J H Maindonald

Learning and Exploring R

Copyright © 2018 J H Maindonald
This text is dual licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License (CC-BY-SA) and the GNU Free Documentation License (GFDL).

March 2018

S has forever altered the way people analyze, visualize, 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.

From the citation for the 1998 Association for Computing Machinery Software award.

A big computer, a complex algorithm and a long time does not equal science.

Robert Gentleman

Contents

1 *Getting started with R* 11

2 *The R Working Environment* 27

3 *Examples — Data analysis with R* 41

4 *Data Objects and Functions* 53

5 *Data Input and Storage* 77

6 *Data Manipulation and Management* 87

7 *Graphics – base, lattice, ggplot2, rgl, googleVis...* 101

8 *Regression with Linear Terms and Factors* 133

9 *A Miscellany of Models & Methods* 161

10 *Map Overlays and Spatial Modeling* 183

11 *Brief Notes on Text Mining* 193

12 **Leveraging R Language Abilities* 199

A *R System Configuration 207

B The R Commander Graphical User Interface 213

C Color Versions of Selected Graphs 215

Introduction

The scope of these notes

These notes were developed over the course of more than a decade, for use in R courses that were presented to groups within universities, within CSIRO, and within Government departments. Each new course offered the chance to extend and refine the content, and to add content that was tuned to the requirements of the new audience. The result is a somewhat eclectic mix of material. The notes are provided here with the intention that others will be free to add to them, or to develop them for their own purposes.

Commentary on R

General

R has extensive graphical abilities that are tightly linked with its analytic abilities. A new release of base R, on which everything else is built appears every few months.

The major part of R's abilities for statistical analysis and for specialist graphics comes from the extensive enhancements that the packages build on top of the base system. Its abilities are further extended by an extensive range of interfaces into other systems[1]

The main part of the R system – base R plus the recommended packages – is under continuing development.

R is free to download from a CRAN site (see above). It runs on all common types of system – Windows, Mac, Unix and Linux.

[1] These include Python, SQL and other databases, parallel computing using MPI, and Excel.

The R user base

Statistical and allied professionals have found R especially attractive, both for the access that it gives to cutting edge tools, and as a platform for developing new tools. Additionally, the R system has found wide use among working scientists whose data analysis requirements justify the time needed to gain the necessary R skills. It is finding use, also, as an environment in which to embed applications whose primary focus is not data analysis or graphics.

The R Task Views web page (`http://cran.csiro.au/web/views/`) notes, for application areas where R is widely used, relevant packages.

Getting help

Note the web sites:

Wikipedia:
http://en.wikipedia.org/wiki/R_(programming_language)

R-downunder (low traffic, friendly):
http://www.stat.auckland.ac.nz/mailman/listinfo/r-downunder

Stackoverflow
http://stackoverflow.com/questions/tagged/r.

The r-help mailing list serves, especially for users with a technical bent, as an informal support network. The R community expects users to want more than a quick cook-book fix, and to show a willingness to work at improving statistical knowledge.

Novices will find the low traffic R-downunder list more friendly and helpful than the main R mailing list. Its subscribers include some highly expert individuals.

Details of this and other lists can be found at: http://www.r-project.org. Be sure to check the available documentation before posting to r-help. List archives can be searched for previous questions and answers.

Important R web links

Note the following web sites:

CRAN (Comprehensive R Archive Network):
http://cran.r-project.org

Obtain R and R packages from a CRAN mirror in the local region. An Australian mirror (one of two) is: https://cran.csiro.au/

For New Zealand, use http://cran.stat.auckland.ac.nz
R homepage: https://www.r-project.org/

For various useful links click, from an R session that uses the GUI, on the menu item R help. Then, on the browser window that pops up, look under Resources

CRAN is the primary R 'repository'. Among package repositories that supplement CRAN, note in particular the Bioconductor repository (http://www.bioconductor.org), which caters for high throughput genomic data.

The origins and future of R

The R system implements a dialect of the S language that was developed at AT&T Bell Laboratories as a general purpose scientific language, with especial strengths in data manipulation, graphical presentation and statistical analysis. The commercial S-PLUS implementation popularized the S language, giving a user base into which R could tap.

Ross Ihaka and Robert Gentleman, both at that time from the University of Auckland, developed the initial version of R, for use in teaching. Since mid-1997, development has been overseen by a 'core team' of about a dozen people, drawn from different institutions worldwide.

With the release of version 1.0 in early 2000, R became a serious tool for professional use. Since 2004, the number of packages has increased at a rate of slightly more than 25% per annum.

Open source systems that might have been the basis for an R-like project include Scilab, Octave, Gauss, Lisp-Stat, Python, and now Julia. None of these match the range and depth of R's packages. Julia, which strongly outperforms R in execution time comparisons that appear on the Julia website http://julialang.org, has not had time to establish a clear place for itself.

More than 12,000 packages are, as of January 2018, available through the CRAN sites.

R code looks at first glance like C code. The R interpreter is modeled on the Scheme LISP dialect.

The R system uses a language model that dates from the 1980s. Any change to a more modern language model is likely to be evolutionary. Details of the underlying computer implementation are in a process of limited continual change.

Interactive development environments – editors and more

RStudio (`http://rstudio.org/`) is a very attractive run-time environment for R, available for Windows, Mac and Linux/Unix systems. This has extensive abilities for managing projects, and for working with code. It is a highly recommended alternative to the GUIs that come with the Windows and Mac OS X binaries that are available from CRAN sites.

Note also Emacs, with the ESS (Emacs Speaks Statistics) addon. is This is a feature-rich environment that can be daunting for novices. It runs on Windows as well as Linux/Unix and Mac. Note also, for Windows, the Tinn-R editor (`http://www.sciviews.org/Tinn-R/`).

Pervasive unifying ideas

Ideas that pervade R include:

- Generic functions for common tasks – print, summary, plot, etc. (the Object-oriented idea; do what that "class" of object requires)
- Formulae, for specifying graphs, models and tables.
- Language structures can be manipulated, just like any data object (Manipulate formulae, expressions, function argument lists, . . .)
- Lattice (trellis) and ggplot graphics offer innovative features that are widely used in R packages. They aid the provision of graphs that reflect important aspects of data structure.

Note however that these are not uniformly implemented through R. This reflects the incremental manner in which R has developed.

Expressions can be:

- evaluated (of course)
- printed on a graph (come to think of it, why not?)

There are many unifying computational features. Thus any 'linear' model (lm, lme, etc) can use spline basis functions to fit spline terms.

Data set size

R's evolving technical design has allowed it, taking advantage of advances in computing hardware, to steadily improve its handling of large data sets. The flexibility of R's memory model does however have a cost[2] for some large computations, relative to systems that process data from file to file.

An important step was the move, with the release of version 1.2, to a dynamic memory model.

[2] The difference in cost may be small or non-existent for systems that have a 64-bit address space.

Good planning, informed analysis and reliable software

While the R system is unique in the extent of close scrutiny that it receives from highly expert users, the same warnings apply as to any statistical system. The base system and the recommended packages get unusually careful scrutiny.

The scientific context, has crucial implications for the experiments that it is useful to do, and for the analyses that are meaningful. Available statistical methodology, and statistical and computing software and hardware, bring their own constraints and opportunities.

Take particular care with newer or little-used abilities in contributed packages. These may not have been much tested, unless by their developers.

Statistics of data collection encompasses statistical *experimental*

Always, one has to ask whether data are available, or can be collected, that allow the required inferences.

design, sampling design, and data collection more generally. Subject area insights can be crucial.

Once the data have been collected, the challenges are then those of data analysis and of interpretation and presentation of results. Effective data analysis must take account of the limitations inherent in the data, an understanding of the statistical issues, and risks that arise from inadequate understanding of the statistical issues. For this, software that is of high quality must be complemented with the critical resources of well-trained and well-informed minds.

Documentation and Learning Aids

R podcasts: See for example `http://www.r-podcast.org/`

Official Documentation: Users who are working through these notes on their own should have available for reference the document "An Introduction to R", written by the R Development Core Team. To download an up-to-date copy, go to CRAN.

Web-based Documentation: Go to `http://www.r-project.org` and look under Documentation. There are further useful links under Other.

Also `http://wiki.r-project.org/rwiki/doku.php`

The R Journal (formerly R News): Successive issues are a mine of useful information. These can be copied down from a CRAN site.

Books: See `http://www.R-project.org/doc/bib/R.bib` for a list of R-related books that is updated regularly. Here, note especially:

Maindonald, J. H. & Braun, J. H. 2010. Data Analysis & Graphics Using R. An Example-Based Approach. 3rd edn, Cambridge University Press, Cambridge, UK, 2010.
`http://www.maths.anu.edu.au/~johnm/r-book.html`

Notes for Readers of this Text

Asterisked Sections or Subsections

Asterisks are used to identify material that is more technical or specialized, and that might be omitted at a first reading.

The DAAGviz *package*

This package, available from Github, is an optional companion to these notes.

Once attached, this package gives access to:

The *DAAGviz* package collects scripts and datasets together in a way that may be useful to readers of these notes.

Assuming that the *DAAGviz* package has been installed, it can be attached thus:

```
library(DAAGviz)
```

- Scripts that include all the code. To access these scripts do, e.g.

```
## Check available scripts
dir(system.file('scripts', package='DAAGviz'))
## Show chapter 5 script
script5 <- system.file('scripts/5examples-code.R',
                       package='DAAGviz')
file.show(script5)
```

- Source files (also scripts) for functions that can be used to reproduce the graphs. These are available for Chapters 5 to 15 only. To load the Chapter 5 functions into the workspace, use the command:

```
path2figs5 <- system.file('doc/figs5.R',
                          package='DAAGviz')
source(path2figs5)
```

More succinctly, use the function `getScript()`:

```
## Place Ch 5 script in
## working directory
getScript(5)
```

- The datasets `bronchit`, `eyeAmp`, and `Crimean`, which feature later in these notes.

