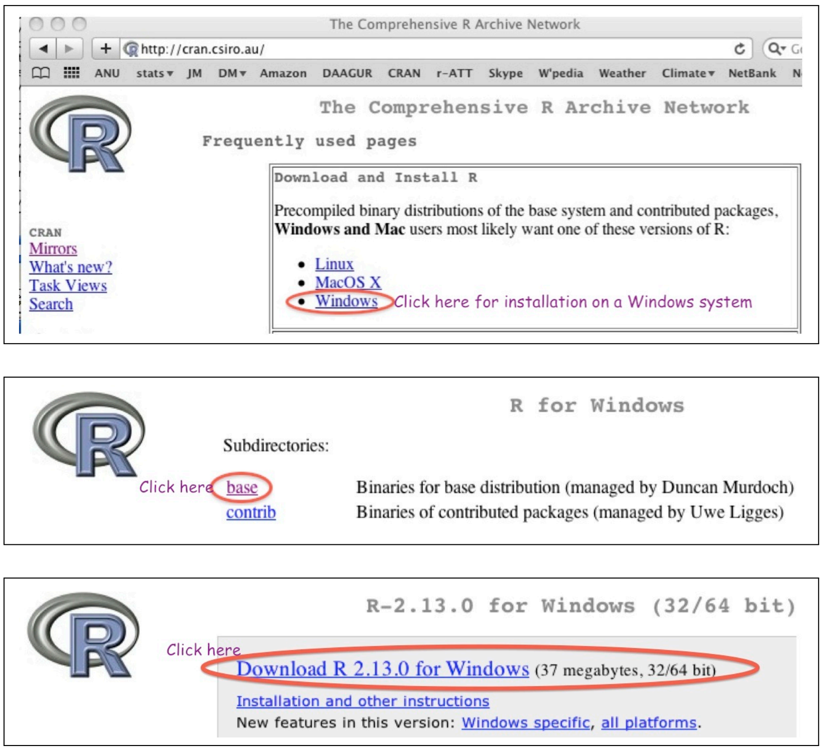
1 Getting started with R

1.1 Installation of R

Click as indicated in the successive panels to download R for Windows from the web page `http://cran.csiro.au`:



The Comprehensive R Archive Network

http://cran.csiro.au/

ANU stats JM DM Amazon DAAGUR CRAN r-ATT Skype W'pedia Weather Climate NetBank

The Comprehensive R Archive Network

Frequently used pages

CRAN
Mirrors
What's new?
Task Views
Search

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Linux
- MacOS X
- Windows

R for Windows

Subdirectories:

base	Binaries for base distribution (managed by Duncan Murdoch)
contrib	Binaries of contributed packages (managed by Uwe Ligges)

R-2.13.0 for Windows (32/64 bit)

Download R 2.13.0 for Windows (37 megabytes, 32/64 bit)

Installation and other instructions

New features in this version: Windows specific, all platforms.

Figure 1.1: This shows a sequence of clicks that will download the R installation file from `cran.csiro.edu`. At the time of writing, the website will offer R-3.4.3 rather than R-2.13.0. The site `cran.csiro.edu` is one of two Australian CRAN (Comprehensive R Archive Network) sites. The other is: `http://cran.ms.unimelb.edu.au/`

Figure 1.2: On 64-bit Windows systems the default installation process creates two icons, one for 32-bit R and one for 64-bit R. Additional icons can be created as desired.

Click on the downloaded file to start installation. Most users will want to accept the defaults. The effect is to install the R base system, plus recommended packages, with a standard “off-the-shelf” setup. Windows users will find that one or more desktop R icons have been created as part of the installation process.

Depending on the intended tasks, it may be necessary to install further packages. Section 1.3 describes alternative ways to install packages.

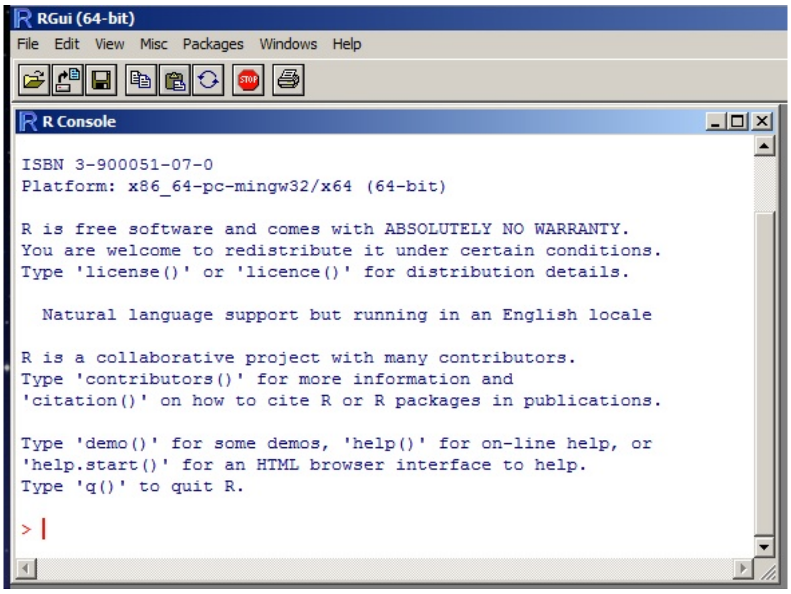
An optional additional step is to install RStudio. RStudio has abilities that help in managing workflow, in navigating between projects, and in accessing R system information. See Section 2.4.

Clicking on the RStudio icon to start a session will at the same time start R. RStudio has its own command line interface, where users can type R commands.

1.2 First steps

Click on an R icon to start an R session. This opens an R command window, prints information about the installed version of R, and gives a command prompt.

Readers who have RStudio running can type their commands in the RStudio command line panel.



```
ISBN 3-900051-07-0
Platform: x86_64-pc-mingw32/x64 (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.

> |
```

Figure 1.3: Windows command window at startup. This shows the default MDI (multiple display) interface. For running R from the R Commander, the alternative SDI (single display) interface may be required, or may be preferable. The Mac GUI has a SDI type interface; there is no other option.

The > prompt that appears on the final line is an invitation to start typing R commands:

Thus, type 2+5 and press the Enter key. The display shows:

```
> 2+5
```

```
[1] 7
```

The [1] says, a little strangely, “first requested element will follow”. Here, there is just one element.

The result is 7. The output is immediately followed by the > prompt, indicating that R is ready for another command.

Try also:

```
> result <- 2+5
> result
```

```
[1] 7
```

Typing `result` on the command line has printed the value 7.

Technically, the *workspace* is one of a number of *databases* where objects may be stored.

The object `result` is stored in the *workspace*. The *workspace* holds objects that the user has created or input, or that were there at the start of the session and not later removed

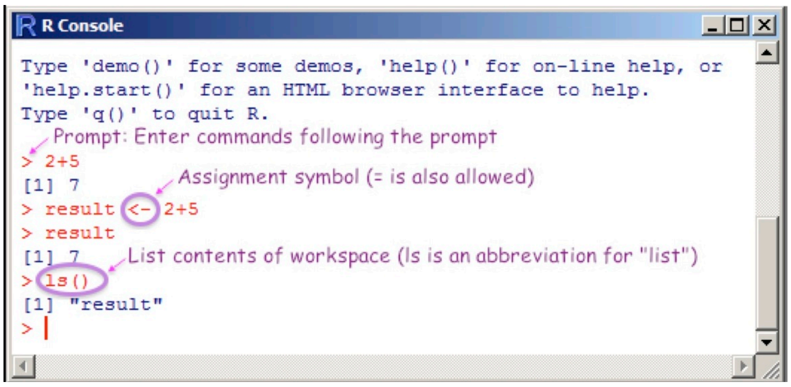
Type `ls()` to list the objects in the workspace, thus:

```
> ls()
```

```
[1] "result"
```

The object `result` was added to a previously empty workspace.

Figure 1.4 shows, with annotations, the screen as it appears following the above sequence of commands.



```
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.
> 2+5
[1] 7
> result <- 2+5
> result
[1] 7
> ls()
[1] "result"
>
```

R Console
Prompt: Enter commands following the prompt
Assignment symbol (= is also allowed)
List contents of workspace (ls is an abbreviation for "list")

Figure 1.4: This shows the sequence of commands that are demonstrated in the text, as they appear on the screen, with added annotation.

An R session is structured as a hierarchy of databases. Functions that were used or referred to above — such as `ls()` – are from a database or *package* that is part of the R system. Objects that the user has created or input, or that were there at the start of the session and not later removed, are stored in the *workspace*.

The workspace is the user's database for the duration of a session. It is a volatile database, i.e., it will disappear if not explicitly saved prior to or at the end of the session.

Technically, the R system refers to the workspace as `.Globalenv`.

1.2.1 *Points to note*

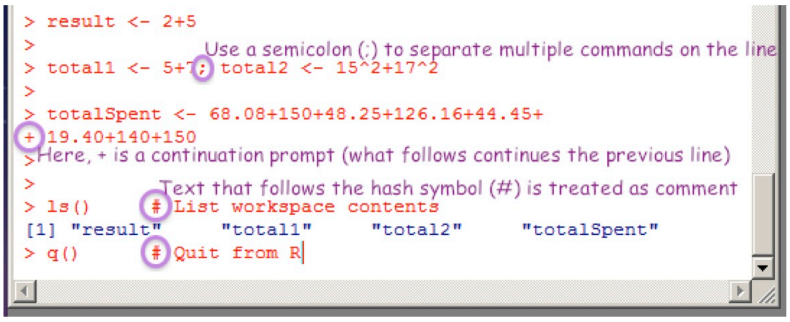
Printing	Typing the name of an object (and pressing Enter) displays (prints) its contents.
Quitting	To quit, type `q()`, (not `q`)
Case matters	`volume` is different from `Volume`

Typing the name of an object (and pressing the Enter key) causes the printing of its contents, as above when `result` was typed. This applies to functions also. Thus type `q()` in order to quit, not `q`.[1] One types `q()` because this causes the function `q` to spring into action.

Upon typing `q()` and pressing the Enter key, a message will ask whether to save the workspace image.[2] Clicking Yes (usually the safest option) will save the objects that remain in the workspace – any that were there at the start of the session (unless removed or overwritten) and any that have been added since. The workspace that has been thus saved is automatically reloaded when an R session is restarted in the working directory to which it was saved.

[1] Typing `q` lists the code for the function.

[2] Such an *image* allows reconstruction of the workspace of which it forms an image!



```
> result <- 2+5
>
> total1 <- 5+7; total2 <- 15^2+17^2
>
> totalSpent <- 68.08+150+48.25+126.16+44.45+
+ 19.40+140+150
>
>
> ls()     # List workspace contents
[1] "result"      "total1"      "total2"      "totalSpent"
> q()      # Quit from R
```

Use a semicolon (;) to separate multiple commands on the line
Here, + is a continuation prompt (what follows continues the previous line)
Text that follows the hash symbol (#) is treated as comment

Figure 1.5: Note the use of the special characters: ; to separate multiple commands on the one line, + (generated by the system) to denote continuation from previous line, and # to introduce comment that extends to end of line.

Note that for names of R objects or commands, case is significant. Thus `Myr` (millions of years, perhaps) differs from `myr`. For file names,[3] the operating system conventions apply.

Commands may, as demonstrated in Figure 1.5, continue over more than one line. By default, the continuation prompt is +. As with the > prompt, this is generated by R, and appears on the left margin. Including it when code is entered will give an error!

[3] Under Windows case is ignored. For Unix case does distinguish. (Mac OS X Unix is a partial exception.)

Here is a command that extends over two lines:

```
> result <-
+ 2+5
```

1.2.2 Some further comments on functions in R

Common functions that users should quickly get to know include `print()`, `plot()` and `help()`. Above, we noted the function `q()`, used to quit from an R session.

