

Learning R

Peter K Dunn

Contents

1	Introduction	1
1.1	Why learn R?	1
1.2	About R	2
1.2.1	What is R?	2
1.2.2	What are the differences between S-Plus and R?	3
1.3	Basic R commands	3
2	Basic arithmetic in R	5
3	Getting help in R	9
3.1	Help in R itself	9
3.2	Help outside R	9
3.3	Using R functions	10
3.3.1	An initial look	10
3.3.2	Another look	12
4	Loading data	15
4.1	Introduction	15
4.2	Entering data directly	15
4.3	Loading text data files	16
4.4	More on loading data	17
4.5	Data frames	19
4.6	Examining data	20
4.6.1	Initial examination of the data frame	20
4.6.2	Initial examination of the variables	21
4.6.3	Initial plotting of the data	22
5	Matrices in R	27
6	Plotting and graphics in R	29
6.1	Plotting data	29

6.2	Plotting functions	31
7	Simple analyses	33
7.1	Formula notation	33
7.2	Function output	35
8	How to run your code in R	39
9	More on plotting	41
9.1	Introduction	41
9.2	Adding labels	42
9.3	Changing the plotting characters	43
9.4	Changing colours	43
9.5	Changing the format of the axes	43
9.6	Selecting parts of a data frame	44
9.7	Adding a legend	45
9.8	Saving figures	46
9.9	Other options	47

Chapter 1

Introduction

1.1 Why learn R?

- R is available for most operating systems (including Windows, Mac, and most flavours of Linux including BSD)¹.
- R is open source.
- Because R is open source, the code is freely available to anyone. This has the advantages that:
 - Anyone can examine the code to find (and fix) errors;
 - Unlike some statistical (and mathematical) packages, it is possible to know *exactly* what the software does by looking at the code.

This means the software has been examined by many experts to ensure quality of the statistical procedures.

- Because R is free, it is easy and cheap to ensure you have the most up-to-date versions.
- R is very powerful.
- R is extendable, with numerous add-on packages available.
- Because R is open source and used by many active and renowned scientists and statisticians, R is often the software of choice for implementing new ideas.

¹For example, SPSS is not available for Macs, and only for some flavours of Linux, and not for BSD

- Many research institutions, universities and commercial companies use R.
- R is actively supported through a variety of means (see Section 3).
- R is actively supported by many well-known and accomplished statisticians and scientists.
- R is programmable: if R can't do a particular task, you can *program* R to do it.
- R produces publication quality graphics.
- In many areas, R is (or is becoming) the standard environment for analysis, notably bioinformatics (using the BioConductor package for R).
- R is an object-oriented language.

1.2 About R

1.2.1 What is R?

R is a statistics package, or a statistics environment. It is a command-line package, in a similar way that MATLAB is command-line driven, unlike many popular statistics packages.

R is based on the S language, a very high level language and an environment for data analysis and graphics. In 1998, the Association for Computing Machinery (ACM) presented its Software System Award to John M. Chambers, the principal designer of S, for

- the S system, which has forever altered the way people analyse, visualise, and manipulate data. . .
- S is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.

The two main expression of the S language are:

- S-Plus, a commercial (and often expensive) implementation of S; and
- R, an free, Open Source implementation of S.

1.2.2 What are the differences between S-Plus and R?

Since both S-Plus and R are implementations of S, they are very similar. There are some differences, however:

- S-Plus is commercial; R is free and Open Source.
- There are a few differences in the language also.
- S-Plus has a fancier GUI.
- S-Plus often has fancier graphics capabilities.
- R has some useful functions not found in S-Plus (such as adding Greek letters to plots). Likewise, S-Plus has some functionality not found in R.
- S-Plus has many (additional cost) add-ons that extend its functionality beyond R, or are free in R.
- R is available “out of the box” on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux). It also compiles and runs on Windows and MacOS. In contrast, S-Plus is available for Windows and some flavours of Linux, but not MacOS.

The functionality of R is growing rapidly, and for most purposes they can be considered together apart from the interface. Low-level users might never notice the differences.

1.3 Basic R commands

- Once R has been started, you can quit by typing `q()`.
- R is command-line based (like, say, MATLAB), not point-and-click (like, say, SPSS or Word).
- In the command window, a `>` character at the far left (the ‘command prompt’) indicates R is awaiting your instructions.

Chapter 2

Basic arithmetic in R

- Arithmetic is performed as one would expect:

```
> 3^2 * exp(2) - 1
```

```
[1] 65.5015
```

```
> 67/(log(2) + 1)
```

```
[1] 39.57128
```

```
> 2 * pi
```

```
[1] 6.283185
```

- R can do complex arithmetic; try, for example, $(1 - 2i) * (-3 - 1i)$

You must tell R you wish to do complex arithmetic; compare the output of

```
> sqrt(-1)
```

```
[1] NaN
```

with the output of

```
sqrt(-1 + 0i)
```

```
[1] 0+1i
```

- R is case-sensitive. This means that `abc`, `ABC` and `aBc` are different.

- All commands must be followed by parentheses; usually, these parentheses contain arguments to the function. But, as the example above for `q()` shows, even if there are no arguments to the function the parentheses are necessary (otherwise we see the function definition, which is scary as well as not what we want).
- Round brackets are used for function arguments; square brackets are used for indexing vectors and matrices.
- R works happily with *variable* names. Variables must start with a letter, but can contain numbers and dots (and, more recently, underscores). It is common in R to use the period in variable names; for example, `model1.residual` is a valid variable name. Numbers can be used in variable names, except as the first character. The following are all valid variable names: `x3`, `x2.2`, `D.2.x.2`
- Variables can be defined using the assignment operator `<-`

```
> radius <- 3
> circumference <- 2 * pi * radius
> area <- pi * radius^2
> area

[1] 28.27433

> circumference

[1] 18.84956
```

Note that spacing is not important: `x <- 2` is the same as `x <-2`. However, use spaces wisely for readability¹.

- Assignment is also allowed using `=`, though `<-` is preferred. All these are the same
 - `x <- 3` (Best)
 - `x = 3` (OK)
 - `3 -> x` (OK)
 - Not so many years ago, this syntax was also valid: `x _ 3` which is very hard to read. This syntax is being phased out (thankfully).

¹But note: `x<-3` can be interpreted two ways: `x < (-3)` or `x <- (3)`. So use spaces wisely and carefully!

You can even do things like this: `x <- 2 -> y` (but this looks awful) and `x <- y <- z <- 2`.