More succinctly, use the function `sourceFigFuns()`:

```
## Load Ch 5 functions
## into workspace
sourceFigFuns(5)
```

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`:

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.

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.

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

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

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.

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!

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.

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

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!

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.

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.

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

```
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.

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.

Examples of use of ??:

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

1.2.4 The working directory

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

- 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**.

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.

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:

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

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

Additional packages can be installed thus:

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

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

1.4 Practice with R commands

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.

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.

Try the following

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.

```
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:

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

Images and perspective plots:

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

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

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

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:

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.

```
args(round)
```

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

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()`:

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

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

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)
```

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

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

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

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")`.

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.

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.

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`

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

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

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.

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.

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

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

```
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.

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

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

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

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:

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

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

NB also the function `round()`.

```
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
```

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 screenshots here are for version 0.98.501 of 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.

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.

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

- Abilities that help in the creation of packages.

2.4.1 *The RStudio file menu*

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.

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.

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`.

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

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:

$$\widehat{\text{Time}} = e^{0.7316} \times \text{Distance}^{1.1248} = 2.08 \times \text{Distance}^{1.1248}$$

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`.

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)`.

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

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 R^2:  0.999,    Adjusted R^2:  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"))
```

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

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

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.

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

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

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

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

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

```
                          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

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

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

   (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.

4

Data Objects and Functions

Different types of data objects:	
Vectors	These collect together elements of the same mode. (Possible modes are "logical", "integer", "numeric", "complex", "character" and "raw")
Factors	Factors identify category levels in categorical data. Modeling functions know how to represent factors. (Factors do not quite manage to be vectors! Why?)
Data frame	A list of columns – same length; modes may differ. Data frames are a device for organizing data.
Lists	Lists group together an arbitrary set of objects (Lists are recursive; elements of lists are lists.)
`NA`s	Use `is.na()` to check for `NA`s.

Data objects and functions are two of several types of objects (others include model objects, formulae, and expressions) that are available in R. Users can create and work with such objects in a user workspace. All can, if the occasion demands, be treated as data!

We start this chapter by noting data objects that may appear as columns of a data frame.

4.1 Column Data Objects – Vectors and Factors

Column objects is a convenient name for one-dimensional data structures that can be included as columns in a data frame. This includes vectors[1], factors, and dates.

[1] Strictly, the vectors that we discuss here are *atomic* vectors. Their elements are not, as happens with lists, wrappers for other language objects.

4.1.1 Vectors

Examples of vectors are

```
c(2,3,5,2,7,1)
3:10 # The numbers 3, 4,.., 10
c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
c("fig","mango","apple","prune")
```

Common vector modes are logical, numeric and character. The 4 lines of code create vectors that are, in order: numeric, numeric, logical, character.

Use `mode()` to show the storage mode of an object, thus:

```
x <- c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
mode(x)
```

```
[1] "logical"
```

The missing value symbol is NA. Subsection 4.1.3 will discuss issues that arise when one or more vector elements is an NA.

Subsets of Vectors

There are four common ways to extract subsets of vectors.

1. Specify the subscripts of elements that are to be extracted:

```
x <- c(3,11,8,15,12)    # Assign to x the values
x[c(2,4)]               # Extract elements 2 and 4
```

```
[1] 11 15
```

Negative numbers may be used to omit elements:[2]

[2] Mixing of positive and negative subscripts is not allowed.

```
x <- c(3,11,8,15,12)
x[-c(2,3)]
```

```
[1]  3 15 12
```

2. Specify a vector of logical values. The elements that are extracted are those for which the logical value is TRUE. Thus suppose we want to extract values of x that are greater than 10.

Arithmetic relations that may be used for extraction of subsets are >=, ==, != and %in%. The first four compare magnitudes, == tests for equality, != tests for inequality, and %in% tests whether any element matches.

```
x>10    # Values are logical (TRUE or FALSE)
```

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

```
x[x > 10]
```

```
[1] 11 15 12
```

```
"John" %in% c("Jeff", "Alan", "John")
```

```
[1] TRUE
```

3. Where elements have names, these can be used to extract elements:

```
altitude <- c(Cambarville=800, Bellbird=300,
              "Allyn River"=300,
              "Whian Whian"=400,
              Byrangery=200, Conondale=400,
              Bulburin=600)
##
## Names can be used to extract elements
altitude[c("Cambarville", "Bellbird")]
```

```
Cambarville    Bellbird
        800         300
```

4. Use subset(), with the vector as the first argument, and a logical statement that identifies the elements to be extracted as the second argument. For example:

```
subset(altitude, altitude>400)
```

```
Cambarville     Bulburin
        800          600
```

4.1.2 Factors

Factors are column objects whose elements are integer values 1, 2, . . . , k, where k is the number of levels. They are distinguished from integer vectors by having the class `factor` and a `levels` attribute.

Factors are an economical way to store vectors of repetitive text strings. By default, when a vector of text strings becomes a column in a data frame, it is incorporated as a factor.

For example, create a character vector `fruit`, thus:

```
fruit <- c("fig","mango","apple","plum", "fig")
```

This might equally well be stored as a factor, thus:

```
fruitfac <- factor(fruit)
```

Internally, the factor is stored as the integer vector 2, 3, 1, 4, 2. These numbers are interpreted according to the attributes table:

Thus 1 is interpreted as `"apple"`; 2:`"fig"`; 3:`"mango"`; 4:`"plum"`.

1	2	3	4
"apple"	"fig"	"mango"	"plum"

By default, the levels are taken in alphanumeric order.

The function `factor()`, with the `levels` argument specified, can be used both to specify the order of levels when the factor is created, or to make a later change to the order.[3] For example, the following orders levels according to stated glycemic index:

[3] Where counts are tabulated by factor level, or *lattice* or other graphs have one panel per factor level, these are in order of the levels.

```
glycInd <- c(apple=40, fig=35, mango=55, plum=25)
## Take levels in order of stated glycInd index
fruitfac <- factor(fruit,
                   levels=names(sort(glycInd)))
levels(fruitfac)
```

```
[1] "plum"  "fig"   "apple" "mango"
```

```
unclass(fruitfac)  # Examine stored values
```

```
[1] 2 4 3 1 2
attr(,"levels")
[1] "plum"  "fig"   "apple" "mango"
```

Incorrect spelling of the level names generates missing values, for the level that was mis-spelled. Use the `labels` argument if you wish to change the level names, but be careful to ensure that the label names are in the correct order.

Mis-spelt name, example:

```
trt <- c("A","A","Control")
trtfac <- factor(trt,
  levels=c("control","A"))
table(trtfac)
```

```
trtfac
control       A
      0       2
```

In most places where the context seems to demand it, the integer levels are translated into text strings, thus:

```
fruit <- c("fig","mango","apple", "plum","fig")
fruitfac <- factor(fruit)
fruitfac == "fig"
```

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

Section 8.5 has detailed examples of the use of factors in model formulae.

Ordered factors

In addition to factors, note the existence of ordered factors, created using the function `ordered()`. For ordered factors, the order of levels implies a relational ordering. For example:

```
windowTint <- ordered(rep(c("lo","med","hi"), 2),
                      levels=c("lo","med","hi"))
windowTint
```

```
[1] lo  med hi  lo  med hi
Levels: lo < med < hi
```

```
sum(windowTint > "lo")
```

```
[1] 4
```

Subsetting of factors

Consider the factor `fruitfac` that was created earlier:

```
fruitfac <- factor(c("fig","mango","apple","plum", "fig"))
```

We can remove elements with levels `fig` and `plum` thus:

```
ff2 <- fruitfac[!fruitfac %in% c("fig","plum")]
ff2
```

```
[1] mango apple
Levels: apple fig mango plum
```

```
table(ff2)
```

```
ff2
apple   fig mango  plum
    1     0     1     0
```

The levels `fig` and `plum` remain, but with the table showing 0 values for these levels. Use the function `droplevels()` to remove levels that are not present in the data:

```
droplevels(ff2)
```

```
[1] mango apple
Levels: apple mango
```

Note also:

```
table(droplevels(ff2))
```

```
apple mango
    1     1
```

Why is a factor not a vector?

Two factors that have different levels vectors are different types of object. Thus, formal concatenation of factors with different levels vectors is handled by first coercing both factors to integer vectors. The integer vector that results is not, in most circumstances, meaningful or useful.

Vectors can be concatenated (joined). Two or mare factors can be sensibly concatenated only if they have identical levels vectors.

4.1.3 Missing Values, Infinite Values and NaNs

Any arithmetic or logical operation with NA generates an NA. The consequences are more far-reaching than might be immediately obvious. Use `is.na()` to test for a missing value:

Failure to understand the rules for calculations with NAs can lead to unwelcome surprises.

```
is.na(c(1, NA, 3, 0, NA))
```

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

An expression such as `c(1, NA, 3, 0, NA) == NA` returns a vector of NAs, and cannot be used to test for missing values.

```
c(1, NA, 3, 0, NA) == NA
```

```
[1] NA NA NA NA NA
```

As the value is unknown, it might or might not be equal to 1, or to another NA, or to 3, or to 0.

Note that different functions handle NAs in different ways. Functions such as `mean()` and `median()` accept the argument `na.rm=TRUE`, which causes observations that have NAs to be ignored. The `plot()` function omits NAs, infinities and NaNs. For use of `lowess()` to put a smooth curve through the plot, NAs must first be removed. By default, `table()` ignores NAs.