Consider the function `print()`. One can explicitly invoke it to print the number 2 thus:

R is a functional language. Whenever a command is entered, this causes a function to run. Addition is implemented as a function, as are other such operations.

```
print(2)
```

```
[1] 2
```

Objects on which the function will act are placed inside the round brackets. Such quantities are known as *arguments* to the function.

An alternative to typing `print(2)` is to type `2` on the command line. The function `print()` is then invoked implicitly:

```
2
```

```
[1] 2
```

1.2.3 Help information

Included on the information that appeared on the screen when R started up, and shown in Figures 1.4 and 1.5, were brief details on how to access R's built-in help information:

```
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
```

The shorthand `?plot` is an alternative to typing `help(plot)`.

Replace '?' by '??' for a wider search. This invokes the function `help.search()`, which looks for a partial match in the title or concept fields as well as in the name.

Examples of use of ??:

```
??Arithmetic
??base::Arith
  # Search base R only
```

R has extensive built-in help information. Be sure to check it out as necessary. Section 1.8 has further details on what is available, beyond what you can get by using the help function.

1.2.4 The working directory

Associated with each session is a working directory where R will by default look for files. In particular:

Under Windows, if R is started by clicking on an R icon, the working directory is that specified in the Start in directory specified in the icon Preferences. Subsection A.1 has details on how to specify the Start in directory for an icon.

- If a command inputs data from a file into the workspace and the path is not specified, this is where R will look for the file.
- If a command outputs results to a file, and the path is not specified, this is where R will place the file.
- Upon quitting a session, the "off-the-shelf" setup will ask whether to save an "image" of the session. Answering "Yes" has the result that the contents of the workspace are saved into a file, in the working directory, that has the name **.RData**.

When R finds a **.RData** file in the working directory at startup, that file will, in an off-the-shelf setup, be used to restore the workspace.

For regular day to day use of R, it is advisable to have a separate working directory for each different project. RStudio users will be asked to specify a working directory when setting up a new "project".

1.3 Installation of R Packages

Installation of R Packages (Windows & MacOS X)

Start R (e.g., click on the R icon). Then use the relevant menu item to install packages via an internet connection. This is (usually) easier than downloading, then installing.

For command line instructions to install packages, see below.

A fresh install of R packages is typically required when moving to a new major release (e.g., from a 3.0 series release to a 3.1 series release).

The functions that R provides are organised into packages. The packages that need to be installed, additional to those that come with the initial ready-to-run system, will vary depending on individual user requirements. The GUIs — MacOS X, Windows or Linux — make package installation relatively straightforward.

Installation of packages from the command line

To install the R Commander from the command line, enter:

```
install.packages("Rcmdr", dependencies=TRUE)
```

By default, a CRAN mirror is searched for the required package. Refer back to the introduction for brief comments on CRAN. Subsection 2.3.1 gives details of alternatives to CRAN. Note in particular the Bioconductor repository.

The R Commander has a number of dependencies, i.e., packages that need to be installed for the R Commander to run. Graphics packages that are dependencies include *rgl* (3D dynamic graphics), *scatterplot3d*, *vcd* (visualization of categorical data) and *colorspace* (generation of color palettes, etc).

Installation of Bioconductor packages

To set your system up for use of Bioconductor packages, type:

```
source("http://bioconductor.org/biocLite.R")
biocLite()
```

For installation of Bioconductor packages from the GUI, see Subsection A.4.

Additional packages can be installed thus:

```
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

See further http://www.bioconductor.org/install/.

1.4 *Practice with R commands*

Read `c` as "concatenate", or perhaps as "column".

Lists are widely used in R. A data frame is a special type of list, used to collect together column objects under one name.

Column Objects

```
    width = c(11.3, 13.1, 20, 21.1, 25.8, 13.1)
    height = c(23.9, 18.7, 27.6, 28.5, 36, 23.4)
```

Data frame

A data frame is a list of column objects, all of the same length.

```
    widheight <- data.frame(
        width = c(11.3, 13.1, 20, 21.1, 25.8, 13.1),
        height = c(23.9, 18.7, 27.6, 28.5, 36, 23.4)
     )
```

Also: Arithmetic operations; simple plots; input of data.

The R language has the standard abilities for evaluating arithmetic and logical expressions. There are numerous functions that extend these basic arithmetic and logical abilities.

Try the following

```
2+3         # Simple arithmetic
```

```
[1] 5
```

```
1:5         # The numbers 1, 2, 3, 4, 5
```

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

```
mean(1:5)   # Average of 1, 2, 3, 4, 5
```

```
[1] 3
```

```
sum(1:5)    # Sum of 1, 2, 3, 4, 5
```

```
[1] 15
```

```
(8:10)^2    # 8^2 (8 to the power of 2), 9^2, 10^2
```

```
[1]  64  81 100
```

In addition to `log()`, note `log2()` and `log10()`:

A change by a factor of 2 is a one unit change on a log2 scale. A change by a factor of 10 is a one unit change on a log10 scale.

```
log2(c(0.5, 1, 2, 4, 8))
```

```
[1] -1  0  1  2  3
```

```
log10(c(0.1, 1, 10, 100, 1000))
```

```
[1] -1  0  1  2  3
```

It turns out, surprisingly often, that logarithmic scales are appropriate for one or other type of graph. Logarithmic scales focus on relative change — by what factor has the value changed?

The following uses the relational operator >:

Other relational operators are $< \;\; >= \;\; < \;\; <= \;\; == \;\; !=$

```
(1:5) > 2  # Returns FALSE FALSE  TRUE  TRUE  TRUE
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

Demonstrations

Demonstrations can be highly helpful in learning to use R's functions. The following are some of demonstrations that are available for graphics functions:

Images and perspective plots:

```
demo(image)
demo(persp)
```

For the following, the *vcd* package must be installed:

```
library(vcd)
demo(mosaic)
```

```
demo(graphics)   # Type <Enter> for each new graph
library(lattice)
demo(lattice)
```

Especially for `demo(lattice)`, it pays to stretch the graphics window to cover a substantial part of the screen. Place the cursor on the lower right corner of the graphics window, hold down the left mouse button, and pull.

The following lists available demonstrations:

```
## List demonstrations in attached packages
demo()
## List demonstrations in all installed packages
demo(package = .packages(all.available = TRUE))
```

1.5 A Short R Session

We will work with the data set shown in Table 1.1:

	thickness	width	height	weight	volume	type
Aird's Guide to Sydney	1.30	11.30	23.90	250	351	Guide
Moon's Australia handbook	3.90	13.10	18.70	840	955	Guide
Explore Australia Road Atlas	1.20	20.00	27.60	550	662	Roadmaps
Australian Motoring Guide	2.00	21.10	28.50	1360	1203	Roadmaps
Penguin Touring Atlas	0.60	25.80	36.00	640	557	Roadmaps
Canberra - The Guide	1.50	13.10	23.40	420	460	Guide

Table 1.1: Weights and volumes, for six Australian travel books.

Entry of columns of data from the command line

The following enters data as numeric vectors:

```
volume <- c(351, 955, 662, 1203, 557, 460)
weight <- c(250, 840, 550, 1360, 640, 420)
```

Read `c` as "concatenate", or perhaps as "column". It joins elements together into a vector, here numeric vectors.

Now store details of the books in the character vector `description`:

```
description <- c("Aird's Guide to Sydney",
 "Moon's Australia handbook",
 "Explore Australia Road Atlas",
 "Australian Motoring Guide",
 "Penguin Touring Atlas", "Canberra - The Guide")
```

The end result is that objects `volume`, `weight` and `description` are stored in the workspace.

Listing the workspace contents

Use `ls()` to examine the current contents of the workspace.

```
ls()
```

```
[1] "description" "result"      "volume"      "weight"
```

Use the argument `pattern` to specify a search pattern:

```
ls(pattern="ume")   # Names that include "ume"
```

```
[1] "volume"
```

Note also:

```
ls(pattern="^des")
  ## begins with 'des'
ls(pattern="ion$")
  ## ends with 'ion'
```

Operations with numeric `vectors`

Here are the values of `volume`

```
volume
```

```
[1]  351  955  662 1203  557  460
```

To extract the final element of `volume`, do:

```
volume[6]
```

```
[1] 460
```

For the ratio of weight to volume, i.e., the density, we can do:

```
weight/volume
```

```
[1] 0.7123 0.8796 0.8308 1.1305 1.1490 0.9130
```

A note on functions

For the `weight/volume` calculation, two decimal places in the output is more than adequate accuracy. The following uses the function `round()` to round to two decimal places:

```
round(x=weight/volume, digits=2)
```

More simply, type:

```
round(weight/volume, 2)
```

Providing the arguments are in the defined order, they can as here be omitted from the function call.

```
[1] 0.71 0.88 0.83 1.13 1.15 0.91
```

Functions take *arguments* — these supply data on which they operate. For `round()` the arguments are '`x`' which is the quantity that is to be rounded, and '`digits`' which is the number of decimal places that should remain after rounding.

Use the function `args()` to get details of the named arguments:

```
args(round)
```

```
function (x, digits = 0)
NULL
```

Many functions, among them `plot()` that is used for Figure 1.6, accept unnamed as well as named arguments. The symbol '`...`' is used to denote the possibility of unnamed arguments. If a '`...`' appears, indicating that there can be unnamed arguments, check the help page for details.

Tabulation

Use the function `table()` for simple numeric tabulations, thus:

```
type <- c("Guide","Guide","Roadmaps","Roadmaps",
          "Roadmaps","Guide")
table(type)
```

```
type
   Guide Roadmaps
       3        3
```

A simple plot

Figure 1.6 plots `weight` against `volume`, for the six Australian travel books. Note the use of the graphics formula `weight ~ volume` to specify the $x-$ and $y-$variables. It takes a similar from to the "formulae" that are used in specifying models, and in the functions `xtabs()` and `unstack()`.

Code for Figure 1.6 is:

```
## Code
plot(weight ~ volume, pch=16, cex=1.5)
  # pch=16: use solid blob as plot symbol
  # cex=1.5: point size is 1.5 times default
## Alternative
plot(volume, weight, pch=16, cex=1.5)
```

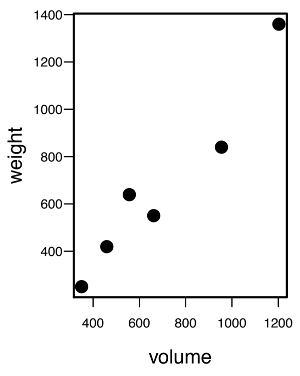


Figure 1.6: Weight versus volume, for six Australian travel books.

The axes can be labeled:

```
plot(weight ~ volume, pch=16, cex=1.5,
     xlab="Volume (cubic mm)", ylab="Weight (g)")
```

Interactive labeling of points (e.g., with species names) can be done interactively, using `identify()`:

```
identify(weight ~ volume, labels=description)
```

Use `text()` for non-interactive labeling of points.

Then click the left mouse button above or below a point, or on the left or right, depending on where you wish the label to appear. Repeat for as many points as required.

On most systems, the labeling can be terminated by clicking the right mouse button. On the Windows GUI, an alternative is to click on the word “Stop” that appears at the top left of the screen, just under “Rgui” on the left of the blue panel header of the R window. Then click on “Stop locator”.

Formatting and layout of plots

There are extensive abilities that may be used to control the formatting and layout of plots, and to add features such as special symbols, fitted lines and curves, annotation (including mathematical annotation), colors, . . .

1.6 Data frames – Grouping columns of data

Data frames	Store data that have a cases by columns layout.
Creating data frames	Enter from the command line (small datasets) Or: Use `read.table()` to input from a file.
Columns of data frames	`travelbooks$weight` or `travelbooks[, 4]` or `travelbooks[, "weight"]`

Data frames are pervasive in R. Most datasets that are included with R packages are supplied as data frames.

