```

5. There are various ways to extract specific data from a data frame. Think of it here as a list. Entering `names(events)` will tell us the defined variable names, and the order in which they occur. Since longitude is the second variable, and the “events” object is also a list, then we could extract the longitude vector as `events$longitude` or `events[[2]]` (note the double square brackets), consistent with the discussion in §4.1. Since longitude is a vector, if we wanted to extract the 10th element, this could be done as `events$longitude[10]` or `events[[2]][10]`; notice here the single square brackets.
6. But the events object is a special type of list, i.e. a data frame. A data frame has matrix like properties. Hence, the following statements are equivalent:

```
print(events$longitude[10])
print(events[[2]][10])
print(events[10,2])
print(events[10,"longitude"])
```

As in matrix algebra, we need to specify both the row and column to uniquely determine a given element. Note that a variable (longitude here) can be specified by its number or name. Further, the following statements are equivalent:

```
print(events$longitude)
print(events[[2]])
```

```
print(events[,2])
print(events[, "longitude"])
```

In the last two statements, nothing is specified for the rows. This defaults to all rows.

7. We can also extract blocks of rows and columns:

```
# longitude and latitude for events 10 and 20
print(events[c(10,20), c("longitude","latitude")])
# all variables for events 10,12,14,...,20
print(events[seq(10,20,2),])
# all events where depth is <= 40km
print(events[events$depth<=40,])
# all events where depth is <= 40km occurring after 1980
# a is a boolean vector of the same length
a <- (events$depth<=40 & events$year>1980)
print(is.logical(a))
print(events[a,])
```

8. Beware with row numbers and row names. They are not necessarily the same. For example, `print(events)`, and note the *row names* on the left. We could print the second row with either of these statements:

```
print(events["2",])
print(events[2,])
```

The first says print the row with row name "2", the 2nd says print the second row. Now try:

```
tmp <- events[-1,]
print(tmp["2",])
print(tmp[1:2,])
```

The first statement deletes the first row. Hence, now the first row is that record with row name "2", whereas the 2nd record has row name "3".

4.3 Factors

A factor is essentially a coded variable, usually a character variable. Representing character strings, particularly long ones, means the dataset not only occupies less disk space, but since it is smaller, it can be processed much faster.

1. Consider again the earthquake listed in Appendix B.1. We read them again using `read.csv`.

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"
events <- read.csv(file=fname, as.is=TRUE)
print(is.data.frame(events))
print(mode(events$month))
```

This creates a data frame called “events”. The variable `events$month` is character because of the option `as.is=TRUE`. If this was excluded, then it defaults to a factor, sometimes intensely irritating.

2. Now we create a new variable (keep the old for comparison) within the events object called “month1” which is a factor:

```
events$month1 <- as.factor(events$month)
print(events$month1)
print(as.numeric(events$month1))
```

Notice that the number ascribed to each month depends on their rank when the months are sorted alphabetically! Hence April is month 1, August month 2, etc. Not what we want!

3. We actually want months to be an ordered factor, i.e. there is a natural ordering that does not coincide with the alphabetical ordering. Hence, try this:

```
# R has a built in vector of month abbreviations
print(month.abb)
events$month <- factor(events$month, levels=month.abb, ordered=TRUE)
print(events$month)
print(as.numeric(events$month))
# remove the incorrect representation of months
events$month1 <- NULL
```

4. Note that a factor is stored as a numeric variable but it has some “attributes” which tell R how it must interpret these numbers:

```
print(attributes(events$month))
print(mode(events$month))
```

Note that the attributes tell us that it is an ordered factor, along with the correct ordering of the levels. The mode says it is stored as a number coinciding with the level, hence:

```
print(events$month)
print(as.numeric(events$month))
```

5. Now assume that we want to divide the depth into two categories, “deep” and “shallow”. Say an event is shallow if its depth ≤ 40 km. This can be achieved as follows:

```
events$depth.cat <- c("deep", "shallow")[(events$depth<=40)+1]
print(events$depth.cat)
print(is.character(events$depth.cat))
```

Note that the “data” are stored as character strings, i.e. “deep” or “shallow”, which would be inefficient for a large dataset. Convert the variable `events$depth.cat` into a factor as follows:

```
events$depth.cat <- as.factor(events$depth.cat)
```

Recall that, by default, the levels will be sorted alphabetically.