The modeling function `lm()` accepts any of the arguments `na.action=na.omit` (omit), `na.action=na.exclude` (omit NAs when fitting; replace by NAs when fitted values and residuals are calculated), and `na.action=na.fail`.

Problems with missing values are a common reason why calculations fail. Infinite values and NaNs are a further potential source of difficulty.

Inf and NaN

The expression `1/0` returns `Inf`. The expression `log(0)` returns `-Inf`, i.e., smaller than any real number. The expressions `0/0` and `log(-1)` both return `NaN`.

Note that `sqrt(-1+0i)` returns `0+1i`. R distinguishes between the real number `-1` and the complex number `-1+0i`.

NAs in subscripts?

It is best to ensure that NAs do not appear, when there is an assignment, in subscript expressions on either side of the expression.

4.2 Data Frames, Matrices, Arrays and Lists

Data frames: Data frames are lists of column objects. The requirement that all of the column objects have the same length gives data frames a row by column rectangular structure. Different columns can have different column classes — commonly numeric or character or factor or logical or date.

Data frames with all columns numeric can sometimes be handled in the same way as matrices. In other cases, a different syntax may be needed, or conversion from one to the other. Proceed with care!

Matrices – vectors with a Dimension: When printed, matrices appear in a row by column layout in which all elements have the same mode – commonly numeric or character or logical.

Internally, matrices are stored as one long vector in which the columns are stacked one above the other. The first element in the dimension attribute gives the number of rows in each column.

Arrays and tables: Matrices are two-dimensional arrays. Arrays more generally can have an arbitrary number of dimensions. Tables have a structure that is identical to that of arrays.

The data frame `travelbooks` will feature in the subsequent discussion. Look back to Section 1.7 to see how it can be entered.

4.2.1 Data frames versus matrices and tables

Modeling functions commonly return larger numeric objects as matrices rather than data frames. The principal components function `prcomp()` returns scores as a matrix, as does the linear discriminant analysis function `lda()` from the *MASS* package.

Computations that can be performed with matrices are typically much faster than their equivalents with data frames. See Section 6.4.

Functions are available to convert data frames into matrices, and vice versa. For example:

```
travelmat <- as.matrix(travelbooks[, 1:4])
  # From data frame to matrix
newtravelbooks <- as.data.frame(travelmat)
  # From matrix to data frame
```

In comparing data frames with matrices, note that:

- Both for data frames and for matrices or two-way tables, the function `dim()` returns number of rows by number of columns, thus:

  ```
  travelmat <- as.matrix(travelbooks[, 1:4])
  dim(travelmat)
  ```

  ```
  [1] 6 4
  ```

  Alternatively, do:

  ```
  attr(travelmat, "dim")
  ```

  ```
  [1] 6 4
  ```

- For a matrix, `length()` returns the number of elements. For a data frame it returns the number of columns.

  A data frame is a list of columns. The function `length()` returns the list length.

  ```
  c(dframelgth=length(travelbooks),
    matlgth=length(travelmat))
  ```

  ```
  dframelgth     matlgth
           6          24
  ```

- The notation that uses single square left and right brackets to extract subsets of data frames, introduced in Section 1.6 works in just the same way with matrices. For example

  ```
  travelmat[, 4]
  travelmat[, "weight"]
  travelmat[, 1:3]
  travelmat[2,]
  ```

  Negative indices can be used to omit rows and/or columns.

- Use of the subscript notation to extract a row from a data frame returns a data frame, whereas extraction of a column yields a column vector. Thus:

  – Extraction of a row from a data frame, for example `travelbooks["Canberra - The Guide", ]` or `travelbooks[6, ]`, yields a data frame, i.e., a special form of list.

  Use `unlist(travelbooks[6, ])` to turn row from the data frame into a vector. All elements are coerced to a common mode, in this case numeric. Thus the final element becomes 1.0 (the code that is stored), rather than `Guide` which was the first level of the factor `type`.

  - `travelbooks$volume` (equivalent to `travelbooks[,1]` or `travelbooks[,"volume"]`)) is a vector.

- For either a data frame or a matrix, the function `rownames()` can be used to extract row names, and the function `colnames()` to extract column names. For data frames, `row.names()` is an alternative to `rownames()`, while `names()` is an alternative to `colnames()`.

Note also a difference in the mechanisms for adding columns. The following adds new columns `area` (area of page), and `density` (`weight` to `volume` ratio) to the data frame `travelbooks`:

```
travelbooks$area <- with(travelbooks, width*height)
travelbooks$density <- with(travelbooks,
                            weight/volume)
names(travelbooks)   # Check column names
```

```
[1] "thickness" "width"     "height"    "weight"
[5] "volume"    "type"      "area"      "density"
```

Columns are added to the data frame as necessary.

For matrices, use `cbind()`, which can also be used for data frames, to bind in new columns.

4.2.2 *Inclusion of character vectors in data frames*

When data frames are created, whether by use of `read.table()` or another such function to input data from a file, or by use of the function `data.frame()` to join columns of data together into a data frame, character vectors are converted into factors. Thus, the final column (`type`) of `travelbooks` became, by default, a factor.[4] To prevent such type conversions, specify `stringsAsFactors=FALSE` in the call to `read.table()` or `data.frame()`.

[4] This assumes that the global option `stringsAsFactors` is `FALSE`. To check, interrogate `options()$stringsAsFactors`.

4.2.3 *Factor columns in data frame subsets*

The data frame `ais` (*DAAG*) has physical charateristics of athletes, divided up thus between ten different sports:

```
with(ais, table(sport))
```

```
sport
 B_Ball   Field     Gym Netball     Row    Swim  T_400m T_Sprnt  Tennis
     25      19       4      23      37      22      29      15      11
 W_Polo
     17
```

Figure 7.2.1 in Subsection 7.9 limits the data to swimmers and rowers. For this, at the same time removing all levels except `Row` and `Swim` from the factor `sport`, do:

```
rowswim <- with(ais, sport %in% c("Row", "Swim"))
aisRS <- droplevels(subset(ais, rowswim))
xtabs(~sport, data=aisRS)
```

If redundant levels were left in place, the graph would show empty panels for each such level.

```
sport
 Row Swim
  37   22
```

Contrast the above with:

```
xtabs(~sport, data=subset(ais, rowswim))
```

```
sport
 B_Ball   Field     Gym Netball     Row    Swim  T_400m T_Sprnt  Tennis
      0       0       0       0      37      22       0       0       0
 W_Polo
      0
```

4.2.4 *Handlng rows that include missing values*

The function `na.omit()` omits rows that contain one or more missing values. The argument may be a data frame or a matrix. The function `complete.cases()` identifies such rows. Thus:

```
test.df <- data.frame(x=c(1:2,NA), y=1:3)
test.df
```

```
   x y
1  1 1
2  2 2
3 NA 3
```

```
## complete.cases()
complete.cases(test.df)
```

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

```
## na.omit()
na.omit(test.df)
```

```
  x y
1 1 1
2 2 2
```

4.2.5 *Arrays — some further details*

A matrix is a two-dimensional array. More generally, arrays can have an arbitrary number of dimensions.

Tables, which will be the subject of the next subsection, have a very similar structure to arrays.

Removal of the dimension attribute

The dimension attribute of a matrix or array can be changed or removed, thus:

```
travelvec <-  as.matrix(travelbooks[, 1:4])
dim(travelvec) <- NULL  # Columns of travelmat are stacked into one
                        # long vector
travelvec
```

```
 [1]     1.3     3.9     1.2     2.0     0.6     1.5    11.3    13.1    20.0    21.1
[11]    25.8    13.1    23.9    18.7    27.6    28.5    36.0    23.4   250.0   840.0
[21]   550.0  1360.0   640.0   420.0
```

```
  # as(travelmat, "vector") is however preferable
```

Note again that the $ notation, used with data frames and other list objects to reference the contents of list elements, is not relevant to matrices.

4.2.6 *Lists*

A list is a collection of arbitrary objects. As noted above, a data frame is a specialized form of list. Consider for example the list

Elements of lists are themselves lists. Distinguish `rcanberra[4]`, which is a sub-list and therefore a list, from `rcanberra[[4]]` which extracts the contents of the fourth list element.

```
rCBR <- list(society="ssai", branch="Canberra",
             presenter="John",
             tutors=c("Emma", "Chris", "Frank"))
```

First, extract list length and list names:

```
length(rCBR)       # rCBR has 4 elements
names(rCBR)
```

```
[1] 4
```

```
[1] "society"   "branch"    "presenter" "tutors"
```

The following extracts the 4th list element:

```
rCBR[4]            # Also a list,  name is 'tutors'
```

```
$tutors
[1] "Emma"  "Chris" "Frank"
```

Alternative ways to extract the contents of the 4th element are:

List elements can be accessed by name. Thus, to extract the contents of the 4th list element, alternatives to `rcanberra[[4]]` are `rcanberra[["tutors"]]` or `rcanberra$tutors`.

```
rCBR[[4]]          # Contents of 4th list element
```

```
[1] "Emma"  "Chris" "Frank"
```

```
rCBR$tutors        # Equivalent to rCBR[["tutors"]]
```

```
[1] "Emma"  "Chris" "Frank"
```

Model objects are lists

As noted in Subsection 3.2.2, the various R modeling functions all return their own particular type of model object, either a list or as an S4 object.

Recall again, also, that data frames are a specialized form of list, with the restriction that all columns must all have the same length.

4.3 Functions

Different Kinds of Functions:	
Generic	The 'class' of the function argument determines the action taken. E.g., `print()`, `plot()`, `summary()`
Modeling	For example, `lm()` fits *linear* models. Output may be stored in a model object.
Extractor	These extract information from model objects. Examples include `summary()`), `coef()`), `resid()`), and `fitted()`
User	Use, e.g., to automate and document computations
Anonymous	These are user functions that are defined at the point of use, and do not need a name.