The following code groups the several columns of Table 1.1 together, under the name `travelbooks`. It is tidier to have matched columns of data grouped together into a data frame, rather than separate objects in the workspace.

```
## Group columns together into a data frame
travelbooks <- data.frame(
   thickness = c(1.3, 3.9, 1.2, 2, 0.6, 1.5),
   width = c(11.3, 13.1, 20, 21.1, 25.8, 13.1),
   height = c(23.9, 18.7, 27.6, 28.5, 36, 23.4),
   weight = weight,  # Use values entered earlier
   volume = volume,  # Use values entered earlier
   type = c("Guide", "Guide", "Roadmaps", "Roadmaps",
            "Roadmaps", "Guide"),
   row.names = description
)
## Remove objects that are not now needed.
rm(volume, weight, description)
```

The vectors `weight`, `volume` and `description` were entered earlier, and (unless subsequently removed) can be copied directly into the data frame.

It is a matter of convenience whether the description information is used to label the rows, or alternatively placed in a column of the data frame.

The storage of character data as factors

Vectors of character, such as `type`, are by default stored in the data frame as *factors*. In the data as stored, `"Guide"` is 1 and `"Roadmaps"` is 2. Stored with the factor is an attribute table that interprets 1 as `"Guide"` and 2 as `"Roadmaps"`.

While in most contexts factors and character vectors are interchangeable, there are important exceptions.

Accessing the columns of data frames

The following are alternative ways to extract the column `weight` from the data frame:

For a matrix or array, users are restricted to the first and second of these alternatives. With a matrix `travelmat` use, e.g., `travelmat[,4]` or `travelmat[,"weight"]`.

```
travelbooks[, 4]
travelbooks[, "weight"]
travelbooks$weight
travelbooks[["weight"]]   # Reference as a list.
```

There are several mechanisms that avoid repeated reference to the name of the data frame. The following are alternative ways to plot `weight` against `volume`:

1. *Use the parameter* `data`*, where available, in the function call*

   ```
   plot( weight ~ volume, data=travelbooks)
   ```

2. *Use* `with()`*: Take columns from specified data frame*

   ```
   ## Take columns from the specified data frame
   with(travelbooks, plot(weight ~ volume))
   ```

Most modeling functions and many plotting functions accept a `data` argument.

Both these mechanisms look first for a data frame column with a needed name. The workspace is searched only if this fails.

A third option, usually best avoided, is to use `attach()` to add the data frame to the search list. In this case, names in the workspace take precedence over column names in the attached data frame – not usually what is wanted if there are names in common.

Attachment of a data frame:

```
attach(travelbooks)
plot( weight ~ volume)
detach(travelbooks)
 ## Detach when no longer
 ## required.
```

Subsection 2.3.2 will discuss the attaching of packages and image files.

1.7 *Input of Data from a File*

The function `read.table()` is designed for input from a rectangular file into a data frame. There are several variants on this function — notably `read.csv()` and `read.delim()`.

First use the function `datafile()` (*DAAG*) to copy from the *DAAG* package and into the working directory a data file that will be used for demonstration purposes.

This use of `datafile()`, avoiding use of the mouse to copy the file and the associated need to navigate the file system, is a convenience for teaching purposes.

```
## Place the file in the working directory
## NB: DAAG must be installed
DAAG::datafile("travelbooks")
```

Use `dir()` to check that the file is indeed in the working directory:

```
dir(pattern="travel")
  # File(s) whose name(s) include 'travel'
```

The first two lines hold the column headings and first row, thus:

	thickness	width	height	weight	volume	type
Aird's Guide to Sydney	1.30	11.30	23.90	250	351	Guide
. . .						

Observe that column 1, which has the row names, has no name.

The following reads the file into an R data frame:

Row 1 has column names.
Column 1 has row names.

```
## Input the file to the data frame travelbooks
travelbooks <- read.table("travelbooks.txt",
                          header=TRUE, row.names=1)
```

The assignment places the data frame in the workspace, with the name `travelbooks`. The first seven columns are numeric. The character data in the final column is by default stored as a factor.

Data input – points to note:

- Alternatives to command line input include the R Commmander menu and the RStudio menu. These make it easy to check that data are being correctly entered.
- If the first row of input gives column names, specify `heading=TRUE`. If the first row of input is the first row of data, specify `heading=FALSE`.
- See `help(read.table)` for details of parameter settings that may need changing to match the data format.
- Character vectors that are included as columns in data frames become, by default, factors.

Section 5.1 discusses common types of input errors.

Character vectors and factors can often, but by no means always, be treated as equivalent.

1.8 Sources of Help

```
help()              # Help for the help function
help(plot)          # Show the help page for plot
?plot               # Shorthand for help(plot)
example(plot)       # Run examples from help(plot)
demo()              # List available demonstrations
vignette()          # Get information on vignettes
                    # NB also browseVignettes()
```

Note also:
`help.search()`
`apropos()`
`help.start()`
`RSiteSearch()`

This section enlarges on the very brief details in Subsection 1.2.3

Access to help resources from a browser screen

Type `help.start()` to display a screen that gives a browser interface to R's help resources. Note especially Frequently Asked Questions and Packages. Under Packages, click on base to get information on base R functions. Standard elementary statistics functions are likely to be found under stats, and base graphics functions under graphics.

Also available, after clicking on a package name, is a link User guides, package vignettes and other documentation. Click to get details of any documentation that is additional to the help pages.

Official R manuals include An Introduction to R, a manual on Writing R Extensions, and so on.

Searching for key words or strings

Use `help.search()` to look for functions that include a specific word or part of word in their alias or title. Thus, functions for operating on character strings are likely to have "str" or "char" in their name. Try

By default, all installed packages are searched. Limiting the search, here to `package="base"`, will often give more manageable and useful output.

```
help.search("str", package="base")
help.search("char", package="base")
```

The function `RSiteSearch()` searches web-based resources, including R mailing lists, for the text that is given as argument.

Examples that are included on help pages

All functions have help pages. Most help pages include examples, which can be run using the function `example()`. Be warned that, even for relatively simple functions, some of the examples may illustrate non-trivial technical detail.

To work through the code for an example, look on the screen for the code that was used, and copy or type it following the command line prompt. Or get the code from the help page.

Vignettes

Many packages have vignettes; these are typically pdf or (with version ≥ 3.0.0 of R) HTML files that give information on the package or on specific aspects of the package. To get details of vignettes that are available in a package, call `browseVignettes()` with the package name (as a character string) as argument. Thus, for the *knitr* package, enter `browseVignettes(package="knitr")`.

Vignettes are created from a Markdown or HTML or LaTeX document in which R code is embedded, surrounded by markup that controls what is to be done with the code and with any output generated. See Section 2.4.

The browser window that appears will list the vignettes, with the option to click on links that, in most cases, offer a choice of one of PDF and HTML, source, and R code.

Searching for Packages

A good first place to look, for information on packages that relate to one or other area of knowledge, is the R Task Views web page, at: `http://cran.r-project.org/web/views/`. See also the website `http://crantastic.org/`, which has details on what packages are popular, and what users think of them.

1.9 Summary and Exercises

1.9.1 Summary

One use of R is as a calculator, to evaluate arithmetic expressions. Calculations can be carried out in parallel, across all elements of a vector at once.

The R Commander GUI can be helpful in getting quickly into use of R for many standard purposes. It may, depending on requirements, be limiting for serious use of R.

Use `q()` to quit from an R session. To retain objects in the workspace, accept the offer to save the workspace.

- Useful help functions are `help()` (for getting information on a known function) and `help.search()` (for searching for a word that is used in the header for the help file).

NB also: Use `apropos()` to search for functions that include a stated text string as part of their name.

- The function `help.start()` starts a browser window from which R help information can be accessed.
- Use the GUI interface in RStudio or R Commander to input rectangular files. Or, use `read.table()` or one of its aliases.

Aliases of `read.table()` include `read.csv()` and `read.delim()`

- Data frames collect together under one name columns that all have the same length. Columns of a data frame can be any mix of, among other possibilities: logical, numeric, character, or factor.
- The function `with()` attaches a data frame temporarily, for the duration of the call to `with()`.

Use `with()` in preference to the `attach()` / `detach()` combination.

- For simple forms of scatterplot, use `plot()` and associated functions, or perhaps the *lattice* function `xyplot()`.

1.9.2 *Exercises*

1. Use the following code to to place the file `bestTimes.txt` in the working directory:

(a) Examine the file, perhaps using the function `file.show()`. Read the file into the workspace, with the name `bestTimes`.

```
## file.show("bestTimes.txt")
bestTimes <- read.table("bestTimes.txt")
```

(b) The `bestTimes` file has separate columns that show hours, minutes and seconds. Use the following to add the new column `Time`, then omitting the individual columns as redundant

```
## Exercise 1b
bestTimes$Time <- with(bestTimes,
                       h*60 + min + sec/60)
  # Time in minutes
names(bestTimes)[2:4]   # Check column names
bestTimes <- bestTimes[, -(2:4)]
                        # omit columns 2:4
```

(c) Here are alternative ways to plot the data

```
plot(Time ~ Distance, data=bestTimes)
## Now use a log scale
plot(log(Time) ~ log(Distance), data=bestTimes)
plot(Time ~ Distance, data=bestTimes, log="xy")
```

(d) Now save the data into an image file in the working directory

```
save(bestTimes, file="bestTimes.RData")
```

Subsection 2.2.2 discusses the use of the function `save()`.

2. Re-enter the data frame `travelbooks`.[4] Add a column that has the density (`weight/volume`) of each book.

[4] If necessary, refer back to Section 1.6 for details.

3. The functions `summary()` and `str()` both give summary information on the columns of a data frames Comment on the differences in the information provided, when applied to the following data frames from the *DAAG* package:

(a) `nihills`;
(b) `tomato`.

4. Examine the results from entering:

(a) `?minimum`
(b) `??minimum`
(c) `??base::minimum`
(d) `??base::min`

For finding a function to calculate the minimum of a numeric vector, which of the above gives the most useful information?

The notation `base::minimum` tells the help function to look in R's base package.

5. For each of the following tasks, applied to a numeric vector (numeric column object), find a suitable function. Test each of the functions that you find on the vector `volume` in Section 1.5:

(a) Reverse the order of the elements in a column object;
(b) Calculate length, mean, median, minimum maximum, range;
(c) Find the differences between successive values.

2
The R Working Environment

Object	Objects can be data objects, function objects, formula objects, expression objects, . . . Use `ls()` to list contents of current workspace.
Workspace	User's "database", where the user can make additions or changes or deletions.
Working directory	Default directory for reading or writing files. Use a new working directories for a new project.
Image files	Use to store R objects, e.g., workspace contents. (The expected file extension is **.RData** or **.rda**)
Search list	`search()` lists 'databases' that R will search. `library()` adds packages to the search list

Important R technical terms include *object*, workspace*, working directory*, *image file*, *package*, *library*, *database* and *search list*.

Use the relevant menu. or enter `save.image()` on the command line, to store or back up workspace contents. During a long R session, do frequent saves!

2.1 The Working Directory and the Workspace

Each R session has a *working directory* and a workspace. If not otherwise instructed, R looks in the *working directory* for files, and saves files that are output to it.

The *workspace* is at the base of a list of search locations, known as *databases*, where R will as needed search for objects. It holds objects that the user has created or input, or that were there at the start of the session and not later removed.

The workspace is a *volatile* database that, unless saved, will disappear at the end of the session.

The workspace changes as objects are added or deleted or modified. Upon quitting from R (type `q()`, or use the relevant menu item), users are asked whether they wish to save the current workspace. If saved, it is stored in the file **.RData**, in the current working directory. When an R session is next started in that working directory, the off-the-shelf action is to look for a file named **.RData**, and if found to reload it.

The file `.RData` has the name *image* file. From it the workspace can, as and when required, be reconstructed.