- Strings are usually given in double quotes (though single quotes work in more recent versions too); for example: `plot.title <- "Title for my plot"`
- Comments are indicated by the `#` character: Any text following this character is ignored by R. This is useful for explaining what your commands do.

```
area <- pi * radius^2    # Circle area
```

- Two or more commands can be entered on a line at once if separated by a semi-colon; try entering this all on one line in R:
`length <- 2; width <- 5; length * width`
- In the command window, a `+` character means R is waiting for you to *complete* an instructions that is unfinished (for example, brackets are unmatched).
- Commands not completed are automatically continued on to the next line with the `+` prompt. For example, try typing the incomplete command `5 + log(` and then pressing RETURN. The command prompt will change to `+` when you should type 1) so R will have a complete command; it will then give you an answer.
- R is a vector-based language, and so instructions generally work with vectors as easily as scalars:

```
> x <- seq(-3, 3, by = 0.25)
> x
```

```
[1] -3.00 -2.75 -2.50 -2.25 -2.00 -1.75 -1.50 -1.25 -1.00 -0.75 -0.50
[12] -0.25  0.00  0.25  0.50  0.75  1.00  1.25  1.50  1.75  2.00  2.25
[23]  2.50  2.75  3.00
```

```
> x^2
```

```
[1] 9.0000 7.5625 6.2500 5.0625 4.0000 3.0625 2.2500 1.5625 1.0000
[10] 0.5625 0.2500 0.0625 0.0000 0.0625 0.2500 0.5625 1.0000 1.5625
[19] 2.2500 3.0625 4.0000 5.0625 6.2500 7.5625 9.0000
```

```
> x + 10
```

```
[1] 7.00 7.25 7.50 7.75 8.00 8.25 8.50 8.75 9.00 9.25 9.50
[12] 9.75 10.00 10.25 10.50 10.75 11.00 11.25 11.50 11.75 12.00 12.25
[23] 12.50 12.75 13.00
```

- A list of the current variables can be found using `objects()` or `ls()`. It is good practice to remove unwanted variables using `rm(variable.name)`.
- When calling functions, the names of the input variable can be used explicitly (this is unlike MATLAB). For example, `plot(x, y, xlab="The x-axis label")` explicitly assigns a value to `xlab` wherever it appears in the definition of the function `plot`. More on this in Section 3.3.2.

Chapter 3

Getting help in R

3.1 Help in R itself

- Open an html search page using `help.start()`
- If you *do not know* the function name, try, for example, `RSiteSearch("regression")`. This function searches for key words or phrases in the R-help mailing list archives, or R manuals and help pages, using the search engine at <http://search.r-project.org> and enables them to be viewed in a web browser.
- If you *do not know* the function name, another way is to try, for example, `help.search("regression")`.
- If you *do know* the function name, try, for example, `help("lm")` or `?lm`
- You can also try using `example`; for example, `example(lm)`.
- Demonstration of R capabilities are given using `demo()`; for example, `demo(graphics)`. Type `demo()` for a list of available demonstrations.

3.2 Help outside R

- You can search the web pages:
 - The R Project at <http://www.r-project.org/>
 - The Comprehensive R Archive Network (CRAN) at <http://cran.au.r-project.org/>
 - Search the mailing list of three R groups: <http://cran.au.r-project.org/search.html>

- The library has books on R. (Most books about S-Plus are also relevant; see Section 1.2). Some useful books include (in subjective rank order from very useful to useful):
 - Dalgaard, Peter. *Introductory statistics with R*, New York: Springer, 2002. (519.5 Dal)
 - Maindonald, John and Braun, John. *Data Analysis and Graphics Using R: An Example-based Approach*, Cambridge University Press, 2003. (519.50285 Mai)
 - Krause, Andreas and Olson, Melvin. *The basics of S and S-Plus*, New York: Springer, 1997. (519.50285 Spl/Kra). This book is also available as an e-book through the library catalogue.
 - Venables, W. N. and Ripley, B. D. *Modern applied statistics with S-PLUS*, New York: Springer, 1999. (519.5028553 Ven)

3.3 Using R functions

3.3.1 An initial look

Suppose we wish to create a sequence (like $-1, 0, 1, 2, \dots$). To find such a function, type

```
help.search("seq")
```

`seq` appears to be the appropriate function.

Part of an example help page for `seq`, obtained by typing `?seq` at the R prompt, is shown in Figure 3.1.

Here is a brief explanation:

- Line 5: A brief description of what the function does
- Lines 9–18: How to use the function.
- Lines 20–32: The argument to the function, and what they mean.
- Lines 38–42: Other function that might be relevant.
- Lines 44–63: Some examples of using the function. When the command `example(seq)` is given, these commands are executed. Sometimes, these example are for non-beginners.

We learn that `seq` is for generating regular sequences, like: 0, 1, 2, 3, 4, 5. Consider the following:

```
1  seq                                package:base                                R Documentation
2
3  Sequence Generation
4
5  Description:
6
7      Generate regular sequences.
8
9  Usage:
10
11      from:to
12          a:b
13
14      seq(from, to)
15      seq(from, to, by= )
16      seq(from, to, length.out= )
17      seq(along.with= )
18      seq(from)
19
20  Arguments:
21
22      from: starting value of sequence.
23
24      to: (maximal) end value of the sequence.
25
26      by: increment of the sequence.
27
28  length.out: desired length of the sequence.
29
30  along.with: take the length from the length of this argument.
31
32      a,b: 'factor's of same length.
33
34  ---      <snip> ---
35
36  See Also:
37
38      The method 'seq.POSIXt'.
39
40      'rep', 'sequence', 'row', 'col'.
41
42      As an alternative to using ':' for factors, 'interaction'.
43
44  Examples:
45
46      1:4
47      pi:6 # float
48      6:pi # integer
49
50      seq(0,1, length=11)
51      seq(rnorm(20))
52      seq(1,9, by = 2) # match
53      seq(1,9, by = pi)# stay below
54      seq(1,6, by = 3)
55      seq(1.575, 5.125, by=0.05)
56      seq(17) # same as 1:17
57
58  <snip>
```

- Generally, the input arguments are *named*:

```
> seq(from = 0, to = 5, by = 1)
```

```
[1] 0 1 2 3 4 5
```

- If the arguments appear in the order given, they need not be named:

```
> seq(0, 5, by = 1)
```

```
[1] 0 1 2 3 4 5
```

- If the arguments are named, they can appear in *any* order:

```
> seq(to = 5, by = 1, from = 0)
```

```
[1] 0 1 2 3 4 5
```

- A variation is:

```
> seq(0, 5, length = 6)
```

```
[1] 0 1 2 3 4 5
```

- Since this quite a common command, we can also use:

```
> 0:5
```

```
[1] 0 1 2 3 4 5
```

3.3.2 Another look

Consider the help for the R function `rnorm`, for generating random numbers from a Normal distribution. This help page also discusses the help for other functions related to the Normal distribution; see Figure 3.2.