The above list is intended to include the some of the most important types of function. These categories may overlap.

The language that R implements has many of the features of a functional language. Functions have accordingly featured throughout the earlier discussion. Here will be noted functions that are commonly important.

Functions for working with dates are discussed in Section 4.3.9 immediately following.

4.3.1 Built-In Functions

Common useful functions

```
## Use with any R object as argument
print()            # Prints a single R object
length()           # Number of elements in a vector or of a list

## Concatenate and print R objects [does less coercion than print()]
cat()              # Prints multiple objects, one after the other

## Use with a numeric vector argument
mean()             # If argument has NA elements, may want na.rm=TRUE
median()           # As for mean(), may want na.rm=TRUE
range()            # As for mean(), may want na.rm=TRUE
unique()           # Gives the vector of distinct values
diff()             # Vector of first differences
                   # N. B. diff(x) has one less element than x
cumsum()           # Cumulative sums, c.f., also, cumprod()

## Use with an atomic vector object
sort()             # Sort elements into order, but omitting NAs
order()            # x[order(x)] orders elements of x, with NAs last
rev()              # reverse the order of vector elements
any()              # Returns TRUE if there are any missing values
as()               # Coerce argument 1 to class given by argument 2
                   # e.g. as(1:6, "factor")
is()               # Is argument 1 of class given by argument 2?
                   # is(1:6, "factor") returns FALSE
```

```
                    # is(TRUE, "logical") returns TRUE
is.na()             # Returns TRUE if the argument is an NA

## Information on an R object
str()               # Information on an R object
args()              # Information on arguments to a function
mode()              # Gives the storage mode of an R object
                    # (logical, numeric, character, . . ., list)

## Create a vector
numeric()           # numeric(5) creates a numeric vector, length 5,
                    # all elements 0.
                    # numeric(0) (length 0) is sometimes useful.
character()         # Create character vector; c.f. also logical()
```

The function `mean()`, and a number of other functions, takes the argument `na.rm=TRUE`; i.e., remove `NA`s, then proceed with the calculation. For example

```
mean(c(1, NA, 3, 0, NA), na.rm=T)
```

```
[1] 1.333
```

Note that the function `as()` has, at present, no method for coercing a matrix to a data frame. For this, use `as.data.frame()`.

Functions in different packages with the same name

For example, as well as *lattice* function `dotplot()` the graphics package has a defunct function `dotplot()`. To be sure of getting the *lattice* function *dotplot()*, refer to it as `lattice::dotplot`.

4.3.2 Functions for data summary and/or manipulation

4.3.3 Functions for creating and working with tables

4.3.4 Tables of Counts

For data manipulation, note:

- the apply family of functions (Subsection 4.3.7).
- data manipulation functions in the *reshape2* and *plyr* packages (Chapter 6).

Use either `table()` or `xtabs()` to make a table of counts. Use `xtabs()` for cross-tabulation, i.e., to determine totals of numeric values for each table category.

The `table()` function

For use of `table()`, specify one vector of values (often a factor) for each table margin that is required. For example:

```
library(DAAG)         # possum is from DAAG
with(possum, table(Pop, sex))
```

```
       sex
Pop      f  m
  Vic   24 22
  other 19 39
```

NAs in tables

By default, `table()` ignores NAs. To show information on NAs, specify `exclude=NULL`, thus:

```
library(DAAG)
table(nswdemo$re74==0, exclude=NULL)
```

```
FALSE  TRUE  <NA>
  119   326   277
```

The `xtabs()` function

This more flexible alternative to `table()` uses a table formula to specify the margins of the table:

```
xtabs(~ Pop+sex, data=possum)
```

```
       sex
Pop      f  m
  Vic   24 22
  other 19 39
```

A column of frequencies can be specified on the left hand side of the table formula. In order to demonstrate this, the three-way table `UCBAdmissions` (*datasets* package) will be converted into its data frame equivalent. Margins in the table become columns in the data frame:

Manipulations with data frames are in general conceptually simpler than manipulations with tables. For tables that are not unreasonably large, it is in general a good strategy to first convert the table to a data frame and make that the starting point for further calculations.

```
UCBdf <- as.data.frame.table(UCBAdmissions)
head(UCBdf, n=3)
```

```
     Admit Gender Dept Freq
1 Admitted   Male    A  512
2 Rejected   Male    A  313
3 Admitted Female    A   89
```

The following then forms a table of total admissions and rejections in each department:

```
xtabs(Freq ~  Admit+Dept, data=UCBdf)
```

```
          Dept
Admit        A   B   C   D   E   F
  Admitted 601 370 322 269 147  46
  Rejected 332 215 596 523 437 668
```

Information on data objects

The function `str()` gives basic information on the data object that is given as argument.

```
library(DAAG)
str(possumsites)
```

```
'data.frame':   7 obs. of  3 variables:
 $ Longitude: num  146 149 151 153 153 ...
 $ Latitude : num  -37.5 -37.6 -32.1 -28.6 -28.6 ...
 $ altitude : num  800 300 300 400 200 400 600
```

4.3.5 *Utility functions*

```
dir()              # List files in the working or other specified directory
sessionInfo()      # Print version numbers for R and for attached packages
system.file()      # By default, show path to 'package="base"'
R.home()           # Path to R home directory
.Library           # Path to the default library
.libPaths()        # Get/set paths to library directories
```

Section A has further details.

4.3.6 *User-defined functions*

The function `mean()` calculates means, The function `sd()` calculates standard deviations. Here is a function that calculates mean and standard deviation at the same time:

Note also that functions can be defined at the point of use. Such functions do not need a name, and are called anonymous functions. Section 4.3.4 has an example.

```
mean.and.sd <- function(x){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, sd = sdev)    # return value
}
```

The parameter `x` is the argument that the user must supply. The body of the function is enclosed between curly braces. The value that the function returns is given on its final line. Here the return value is a vector that has two named elements.

The following calculates the mean and standard deviation of heterozygosity estimates for seven different *Drosophila* species.[5]

[5] Data are from Lewontin, R. 1974. *The Genetic Basis of Evolutionary Change*.

```
hetero <- c(.43,.25,.53,.47,.81,.42,.61)
mean.and.sd(hetero)
```

```
  mean     sd
0.5029 0.1750
```

It is useful to give the function argument a default value, so that it can be run without user-supplied parameters, in order to see what it does. A possible choice is a set of random normal numbers, perhaps generated using the `rnorm()` function. Here is a revised function definition. Because the function body has been reduced to a single line, the curly braces are not needed.

Note that a different set of random numbers will be returned, giving a different mean and SD, each time that the function is run with its default argument.

```
mean.and.sd <- function(x = rnorm(20))
                  c(mean=mean(x), sd=sd(x))
mean.and.sd()
```

```
    mean       sd
-0.04961  1.01807
```

```
mean.and.sd()
```

```
  mean      sd
0.3284 1.0190
```

4.3.7 *The* `apply` *family of functions*

`apply()`, `sapply()` and friends

`apply()`	Use `apply()` to apply a function across rows or columns of a matrix (or data frame)
`sapply()` & friends	`sapply()` and `lapply()` apply functions in parallel across columns of a data frame, or across elements of a list, or across elements of a vector.

For the `apply` family of functions, specify as the `FUN` argument any function that will not generate an error. Obviously, `log("Hobart")` is not allowed!

Note also the function `tapply()`, which will not be discussed here.

`apply()`: The function `apply()` is intended for use with matrices or, more generally, with arrays. It has three mandatory arguments, a matrix or data frame, the dimension (1 for rows; 2 for columns) or dimensions, and a function that will be applied across that dimension of the matrix or data frame.

If used with a data frames, the data frame is first coerced to `matrix`.

Here is an example:

```
apply(molclock, 2, range)
```

Code that will input `molclock1`:

```
library(DAAG)
datafile("molclock1")
molclock <-
  read.table("molclock1.txt")
```

The following tabulates admissions, in the three-way table `UCBAdmissions`, according to `sex`:

```
apply(UCBAdmissions, c(1,2), sum)
```

```
          Gender
Admit      Male Female
  Admitted 1198    557
  Rejected 1493   1278
```

`sapply()` and `lapply()`: Use `sapply()` and `lapply()` to apply a function (e.g., `mean()`, `range()`, `median()`) in parallel to all columns of a data frame. They take as arguments the name of the data frame, and the function that is to be applied.

Warning: Use `apply()` with `COLUMN=2`, to apply a function to all columns of a matrix. If `sapply()` or `lapply()` is given a matrix as argument, the function is applied to each element (the matrix is treated as a vector).

The function `sapply()` returns the same information as `lapply()`. But whereas `lapply()` returns a list, `sapply()` tries if possible to simplify the result to give a vector or matrix or array.

Here is an example of the use of `sapply()`:

```
sapply(molclock, range)
```

```
     Gpdh  Sod  Xdh AvRate  Myr
[1,]  1.5 12.6 11.5   11.9   55
[2,] 40.0 46.0 31.7   24.9 1100
```

A third argument `na.rm=TRUE` can be supplied to the function `sapply`. This argument is then automatically passed to the function that is given in the second argument position.

Use of `na.rm=TRUE`:

```
sapply(molclock, range,
       na.rm=TRUE)
```

```
     Gpdh  Sod  Xdh AvRate  Myr
[1,]  1.5 12.6 11.5   11.9   55
[2,] 40.0 46.0 31.7   24.9 1100
```

More generally, the first argument to `sapply()` or `lapply()` can be any vector.

`sapply()` – Application of a user function

We will demonstrate the use of `sapply()` to apply a function that counts the number of `NA`s to each column of a data frame. A suitable function can be defined thus:

```
countNA <- function(x)sum(is.na(x))
```

An alternative is to define a function[6] in place, without a name, that counts number of `NA`s. The alternatives are:

[6] This is called an *anonymous* function.