Setting the Working Directory

When a session is started by clicking on a Windows icon, the icon's Properties specify the Start In directory.[1] Type `getwd()` to identify the current working directory.

[1] When a Unix or Linux command starts a session, the default is to use the current directory.

It is good practice to use a separate working directory, and associated workspace or workspaces, for each different project. On Windows systems, copy an existing R icon, rename it as desired, and change the Start In directory to the new working directory. The working directory can be changed[2] once a session has started, either from the menu (if available) or from the command line. Changing the working directory from within a session requires a clear head; it is usually best to save one's work, quit, and start a new session.

[2] To make a complete change to a new workspace, first save the existing workspace, and type `rm(list=ls(all=TRUE)` to empty its contents. Then change the working directory and load the new workspace.

2.2 *Code, data, and project Maintenance*

2.2.1 *Maintenance of R scripts*

It is good practice to maintain a transcript from which work done during the session, including data input and manipulation, can as necessary be reproduced. Where calculations are quickly completed, this can be re-executed when a new session is started, to get to the point where the previous session left off.

Note again RStudio's abilities for managing and keeping R scripts.

2.2.2 *Saving and retrieving R objects*

Use `save()` to save one or more named objects into an image file. Use `load()` to reload the image file contents back into the workspace. The following demonstrate the explicit use of `save()` and `load()` commands:

The command `save.image())` saves everything in the workspace, by default into a file named **.RData** in the working directory. Or, from a GUI interface, click on the relevant menu item.

```
volume <- c(351, 955, 662, 1203, 557, 460)
weight <- c(250, 840, 550, 1360, 640, 420)
save(volume, weight, file="books.RData")
  # Can save many objects in the same file
rm(volume, weight)      # Remove volume and weight
load("books.RData")     # Recover the saved objects
```

See Subsection 2.3.2 for use of `attach("books.RData")` in place of `load("books.RData")`.

Where it will be time-consuming to recreate objects in the workspace, users will be advised to save (back up) the current workspace image from time to time, e.g., into a file, preferably with a suitably mnemonic name. For example:

```
fnam <- "2014Feb1.RData"
save.image(file=fnam)
```

Before saving the workspace, consider use of `rm()` to remove objects that are no longer required.

Two further possibilities are:

- Use `dump()` to save one or more objects in a text format. For example:

  ```
  volume <- c(351, 955, 662, 1203, 557, 460)
  weight <- c(250, 840, 550, 1360, 640, 420)
  ```

```
dump(c("volume", "weight"), file="volwt.R")
rm(volume, weight)
source("volwt.R")       # Retrieve volume & weight
```

- Use `write.table()` to write a data frame to a text file.

2.3 *Packages and System Setup*

Packages	Packages are structured collections of R functions and/or data and/or other objects.
Installation of packages	R Binaries include 'recommended' packages. Install other packages, as required,
`library()`	Use to attach a package, e.g., `library(DAAG)` Once attached, a package is added to the list of "databases" that R searches for objects.

For download or installation of R or CRAN packages, use for preference a local mirror. In Australia `http://cran.csiro.au` is a good choice. The mirror can be set from the Windows or Mac GUI. Alternatively (on any system), type `chooseCRANmirror()` and choose from the list that pops up.

An R installation is structured as a library of packages.

- All installations should have the base packages (one of them is called *base*). These provide the infrastructure for other packages.
- Binaries that are available from CRAN sites include, also, all the recommended packages.
- Other packages can be installed as required, from a CRAN mirror site, or from another repository.

A number of packages are by default attached at the start of a session. To attach other packages, use `library()` as required.

To discover which packages are attached, enter one of:

```
search()
sessionInfo()
```

Use `sessionInfo()` to get more detailed information.

2.3.1 *Installation of R packages*

Section 1.3 described the installation of packages from the internet. Note also the use of `update.packages()` or its equivalent from the GUI menu. This identifies packages for which updates are available, offering the user the option to proceed with the update.

The function `download.packages()` allows the downloading of packages for later installation. The menu, or `install.packages()`, can then be used to install the packages from the local directory. For command line installation of packages that are in a local directory, call `install.packages()` with `pkgs` giving the files (with path, if necessary), and with the argument `repos=NULL`.

Arguments are a vector of package names and a destination directory `destdir` where the latest file versions will be saved as **.zip** or (MacOS X) **.tar.gz** files.

If for example the binary **DAAG_1.22.zip** has been downloaded to **D:\tmp\**, it can be installed thus

```
install.packages(pkgs="D:/DAAG_1.22.zip",
                 repos=NULL)
```

On Unix and Linux systems, gzipped tar files can be installed using the shell command:

R CMD INSTALL xx.tar.gz

(replace xx.tar.gz by the file name.)

On the R command line, be sure to replace the usual Windows backslashes by forward slashes.

Use `.path.package()` to get the path of a currently attached package (by default for all attached packages).

2.3.2 *The search path: library() and attach()*

The R system maintains a *search path* (a list of *databases*) that determines, unless otherwise specified, where and in what order to look for objects. The search list includes the workspace, attached packages, and a so-called `Autoloads` database. It may include other items also.

Database 1, where R looks first, is the user workspace, called `".GlobalEnv"`.

To get a snapshot of the search path, here taken after starting up and entering `library(MASS)`, type:

Packages other than *MASS* were attached at startup.

```
search()
```

```
 [1] ".GlobalEnv"         "package:MASS"
 [3] "tools:RGUI"         "package:stats"
 [5] "package:graphics"   "package:grDevices"
 [7] "package:utils"      "package:datasets"
 [9] "package:methods"    "Autoloads"
[11] "package:base"
```

If the process runs from RStudio, `"tools:rstudio"` will appear in place of `"tools:RGUI"`.

For more detailed information that has version numbers of any packages that are additional to base packages, type:

```
sessionInfo()
```

The '`::`' notation

Use notation such as `base::print()` to specify the package where a function or other object is to be found. This avoids any risk of ambiguity when two or more objects with the same name can be found in the search path.

In Subsection 7.2.9 the notation `latticeExtra::layer()` will be used to indicate that the function `layer()` from the *latticeExtra* package is required, distinguishing it from the function `layer()` in the *ggplot2* package. Use of the notation `latticeExtra::layer()` makes unnecessary prior use of `library(latticeExtra)` or its equivalent.

It is necessary that the *latticeExtra* package has been installed!

Attachment of image files

The following adds the image file `books.RData` to the search list:

Objects that are attached, whether workspaces or packages (using `library()`) or other entities, are added to the search list.

```
attach("books.RData")
```

The session then has access to objects in the file **books.RData**. Note that if an object from the image file is modified, the modified copy becomes part of the workspace.

The file becomes a further "database" on the search list, separate from the workspace.

In order to detach `books.RData`, proceed thus:

```
detach("file:books.RData")
```

Alternatively, supply the numeric position of `books.RData` on the search list (if in position 2, then 2) as an argument to `detach()`.

2.3.3 **Where does the R system keep its files?*

Type `R.home()` to see where the R system has stored its files.

Note that R expects (and displays) either a single forward slash or double backslashes, where Windows would show a single backslash.

```
R.home()
```

```
[1] "/Library/Frameworks/R.framework/Resources"
```

Notice that the path appears in abbreviated form. Type `normalizePath(R.home())` to get the more intelligible result

```
[1] "C:\\Program Files\\R\\R-2.15.2"
```

By default, the command `system.file()` gives the path to the base package. For other packages, type, e.g.

```
system.file(package="DAAG")
```

```
[1] "/Users/johnm1/Library/R/3.4/library/DAAG"
```

To get the path to a file **viewtemps.RData** that is stored with the *DAAG* package in the **misc** subdirectory, type:

```
system.file("misc/viewtemps.RData", package="DAAG")
```

2.3.4 Option Settings

Type `help(options)` to get full details of option settings. There are a large number. To change to 60 the number of characters that will be printed on the command line, before wrapping, do:

```
options(width=60)
```

To display the setting for the line width (in characters), type:

```
options()$width
```

```
[1] 54
```

The printed result of calculations will, unless the default is changed (as has been done for most of the output in this document) often show more significant digits of output than are useful. The following demonstrates a global (until further notice) change:

```
sqrt(10)
```

```
[1] 3.162
```

```
opt <- options(digits=2) # Change until further notice,
                         # or until end of session.
sqrt(10)
```

```
[1] 3.2
```

```
options(opt)             # Return to earlier setting
```

Use `signif()` to affect one statement only. For example

```
signif(sqrt(10),2)
```

NB also the function `round()`.

Note that `options(digits=2)` expresses a wish, which R will not always obey!

Rounding will sometimes introduce small inconsistencies!

For example:

```
round(sqrt(85/7), 2)
```

```
[1] 3.48
```

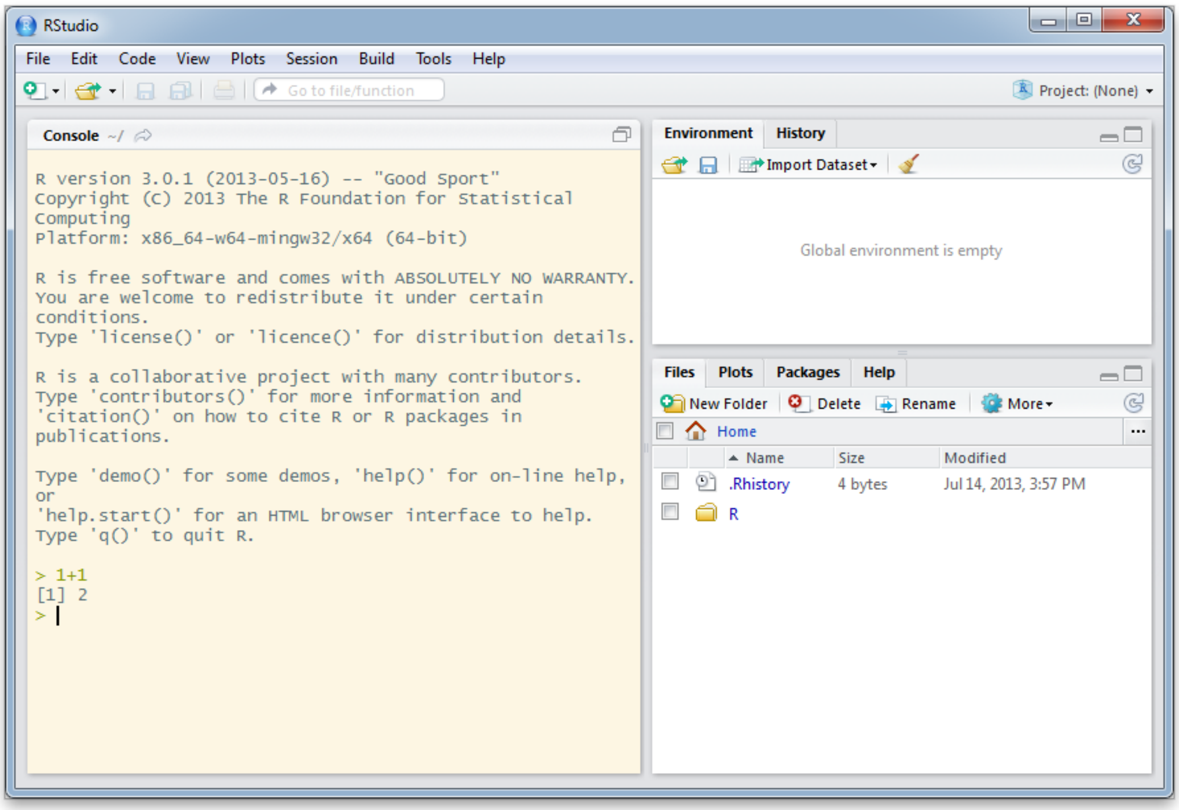
```
round(c(sqrt(85/7)*9,  3.48*9), 2)
```

```
[1] 31.36 31.32
```

2.4 Enhancing the R experience — RStudio

The url for RStudio is `http://www.rstudio.com/`. Click on the icon for the downloaded installation file to install it. An RStudio icon will appear. Click on the icon to start RStudio. RStudio should find any installed version of R, and if necessary start R. Figure 2.1 shows an RStudio display, immediately after starting up and entering, very unimaginatively, 1+1.