Observe the following¹:

- To generate two random numbers from $N(0, 1)$, use:

```
> rnorm(n = 2, mean = 0, sd = 1)
```

```
[1] -0.549638  1.091054
```

¹Note that, since random numbers are used, each time the function is run produces a different value, and you will see different values if you type these commands in yourself. If you want to set the random number seed (and so have reproducible ‘random’ numbers), use `set.seed()`, and place some integer in the parentheses; for example `set.seed(518768)`.

```
1 Normal                package:stats                R Documentation
2
3 The Normal Distribution
4
5 Description:
6
7     Density, distribution function, quantile function and random
8     generation for the normal distribution with mean equal to 'mean'
9     and standard deviation equal to 'sd'.
10
11 Usage:
12
13     dnorm(x, mean=0, sd=1, log = FALSE)
14     pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
15     qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
16     rnorm(n, mean=0, sd=1)
17
18 Arguments:
19
20     x,q: vector of quantiles.
21
22     p: vector of probabilities.
23
24     n: number of observations. If 'length(n) > 1', the length is
25     taken to be the number required.
26
27     mean: vector of means.
28
29     sd: vector of standard deviations.
30
31 log, log.p: logical; if TRUE, probabilities p are given as log(p).
32
33 lower.tail: logical; if TRUE (default), probabilities are P[X <= x],
34     otherwise, P[X > x].
35
36 Details:
37
38     If 'mean' or 'sd' are not specified they assume the default values
39     of '0' and '1', respectively.
40
41 --- <snip> ---
42 Value:
43
44     'dnorm' gives the density, 'pnorm' gives the distribution
45     function, 'qnorm' gives the quantile function, and 'rnorm'
46     generates random deviates.
47
```

Figure 3.2: Part of the help for the function `rnorm`

- In the definition, the default values for the mean and standard deviation are specified as 0 and 1 (see Line 16) respectively; so we could use:

```
> rnorm(n = 2)

[1] 0.6397777 1.0425760
```

- Since arguments need not be named if given in the specified order, we could use:

```
> rnorm(2)

[1] 0.1697033 1.1378004
```

- We could, of course, type:

```
> rnorm(mean = 0, n = 2, sd = 1)

[1] -0.9705545 -0.1318273
```

If we produce a lot of random numbers you see how R displays long vectors:

```
> rnorm(30)

[1] 0.14622623 1.44129222 -2.94138513 -0.24283860 -0.14058509
[6] -0.03265396 0.27981588 0.59014791 1.02429531 2.10732266
[11] 0.15463424 0.91306984 -0.25422449 1.51943184 1.78126982
[16] -0.87884444 -1.52921818 0.13591141 -0.70889985 -1.40957238
[21] 1.83093656 1.29019398 -2.36243538 -0.55097480 -0.30466297
[26] -0.74996129 0.14372175 -0.54943140 0.15992394 -0.08767391
```

The numbers in brackets on the far-left indicate the element number in the vector.

Chapter 4

Loading data

4.1 Introduction

The main purpose of R is analysis of data. Therefore, it is vitally important to be able to load data. Fortunately, R is very flexible about importing data. Certain functions can be used to read and write data stored in a variety of text formats, as well as data saved by statistical packages such as Minitab, S, SAS, SPSS, Stata, Systat, and for reading and writing `.dbf` (dBase) files. Data can also be entered directly into R.

4.2 Entering data directly

Data can be entered directly into R using the function `c()` (for concatenate):

```
> attendance <- c(9, 10, 15, 13, 12)
> attendance
```

```
[1]  9 10 15 13 12
```

The function `scan()` can also be used to read the data line-by-line from the console (input is terminated by entering a *blank line*):

```
> attendance <- scan()
9
10
15
13
12
```

```
Read 5 items
> attendance
```

```
[1]  9 10 15 13 12
```

4.3 Loading text data files

Most users load their data into R from text files. Consider this (artificial) data set, named `datafile1.dat` say, where tabs separate the data within each row:

Gender	Age	Height	Income
Male	34	173	34556
Male	45	184	67780
Female	NA	175	56449
Male	24		22089
Female	45	169	34402
Female	21	172	26709
Female	26	166	45221
Male	NA	189	56223

The categorical variable `Gender` takes the value `Male` or `Female`. `Age`, `Height` and `Income` are quantitative. There are three missing values, two coded as `NA` and one simply missing altogether. The first line also has a header row containing the variable names.

The data could be loaded as follows, directly from the web page where I have this file:

```
> read.table("http://www.sci.usq.edu.au/staff/dunn/Datasets/datafile1.dat",
+           header = TRUE, sep = "\t")
```

	Gender	Age	Height	Income
1	Male	34	173	34556
2	Male	45	184	67780
3	Female	NA	175	56449
4	Male	24	NA	22089
5	Female	45	169	34402
6	Female	21	172	26709
7	Female	26	166	45221
8	Male	NA	189	56223

The input `\sep="\t"` means the columns are separated with tab characters. If the data file is ‘nice’ (no missing values, no textstrings, etc.), you can probably omit the `sep` input. Note that R can load textual data as well as numerical data. Also, note that the missing values posed no problem.

If the columns are not separated with tabs, other separators can be used.

Almost always, when data is loaded it is allocated to an object name:

```
> df <- read.table("http://www.sci.usq.edu.au/staff/dunn/Datasets/datafile1.dat",
+                 header = TRUE, sep = "\t")
```

Soon we will look at the structure of these types of objects.

4.4 More on loading data

- R includes many data sets by default (for example, for use in the R help). A list of these is shown by typing

```
> data()
```

These data can be loaded as follows:

```
> data(USArrests)
```

In most instances, however, data will need to be loaded from a data file.

- If the first row of your data file does not have variable names, omit the argument `header=TRUE`. R then names the variables V1, V2, and so on, by default.
- If data are separated by commas (for example, `.csv` files stands for comma separated variables), you can use `read.table` with `sep=","` or `read.csv` which is specifically designed for comma-separated data files.
- Almost any separator can be specified using `sep=`; see `?read.table`.
- If you omit the `sep` argument, R thinks variables are separated by white space (spaces, tabs, new lines or carriage returns). This generally works well, but sometimes you need to explicitly specify the separator.
- Data are often missing. Sometime, many different codes are used to denote this in one data file. For example, `na.strings=c("NA", "99999", "-1")` means that NA, 99999 and -1 all denote missing values.
- R copes if missing values are simply absent and nothing appears; for example, if a comma-separated file contained this line:

```
6, -1, "Male",, 173,12,,0
```

R would see two missing values.
- Often, text is quoted in quotation marks. By default, R looks for double quote marks: `"`. Other quote characters can be defined also. The default is `quote="\\"`. (Note the quotation character is entered between double quotes (so we could also say `quote="!"` if the quotation marks are ! wich is unlikely); so to indicate the quotation character itself is a double quote, the double quote must be *escaped*.)

- There are many other options; see `?read.table`.
- For more powerful handling of data importing, see `?scan`.
- See `read.fwf` specifically designed for reading data in fixed width format.
- Type `library(help=foreign)` to learn about how R reads data from other file formats such as SPSS, SAS and Minitab.
- As already seen, the data file can be a `url` so that data can be loaded from the web.
- Unless told otherwise, R loads data from the current working directory/folder, which can be obtained from

```
> getwd()
```

```
[1] "/home/mcsci/dunn/pkd/Admin/Webpages/Homepage/LearnR/Docs"
```

This directory/folder can be *set* as follows in Linux:

```
setwd("~/datasets/thiscourse/datatfiles")
```

In Windows, use one of:

```
setwd("c:/datasets/thiscourse/datasets")
```

or

```
set.wd("c:\\datasets\\thiscourse\\datasets")
```

or use the menu to set the current working directory.