Use function defined earlier:

```
library(MASS)
sapply(Pima.tr2[, 1:5], countNA)
```

```
npreg   glu    bp  skin   bmi
    0     0    13    98     3
```

Define function at place of call:

```
sapply(Pima.tr2[, 1:5],
       function(x)sum(is.na(x)))
```

```
npreg   glu    bp  skin   bmi
    0     0    13    98     3
```

4.3.8 Functions for working with text strings

The functions `paste()` and `paste0()` join text strings. The function `sprintf()`, primarily designed for formatting output for printing, usefully extends the abilities of `paste()` and `paste0()`.

For `paste()`, the default is to use a space as a separator; `paste0()` omits the space.

Other simple string operations include `substring()` and `nchar()` (number of characters). Both of these, and `strsplit()` noted in the next paragraph, can be applied to character vectors.

The function `strsplit()`, used to split strings, has an argument `fixed` that by default equals `FALSE`. The effect is that the argument `split`, which specifies the character(s) on which the string will split, is assumed to be a regular expression. See `help(regexp)` for details. For use of a `split` character argument, call `strsplit()` with `fixed=FALSE`.

Other functions that accept an argument `fixed` include the search functions `grep()` and `regexpr()`, and the search and replace functions `sub()` and `gsub()`.

Bird species in the dataset `cuckoos` (*DAAG*) are:

```
(spec <- levels(cuckoos$species))
```

```
[1] "hedge.sparrow" "meadow.pipit"  "pied.wagtail"
[4] "robin"         "tree.pipit"    "wren"
```

Now replace the periods in the names by spaces:

```
(specnam <- sub(".", " ", spec, fixed=TRUE))
```

```
[1] "hedge sparrow" "meadow pipit"  "pied wagtail"
[4] "robin"         "tree pipit"    "wren"
```

Regular expression substitution:

```
specnam <- sub("\\.",
               " ", spec)
```

In regular expressions enter a period (`"."`) as `"\\."`

For string matching, use `match()`, `pmatch()` and `charmatch()`. For matching with regular expressions, note `grep()` and `regexpr()`. For string substitution, use `sub()` and `gsub()`.

See `help(regex)` for information on the use of regular expressions.

Web pages with information on string manipulation in R include:

http://www.stat.berkeley.edu/classes/s133/R-6.html
http://en.wikibooks.org/wiki/R_Programming/Text_Processing

The first is an overview, with the second more detailed.

The package *stringr*, due to Hadley Wickham, provides what may be a more consistent set of functions for string handling than are available in base R.

For strings representing biological sequences, install the well-documented Bioconductor package *Biostrings*.

4.3.9 Functions for Working with Dates (and Times)

Use `as.Date()` to convert character strings into dates. The default format has year, then month, then day of month, thus:

Good starting points for learning about dates in R are the help pages `help(Dates)`, `help(as.Date)` and `help(format.Date)`.

```
# Electricity Billing Dates
dat <- c("2003-08-24","2003-11-23","2004-02-22",
         "2004-05-03")
dd <- as.Date(dat)
```

Use `format()` to set or change the way that a date is formatted. The following is a selection of the available symbols:

%d:	day, as number
%a:	abbreviated weekday name (%A: unabbreviated)
%m:	month (00-12)
%b:	month abbreviated name (%B: unabbreviated)
%y:	final two digits of year (%Y: all four digits)

The default format is `"%Y-%m-%d"`. The character `/` can be used in place of `-`. Other separators (e.g., a space) must be explicitly specified, using the `format` argument, as in the examples below.

Date objects can be subtracted:

Subtraction yields a time difference object. If necessary, use `unclass()` to convert this to a numeric vector.

```
as.Date("1960-12-1") - as.Date("1960-1-1")
```

```
Time difference of 335 days
```

There is a `diff()` method for date objects:

Use `unclass()` to turn a time difference object into an integer vector:

```
unclass(diff(dd))
```

```
dd <- as.Date(c("2003-08-24","2003-11-23",
                "2004-02-22", "2004-05-03"))
diff(dd)
```

```
Time differences in days
[1] 91 91 71
```

Formatting dates for printing: Use `format()` to fine tune the formatting of dates for printing.

See `help(format.Date)`.

```
dec1 <- as.Date("2004-12-1")
format(dec1, format="%b %d %Y")
```

```
[1] "Dec 01 2004"
```

```
format(dec1, format="%a %b %d %Y")
```

```
[1] "Wed Dec 01 2004"
```

Such formatting may be used to give meaningful labels on graphs. Figure 4.1 provides an example:

```
## Labeling of graph: data frame jobs (DAAG)
library(DAAG); library(lattice)
fromdate <- as.Date("1Jan1995", format="%d%b%Y")
startofmonth <- seq(from=fromdate, by="1 month",
                    length=24)
atdates <- seq(from=fromdate, by="6 month",
               length=4)
xyplot(BC ~ startofmonth, data=jobs,
       scale=list(x=list(at=atdates,
                         labels=format(atdates,
                                       "%b%y"))))
```

Figure 4.1: Canadian worker force numbers, with dates used to label the x-axis. See Figure 7.12 in Subsection 7.2.6 for data from all Canadian provinces.

Conversion of dates to and from integer number of days: By default, dates are stored in integer numbers of days. Use `julian()` to convert a date into its integer value, by default using January 1 1970 as origin. Use the argument `option` to specify some different origin:

```
dates <- as.Date(c("1908-09-17", "1912-07-12"))
julian(dates)
```

```
[1] -22386 -20992
attr(,"origin")
[1] "1970-01-01"
```

```
julian(dates, origin=as.Date("1908-01-01"))
```

```
[1]  260 1654
attr(,"origin")
[1] "1908-01-01"
```

Note also `weekdays()`, `months()`, and `quarters()`:

```
dates <- as.Date(c("1908-09-17", "1912-07-12"))
weekdays(dates)
```

```
[1] "Thursday" "Friday"
```

```
months(dates)
```

```
[1] "September" "July"
```

```
quarters(dates)
```

```
[1] "Q3" "Q3"
```

Regular sequences of dates: Use the function `help(seq.Date)`.

Given a vector of 'event' times, the following function can be used to count the number of events in each of a regular sequence of time intervals:

```
intervalCounts <- function(date, from=NULL, to=NULL, interval="1 month"){
  if(is.null(from))from <- min(date)
  if(is.null(to))to <- max(date)
  dateBreaks <- seq(from=from, to=to, by=interval)
  dateBreaks <- c(dateBreaks, max(dateBreaks)+diff(dateBreaks[1:2]))
  cutDates <- cut(date, dateBreaks, right=FALSE)
  countDF <- data.frame(Date=dateBreaks[-length(dateBreaks)],
                        num=as.vector(table(cutDates)))
  countDF
}
```

The following counts the number of events by year:

```
dates <- c("1908-09-17", "1912-07-12", "1913-08-06", "1913-09-09", "1913-10-17")
dates <- as.Date(dates)
(byYear <- intervalCounts(dates, from=as.Date("1908-01-01"), interval='1 year'))
```

```
        Date num
1 1908-01-01   1
2 1909-01-01   0
3 1910-01-01   0
4 1911-01-01   0
5 1912-01-01   1
6 1913-01-01   3
```

Further useful functions for working with dates: Note also `date()` which returns the current date and time, and `Sys.Date()` which returns the date. For information on functions for working with times, see `help(ISOdatetime)`.

The CRAN Task View for Time Series Analysis has notes on classes and methods for times and dates, and on packages that give useful functionality

4.3.10 Summaries of Information in Data Frames

A common demand is to obtain a tabular summary of information in each of several columns of a data frame, broken down according to the levels of one or more grouping variables. Consider the data frame `nswdemo` (*DAAG*). Treatment groups are control (`trt==0`) and treatment (`trt==1`) group, with variables `re74` (1974 income), `re75` (1975) and `re78` (1978),

The following calculates the number of zeros for each of the three variables, and for rach of the two treatment categories:

```
## Define a function that counts zeros
countzeros <- function(x)sum(!is.na(x) & x==0)
aggregate(nswdemo[, c("re74", "re75", "re78")],
          by=list(group=nswdemo$trt),
          FUN=countzeros)
```

The data frame is split according to the grouping elements specified in the `by` argument. The function is then applied to each of the columns in each of the splits.

```
  group re74 re75 re78
1     0  195  178  129
2     1  131  111   67
```

Now find the proportion, excluding `NA`s, that are zero. The result will be printed out with improved labeling of the rows:

```
## countprop() counts proportion of zero values
countprop <- function(x){
    sum(!is.na(x) & x==0)/length(na.omit(x))}
```

```
prop0 <-
  aggregate(nswdemo[, c("re74","re75","re78")],
            by=list(group=nswdemo$trt),
            FUN=countprop)
## Now improve the labeling
rownames(prop0) <- c("Control", "Treated")
round(prop0,2)
```

```
        group re74 re75 re78
Control     0 0.75 0.42 0.30
Treated     1 0.71 0.37 0.23
```

The calculation can alternatively be handled by two calls to the function `sapply()`, one nested within the other, thus:

```
prop0 <-
  sapply(split(nswdemo[, c("re74","re75","re78")],
               nswdemo$trt),
         FUN=function(z)sapply(z, countprop))
round(t(prop0), 2)
```

The argument `z` in the 'in place' function is a data frame. The argument `x` to `countprop()` is a column of a data frame.

```
  re74 re75 re78
0 0.75 0.42 0.30
1 0.71 0.37 0.23
```

4.4 *Classes and Methods (Generic Functions)

Key language constructs:	
Classes	Classes make generic functions (methods) possible.
Methods	Examples are `print()`, `plot()`, `summary()`, etc.