The screenshots here are for version 0.98.501 of RStudio.


RStudio
File Edit Code View Plots Session Build Tools Help
Go to file/function
Project: (None)
Console ~/
Environment History
Import Dataset
Global environment is empty
Files Plots Packages Help
New Folder Delete Rename More
Home
Name Size Modified

```
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical
Computing
Platform: x86_64-w64-mingw32/x64 (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.

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
[1] 2
>
```

Name	Size	Modified
.Rhistory	4 bytes	Jul 14, 2013, 3:57 PM
R		

Figure 2.1: Here is shown the RStudio interface, after starting up and entering 1+1.

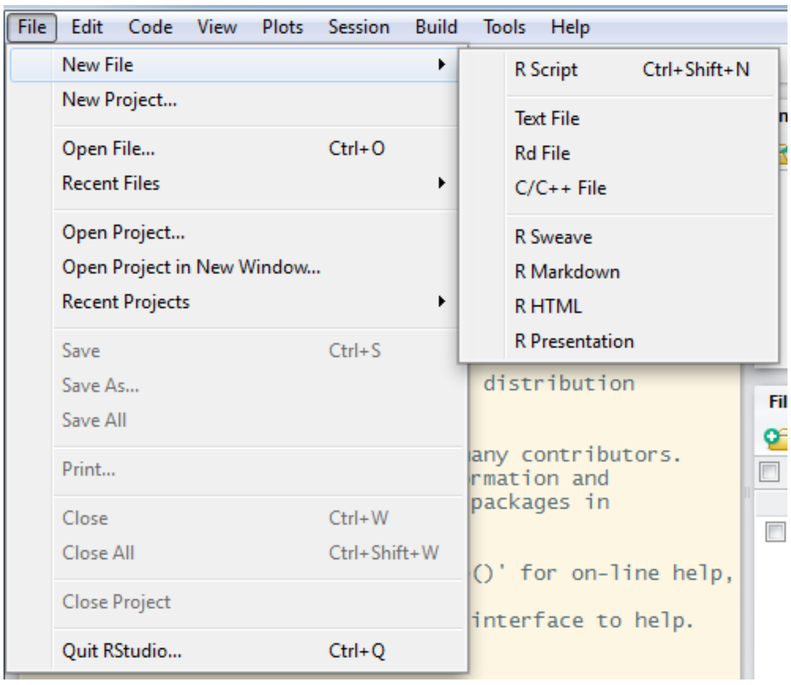
Techncally, RStudio offers an Interactive Development Environment. It provides, from a graphical user interface, a range of abilities that are helpful for organizing and managing work with R. Helpful features of RStudio include:

Extensive and careful RStudio documentation can be accessed, assuming an internet connection, from the Help drop-down menu. The notes included here are designed to draw attention to some of the more important RStudio abilities and features.

- The organisation of work into projects.
- The recording of files that have been accessed from RStudio, of help pages accessed, and of plots. The record of files is maintained from one session of a project to the next.
- By default, a miniature is displayed of any graph that is plotted. A single click expands the miniature to a full graphics window.
- The editing, maintenance and display of code files.
- Abilities that assist reproducible reporting. Markup text surrounds R code that is incorporated into a document, with option settings used to control the inclusion of code and/or computer output in the final document. Output may include tables and graphs.
- Abilities that help in the creation of packages.

Alternative available types of markup are R Markdown or R HTML or Sweave with LaTeX.

2.4.1 *The RStudio file menu*



```
distribution

any contributors.
rmation and
packages in

()' for on-line help,

interface to help.
```

Figure 2.2: The RStudio File drop-down menu. The New File submenu has been further expanded.

For now, the RStudio drop-down menus that are of most immediate importance are File and Help. Here (Figure 2.2) is the File menu, with the New File submenu also shown.

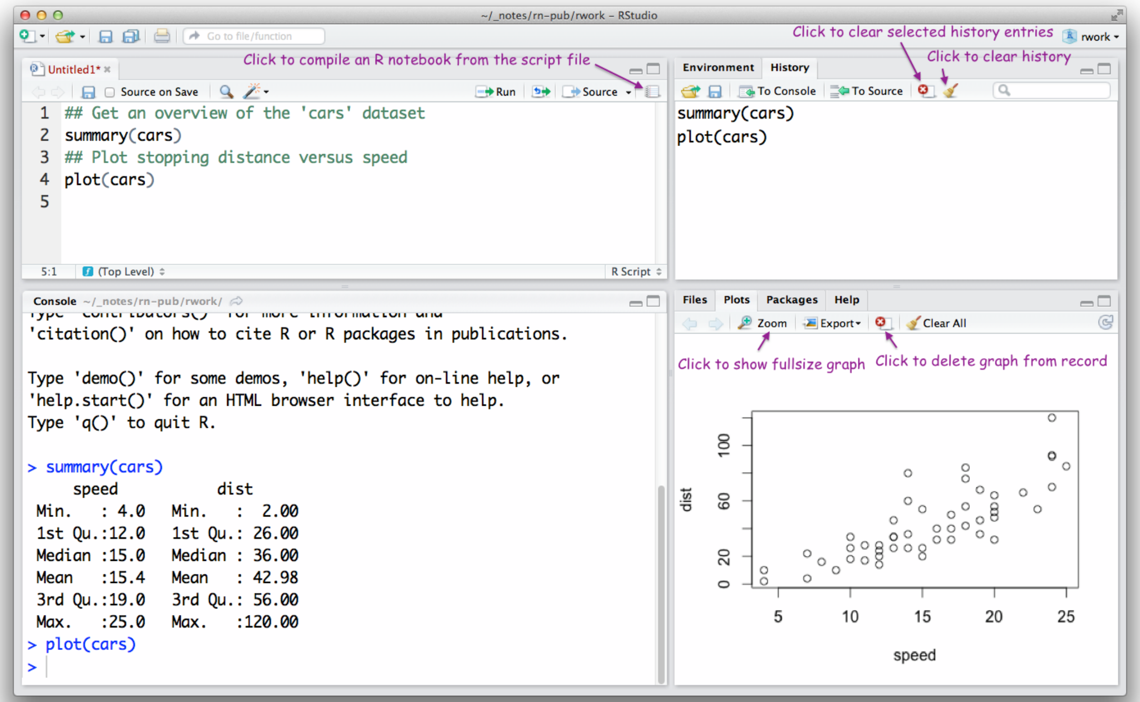
Here, note the possibility of opening a new R script file, and entering code into that file. Or, to open an existing R code file, click on the Open File... submenu.

The key combination <CTRL><ENTER> can be used to send code to the command line. Code that has been selected will be sent to the command line. Or if no code has been selected, the line on which the cursor is placed will be sent to the command line.

Here, <CTRL> is the control key and <ENTER> is the Enter key.

2.4.2 *Compile a code notebook*

Figure 2.3 shows a script file in the upper left panel. The code has been sent to the command line, so that it also appears in the code history panel on the upper right.


~/_notes/rn-pub/rwork - RStudio
Click to clear selected history entries
Click to clear history
Click to compile an R notebook from the script file
Untitled1*
Source on Save
Run
Source
Environment
History
To Console
To Source
5:1
(Top Level)
R Script
Console ~/_notes/rn-pub/rwork/
Files
Plots
Packages
Help
Zoom
Export
Clear All
Click to show fullsize graph
Click to delete graph from record
dist
speed

```
## Get an overview of the 'cars' dataset
summary(cars)
## Plot stopping distance versus speed
plot(cars)
```

```
summary(cars)
plot(cars)
```

```
'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.

> summary(cars)
     speed           dist       
 Min.   : 4.0   Min.   :  2.00  
 1st Qu.:12.0   1st Qu.: 26.00  
 Median :15.0   Median : 36.00  
 Mean   :15.4   Mean   : 42.98  
 3rd Qu.:19.0   3rd Qu.: 56.00  
 Max.   :25.0   Max.   :120.00  
> plot(cars)
>
```

Figure 2.3: Code from the script window has been sent to the command line.

In Figure 2.3, take particular note of the icon on which you can click to create an R notebook. Upon clicking this icon, the system will ask for a name for the file. It will then create an HTML file that has, along with the code and comment, the compluter output. An alternative to clicking on the icon is to click on the File drop-down menu, and then on Compile Notebook... .

For the code that is shown, the HTML file that results will include the output from `summary(cars)` and the graph from `plot(cars)`.

2.5 *Abilities for reproducible reporting*

Markdown editors use simple markup conventions to control how text and other document features will appear. For example:

`**Help**` or `__Help__` will be rendered as **Help**

`*Help*` or `_Help_` will be rendered as *Help*.

2.5.1 *R Markdown*

Click on File | New File | R Markdown.... Clicking on HTML (alternatives are PDF, Word), on Document (alternatives are Presentation, Shiny, From Template) and then on OK displays a simple skeleton R Markdown document thus:

R Markdown, as available under RStudio, is an enhanced version of Markdown. It adds the ability to include R code, surrounded by markup that controls what code and/or output will appear in the final document.

R users are strongly encouraged to use R Markdown, or another such markup system that allows embedded R code, for documenting any work that is more than trivial. Those who are familiar with more sophisticated markdown languages may still, for some types of work, find benefit in the simplicity and speed of working with R markdown.

~~~
---
title: "Untitled"
output: html_document
---

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS
Word documents. For more details on using R Markdown
see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will
be generated that includes both content as well as
the output of any embedded R code chunks within the
document. You can embed an R code chunk like this:

```{r}
summary(cars)
~~~

You can also embed plots, for example:

```{r, echo=FALSE}
plot(cars)
```

Note that the `echo = FALSE` parameter was added
to the code chunk to prevent printing of the R
code that generated the plot.
```

For tutorial purposes, the file can be processed as it stands. Click the Knit HTML button to start the process of generating the HTML file. When prompted, enter a name for the file. An HTML file will be generated and displayed in a browser.

In actual use, one would edit out the text and R code and replace it with one's own text and R code chunks, then clicking on Knit HTML. When prompted, enter a name for the file.
```

## *R Markdown code chunk options*

The markup that surrounds R code can include instructions on what to do with R code and/or any output, including tables and graphs. Should code be executed, should it be echoed, and what output text and/or tables and/or graphs should appear in the final document?

Here is an example of code with surrounding markup, with the code chunk options `fig.width` and `fig.height` giving the width and height of the initial figure, and `out.width` giving the width to which it should be scaled in the final document:

```
‘‘‘{r plotgph, fig.width=7, fig.height=6, out.width="80%"}
plot(cars)
‘‘‘
```

Giving the code chunk a name, here `plotgph`, is optional. The `fig.width` and `fig.height` settings control the size of the output plot, before it is scaled to fit within the available line width. The `out.width` setting controls the width (here given as a percentage of the line width) in the final HTML document. The width may alternatively be given in pixels, e.g., ‘out.width="600px"‘.

Other possible settings include: `echo=FALSE` (do not show code), & `eval=FALSE` (do not evaluate).

An image from a file **pic.png** that has been generated separately from the markup R code can be input thus:

```
‘‘‘{r, out.width="80%"}
knitr::include\_graphics("pic.png")
‘‘‘
```

**Inclusion of HTML in R Markdown documents*

Note also that HTML markup can be included in R Markdown documents. The following is a less preferred alternative to the R code `knitr::include_graphics("pic.png")` whose use was demonstrated above:

```
<IMG SRC="pic.png" alt="Show this, if no image" STYLE="width: 1200px"/>
```

The image position can if necessary be adjusted thus:

```
<IMG SRC="pic.png" alt="Show this, if no image" STYLE="position:absolute;
TOP:-25px; LEFT:40px; WIDTH:800px; HEIGHT:500px"/>
```

R Presentation

Note the R Presentation variant of R Markdown. To display a simple skeleton document, click on:

File | New File | R Presentation

An R Presentation document is a specific type of R Markdown document that is formatted to provide slides that can be displayed using a browser.

Click on Knit HTML to process the document, either as it stands or after replacing the sample text and code with one’s own text and code.

2.5.2 *Other markup types – HTML, LaTeX, . . .

R HTML

Click on File | New File | R HTML to display a skeleton HTML document that has embedded R code. The following shows the markup format:

Also available is reStructuredText (reST), which is an extended variant of R Markdown.