6. If there is a natural ordering to the factor levels (e.g. “young”, “middle age” and “old”), then one should use an ordered factor. We encountered *ordered factors* in §3.3 too.

4.4 Attributes

A variable can have a number of attributes. These are characteristics of the variable, and some determine the manner in which the variable is printed, and so on.

1. Consider the example in §4.3, in particular, the variable `events$depth.cat`. The attributes can be printed as:

```
print(attributes(events$depth.cat))
```

Note that there are two attributes, `levels` and `class` as follows:

```
$levels
[1] "deep"      "shallow"

$class
[1] "factor"
```

The levels were discussed in §4.3. The “class” attribute is a central concept in the R language, and determines the manner in which other functions interact with this variable. This is discussed further in §4.5.

2. We can also attach our own attributes using the `attr` function. Again, consider the data in §4.3. The magnitudes are on a “local” scale. This information could be attached to the variable `events$magnitude` as follows:

```
attr(events$magnitude, "magn.type") <- "local"
print(events$magnitude)
print(attributes(events$magnitude))
```

Hence, we have attached an attribute called `"magn.type"` which has a value of `"local"`.

3. The longitude vector can be given a note to say that they are represented as degrees east, and positive latitudes are degrees north:

```
attr(events$longitude, "note") <- "Degrees East"
attr(events$latitude, "note") <- "Degrees North"
print(events$longitude)
print(attributes(events$longitude))
print(events$latitude)
print(attributes(events$latitude))
```

4. Consider the matrix `colours` from §1.2.2:

```
colours <- matrix(c("red", "blue", "green", "cyan", "yellow", "magenta"),
                 byrow=TRUE, nrow=2)
print(attributes(colours))
```

The `colours` object only has one attribute, `dim`, being the dimensions of the matrix.

5. Column and row names can be added to the matrix as:

```
dimnames(colours) <- list(c("Primary", "Secondary"), c("R", "B", "G"))
print(colours)
print(attributes(colours))
```

Note that the row and column names are actually stored as attributes.

4.5 Class Attribute, Generic and Method Functions

When analysing data, we often want to *print*, *plot* or create a *summary* of the dataset. These can be thought of as *generic* operations. However, the most appropriate way to print, plot or summarise the data will often depend on the type of data. This is achieved by giving the data object a *class*, which then determines the *method* that is used to print, plot or summarise the data. There are many generic like operations: calculate the model likelihood, plot model residuals, etc.

1. We again refer to the dataset in §4.3. Enter `class(events$depth.cat)`, it will be `"factor"`.
2. Note that whenever an object name is entered on the command line followed by *Enter*, for example `events$depth.cat`, it is interpreted as `print(events$depth.cat)`.
3. The function `print` is *generic*. When `print(object)` is entered on the command line, the print function checks to see what the class of `object` is. If `object` has no set class, the function `print` issues the command

```
print.default(object)
```

4. Since `class(events$depth.cat)` is `"factor"`, then the `print` function looks for another function called `print.factor`, and since there is such a function, it issues the command `print.factor(events$depth.cat)`. It is this function that causes `events$depth.cat` to be printed using the values `deep` and `shallow` rather than one's and two's.
5. The function `print.factor` is referred to as the *method* to be used on objects of class `"factor"` by the *generic* function `print`.
6. The function `print.default` will print the object without utilising the method for the defined class. For example,

```
print.default(events$depth.cat)
```

will give a vector of one's and two's, because this is how the data are stored.

7. It is possible to define our own generic functions and the associated methods. One can also add methods for system supplied generic functions. For example, say we had a peculiar model object that we wanted to print in a particular format. Say the model object is called `xyz`, and we gave our model a class called `"peculiar"`. This can be done as follows:

```
xyz <- seq(0, 10, 2)
class(xyz) <- "peculiar"

# we need to define how we want to print this
print.peculiar <- function(x, ...){
  cat("This is a very peculiar model object\n\n")
  cat(x^2, "\n")
}

# now print as:
print(xyz)
```