- To specify the location of a file directly, use, for example, `read.table("~/datasets/thiscourse/tutorial1.dat")`.
In Windows, use `read.table("c:/datasets/thiscourse/tutorial1.dat")`
or `read.table("c:\\datasets\\thiscourse\\tutorial1.dat")`

4.5 Data frames

Data files may contain different types of variables, such as

- Quantitative variables (such as heights);
- Categorical variables (such as gender);
- Text (or string) variables (such as people's names);

and many others. R also handles missing values well.

When data is loaded into R, it is stored as an object called a *data frame*. A data frame is like a matrix, but it can contain numerical as well as textual variables. For example,

```
> dummy.data <- read.table("http://www.sci.usq.edu.au/staff/dunn/Datasets/datafile1.dat",
+   header = TRUE, sep = "\t")
> dummy.data

  Gender Age Height Income
1  Male  34   173  34556
2  Male  45   184  67780
3 Female NA   175  56449
4  Male  24    NA  22089
5 Female 45   169  34402
6 Female 21   172  26709
7 Female 26   166  45221
8  Male  NA   189  56223
```

So we have a data frame called `dummy.data`. How do we just access *one* variable in that data frame? As follows¹:

```
> dummy.data$Gender

[1] Male  Male  Female Male  Female Female Female Male
Levels: Female Male

> summary(dummy.data$Gender)

Female  Male
      4      4

> summary(dummy.data$Income)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
22090  32480   39890  42930  56280  67780
```

¹Incidentally, this example demonstrates that R is *object oriented*: the `summary` command produces different output depending of what type of input it is asked to summarize.

In this way, you can have several data sets (“data frames”) loaded at once. Try typing `Gender`, and see what happens: R doesn’t know any variable by the name `Gender`, only `dummy.data$Gender`. It gets pretty annoying after a while typing `dummy.data$Gender` all the time...so consider this:

```
> attach(dummy.data)
> Gender

[1] Male   Male   Female Male   Female Female Female Male
Levels: Female Male

> Income

[1] 34556 67780 56449 22089 34402 26709 45221 56223
```

The command `attach` makes the variables in the specified data frame available without having to go through the cumbersome `data.frame$variable.name` process.

When you have finished with a data frame, remember to use

```
> detach(dummy.data)
```

If you have two data frames open that share variable names, anything could happen...

4.6 Examining data

4.6.1 Initial examination of the data frame

The data file `jelly-R.dat` is available for loading in to R, and comes from Hand et al. [1]. Proceed as follows: First ensure you are in the correct directory where you saved the data file, load it from a URL (you’ll have to be connected to the internet!), or give the full path:

```
> jf <- read.table("http://www.sci.usq.edu.au/staff/dunn/Datasets/Books/Hand/Hand-R/jelly-1
+   header = TRUE)
> attach(jf)
> names(jf)

[1] "Breadth" "Length" "Site"
```

Once data is loaded, the first task is to quickly examine the data. The following commands are useful for quick looks, especially to ensure the data loaded correctly:

```
> head(jf)
```

```
  Breadth Length Site
1     6.5    8.0    1
2     6.0    9.0    1
3     6.5    9.0    1
4     7.0    9.0    1
5     8.0    9.5    1
6     7.0   10.0    1
```

```
> tail(jf)
```

```
  Breadth Length Site
41     19    20    2
42     15    21    2
43     16    21    2
44     21    21    2
45     19    22    2
46     20    22    2
```

```
> summary(jf)
```

```
  Breadth      Length      Site
Min.   : 6.00   Min.   : 8.00   Min.   :1.000
1st Qu.:10.25   1st Qu.:13.00   1st Qu.:1.000
Median :15.00   Median :16.25   Median :2.000
Mean   :13.34   Mean   :15.80   Mean   :1.522
3rd Qu.:16.00   3rd Qu.:19.00   3rd Qu.:2.000
Max.   :21.00   Max.   :22.00   Max.   :2.000
```

Notice that `summary` produces different information for different sort of variables: again, this is the object-oriented aspect of R.

4.6.2 Initial examination of the variables

All the usual statistical summaries of data are available in R; for example, with quantitative variables:

- measures of centre, like the mean and median:

```
> mean(Length)
```

```
[1] 15.80435
```

```
> median(Length)
```

```
[1] 16.25
```

- measures of spread, like the standard deviation, variance and inter-quartile range:

```
> sd(Length)

[1] 4.167971

> var(Length)

[1] 17.37198

> IQR(Length)

[1] 6
```

A simpler overview of a variable is provided using `summary`, much like we saw above:

```
> summary(Length)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  8.00  13.00   16.25   15.80   19.00   22.00

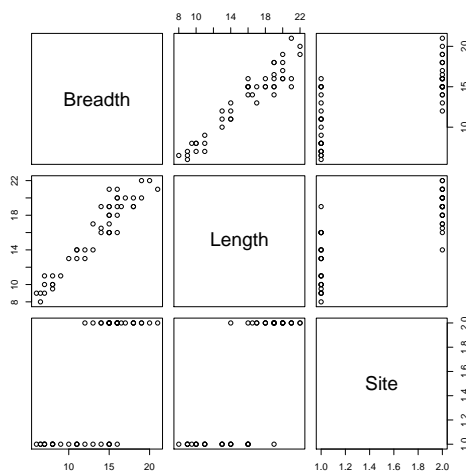
> summary(Site)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.000  1.000   2.000   1.522  2.000   2.000
```

4.6.3 Initial plotting of the data

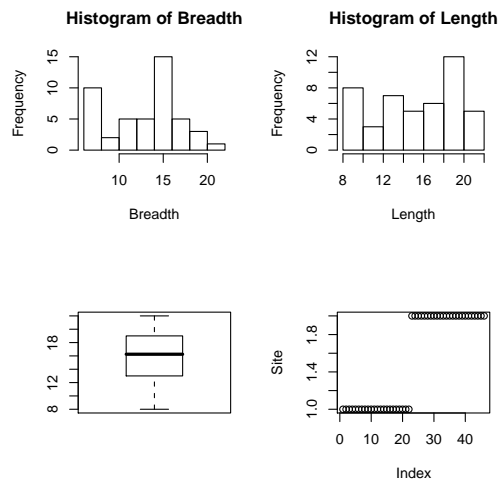
We should also do some plots of the data:

```
> plot(jf)
```