```
<!--begin.rcode fig.width=7, fig.height=6, out.width="600px"
plot(cars)
end.rcode-->
```

Again, the document that appears can be processed as it stands – click on Knit HTML.

R Sweave:

Click on File | New File | R Sweave to display a template for a LaTeX file. The web page http://maths-people.anu.edu.au/~johnm/r-book/knitr/ has files that demonstrate the use of *knitr* Sweave type markup.

2.5.3 RStudio documentation – markup and other

Extensive RStudio documentation is available online. Click on Help | RStudio Docs to go to the relevant web page. For R Markdown and R Presentation, note the documentation files for **Using R Markdown**. LaTeX users should note the **Sweave and knitr** documentation files.

2.5.4 A strategy for RStudio project management

RStudio is designed to encourage good project management practices, using a strategy akin to the following:

Set up each new project in its own working directory.

For each project, maintain one or more script files that holds the code. Script files can be compiled into "notebooks" for purposes of keeping a paper record.

Script files are readily expanded into R Markdown documents – a simple form of "reproducible reporting" document. They can as required be expanded into a draft for a paper.

2.6 Summary and Exercises

2.6.1 Summary

Each R session has a working directory, where R will by default look for files or store files that are external to R.

User-created R objects are added to the workspace, which is at the base of a search list, i.e., a list of “databases” that R will search when it looks for objects.

From within functions, R will look first in the functions environment, and then if necessary look within the search list.

It is good practice to keep a separate workspace and associated working directory for each major project. Use script files to keep a record of work.

At the end of a session an image of the workspace will typically (respond “y” when asked) be saved into the working directory.

Before making big changes to the workspace, it may be wise to save the existing workspace under a name (e.g., `Aug27.RData`) different from the default `.RData`.

Note also the use of `attach()` to give access to objects in an image (**.RData** or **.rda**) file.[3]

[3] Include the name of the file (optionally preceded by a path) in quotes.

R has an extensive help system. Use it!

2.6.2 *Exercises*

Data files used in these exercises are available from the web page `http://www.maths.anu.edu.au/~johnm/datasets/text/`.

The function `DAAG::datafile()` is able to place in the working directory any of the files: **fuel.txt** **molclock1.txt**, **molclock2.txt**, **travelbooks.txt**. Specify, e.g.
```
datafile(file="fuel")
```

1. Place the file **fuel.txt** to your working directory.

2. Use `file.show()` to examine the file, or click on the RStudio Files menu and then on the file name to display it. Check carefully whether there is a header line. Use the RStudio menu to input the data into R, with the name `fuel`. Then, as an alternative, use `read.table()` directly. (If necessary use the code generated by RStudio as a crib.) In each case, display the data frame and check that data have been input correctly.

A shortcut for placing these files in the working directory is:
```
datafile(file=c("molclock1",
                "molclock2"))
```

3. Place the files **molclock1.txt** and **molclock2.txt** in a directory from which you can read them into R. As in Exercise 1, use the RStudio menu to input each of these, then using `read.table()` directly to achieve the same result. Check, in each case, that data have been input correctly.

   Use the function `save()` to save `molclock1`, into an R image file. Delete the data frame `molclock1`, and check that you can recover the data by loading the image file.

4. The following counts, for each species, the number of missing values for the column `root` of the data frame `DAAG::rainforest`:

```
library(DAAG)
with(rainforest, table(complete.cases(root), species))
```

   For each species, how many rows are “complete”, i.e., have no values that are missing?

5. For each column of the data frame `MASS::Pima.tr2`, determine the number of missing values.

6. The function `dim()` returns the dimensions (a vector that has the number of rows, then number of columns) of data frames and matrices. Use this function to find the number of rows in the data frames `tinting`, `possum` and `possumsites` (all in the *DAAG* package).

7. Use `mean()` and `range()` to find the mean and range of:

   (a) the numbers 1, 2, . . . , 21

   (b) the sample of 50 random normal values, that can be generated from a normaL distribution with mean 0 and variance 1 using the assignment `y <- rnorm(50)`.

   (c) the columns `height` and `weight` in the data frame `women`.

   Repeat (b) several times, on each occasion generating a nwe set of 50 random numbers.

The *datasets* package that has the data frame `women` is by default attached when R is started.

8. Repeat exercise 6, now applying the functions `median()` and `sum()`.

9. Extract the following subsets from the data frame `DAAG::ais`

   (a) Extract the data for the rowers.

   (b) Extract the data for the rowers, the netballers and the tennis players.

   (c) Extract the data for the female basketballers and rowers.

10. Use `head()` to check the names of the columns, and the first few rows of data, in the data frame `DAAG::rainforest`. Use `table(rainforest$species)` to check the names and numbers of each species that are present in the data. The following extracts the rows for the species *Acmena smithii*

```
Acmena <- subset(rainforest, species=="Acmena smithii")
```

The following extracts the rows for the species `Acacia mabellae` and `Acmena smithii`:

```
AcSpecies <- subset(rainforest, species %in% c("Acacia mabellae",
                                               "Acmena smithii"))
```

Now extract the rows for all species except `C. fraseri`.

3
Examples — Data analysis with R

Scatterplot matrices	Scatterplot matrices can give useful insights on data that will be used for regression or related calculations.
Transformation	Data often require transformation prior to entry into a regression model.
Model objects	Fitting a regression or other such model gives, in the first place, a model object.
Generic functions	`plot()`, `print()` and `summary()` are examples of *generic* functions. With a dataframe as argument `plot()` gives a scatterplot matrix. With an `lm` object, it gives diagnostic plots.
Extractor function	Use an extractor function to extract output from a model object. Extractor fucntions are *generic* functions
List objects	An `lm` model object is a list object. Lists are used extensively in R.

This chapter will use examples to illustrate common issues in the exploration of data and the fitting of regression models. It will round out the discussion of Chapters 1 and 2 by adding some further important technical details.

Issues that will be noted include the use of *generic* functions such as `plot()` and `print()`, the way that regression model objects are structured, and the use of extractor functions to extract information from model objects.

Notation, when referring to datasets

Data will be used that is taken from several different R packages. The notation `MASS::mammals`, which can be used in code as well as in the textual description, makes it clear that the dataset `mammals` that is required is from the *MASS* package. Should another attached package happen to have a dataset `mammals`, there is no risk of confusion.

3.1 The Uses of Scatterplots

```
## Below, the dataset MASS::mammals will be required
library(MASS, quietly=TRUE)
```

3.1.1 Transformation to an appropriate scale

A first step is to elicit basic information on the columns in the data, including information on relationships between explanatory variables. Is it desirable to transform one or more variables?

Among other issues, is there a wide enough spread of distinct values that data can be treated as continuous.

Transformations are helpful that ensure, if possible, that:

- All columns have a distribution that is reasonably well spread out over the whole range of values, i.e., it is unsatisfactory to have most values squashed together at one end of the range, with a small number of very small or very large values occupying the remaining part of the range.
- Relationships between columns are roughly linear.
- the scatter about any relationship is similar across the whole range of values.

It may happen that the one transformation, often a logarithmic transformation, will achieve all these at the same time.

The scatterplot in Figure 3.1A, showing data from the dataset `MASS::mammals`, is is an extreme version of the common situation where positive (or non-zero) values are squashed together in the lower part of the range, with a tail out to the right. Such a distribution is said to be "skewed to the right".

Code for Figure 3.1A

```
plot(brain ~ body, data=mammals)
mtext(side=3, line=0.5, adj=0,
      "A: Unlogged data", cex=1.1)
```

Figure 3.1B shows the scatterplot for the logged data. Code for Figure 3.1B is:

```
plot(brain ~ body, data=mammals, log="xy")
mtext(side=3, line=0.5, adj=0,
      "B: Log scales (both axes)", cex=1.1)
```

Where, as in Figure 3.1A, values are concentrated at one end of the range, the small number (perhaps one or two) of values that lie at the other end of the range will, in a straight line regression with that column as the only explanatory variable, be a leverage point. When it is one explanatory variable among several, those values will have an overly large say in determining the coefficient for that variable.

As happened here, a logarithmic transformation will often remove much or all of the skew. Also, as happened here, such transformations often bring the added bonus that relationships between the resulting variables are approximately linear.

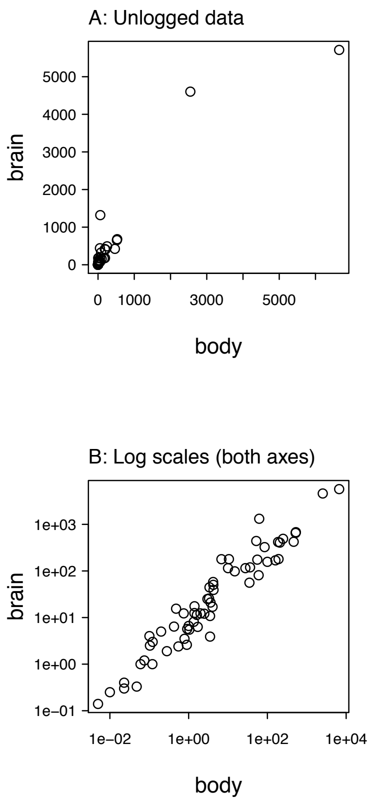


Figure 3.1: Brain weight (g) versus Body weight (kg), for 62 species of mammal. Panel A plots the unlogged data, while Panel B has log scales for both axes, but with axis labels in the original (unlogged) units.

3.1.2 The Uses of Scatterplot Matrices

Subsequent chapters will make extensive use of scatterplot matrices. A scatterplot matrix plots every column against every other column, with the result in the layout used for correlation matrices. Figure 3.2 shows a scatterplot matrix for the `datasets::trees` dataset.

The `datasets` package is, in an off-the-shelf installation, attached when R starts.

Interpreting Scatterplot Matrices:

For identifying the axes for each panel

- look across the row to the diagonal to identify the variable on the vertical axis.
- look up or down the column to the diagonal for the variable on the horizontal axis.

Each below diagonal panel is the mirror image of the corresponding above diagonal panel.