8. Objects can have multiple classes. Hence a generic function will initially search for the appropriate method for first class. If this is not found, it will search for an appropriate method for the next, and so on. If there is no method found, it will simply use the default method for the generic operation.
9. The R software is sometimes rather too eager to turn any character variable into a factor. At times this can be terribly tiresome! Consider again the dataset in §2.1.2:

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/example1.txt"
NZ1 <- scan(file=fname, what=list(latitude=0, longitude=0,
    event="", depth=0, magnitude=0, day=0, month=0,
    year=0, hour=0, minute=0, second=0), sep="")
print(NZ1)
print(is.data.frame(NZ1))
```

If we were to coerce this into a data frame, the variable `NZ1$event` would be turned into a factor. However, the event names are unique and not repeating, so there is little efficiency in storing these data as a factor. We can give the variable `NZ1$event` a class of `AsIs`, which tells R to leave it alone whenever it feels the urge to convert it into factor:

```
NZ1$event <- I(NZ1$event)
print(class(NZ1$event))
NZ1 <- as.data.frame(NZ1)
print(NZ1)
print(is.factor(NZ1$event))
```

Then the option `stringsAsFactors=FALSE` does not need to be specified in the call to `as.data.frame`.

Chapter 5

Some Standard Statistical Analyses

5.1 ANOVA

The data analysed here are the diameters (cm) of Kauri saplings at a height of 1.8m in 3 different blocks. A one way ANOVA is effectively a comparison of the mean diameters between the 3 blocks. The data are listed in Appendix B.2. Notice that they are actually included within R statements (i.e. it is R code). Assume that this code is in a file called “kauri.R”. The code is executed by the statement `source("kauri.R")` below.

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/kauri.R"
source(fname)
n1 <- length(x1)
n2 <- length(x2)
n3 <- length(x3)

kauri <- list()
kauri$diam <- c(x1, x2, x3)
kauri$block <- as.factor(c(rep(1, n1), rep(2, n2), rep(3, n3)))
kauri <- as.data.frame(kauri)

boxplot(diam ~ block, data=kauri)
z <- lm(diam ~ block, data=kauri)
print(summary(z))
hist(residuals(z))
boxplot(residuals(z) ~ kauri$block)
```

Multiway ANOVAs would be specified by including more factors (covariates too if required) in the model formula.

5.2 Generalised Linear Models

Here we use the generalised linear modelling framework to fit a very simple point process model.

A simple Poisson point process has events occurring completely randomly with a constant rate parameter. In this case the inter-event time will have an exponential distribution. A non-homogeneous process has a non-constant rate; it may vary with time. One can see that the number of events here increases over time, hence the inter-event time decreases.

We model the inter-event time below as an exponential distribution, but where the rate changes as a linear function of time. Note that the exponential distribution is a special case of the gamma distribution. We start with the code as in §3.4.

```
fname <- "ftp://ftp.gns.cri.nz/pub/davidh/Rworkshop/earthquakes.csv"
events <- read.csv(fname, as.is=TRUE)
events$month <- factor(events$month, levels=month.abb, ordered=TRUE)
event_date <- as.Date(paste(events$day, events$month, events$year, sep=""),
                      "%d/%b/%Y")
num_days <- julian(event_date, origin=as.Date("1960-01-01")) +
             events$hour/24 + events$minute/(24*60) +
             events$second/(24*60*60)
events$event_date <- event_date
events$num_days <- num_days
events$diff_days <- num_days - c(0, num_days[-length(num_days)])
events <- as.data.frame(events)

z <- glm(diff_days ~ num_days, family=Gamma(link=identity), data=events)
print(summary(z))
par(mfrow=c(2,1))
plot(events$num_days, events$diff_days, type="l")
abline(a=z$coefficients[1], b=z$coefficients[2], col="blue")
plot(events$num_days, residuals(z), type="l")
```

This family of models is quite large. It also includes the ANOVA models (as above), but also binomial logistic models, log-linear models, standard regression models, etc. They can all be included into this framework. For a general text, see the book by McCullagh & Nelder (1989).

5.3 Other Models

An extensive range of different models can be fitted to data. The usual regression model fits into the ANOVA and GLIM frameworks above, where the right hand side of the model formula contains the name of the covariate. The models can also be mixed, and contain interactions (see topic `formula` in the help documentation), for example:

```
response ~ factor1*covariate1 + factor1*factor2
```

contains two factors and a covariate. There is effectively a different slope in the line for each level of `factor1`.