There are two implementation of classes and methods, the original S3 implementation, and the newer S4 implementation that is implemented in the *methods* package. Here, consider the simpler S3 implementation.

All objects have a class. Use the function `class()` to get this information.

For many common tasks there are generic functions – `print()`, `summary()`, `plot()`, etc., whose action varies according to the class of object to which they are applied.

Thus `print()` calls a method thus: factor: `print.factor()`; data frame: `print.data.frame()`; and so on. Ordered factors "inherit" the print method for factors. For objects without an explicit print method, `print.default()` is called.

To get details of the S3 methods that are available for a generic function such as `plot()`, type, e.g., `methods(plot)`. To get a list of the S3 methods that are available for objects of class `lm`, type, e.g., `methods(class="lm")`

4.4.1 *S4 methods

The S4 conventions and mechanisms extend the abilities available under S3, build in checks that are not available with S3, and are more conducive to good software engineering practice.

Packages that use S4 classes and methods include *lme4*, Bioconductor packages, and most of the spatial analysis packages.

Example – a spatial class

The *sp* package defines, among other possibilities, spatial data classes `SpatialPointsDataFrame` and `SpatialGridDataFrame`.

The *sp* function `bubble()`, for plotting spatial measurement data, accepts a spatial data object as argument.[7] The function `coordinates()` can be used, given spatial coordinates, to turn a data frame or matrix into an object of one of the requisite classes.

Data from the data frame `meuse`[8], from the *sp* package, will be used for an example. A first step is to create an object of one of the classes that the function `bubble()` accepts as argument, thus:

Classes defined in the *sp* package are widely used across R spatial data analysis packages.

[7] Each point (location) is shown as a bubble, with area proportional to a value for that point.

[8] Data are from the floodplain of the river Meuse, in the Netherlands. It includes concentrations of various metals (`cadmium`, `copper`, `lead`, `zinc`), with Netherlands topographical map coordinates.

```
library(sp)
data(meuse)
class(meuse)
```

```
[1] "data.frame"
```

```
coordinates(meuse) <- ~ x + y
class(meuse)
```

```
[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

This has created an object of the class `SpatialPointsDataFrame`.

Code that creates the plot, shown in Figure 4.2, is:

```
bubble(meuse, zcol="zinc", scales=list(tck=0.5),
       maxsize=2, xlab="Easting", ylab="Northing")
```

Figure 4.2: Bubble plot for `zinc` concentrations. Areas of bubbles are proportional to concentrations.

The function `bubble()` uses the abilities of the `lattice` package. It returns a `trellis` graphics object.

The coordinates can be extracted using `coordinates(meuse)`. Remaining columns from the original data frame are available from the data frame `meuse@data`.

Use `slotNames()` to examine the structure of the object:

```
slotNames(meuse)
```

```
[1] "data"        "coords.nrs"  "coords"
[4] "bbox"        "proj4string"
```

Typing `names(meuse)` returns the column names for the `data` slot. The effect is the same as that of typing `names(meuse@data)`. To get a list of the S4 methods that are available for a generic function, use `showMethods()`. Section 12.4 has further details.

Note that `meuse@data` is shorthand for `slot(meuse, "data")`.

4.5 *Common Sources of Surprise or Difficulty*

Character vectors, when incorporated as columns of a data frame, become by default factors.

Factors can often be treated as vectors of text strings, with values given by the factor levels. Watch however for contexts where the integer codes are used instead.

Use `is.na()` to check for missing values. Do not try to test for equality with `NA`. Refer back to Section 4.1.3.

If there is a good alternative, avoid the attaching of data frames. If you do use this mechanism, be aware of the traps.

Assignment of new values to an attached data frame creates a new local data frame with the same name. The new local copy remains in the workspace when the data frame is detached.

The syntax `elasticband[,2]`, extracts the second column from the data frame `elasticband`, yielding a numeric vector. Observe however that `elasticband[2, ]` yields a data frame, rather than the numeric vector that the user may require. Use the function `unlist()` to extract the vector of numeric values.

4.6 Summary

Important R data structures are vectors, factors, data frames and lists. Vector modes include numeric, logical, character or complex.

Factors, used for categorical data, can be important in the use of many of R's modeling functions. Ordered factors are appropriate for use with ordered categorical data.

Use `table()` for tables of counts, and `xtabs()` for tables of counts or totals.

R allows the use of infinite Values (`Inf` or `-Inf`) and `NaN`s (not a number) in calculations. Introduce such quantities into your calculations only if you understand the implications.

A matrix is a vector that is stacked column upon column into a rectangular array that has dimensions given by its dimension attribute. A data frame is, by contrast, a list of columns.

Matrices are in some (not all) contexts handled similarly to data frames whose elements are all of one type (typically all numeric).

Calculations with matrices are likely to be much faster than with data frames.

Lists are "non-atomic" vectors. Use the function `c()` (concatenate) to join lists, just as for "atomic" vectors.

Modeling functions typically output a *model object* that has a list structure. This holds information from the model fit, in a form from which generic model functions can then extract commonly required forms of output.

Generic functions that may be used with model objects typically include `print()`, `summary()`, `fitted()`, `coef()` and `resid()`.

4.7 Exercises

1. Find an R function that will sort a vector. Give an example.

2. Modify the function `mean.and.sd()` so that it outputs, in addition to mean and standard deviation, the number of vector elements.

3. *What is the mode of: (i) a factor; (ii) a dataframe?; (iii) a list that is not necessarily a dataframe? Apply the function `mode()` to objects of each of these classes. Explain what you find.

4. The attempt to assign values to an expression whose subscripts include missing values generates an error. Run the following code and explain the error that results:

```
y <- c(1, NA, 3, 0, NA)
y[y > 0]
y[y > 0] <- c(11, 12)
```

5. Run the following code:

```
gender <- factor(c(rep("female", 91), rep("male", 92)))
table(gender)
gender <- factor(gender, levels=c("male", "female"))
table(gender)
gender <- factor(gender, levels=c("Male", "female")) # Note the mistake
                              # The level was "male", not "Male"
table(gender)
rm(gender)                    # Remove gender
```

The output from the final `table(gender)` is

```
gender
  Male female
     0     91
```

Explain the numbers that appear.

6. In the data set `nswpsdi1` (`DAAGxtras`), do the following for each of the two levels of `trt`:

(a) Determine the numbers for each of the levels of `black`;

(b) Determine the numbers for each of the levels of `hispanic`; item Determine the numbers for each of the levels of `marr` (married).

7. Sort the rows in the data frame `Acmena` in order of increasing values of `dbh`.

[Hint: Use the function `order()`, applied to `age` to determine the order of row numbers required to sort rows in increasing order of age. Reorder rows of `Acmena` to appear in this order.]

```
Acmena <- subset(rainforest, species=="Acmena smithii")
ord <- order(Acmena$dbh)
acm <- Acmena[ord, ]
```

Sort the row names of `possumsites` (*DAAG*) into alphanumeric order. Reorder the rows of `possumsites` in order of the row names.

8(a) Create a `for` loop that, given a numeric vector, prints out one number per line, with its square and cube alongside.

(b) Look up `help(while)`. Show how to use a `while` loop to achieve the same result.

(c) Show how to achieve the same result without the use of an explicit loop.

9. Here are examples that illustrate the use of `paste()` and `paste0()`:

```
paste("Leo", "the", "lion")
paste("a", "b")
paste0("a", "b")
paste("a", "b", sep="")
paste(1:5)
paste(1:5, collapse="")
```

What are the respective effects of the parameters `sep` and `collapse`?

10. The following function calculates the mean and standard deviation of a numeric vector.

```
meanANDsd <- function(x){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, sd = sdev) # The function returns this vector
}
```

Modify the function so that: (a) the default is to use `rnorm()` to generate 20 random normal numbers, and return the standard deviation; (b) if there are missing values, the mean and standard deviation are calculated for the remaining values.

11. Try the following:

```
class(2)
class("a")
class(cabbages$HeadWt)     # cabbages is in the datasets package
class(cabbages$Cult)
```

Now do `sapply(cabbages, class)`, and note which columns hold numerical data. Extract those columns into a separate data frame, perhaps named `numtinting`.

[Hint: `cabbages[, c(2,3)]` is not the correct answer, but it is, after a manner of speaking, close!]

12. Functions that may be used to get information about data frames include `str()`, `dim()`, `row.names()` and `names()`. Try each of these functions with the data frames `allbacks`, `ant111b` and `tinting` (all in *DAAG*).

For getting information about each column of a data frame, use `sapply()`. For example, the following applies the function `class()` to each column of the data frame `ant111b`.