```
## Code used for the plot
plot(trees, cex.labels=1.5)
  # Calls pairs(trees)
```

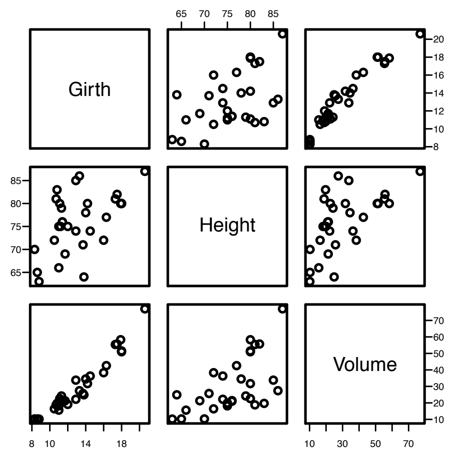


Figure 3.2: Scatterplot matrix for the `trees` data, obtained using the default `plot()` method for data frames. The scatterplot matrix is a graphical counterpart of the correlation matrix.

Notice that `plot()`, called with the dataframe `trees`, has in turn called the plot method for a data frame, i.e., it has called `plot.data.frame()` which has in turn called the function `pairs()`.

The scatterplot matrix may be examined, if there are enough points, for evidence of:

1. Strong clustering in the data, and/or obvious outliers;
2. Clear non-linear relationships, so that a correlation will underestimate the strength of any relationship;
3. Severely skewed distributions, so that the correlation is a biased measure of the strength of relationship.

The scatterplot matrix is best used as an initial coarse screening device. Skewness in the individual distributions is better checked using plots of density estimates.

3.2 World record times for track and field events

The first example is for world track and road record times, as at 9th August 2006. Data, copied down from the web page `http://www.gbrathletics.com/wrec.htm`, are in the dataset `DAAG::worldRecords`.

Note also the use of these data in the exercise at the end of Chapter 2 (Section 1.9.2)

Data exploration

First, use `str()` to get information on the data frame columns:

```
library(DAAG, quietly=TRUE)
```

```
str(worldRecords, vec.len=3)
```

```
'data.frame':   40 obs. of  5 variables:
 $ Distance   : num  0.1 0.15 0.2 0.3 0.4 0.5 0.6 0.8 ...
 $ roadORtrack: Factor w/ 2 levels "road","track": 2 2 2 2 2 2 2 2 ...
 $ Place      : chr  "Athens" "Cassino" "Atlanta" ...
 $ Time       : num  0.163 0.247 0.322 0.514 ...
 $ Date       : Date, format: "2005-06-14" ...
```

Distinguishing points for track events from those for road events is easiest if we use lattice graphics, as in Figure 3.3.

```
## Code
library(lattice)
xyplot(Time ~ Distance, scales=list(tck=0.5),
       groups=roadORtrack, data=worldRecords,
       auto.key=list(columns=2), aspect=1)
## On a a colour device the default is to use
## different colours, not different symbols,
## to distinguish groups.
```

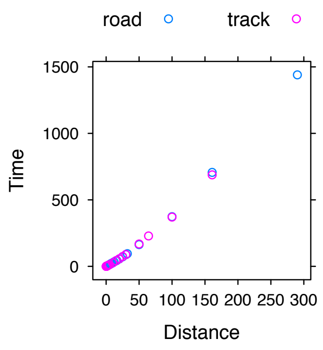


Figure 3.3: World record times versus distance, for field and road events.

Clearly increases in `Time` are not proportional to increases in `Distance`. Indeed, such a model does not make sense; velocity decreases as the length of the race increases. Proportionality when logarithmic scales are used for the two variables does make sense.

Figure 3.4 uses logarithmic scales on both axes. The two panels differ only in the labeling of the scales. The left panel uses labels on scales of $\log_e$, while the right panel has labels in the orginal units. Notice the use of `auto.key` to obtain a key.

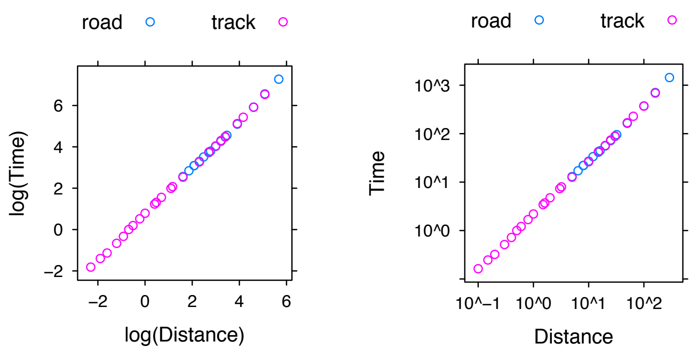


Figure 3.4: World record times versus distance, for field and road events, using logarithmic scales. The left panel uses labels on scales of $\log_e$, while in the right panel, labeling is in the orginal units, expressed as powers of 10.

```
## Code for Left panel
xyplot(log(Time) ~ log(Distance),
       groups=roadORtrack, data=worldRecords,
       scales=list(tck=0.5),
       auto.key=list(columns=2), aspect=1)
## Right panel
xyplot(Time ~ Distance, groups=roadORtrack,
       data=worldRecords,
       scales=list(log=10, tck=0.5),
       auto.key=list(columns=2), aspect=1)
```

Fitting a regression line

The plots suggest that a line is a good fit. Note however that the data span a huge range of distances. The ratio of longest to shortest distance is almost 3000:1. Departures from the line are of the order of 15% at the upper end of the range, but are so small relative to this huge range that they are not obvious.

The following uses the function `lm()` to fit a straight line fit to the logged data, then extracting the regression coefficients:

The name `lm` is a mnemonic for linear model.

```
worldrec.lm <- lm(log(Time) ~ log(Distance),
                  data=worldRecords)
coef(worldrec.lm)
```

```
  (Intercept) log(Distance)
       0.7316        1.1248
```

The equation gives predicted times:

$$\begin{aligned}\widehat{\text{Time}} &= e^{0.7316} \times \text{Distance}^{1.1248}\\ &= 2.08 \times \text{Distance}^{1.1248}\end{aligned}$$

This implies, as would be expected, that kilometers per minute increase with increasing distance. Fitting a line to points that are on a log scale thus allows an immediate interpretation.

There is no difference that can be detected visually between the track races and the road races. Careful analysis will in fact find no difference.

3.2.1 Summary information from model objects

In order to avoid recalculation of the model information each time that some different information is required, we store the result from the `lm()` calculation in the model object `worldrec.lm`.

The name `worldrec.lm` is used to indicate that this is an `lm` object, with data from `worldRecords`. Use any name that seems helpful!

Note that the function `abline()` can be used with the model object as argument to add a line to the plot of `log(Time)` against `log(Distance)`.

Plot points; add line:

```
plot(log(Time) ~ log(Distance),
     data = worldRecords)
abline(worldrec.lm)
```

Diagnostic plots

Insight into the adequacy of the line can be obtained by examining the "diagnostic" plots, obtained by "plotting" the model object. Figure 3.5 following shows the first and last of the default plots:

By default, there are four "diagnostic" plots.

```
## Code
plot(worldrec.lm, which=c(1,5),
     sub.caption=rep("",2))
```

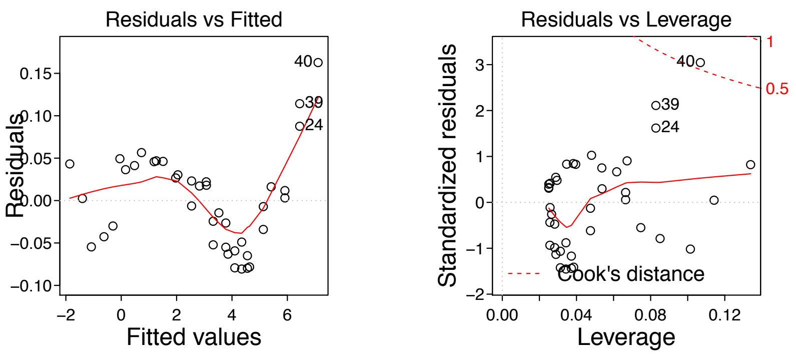


Figure 3.5: First and last of the default diagnostic plots, from the linear model for log(record time) versus log(distance), for field and road events.

Panel A is designed to give an indication whether the relationship really is linear, or whether there is some further systematic component that should perhaps be modeled. It does show systematic differences from a line.

The largest difference is more than a 15% difference.[1] There are mechanisms for using a smooth curve to account for the differences from a line, if these are thought important enough to model.

[1] A difference of 0.05 on a scale of $\log_e$ translates to a difference of just over 5%. A difference of 0.15 translates to a difference of just over 16%, i.e., slightly more than 15%.

The plot in panel B allows an assessment of the extent to which individual points are influencing the fitted line. Observation 40 does have both a very large leverage and a large Cook's distance. The plot on the left makes it clear that this is the point with the largest fitted time. Observation 40 is for a 24h race, or 1440 min. Examine

```
worldRecords["40", ]
```

```
   Distance roadORtrack Place Time       Date
40    290.2        road Basle 1440 1998-05-03
```

3.2.2 *The model object*

Functions that are commonly used to get information about model objects are: `print()`, `summary()` and `plot()`. These are all *generic* functions. The effect of the function depends on the class of object that is printed (ie, by default, displayed on the screen) or or plotted, or summarized.

The function `print()` may display relatively terse output, while `summary()` may display more extensive output. This varies from one type of model object to another.

Compare the outputs from the following:

```
print(worldrec.lm)    # Alternatively, type worldrec.lm
```

```

Call:
lm(formula = log(Time) ~ log(Distance), data = worldRecords)

Coefficients:
  (Intercept)  log(Distance)
        0.732          1.125
```

```
summary(worldrec.lm)
```

```

Call:
lm(formula = log(Time) ~ log(Distance), data = worldRecords)

Residuals:
    Min      1Q  Median      3Q     Max
-0.0807 -0.0497  0.0028  0.0377  0.1627

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.73160    0.01241      59   <2e-16
```

```
log(Distance)  1.12475    0.00437     257    <2e-16

Residual standard error: 0.0565 on 38 degrees of freedom
Multiple R²:  0.999,    Adjusted R²:  0.999
F-statistic: 6.63e+04 on 1 and 38 DF,  p-value: <2e-16
```

Used with `lm` objects, `print()` calls `print.lm()`, while `summary()` calls `summary.lm()`. Note that typing `worldrec.lm` has the same effect as `print(worldrec.lm)`.

Internally, `summary(wtvol.lm)` calls `UseMethod("summary")`. As `wtvol.lm` is an lm object, this calls `summary.lm()`.

3.2.3 The `lm` model object is a list

The model object is actually a list. Here are the names of the list elements:

```
names(worldrec.lm)
```

```
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

These different list elements hold very different classes and dimensions (or lengths) of object. Hence the use of a list; any collection of different R objects can be brought together into a list.

The following is a check on the model call:

```
worldrec.lm$call
```

```
lm(formula = log(Time) ~ log(Distance), data = worldRecords)
```

Commonly required information is best accessed using generic extractor functions. Above, attention was drawn to `print()`, `summary()` and `plot()`. Other commonly used extractor functions are `residuals()`, `coefficients()`, and `fitted.values()`. These can be abbreviated to `resid()`, `coef()`, and `fitted()`.

Use extractor function `coef()`:

```
coef(worldrec.lm)
```

3.3 Regression with two explanatory variables

The dataset `nihills` in the *DAAG* package will be used for a regression fit in Section 8.6. This has record times for Northern Ireland mountain races. Overview details of the data are:

```
str(nihills)
```

```
'data.frame':   23 obs. of  4 variables:
 $ dist : num  7.5 4.2 5.9 6.8 5 4.8 4.3 3 2.5 12 ...
 $ climb: int  1740 1110 1210 3300 1200 950 1600 1500 1500 5080 ...
 $ time : num  0.858 0.467 0.703 1.039 0.541 ...
 $ timef: num  1.064 0.623 0.887 1.214 0.637 ...
```

Figure 3.6 uses the lattice function `splom()` (from the *lattice* package) to give scatterplot matrices, one for the unlogged data,

The function `splom()` is a *lattice* alternative to `pairs()`, giving a different panel layout.