Time series models can also be fitted, for example, see functions `ts`, `arima`, `stl`, and `spec.pgram`. For principal components, see `princomp` and `prcomp`. For further details on these and other models, look in the section “Keywords” on the front page of the `help.start()` pages.

Appendix A

Appendix: Software Installation

A.1 Installing Base R Software

The easiest way to install R is to download the required *binary*, a compiled version of the software that is compatible with your operating system. These binaries can be found on CRAN (Comprehensive R Archive Network); this web address will direct you to the closest mirror: <http://cran.rstudio.com/>. There are binaries for various versions of Linux, Windows and MacOS. Some distributions of Linux (e.g. Fedora and Debian) have R included within their distribution, and can be installed directly from that source.

If a binary is not available for your operating system or you want to configure the installation in a non-standard way, then you need to install the software using the source code; also available from CRAN.

A.2 Installing Contributed Packages

There are about 5000 contributed packages on CRAN. R is the main software used by research statisticians and applied probabilists, hence there is an enormous pool of contributors. The software is also used by many applied data analysts including Google.

Information about how to write an R package is given by [R Development Core Team \(2014g\)](#).

UNIX Like Environment

In UNIX like environments, the packages get installed using the source code. They are generally installed by the “root” user, and will be placed in the system directories with the R base software. Consider a package called “xyz”, version 2.2-3. Its source code will be in a file called “xyz_2.2-3.tar.gz”. To install, one logs on as “root”, and runs the following command from within the directory that contains the downloaded file “xyz_2.2-3.tar.gz”:

```
R CMD INSTALL xyz_2.2-3.tar.gz
```

Note that during installation from source code, the R software finds the most appropriate compilers on the system (FORTRAN and C) and automatically compiles any source code that is included within the package. Further, the created binaries are automatically linked into the R system when the package is required by a user. To subsequently remove the package, one logs on as “root” and runs the following command:

```
R CMD REMOVE XYZ
```

Microsoft Windows

Packages that can be found on CRAN can be installed easily using the GUI menus within the R session. Click on *Packages* then *Install Packages*, then follow the instructions. Similarly, installed packages can be updated by clicking on *Packages* then *Update Packages*

For packages from a non-standard origin, download the appropriate “.zip” file. This is not simply a zip compressed file of the package source code. It is a zip compressed file of the binary compiled version of the package. The binary includes compiled FORTRAN and C objects, compiled html code, etc, all of which is not included within the package source code. Within R, and click on the *Packages* menu tab, then click on *Install packages from local zip files* Select the downloaded “.zip” file, and install.

Other Options

Packages can also be installed and updated by running an R script. The basics of the required commands are given here.

```
# windows binary location
if (options()$pkgType=="win.binary")
  repos <- "http://cran.stat.auckland.ac.nz/bin/windows/contrib/r-release/"

# source code location
if (options()$pkgType=="source")
  repos <- "http://cran.stat.auckland.ac.nz/src/contrib/"

# list packages available for update
tmp <- packageStatus(repositories=repos)
vars <- c('Version', 'Built', 'Status', 'LibPath')
print(tmp$inst[(tmp$inst$Status=='upgrade'), vars])
```

Some listed updates can be installed (*see exceptions below*) by entering:

```
upgrade(tmp)
```

All of the available packages in the repository can listed by entering:

```
available.packages(contriburl=repos)[,1:2]
```

Packages on your machine that were built under an earlier version of R can be reinstalled (*see exceptions in bullet points below*) by entering:

```
update.packages(contriburl=repos, checkBuilt=TRUE, ask=TRUE)
```

Packages can be installed from the repository also (*see exceptions below*); for example, to install the package *chron*, enter:

```
install.packages(pkgs="chron", contriburl=repos)
```

Note the following: Updating or installing could also be blocked by firewalls on your local network. In these situations it may need to be done manually. In an institutional computing environment you may not have administrative rights to install software (both UNIX and Windows). An alternative is to install the packages into your local home directory and modify the search path in R.

For other installation options, including a local install (as non “root”), see [R Development Core Team \(2014d\)](#).