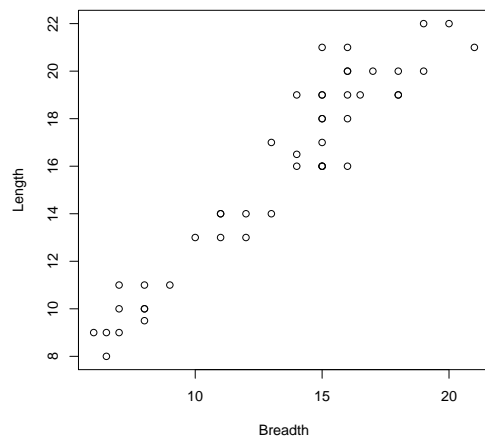
Even better, we can examine each variable separately²:

```
> par(mfrow = c(2, 2))
> hist(Breadth)
> hist(Length)
> boxplot(Length)
> plot(Site)
> par(mfrow = c(1, 1))
```



Relationships may also be examined:

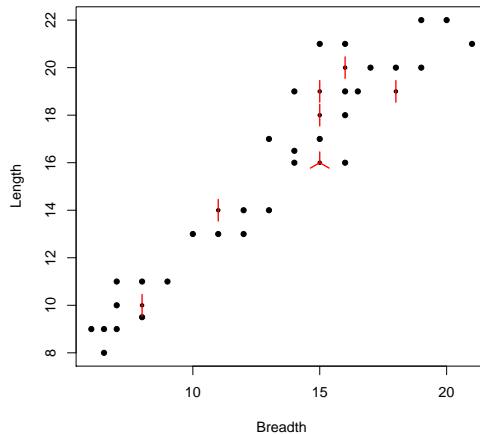
```
> plot(Breadth, Length)
```



Some markers represent more than one observation; here is an alternative plot:

²The command `par(mfrow=c(r,c))` splits the graphics windows into `r` rows and `c` columns. Also see the help (or run the `example`) for `layout` and `split.screen`.

```
> sunflowerplot(Breadth, Length)
```



In either case, there appears to be a linear relationship between Length and Breadth. Perhaps some kind of linear model is appropriate.

We can also be fancier. First, plot the data but don't display the points (that is, use `type="n"`):

```
> plot(Breadth ~ Length, type = "n")
```

Then plot Breadth against Length differently for each Site, using a different plotting character (`pch`) and colour (`col`):

```
> points(Breadth[Site == 1] ~ Length[Site == 1], pch = 19,
+       col = "red")
> points(Breadth[Site == 2] ~ Length[Site == 2], pch = 15,
+       col = "blue")
```

If you want to be really clever, add a legend:

```
> legend("topleft", pch = c(19, 15), col = c("red", "blue"),
+       legend = c("Dangar Island", "Salamander Bay"))
```

The final plot is shown in Figure 4.1.

This shows us that the data for each Site is different, even though the linear relationship looks quite similar: Salamander Bay jellyfish are generally *larger*.

And remember:

```
> detach(jf)
```

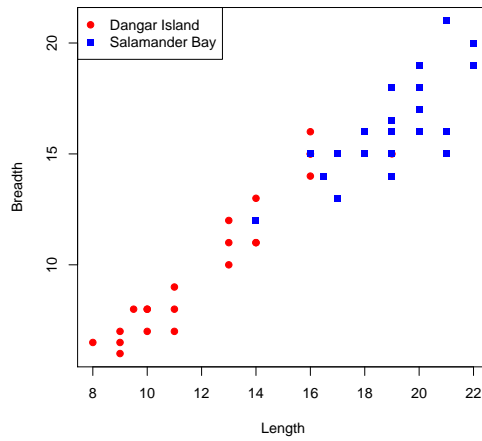


Figure 4.1: Plotting the jellyfish data, using different plotting characters for each location

Chapter 5

Matrices in R

- Matrix arithmetic requires special operators:

```
> matrix1 <- matrix(nrow = 3, ncol = 2, data = c(1, 2,
+ 0, 1, 2, 5), byrow = TRUE)
> matrix2 <- matrix(nrow = 2, ncol = 1, data = c(21, 22),
+ byrow = TRUE)
> matrix1
```

```
      [,1] [,2]
[1,]    1    2
[2,]    0    1
[3,]    2    5
```

```
> matrix2
```

```
      [,1]
[1,]   21
[2,]   22
```

```
> matrix1 %*% matrix2
```

```
      [,1]
[1,]   65
[2,]   22
[3,]  152
```

- Element-by-element multiplication uses no special operators:

```
> matrix1 * matrix1
```

```
      [,1] [,2]
[1,]    1    4
[2,]    0    1
[3,]    4   25
```

```
> matrix1^2
      [,1] [,2]
[1,]    1    4
[2,]    0    1
[3,]    4   25
```

- Transpose of matrices are found using `t()`

```
> XtX <- t(matrix1) %*% matrix1
> XtX
      [,1] [,2]
[1,]    5   12
[2,]   12   30
```

- Inverses of matrices are found using `solve()`:

```
> XtX.inverse <- solve(XtX)
> XtX.inverse %*% XtX
      [,1] [,2]
[1,] 1.000000e+00 8.881784e-15
[2,] 4.440892e-16 1.000000e+00

> zapsmall(XtX.inverse %*% XtX)
      [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Chapter 6

Plotting and graphics in R

6.1 Plotting data

First, load some data:

```
> data(mtcars)
> attach(mtcars)
```

The following object(s) are masked from `mtcars` (position 3) :

```
am carb cyl disp drat gear hp mpg qsec vs wt
```

```
> names(mtcars)
```

```
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am"
[10] "gear" "carb"
```

To plot one variable against another, use

```
> plot(disp, mpg)
```

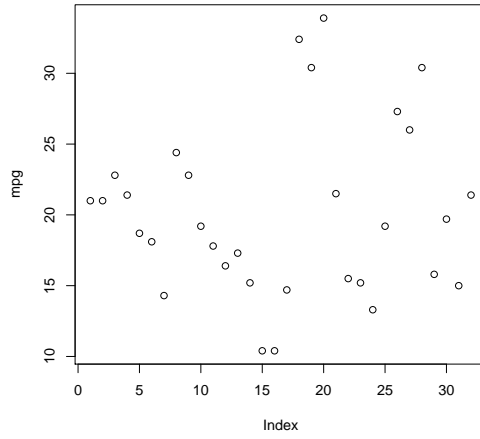
or, using formula notation,

```
> plot(mpg ~ disp)
```

By default, R plots using points as this is usually what is needed for plotting data.

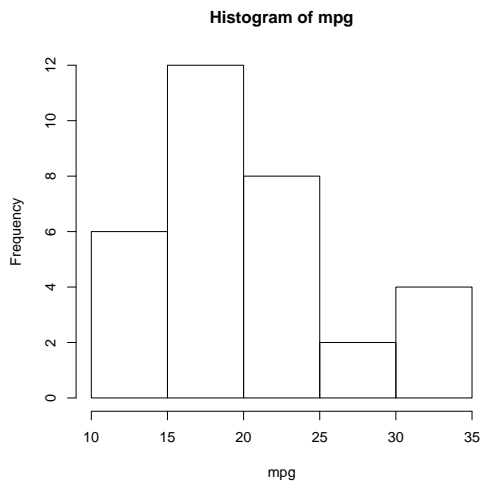
Individual variables can be plotted also:

```
> plot(mpg)
```

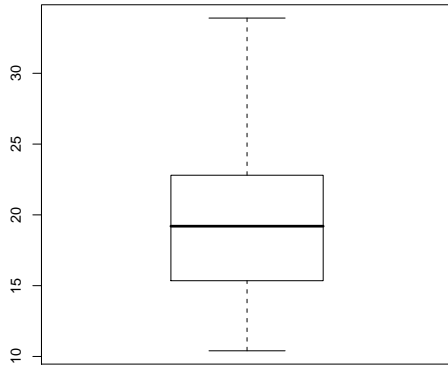


But that's not very useful. Perhaps better is one of the following

```
> hist(mpg)
```



```
> boxplot(mpg)
```



6.2 Plotting functions

- Since R is a vector-based language, plots of functions are easily produced.