```
library(DAAG)
sapply(ant111b, class)
```

For columns in the data frame `tinting` that are factors, use `table()` to tabulate the number of values for each level.

5
Data Input and Storage

5.1 *Data Input from a File

.

Use of the RStudio menu is recommended. This is fast, and allows a visual check of the data layout before input proceeds. If input options are incorrectly set, these can be changed as necessary before proceeding. The code used for input is shown. In those rare cases where input options are required for which the menu does not make provision, the command line code can be edited as needed, before proceeding.

Most data input functions allow import from a file that is on the web — give the URL when specifying the file. Another possibility is to copy the file, or a relevant part of it, to the clipboard. For reading from and writing to the clipboard under Windows, see `http://bit.ly/2sxyOhG`. For MacOS, see `http://bit.ly/2t1nX0I`

5.1.1 Managing input is from the RStudio menu

Data input that is initiated from the RStudio menu uses functions from the package `readr` for input of tabular data. The function `readr::read_table()` replaces `read.table()`, `readr::read_csv()` replaces `read.csv()`, and similarly for other `read.table()` aliases.

It is important to check, when data have been entered, that data values appear sensible. Do minimal checks on: ranges of variable values, the mode of the input columns (numeric or factor, or . . .). Scatterplot matrices are helpful both for checking variable ranges and for identifying impossible or unusual combinations of variable values.

It uses the function `readxl::readxl()` for Excel spreadsheet data. There is provision, also, using functions from the package `haven`, to import data from SPSS (POR and SAV files), from SAS (XPT and SAS files), and from Stata (DTA files).

See `vignette("semantics", package="haven")` for details of the way that labelled data and missing values are handled, for input from SPSS, SAS, and Stata.

Output is in all cases to a tibble, which is a specialized form of data frame. Character columns are not automatically converted to factors, column names are not converted into valid R identifiers, and row names are not set. For subsequent processing, there are important differences between tibbles and data frames that users need to note.

5.1.2 Input using the `read.table()` family of functions

There are several aliases for `read.table()` that have different settings for input defaults. Note in particular `read.csv()`, for reading in comma delimited **.csv** files such as can be output from Excel spreadsheets. See `help(read.table)`. Recall that

Non-default option settings can however, for very large files, severely slow data input.

For factor columns check that the levels are as expected.

- Character vectors are by default converted into factors. To prevent such type conversions, specify `stringsAsFactors=FALSE`.
- Specify `heading=TRUE`[1] to indicate that the first row of input has column names. Use `heading=FALSE` to indicate that it holds data. [If names are not given, columns have default names `V1`, `V2`, ...]
- Use the parameter `row.names`, then specifying a column number, to specify a column for use to provide row names.

[1] By default, if the first row of the file has one less field than later rows, it is taken to be a header row. Otherwise, it is taken as the first row of data.

Issues that may complicate input

Where data input fails, consider using `read.table()` with the argument `fill=TRUE`, and carefully check the input data frame. Blank fields will be implicitly added, as needed, so that all records have an equal number of identified fields.

NB also that `count.fields()` counts the number of fields in each record — albeit watch for differences from input fields as detected by the input function.

Carefully check the parameter settings[2] for the version of the input command that is in use. It may be necessary to change the field separators (specify `sep`), and/or the missing value character(s) (specify `na.strings`). Embedded quotes and comment characters (#; by default anything that follows # on the same line is ignored.) can be a source of difficulty.

[2] For text with embedded single quotes, set `quote = ""`. For text with # embedded; change `comment.char` suitably.

Where a column that should be numeric is converted to a factor this is an indication that it has one or more fields that, as numbers, would be illegal. For example, a "1" (one) may have been mistyped as an "l" (ell), or "0" (zero) as "O" (oh).

Among other possibilities, there may be a non-default missing value symbol (e.g., `"."`), but without using `na.strings` to indicate this.

Note options that allow the limiting of the number of input rows. For `read.table()`) and aliases, set `nrows`. For functions from the `readr` package, set `n_max`. For `scan()`, discussed in the next subsection, set `nlines`. All these functions accept the argument `skip`, used to set the number of lines to skip before input starts.

*5.1.3 *The use of scan() for flexible data input*

Data records may for example spread over several rows. There seems no way for `read.table()` to handle this.

The following code demonstrates the use of `scan()` to read in the file **molclock1.txt**. To place this file in your working directory, attach the *DAAG* package and type `datafile("molclock1")`.

```
colnam <- scan("molclock1.txt", nlines=1, what="")
molclock <- scan("molclock1.txt", skip=1,
                 what=c(list(""), rep(list(1),5)))
molclock <- data.frame(molclock, row.names=1)
  # Column 1 supplies row names
names(molclock) <- colnam
```

There are two calls to `scan()`, each time taking information from the file **molclock1.txt**. The first, with `nlines=1` and `what=""`, input the column names. The second, with `skip=1` and `what=c(list(""), rep(list(1),5)))]`, input the several rows of data.

For repeated use with data files that have a similar format, consider putting the code into a function, with the `what` list as an argument.

The `what` parameter should be a list, with one list element for each field in a record. The "" in the first list element indicates that the data is to be input as character. The remaining five list elements are set to 1, indicating numeric data. Where records extend over several lines, set `multi.line=TRUE`.

5.1.4 The `memisc` package: input from SPSS and Stata

The *memisc* package has highly effective abilities for examining and inputting data from various SPSS formats. These include **.sav**, **.por**, and Stata **.dta** data types. Note in particular the ability to check the contents of the columns of the dataset before importing part or all of the file.

Note also the `haven` package, mentioned above, and the *foreign* package. The *foreign* package has functions that allow input of various types of files from Epi Info, Minitab, S-PLUS, SAS, SPSS, Stata, Systat and Octave. There are abilities for reading and writing some dBase files. For further information, see the R Data Import/Export manual.

An initial step is to use an importer function to create an *importer* object. As of now, *importer* functions are: `spss.fixed.file()`, `spss.portable.file()` (**.por** files), `spss.system.file()` (**.sav** files), and `Stata.file()` (**.dta** files). The importer object has information about the variables: including variable labels, value labels, missing values, and for an SPSS 'fixed' file the columns that they occupy, etc.

Functions that can be used with an importer object include:

Additionally, it has also information from further processing of the file header and/or the file proper that is needed in preparation for importing the file.

- `description()`: column header information;
- `codebook()`: detailed information on each column;
- `as.data.set()`: bring the data into R, as a 'data.set' object;
- `subset()`: bring a subset of the data into R, as a 'data.set' object

The functions `as.data.set()` and `subset()` yield 'data.set' objects. These have structure that is additional to that in data frames. Most functions that are available for use with data frames can be used with data.set class objects.

Use `as.data.frame()` to coerce data.set objects into data frames. Information that is not readily retainable in a data frame format may be lost in the process.

The vignette `anes48` that comes with the *memisc* package illustrates the use of the above abilities.

Example

A compressed version of the file "NES1948.POR" (an SPSS 'portable' dataset) is stored as part of the *memisc* installation. The following does the unzipping, places the file in a temporary directory, and stores the path to the file in the text string `path2file`:

To substitute your own file, store the path to the file in `path2file`.

```
library(memisc)
```

```
## Unzip; return path to "NES1948.POR"
path2file <- unzip(system.file("anes/NES1948.ZIP",package="memisc"),
                   "NES1948.POR",exdir=tempfile())
```

Now create an 'importer' object, and get summary information:

```
# Get information about the columns in the file
nes1948imp <- spss.portable.file(path2file)
show(nes1948imp)
```

```
SPSS portable file '/var/folders/00/_kpyywm16hnbs2c0dvlf0mwr0000gq/T//RtmpmuBIU5/file11c
        with 67 variables and 662 observations
```

There will be a large number of messages that draw attention to duplicate labels.

Before importing, it may be well to check details of what is in the file. The following, which restricts attention to columns 4 to 9 only, indicates the nature of the information that is provided.

Use `labels()`) to change labels, or `missing.values()` to set missing value filters, prior to data import.

```
## Get details about the columns (here, columns 4 to 9 only)
description(nes1948imp)[4:9]
```

```
$v480002
[1] "INTERVIEW NUMBER"

$v480003
[1] "POP CLASSIFICATION"

$v480004
[1] "CODER"

$v480005
[1] "NUMBER OF CALLS TO R"

$v480006
[1] "R REMEMBER PREVIOUS INT"

$v480007
[1] "INTR INTERVIEW THIS R"
```

As there are in this instance 67 columns, it might make sense to look at columns perhaps 10 at a time.

More detailed information is available by using the R function `codebook()`. The following gives the codebook information for column 5:

This is more interesting than what appears for columns (1 - 4).

```
## Get codebook information for column 5
codebook(nes1948imp[, 5])
```

```
=====================================================

   nes1948imp[, 5] 'POP CLASSIFICATION'

-----------------------------------------------------

   Storage mode: double
   Measurement: nominal

         Values and labels    N    Percent

   1   'METROPOLITAN AREA'  182   27.5 27.5
   2   'TOWN OR CITY'       354   53.5 53.5
   3   'OPEN COUNTRY'       126   19.0 19.0
```

The following imports a subset of just four of the columns:

```
vote.socdem.48 <- subset(nes1948imp,
              select=c(
                  v480018,
                  v480029,
```

```
                    v480030,
                    v480045
                    ))
```

To import all columns, do:

```
socdem.48 <- as.data.set(nes1948imp)
```

For more detailed information, type:

```
## Go to help page for 'importers'
help(spss.portable.file)
```

Look also at the vignette:

```
vignette("anes48")
```

5.2 *Input of Data from a web page

This section notes some of the alternative ways in which data that is available from the web can be input into R. The first subsection below comments on the use of a point and click interface to identify and download data.

The web page: `http://www.visualizing.org/data/browse/` has an extensive list of web data sources. The World Bank Development Indicators database will feature prominently in the discussion below.

A point and click interface is often convenient for an initial look. Rather than downloading the data and then inputting it to R, it may be better to input it directly from the web page. Direct input into R has the advantage that the R commands that are used document exactly what has been done.[3]

[3] This may be especially important if a data download will be repeated from time to time with updated data, or if data are brought together from a number of different files, or if a subset is taken from a larger database.

Note that the functions `read.table()`, `read.csv()`, `scan()`, and other such functions, are able to read data directly from a file that is available on the web. There is a limited ability to input part only of a file.

Suppose however that the demand is to downlaod data for several of a large number of variables, for a specified range of years, and for a specified geographical area or set of countries. A number of data archives now offer data in one or more of several markup formats that assist selective access. Formats include XML, GML, JSON and JSONP.

GML, or Geography Markup Language, is based on XML.

A browser interface to World Bank data: The web page `http://databank.worldbank.org/data/home.aspx`[4] gives a point and click interface to, among other possibilities, the World Bank development indicator database. Clicking on any of 20 country names that are displayed shows data for these countries for 1991-2010, for 54 of the 1262 series that were available at last check. Depending on the series, data may be available back to 1964. Once selections have been made, click on DOWNLOAD to download the data. For input into R, downloading as a **.csv** file is convenient.

[4] Click on COUNTRY to modify the choice of countries. To expand (to 246) countries beyond the 20 that appear by default, click on Add more country. Click on SERIES and TIME to modify and/or expand those choices. Click on Apply Changes to set the choices in place.

Manipulation of these data into a form suitable for a motion chart display was demonstrated in Subsection 6.2.3

Australian Bureau of Meteorology data: Graphs of area-weighted time series of rainfall and temperature measures, for various regions of Australia, can be accessed from the Australian Bureau of Meteorology web page `http://www.bom.gov.au/cgi-bin/climate/`

change/timeseries.cgidemo. Click on Raw data set[5] to download the raw data.

Once the web path to the file that has the data has been found, the data can alternatively be input directly from the web. The following gets the annual total rainfall in Eastern Australia, from 1910 through to the present':

[5] To copy the web address, right click on Raw data set and click on Copy Link Location (Firefox) or Copy Link Address (Google Chrome) or Copy Link (Safari).