Appendix B

Appendix: Example Datasets

B.1 Large NZ Earthquake Events

These data are sourced from the NZ catalogue (GeoNet), and are events between 1 Jan 1960 and 31 Dec 2000 with magnitude ≥ 6.5 . The listed dates and times are UTC.

```
latitude,longitude,depth,magnitude,day,month,year,hour,minute,second
-39.04000,174.8100,237.0000,6.600,27,Mar,1960,23,28,26.4
-39.04000,174.7700,209.0000,6.500,27,Mar,1960,23,29,45.8
-32.41000,179.3900,579.0000,6.950,18,Jun,1961,13,55,22.7
-32.43000,181.0400,200.0000,6.550,05,Aug,1964,11,06,01.8
-37.00000,177.6300,190.0000,6.500,08,Dec,1965,18,05,24.0
-41.76000,172.0400,12.0000,6.700,23,May,1968,17,24,17.4
-34.94000,179.3000,297.0000,6.813,08,Jan,1970,17,12,36.6
-39.04000,175.2600,163.0000,7.000,05,Jan,1973,13,54,28.7
-29.22000,184.0600,33.0000,6.500,02,Jul,1974,23,26,27.5
-44.67000,167.3800,12.0000,6.550,04,May,1976,13,56,29.2
-23.19000,184.0800,33.0000,7.200,22,Jun,1977,12,08,28.3
-51.68000,159.2400,33.0000,6.627,07,Jul,1982,10,42,54.1
-30.92000,182.5200,271.0000,7.250,26,Jan,1983,16,02,13.7
-34.64000,181.9800,33.0000,6.957,26,Sep,1985,07,27,49.9
-28.34000,184.3000,33.0000,6.860,20,Oct,1986,06,46,09.5
-36.66094,177.2227,258.6172,6.596,25,May,1989,13,01,35.5
-31.55958,182.4913,232.9927,7.053,21,Mar,1990,16,45,59.6
-36.73566,177.3689,203.0520,6.506,22,Mar,1990,00,00,18.0
-33.00895,180.4838,417.0962,6.820,20,May,1990,09,53,46.6
-41.60475,173.6605,79.1020,6.757,27,May,1992,22,30,36.4
-31.20000,179.9400,395.0000,7.103,08,Aug,1992,01,08,04.8
-34.15421,180.4897,242.9395,6.714,14,Dec,1992,19,12,55.3
-45.21383,166.7085,5.0000,6.702,10,Aug,1993,00,51,51.6
-43.00782,171.4762,4.2668,6.670,18,Jun,1994,03,25,14.6
-37.64958,179.4886,12.0000,6.991,05,Feb,1995,22,51,02.3
-37.91566,179.5141,12.0000,6.570,10,Feb,1995,01,44,56.3
-33.21012,180.6300,432.4891,6.720,03,Jun,1995,20,58,59.2
-39.61305,174.2650,205.8918,6.551,19,Sep,1995,22,52,24.7
-31.16000,180.0000,369.0000,7.288,05,Nov,1996,09,41,34.4
-32.54943,182.0185,135.4866,7.113,03,May,1997,16,46,01.8
-32.30516,181.2100,339.3824,7.875,25,May,1997,23,22,30.6
-39.02385,174.9245,231.7755,6.756,20,Apr,1998,23,34,18.1
-32.12886,184.2260,330.9584,7.172,09,Jul,1998,14,45,37.1
-32.45824,182.4867,207.6340,7.005,20,Apr,1999,19,04,04.5
-38.58762,175.4784,264.4511,6.528,18,May,1999,09,19,35.7
-38.57127,175.9141,160.8110,6.952,25,Oct,1999,20,31,42.5
-31.72232,181.6539,392.3979,7.196,08,May,2000,21,35,41.0
```