```
> x <- seq(-3, 3, by = 0.05)
> y <- x^2
> plot(x, y)
```

This produces the plot in Figure 6.1. (The function `seq` is explained in Section 3.3.1). Since R is a statistical package, points are the default plot (since it is most useful for plotting data). Lines can be specified by

`plot(x, y, type="l")` where the type is the letter ‘ell’ for lines.

- R can produce publication-quality graphics; see Figure 6.2 for a variation of Figure 6.1. You do not need to know how to produce this plot; but it is important to know that high-quality plots are possible.

And don’t forget:

```
> detach(mtcars)
```

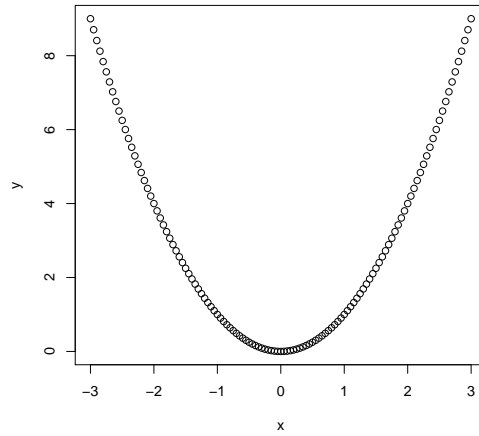
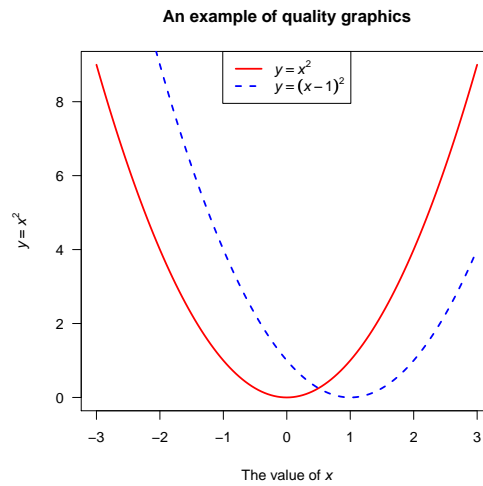
Figure 6.1: An example plot: $y = x^2$.

Figure 6.2: A variation of Figure 6.2

Chapter 7

Simple analyses

From the previous Chapter, a linear relationship is not unreasonable between the length and breadth of jellyfish. To fit a linear model, use the `lm` function, which stands for linear model. Perhaps it will be useful to look at `?lm` now.

7.1 Formula notation

In this section, we use the jellyfish data again:

```
> jf <- read.table("http://www.sci.usq.edu.au/staff/dunn/Datasets/Books/Hand/Hand-R/jelly-R.dat",
+   header = TRUE)
> attach(jf)
```

```
    The following object(s) are masked from jf ( position 3 ) :
```

```
    Breadth Length Site
```

```
    The following object(s) are masked from jf ( position 4 ) :
```

```
    Breadth Length Site
```

```
> names(jf)
```

```
[1] "Breadth" "Length"  "Site"
```

To use this function, we need to understand R's *formula notation*. Formula notation is used frequently in R. It is best seen in practice:

```
> jf.lm <- lm(Breadth ~ Length)
```

The formula `Breadth ~ Length` works as follows:

- The tilde \sim means ‘described by’ or ‘modelled by’.
- On the left of the equation is the *response variable*, or the *dependent variable*, or the ‘Y’ variable.
- On the right of the tilde are the *predictors* or the *covariates*, or the *independent variables*, or the ‘X’. This right-hand side can be quite complex; we will see this later.

So the above model fits a linear regression model of the form

$$\text{Breadth} = \hat{\beta}_0 + \hat{\beta}_1 \text{Length}.$$

Notice the constant term is fitted by default as this is nearly always what is wanted. To explicitly omit the constant term, use

```
> lm(Breadth ~ Length - 1)
```

Call:

```
lm(formula = Breadth ~ Length - 1)
```

Coefficients:

```
Length
0.8497
```

or

```
> lm(Breadth ~ Length + 0)
```

Call:

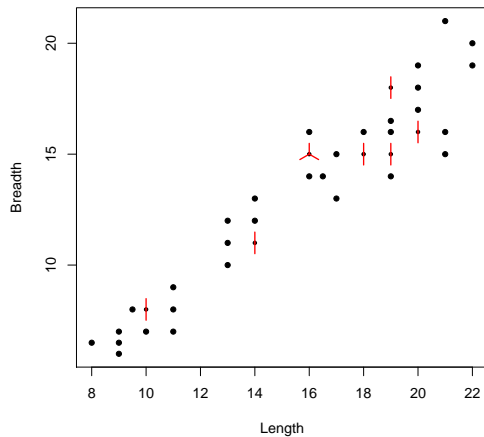
```
lm(formula = Breadth ~ Length + 0)
```

Coefficients:

```
Length
0.8497
```

Note that plotting can also use formula notation, and is often more natural:

```
> sunflowerplot(Breadth ~ Length)
```



7.2 Function output

In the above call to `lm`, the output was directed to a variable called `jf.lm`:

```
> jf.lm <- lm(Breadth ~ Length)
```

What is `jf.lm`? It is an R *object*. In fact, almost everything in R is called an *object*.

But what does this object contain? Let's have a look:

```
> jf.lm
```

Call:

```
lm(formula = Breadth ~ Length)
```

Coefficients:

(Intercept)	Length
-1.4423	0.9351

In fact, this is only a brief overview of the object `jf.lm`. We learn more by typing

```
> summary(jf.lm)
```

Call:

```
lm(formula = Breadth ~ Length)
```

Residuals:

Min	1Q	Median	3Q	Max
-3.19560	-0.81180	0.05847	0.70711	2.80440