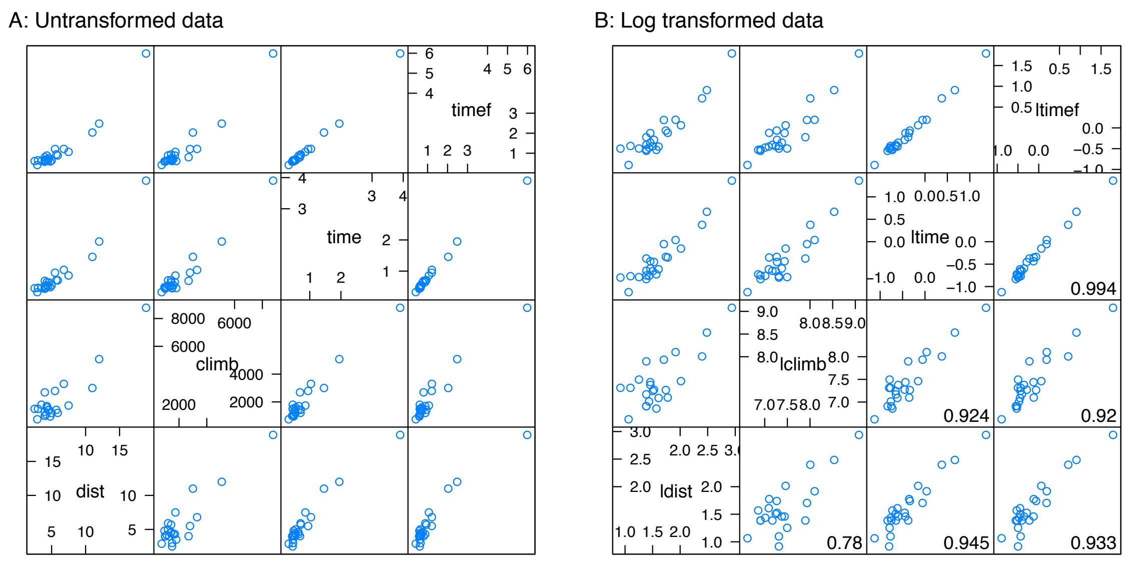


Figure 3.6: Scatterplot matrices for the Northern Ireland mountain racing data. The left panel is for the unlogged data, while the right panel is for the logged data. Code has been added that shows the correlations, in the lower panel.

and the other for the logged data. The left panel shows the unlogged data, while the right panel shows the logged data:

The following panel function was used to show the correlations:

```
showcorr <- function(x,y,...){
    panel.xyplot(x,y,...)
    xy <- current.panel.limits()
    rho <- paste(round(cor(x,y),3))
    eps <- 0.035*diff(range(y))
    panel.text(max(x), min(y)+eps, rho,
               pos=2, offset=-0.2)
}
```

Code for the scatterplot matrix in the left panel is:

```
## Scatterplot matrix; unlogged data
library(lattice)
splom(~nihills,  xlab="",
       main=list("A: Untransformed data", x=0,
       just="left", fontface="plain"))
```

For the right panel, create a data frame from the logged data:

```
lognihills <- log(nihills)
names(lognihills) <- paste0("l", names(nihills))
## Scatterplot matrix; log scales
splom(~ lognihills, lower.panel=showcorr, xlab="",
       main=list("B: Log transformed data", x=0,
       just="left", fontface="plain"))
```

Unlike `paste()`, the function `paste0()` does not leave spaces between text strings that it pastes together.

Note that the data are positively skewed, i.e., there is a long tail to the right, for all variables. For such data, a logarithmic transformation often gives more nearly linear relationships. The relationships between explanatory variables, and between the dependent variable and explanatory variables, are closer to linear when logarithmic scales are used. Just as importantly, issues with large leverage, so that the point for the largest data values has a much greater leverage and hence much greater influence than other points on the the fitted regression, are greatly reduced.

Notice also that the correlation of 0.913 between `climb` and `dist` in the left panel of Figure 3.6 is very different from the correlation of 0.78 between `lclimb` and `ldist` in the right panel. Correlations where distributions are highly skew are not comparable with correlations where distributions are more nearly symmetric. The statistical properties are different.

The following regresses `log(time)` on `log(climb)` and `log(dist)`:

```
nihills.lm <- lm(ltime ~ lclimb + ldist,
                 data=lognihills)
```

3.4 One-way Comparisons

The dataset `tomato` has weights of plants that were grown under one of four different sets of experimental comditions. Five plants were grown under each of the treatments:

A common strategy for getting a valid comparison is to grow the plants in separate pots, with a random arrangement of pots.

- `water only`
- `conc nutrient`
- `2-4-D + conc nutrient`
- `x conc nutrient`

Figure 3.7, created using the function `quickplot()` from the *ggplot2* package, shows the plant weights. Are the apparent differences between treatments large enough that they can be distinguished statistically?

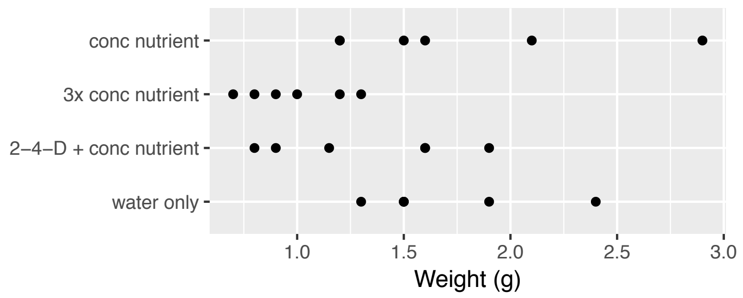


Figure 3.7: Weights (g) of tomato plants grown under four different treatments.

Notice that "water only" is made the reference level. This choice makes best sense for the analysis of variance calculations that appear below.

```
## Code
library(ggplot2)
tomato <- within(DAAG::tomato,
                 trt <- relevel(trt, ref="water only"))
quickplot(weight, trt, data=tomato,
          xlab="Weight (g)", ylab="")
```

The command `aov()`, followed by a call to `summary.lm()`, can be used to analyse these data, thus:

```
tomato.aov <- aov(weight ~ trt, data=tomato)
round(coef(summary.lm(tomato.aov)), 3)
```

Observe that, to get estimates and SEs of treatment effects, `tomato.aov` can be treated as an `lm` (regression) object.

```
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)                  1.683      0.187   9.019    0.000
trt2-4-D + conc nutrient    -0.358      0.264  -1.358    0.190
trt3x conc nutrient         -0.700      0.264  -2.652    0.015
trtconc nutrient             0.067      0.264   0.253    0.803
```

Because we made “`water only`” the reference level, “`(Intercept)`” is the mean for `water only`, and the other coefficients are for differences from `water only`.

A randomized block comparison

Growing conditions in a glasshouse or growth chamber — temperature, humidity and air movement — will not be totally uniform. This makes it desirable to do several repeats of the comparison between treatments[2], with conditions within each repeat ("block") kept as uniform as possible. Each different "block" may for example be a different part of the glasshouse or growth chamber.

[2] In language used originally in connection with agricultural field trials, where the comparison was repeated on different blocks of land, each different location is a "block".

The dataset `DAAG::rice` is from an experiment where there were six treatment combinations — three types of fertilizer were applied to each of two varieties of rice plant. There were two repeats, i.e., two blocks.

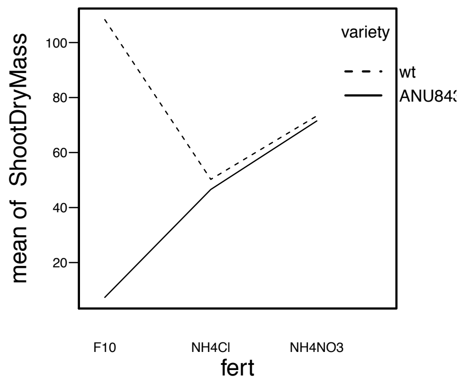


Figure 3.8: Interaction plot for the terms `fert` and `variety`, with `ShootDryMass` as the dependent variable. Notice that for fertilizer F10, there is a huge variety difference in the response. For the other fertilizers, there is no difference of consequence.

```
## Code
library(DAAG)
with(rice, interaction.plot(x.factor=fert,
                            trace.factor=variety,
                            ShootDryMass,
                            cex.lab=1.4))
```

For these data, Figure 3.8 gives a clear picture of the result. For fertilizers `NH4Cl` and `NH4NO3`, any difference between the varieties

The effect of an appropriate choice of clocks, then carrying out an analysis that accounts for block effects, is to allow a more precise comparison between treatments.

is inconsequential. There is strong "interaction" between `fert` and `variety`. A formal analysis, accounting for block differences, will confirm what seems already rather clear.

3.5 Time series – Australian annual climate data

The data frame `bomregions2012` from the *DAAG* package has annual rainfall data, both an Australian average and broken down by location within Australia, for 1900 – 2012. Figure 3.9 shows annual rainfall in the Murray-Darling basin, plotted against year.

Data are from the website `http://www.bom.gov.au/climate/change/`

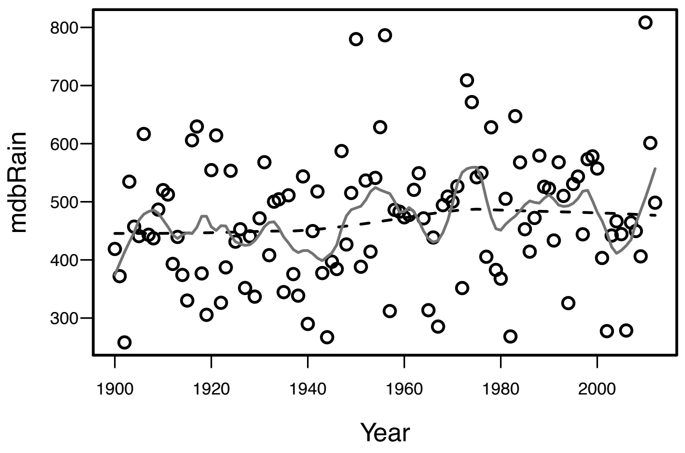


Figure 3.9: Annual rainfall in the Australian Murray-Darling Basin. by year. The `lowess()` function is used to The dashed curve with `f=2/3` captures the overall trend, while the solid curve with `f=0.1` captures trends on a scale of around eleven years. (10% of the 113 year range from 1900 to 2012 is a little more than 11 years.)

```
## Code
library(DAAG)
plot(mdbRain ~ Year, data=bomregions2012)
## Calculate and plot curve showing long-term trend
with(bomregions2012, lines(lowess(mdbRain ~ Year, f=2/3), lty=2))
## Calculate and plot curve of short-term trends
with(bomregions2012, lines(lowess(mdbRain ~ Year, f=0.1),
                           lty=1, col="gray45"))
```

The `lowess()` function has been used to fit smooth curves, formed by moving a smoothing window across the data. The dashed curve with `f=2/3` (include 2/3 of the data in the smoothing window) captures the overall trend in the data. The choice of `f=0.1` for the solid curve has the effect that somewhat more than ten years of data are used in determining each fitted value on the smooth.

For each smoothing window, a line or other simple response function is fitted. Greatest weight to points near the centre of the smoothing window, with weights tailing off to zero at the window edge.

This graph is exploratory. A next step might to model a correlation structure in residuals from the overall trend. There are extensive abilities for this. For graphical exploration, note `lag.plot()` (plot series against lagged series).

The functions `acf()` and `pacf()` might be used to examine the correlation structure in the residuals.

The cube root of average rainfall has a more symmetric distribution than rainfall. Thus, use this in preference to average rainfall when fitting models.

3.6 Exercises

1. Plot `Time` against `Distance`, for the `worldRecords` data. Ignoring the obvious curvature, fit a straight line model. Use `plot.lm` to obtain diagnostic plots. What do you notice?

2. The data set `LakeHuron` (*datasets* package) has mean July average water surface elevations (ft) for Lake Huron, for 1875-1972. The following reates a data frame that has the same information:

```
Year=as(time(LakeHuron), "vector")
huron <- data.frame(year=Year, mean.height=LakeHuron)
```

(a) Plot `mean.height` against year.

(b) To see how each year's mean level is related to the previous year's mean level, use

```
lag.plot(huron$mean.height)
```

This plots the level in each year against the level in the previous year.

(c) *Use the function `acf()` to plot the autocorrelation function. Compare with the result from the `pacf()` (partial autocorrelation). What do the graphs suggest?

For an explanation of the autocorrelation function, look up "Autocorrelation" on Wikepedia.