-31.94136,181.4547,436.1739,7.649,15,Aug,2000,04,30,05.0

B.2 Kauri Saplings

```
# Diameter (cm) of kauri saplings at 1.8m height

# In block 1
x1 <- c(7, 7.7, 5.6, 7.2, 7, 7.2, 6.6, 5.7, 5.8, 5, 8.3, 7.7,
        5.4, 7.5, 6.8, 5.6, 7.4, 8.6, 5.2, 7.5, 7.1, 7.7, 8.2,
        5.6, 8.3, 6.9, 7.4, 8.3, 7.1, 7.2, 6.2, 6.2, 7.6, 5.9,
        7.5, 7.2, 5.8, 8, 9.3, 6.2, 5.3, 8.7, 5.3, 6.6, 5.5,
        6.8, 5.9, 5.7, 5.3, 8, 7.2, 7, 8.7, 5.7, 7.5, 6.6, 6.6,
        7.3, 5.7, 6.6, 7.2, 8.5, 7.5, 6.9, 5.8, 6.8, 7.7)

# In block 2
x2 <- c(10.2, 9, 9.2, 6.2, 7.2, 7.3, 10.9, 7.9, 9.5, 8.1, 8.3,
        8.8, 9.9, 10.2, 7, 8.1, 9.2, 8.2, 8, 8.6, 9, 9.4, 8.3,
        8, 8.7, 8.6, 8.2, 9.8, 10.6, 7.6, 9.6, 9.1, 7.2, 7.3,
        9.9, 9.2, 9, 7.8, 8.9, 6.6, 8.9, 7.7, 8.5, 8.6, 7.5,
        10.2, 8.9, 9.1, 6.6, 10, 7.9, 7.5, 7.2, 10.5, 6.6, 8.5,
        8.5, 5.9, 8.7, 8.8, 9.3, 8.8, 9.1, 7.6, 9.8, 8.2, 9.9,
        9.5, 7.4, 8, 10, 10.9, 7.8, 8.7, 9, 9.1, 9.7, 8, 9.2, 9.5)

# In block 3
x3 <- c(7.4, 8.9, 6.4, 9.2, 9.7, 8.4, 8.9, 9.8, 9.4, 8.3, 6.5,
        7.9, 7.3, 7.8, 8.5, 8.2, 7.3, 6.5, 7.9, 6.9, 7.1, 7, 9.1,
        6.6, 8.1, 7.7, 9.8, 8.5, 7.9, 8, 8.7, 8.8, 8.8, 7.3, 7.6,
        7.6, 8.7, 7.9, 9.4, 10.2, 7.8, 7.6, 7.1, 6.3, 8.8, 7.4,
        9.1, 7.5, 8.6, 8.2, 6.8, 8, 9.4, 8, 7.7, 9.7, 7.8, 8.5,
        6.4, 9, 8, 8.7, 8.9, 7, 6.4, 6.4, 8.4, 8.8, 8.9, 5, 7.1,
        8.5, 8.6, 7, 7.5, 8.4, 10.9, 8.8, 8, 7.9, 7.4, 9.1, 8.5,
        7.8, 7.8, 8.3, 7.5, 7.1, 8.1, 7, 8.4)
```

References

- Chambers, J.M. & Hastie, T. (1991). *Statistical Models in S*. Wadsworth and Brooks-Cole, Pacific Grove CA.
- Ihaka, R. & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* **5**(3), 299–314.
- Maindonald, J. & Braun, J. (2003). *Data Analysis and Graphics Using R - An Example-Based Approach*. Cambridge University Press, Cambridge. ISBN: 0521813360.
- McCullagh, P. & Nelder, J.A. (1989). *Generalized Linear Models (2nd Edition)*. Chapman and Hall, London.
- R Development Core Team. (2014a). *An Introduction to R*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-intro.pdf>.
- R Development Core Team. (2014b). *R: A Language and Environment for Statistical Computing. Reference Index*. R Foundation for Statistical Computing, Vienna. ISBN 3-900051-07-0. URL: <http://cran.r-project.org/doc/manuals/fullrefman.pdf>.
- R Development Core Team. (2014c). *R Data Import/Export*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-data.pdf>.
- R Development Core Team. (2014d). *R Installation and Administration*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-admin.pdf>.
- R Development Core Team. (2014e). *R Internals*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-ints.pdf>.
- R Development Core Team. (2014f). *The R Language Definition*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-lang.pdf>.
- R Development Core Team. (2014g). *Writing R Extensions*. R Foundation for Statistical Computing, Vienna. URL: <http://cran.r-project.org/doc/manuals/R-exts.pdf>.
- Statistical Sciences Inc. (1992). *S-PLUS Programmers Manual, Version 3.0*. Statistical Sciences Inc, Seattle.