```

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -1.44226    0.75199  -1.918  0.0616 .
Length      0.93514    0.04604  20.311 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.287 on 44 degrees of freedom
Multiple R-Squared:  0.9036,    Adjusted R-squared:  0.9014
F-statistic: 412.5 on 1 and 44 DF,  p-value: < 2.2e-16

```

(Recall we used `summary` before to summarize data objects. Since `jf.lm` is a different type of object, `summary` does something different.)

This `summary` provides quite a lot of information: a brief summary of the residuals, the parameter estimates, their standard errors, the corresponding t -ratios and corresponding P -values, plus some more.

`jf.lm` is actually an object that contains a *lot* of information:

```

> names(jf.lm)

[1] "coefficients" "residuals"    "effects"      "rank"
[5] "fitted.values" "assign"       "qr"           "df.residual"
[9] "xlevels"      "call"        "terms"       "model"

```

So, for example, we can see the coefficients of the model by typing

```

> jf.lm$coefficients

(Intercept)      Length
-1.4422622    0.9351363

```

(Note this is similar to how we accessed variables within a data frame if it is not `attach`-ed.) The same format can be used to see all the other components also: try

```

> jf.lm$residuals

```

Typing `jf.lm$residuals` becomes a bit cumbersome after a while; for the commonly requested components, there are usually shortcuts:

```

> coef(jf.lm)

(Intercept)      Length
-1.4422622    0.9351363

> resid(jf.lm)[1:5]

```

```

      1          2          3          4          5
0.46117214 -0.97396413 -0.47396413  0.02603587  0.55846774

```

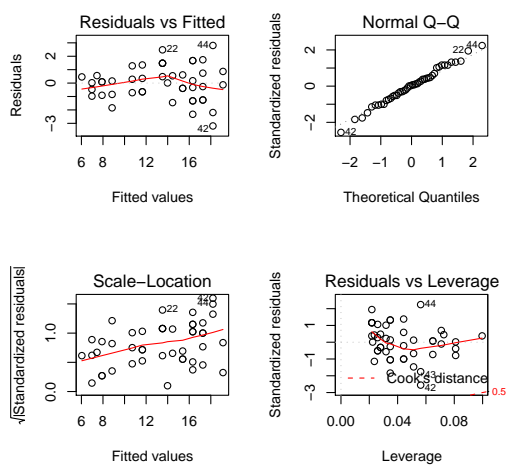
(Note the [1:5] means to extract just the first five residuals.)

You can also plot objects:

```

> par(mfrow = c(2, 2))
> plot(jf.lm)
> par(mfrow = c(1, 1))

```



(Again, this demonstrates R's object oriented approach: when you call `plot`, R determines what type of object you wish to plot, and plots sensible things accordingly.)

And don't forget:

```

> detach(jf)

```


Chapter 8

How to run your code in R

Typing a series of commands at the R prompt is useful for small analyses. But what if you have a lot of analyses to do? Or what if you want to keep track of what you are doing? Or you want to keep repeating very similar but long commands till you get thing just right?

The best way is to proceed as follows:

- Start your favourite text editor, such as Notepad, Wordpad, Kwrite, Emacs.
- Type your R commands into this text editor.
- Save the file somewhere with the extension¹ `.R`; for example, `analysis.R`.
- Ensure the working directory of R is set to where this file is located (for example, using `getwd()` and `setwd()`).
- In R, type, for example, `source("analysis.R")`. In Windows, `source-ing` a file also appears in the File menu item.

This approach has at least two significant advantages over working at the command prompt alone:

- You keep a permanent copy of what you have done.
- If you have any changes you need to make, you can make them in the `.R` file and run the code again quite easily.

¹The extension can be anything you like: `.txt`, `.r` or whatever. However, protocol and tradition encourage one to use the `.R` extension.

If you use Emacs, you can get extra functionality using ESS (Emacs Speaks Statistics).

In the following Chapters, we will assume you are using scripts.

Chapter 9

More on plotting

9.1 Introduction

To learn more about plotting, we will use the data set `mtcars`, concerning fuel consumption and ten aspects of automobile design and performance for 32 US automobiles (1973–74 models). This data set comes with R. Load this data and have a brief look at the data as follows:

```
> data(mtcars)
> attach(mtcars)
```

```
The following object(s) are masked from mtcars ( position 3 ) :
```

```
am carb cyl disp drat gear hp mpg qsec vs wt
```

```
The following object(s) are masked from mtcars ( position 6 ) :
```

```
am carb cyl disp drat gear hp mpg qsec vs wt
```

```
> names(mtcars)
```

```
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am"
[10] "gear" "carb"
```

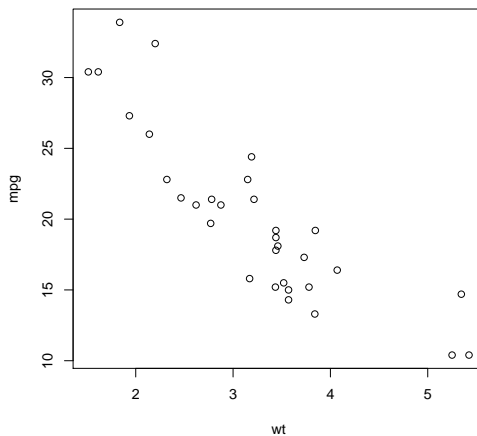
```
> head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

As with any data analysis, we like to start with plots. In this section, we discuss plotting options. You might like to make changes to the plotting commands in a .R file, as explained above, and then `source` this file. This is much easier than retyping similar commands over and over.

Let's initially examine the relationship between fuel consumption (`mpg`, or miles per gallon) and vehicle weight (`wt`):

```
> plot(mpg ~ wt)
```



Note that the *formula notation* has been used, where the tilde `~` means 'is modelled by' or 'is described by'. This plot is pretty basic, but still quite decent for a default plot.

9.2 Adding labels

We can alter the axis labels as follows:

```
> plot(mpg ~ wt, xlab = "Weight (in pounds)", ylab = "Consumption (miles per gallon)")
```

We can also add a major title:

```
> plot(mpg ~ wt, xlab = "Weight (in pounds)", ylab = "Consumption (miles per gallon)",
+      main = "Fuel consumption against Weight")
```

If the main title is too long, you can split it across lines:

```
> plot(mpg ~ wt, xlab = "Weight (in pounds)", ylab = "Consumption (miles per gallon)",
+      main = "Fuel consumption against Weight")
```

There is also a sub-title that can be set, but I usually find it makes the graphics too cluttered and looks too much like an extension of the horizontal-axis label.

After you have drawn a figure, titles can be added using the `title()` command.

9.3 Changing the plotting characters

Points can be plotted using many different types of characters, defined by using `pch`. We can specify the character directly:

```
> plot(mpg ~ wt, pch = "@", xlab = "Weight (in pounds)",  
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight")
```

Or we can use a predefined list (`pch=19` is one of my favourites):

```
> plot(mpg ~ wt, pch = 19, xlab = "Weight (in pounds)",  
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight")
```

To see the list of predefined plotting characters, see `example(points)`.

9.4 Changing colours

Different colours can be used also:

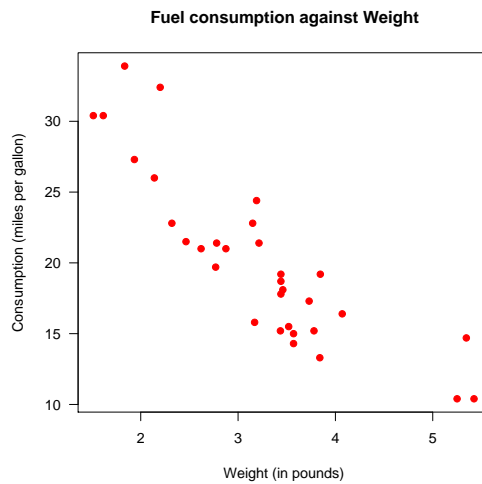
```
> plot(mpg ~ wt, pch = 19, col = "red", xlab = "Weight (in pounds)",  
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight")
```

Colours may be defined explicitly as above, or numerically using a colour code. More on this later. (The standard `x11` colour names are used.)

9.5 Changing the format of the axes

I prefer my axis labels to always be horizontal. In R, sometimes this is almost essential if you make small graphics. Try this:

```
> plot(mpg ~ wt, pch = 19, col = "red", las = 1, xlab = "Weight (in pounds)",  
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight")
```



9.6 Selecting parts of a data frame

Often, a simple scatterplot hides information. For example, in the relationship between fuel consumption and weight, the number of cylinders is probably quite important. In the data frame, the number of cylinders, `cyl`, is numeric. For some purposes, it is useful to think of the number of cylinders as a *factor* or *categorical variable*.

Consider this:

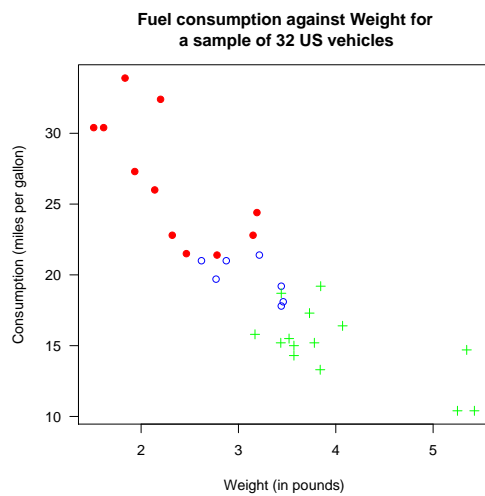
```
> coplot(mpg ~ wt | factor(cyl), columns = 3)
```

But we can do something similar with the standard scatterplot also. First, we plot the data invisibly (using `type="n"`, where `n` means ‘no plot at all (please)’).

```
> plot(mpg ~ wt, type = "n", las = 1, xlab = "Weight (in pounds)",
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight for
```

This sets up the axes, axis labels, and so forth, but doesn’t plot any data. Now we want to *add* data points, but separately for the number of cylinders. To do this, we select *subsets* of the data frame to plot (and note the use of `==` rather than `=`).

```
> plot(mpg ~ wt, type = "n", las = 1, xlab = "Weight (in pounds)",
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight for
> points(mpg[cyl == 4] ~ wt[cyl == 4], pch = 19, col = "red")
> points(mpg[cyl == 6] ~ wt[cyl == 6], pch = 1, col = "blue")
> points(mpg[cyl == 8] ~ wt[cyl == 8], pch = 3, col = "green")
```



Note that this way of selecting part of a data frame applies in general:

```
> mean(mpg[cyl == 4])
```

```
[1] 26.66364
```

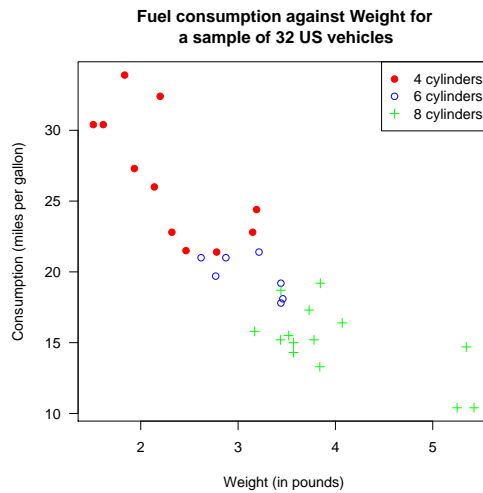
```
> median(mpg[cyl == 8])
```

```
[1] 15.2
```

9.7 Adding a legend

Finally, we might add a legend.

```
> plot(mpg ~ wt, type = "n", las = 1, xlab = "Weight (in pounds)",
+      ylab = "Consumption (miles per gallon)", main = "Fuel consumption against Weight for\n a samp
> points(mpg[cyl == 4] ~ wt[cyl == 4], pch = 19, col = "red")
> points(mpg[cyl == 6] ~ wt[cyl == 6], pch = 1, col = "blue")
> points(mpg[cyl == 8] ~ wt[cyl == 8], pch = 3, col = "green")
> legend("topright", col = c("red", "blue", "green"), pch = c(19,
+      1, 3), legend = c("4 cylinders", "6 cylinders", "8 cylinders"))
```



Now it is clear that cars with more cylinders are, in general, heavier and use more fuel.

9.8 Saving figures

After a picture is produced, it can be saved in a variety of formats. In Windows, you can use the menu to save a figure in some formats¹ Easiest is to save in (encapsulated) postscript:

```
dev.print(postscript, file="PracticePlot.eps")
```

(Notice this is called ‘print-ing’ in R; in reality, the picture is saved in a file.) This is so common, another function is usually used with sensible defaults set:

```
hist( wt )
dev.copy2eps(file="PracticePlot.eps")
```

To save as a png or jpeg file, use

```
hist(wt)
dev.print(png, file="PracticePlot.png")
dev.print(jpeg, file="PracticePlot.jpg")
```

(Again notice this is called ‘print-ing’; in reality, the picture is saved in a file.)

There other method is to open a png, jpg, eps, pdf or another R ‘device’ and plot *directly* to this device:

¹I don’t use Windows, so I know little about it. Perhaps you can right-click on the Windows and save it in some formats also. Nonetheless, everything in this section applies to all operating systems.

```
pdf(file="PracticePlot.pdf")
hist(wt)
dev.off()
```

Note the device must be turned off using `dev.off()` before the file is created. Nothing useful appears on the screen.

Once again, you can specify a different folder/directory by explicitly giving it.

9.9 Other options

There are many options that can be set. In fact, for a beginner the list is intimidating; above we have explored some of the more common options so we didn't need to examine the plethora of options! But it's now time to look at the available options. A list is found by typing:

```
?par
```

And don't forget:

```
> detach(mtcars)
```


Bibliography

- [1] D J Hand, F Daly, A D Lunn, K Y McConway, and E Ostrowski. *A Handbook of Small Data Sets*. Chapman and Hall, London, 1996.