```
webroot <- "http://www.bom.gov.au/web01/ncc/www/cli_chg/timeseries/"
rpath <- paste0(webroot, "rain/0112/eaus/", "latest.txt")
totrain <- read.table(rpath)
```

A function to download multiple data series: The following accesses the latest annual data, for total rainfall and average temperature, from the command line:

```
getbom <-
function(suffix=c("AVt","Rain"), loc="eaus"){
        webroot <- "http://www.bom.gov.au/web01/ncc/www/cli_chg/timeseries/"
        midfix <- switch(suffix[1], AVt="tmean/0112/", Rain="rain/0112/")
        webpage <- paste(webroot, midfix, loc, "/latest.txt", sep="")
        print(webpage)
        read.table(webpage)$V2
        }
##
## Example of use
offt = c(seaus=14.7, saus=18.6, eaus=20.5,  naus=24.7, swaus=16.3,
         qld=23.2, nsw=17.3, nt=25.2,sa=19.5, tas=10.4, vic=14.1,
         wa=22.5, mdb=17.7, aus=21.8)
z <- list()
for(loc in names(offt))z[[loc]] <- getbom(suffix="Rain", loc=loc)
bomRain <- as.data.frame(z)
```

The function can be re-run each time that data is required that includes the most recent year.

*Extraction of data from tables in web pages

The function `readHTMLTable()`, from the *XML* package, will prove very useful for this. It does not work, currenty at least, for pages that use https:.

Historical air crash datra: The web page http://www.planecrashinfo.com/database.htm has links to tables of aviation accidents, with one table for each year. The table for years up to and including 1920 is on the web page http://www.planecrashinfo.com/1920/1920.htm, that for 1921 on the page http://www.planecrashinfo.com/1921/1921.htm, and so on through until the most recent year. The following code inputs the table for years up to and including 1920:

```
library(XML)
```

```
url <- "http://www.planecrashinfo.com/1920/1920.htm"
to1920 <- readHTMLTable(url, header=TRUE)
to1920 <- as.data.frame(to1920)
```

The following inputs data from 2010 through until 2014:

```
url <- paste0("http://www.planecrashinfo.com/",
              2010:2014, "/", 2010:2014, ".htm")
tab <- sapply(url, function(x)readHTMLTable(x, header=TRUE))
```

```
## The following less efficent alternative code spells the steps out in more detail
## tab <- vector('list', 5)
## k <- 0
## for(yr in 2010:2014){
##  k <- k+1
##  url <- paste0("http://www.planecrashinfo.com/", yr, "/", yr, ".htm")
##  tab[[k]] <- as.data.frame(readHTMLTable(url, header=TRUE))
## }
```

Now combine all the tables into one:

```
## Now combine the 95 separate tables into one
airAccs <- do.call('rbind', tab)
names(airAccs) <- c("Date", "Location/Operator",
                    "AircraftType/Registration", "Fatalities")
airAccs$Date <- as.Date(airAccs$Date, format="%d %b %Y")
```

The help page `help(readHTMLTable)` gives examples that demonstrate other possibilities.

5.2.1 *Embedded markup — XML and alternetives

Data are are now widely available, from a number of differet web sites, in one or more of several markup formats. Markup code, designed to make the file self-describing, is included with the data. The user does not need to supply details of the data structure to the software reading the data.

Markup languages that may be used include XML, GML, JSON and JSONP. Queries are built into the web address. Alternatives to setting up the query directly may be:

For details of markup use, as they relate to the World Bank Development Indicators database, see http://data.worldbank.org/node/11.

- Use a function such as `fromJSON()` in the *RJSONIO* package to set up the link and download the data;
- In a few cases, functions have been provided in R packages that assist selection and downloading of data. For the World Bank Development Indicators database, note `WDI()` and other functions in the *WDI* package.

Download of NZ earthquake data: Here the GML markup conventions are used, as defined by the WFS OGC standard. Details can be found on the website http://info.geonet.org.nz/display/appdata/Earthquake+Web+Feature+Service

WFS is Web Feature Service. OGC is Open Geospatial Consortium. GML is Geographic Markup language GML, based on XML.

The **.csv** format is one of several formats in which data can be retrieved.

The following extracts earthquake data from the New Zealand GeoNet website. Data is for 1 September 2009 onwards, through until the current date, for earthquakes of magnitude greater than 4.5.

```
## Input data from internet
from <-
  paste(c("http://wfs-beta.geonet.org.nz/",
          "geoserver/geonet/ows?service=WFS",
          "&version=1.0.0",
          "&request=GetFeature",
          "&typeName=geonet:quake",
          "&outputFormat=csv",
          "&cql_filter=origintime>='2009-08-01'",
          "+AND+magnitude>4.5"),
        collapse="")
quakes <- read.csv(from)
z <- strsplit(as.character(quakes$origintime),
              split="T")
quakes$Date <- as.Date(sapply(z, function(x)x[1]))
quakes$Time <- sapply(z, function(x)x[2])
```

World Bank data — using the WDI *package* Use the function `WDIsearch()` to search for indicators. Thus, to search for indicators with "CO2" in their name, enter `WDIsearch('co2')`. Here are the first 4 (out of 38) that are given by such a search:

```
library(WDI)
```

```
WDIsearch('co2')[1:4,]
```

```
     indicator
[1,] "EN.ATM.CO2E.CP.KT"
[2,] "EN.ATM.CO2E.EG.ZS"
[3,] "EN.ATM.CO2E.FF.KT"
[4,] "EN.ATM.CO2E.FF.ZS"
     name
[1,] "CO2 emissions from cement production (thousand metric tons)"
[2,] "CO2 intensity (kg per kg of oil equivalent energy use)"
[3,] "CO2 emissions from fossil-fuels, total (thousand metric tons)"
[4,] "CO2 emissions from fossil-fuels (% of total)"
```

Use the function `WDI()` to input indicator data, thus:

```
library(WDI)
inds <- c('SP.DYN.TFRT.IN','SP.DYN.LE00.IN', 'SP.POP.TOTL',
 'NY.GDP.PCAP.CD', 'SE.ADT.1524.LT.FE.ZS')
indnams <- c("fertility.rate", "life.expectancy", "population",
             "GDP.per.capita.Current.USD", "15.to.25.yr.female.literacy")
names(inds) <- indnams
wdiData <- WDI(country="all",indicator=inds, start=1960, end=2013, extra=TRUE)
colnum <- match(inds, names(wdiData))
names(wdiData)[colnum] <- indnams
## Drop unwanted "region"
WorldBank <- droplevels(subset(wdiData, !region %in% "Aggregates"))
```

The effect of `extra=TRUE` is to include the additional variables `iso2c` (2-character country code), `country`, `year`, `iso3c` (3-character country code), `region`, `capital`, `longitude`, `latitude`, `income` and `lending`.

The function `WDI()` calls the non-visible function `wdi.dl()`, which in turn calls the function `fromJSON()` from the *RJSONIO* package. To see the code for `wdi.dl()`, type `getAnywhere("wdi.dl")`.

The data frame `Worldbank` that results is in a form where it can be used with the *googleVIS* function `gvisMotionChart()`, as described in Section 7.5.1

5.3 Creating and Using Databases

The *RSQLite* package makes it possible to create an SQLite database, or to add new rows to an existing table, or to add new table(s), within an R session. The SQL query language can then be used to access tables in the database. Here is an example. First create the database:

In addition to the *RSQLite*, note the *RMySQL* and *ROracle* packages. All use the interface provided by the *DBI* package.

```
library(DAAG)
library(RSQLite)
driveLite <- dbDriver("SQLite")
con <- dbConnect(driveLite, dbname="hillracesDB")
dbWriteTable(con, "hills2000", hills2000,
             overwrite=TRUE)
dbWriteTable(con, "nihills", nihills,
             overwrite=TRUE)
dbListTables(con)
```

```
[1] "hills2000" "nihills"
```

The database `hillracesDB`, if it does not already exist, is created in the working directory.

Now input rows 16 to 20 from the newly created database:

```
## Get rows 16 to 20 from the nihills DB
dbGetQuery(con,
  "select * from nihills limit 5 offset 15")
```

```
  dist climb   time  timef
1  5.5  2790 0.9483 1.2086
2 11.0  3000 1.4569 2.0344
3  4.0  2690 0.6878 0.7992
4 18.9  8775 3.9028 5.9856
5  4.0  1000 0.4347 0.5756
```

```
dbDisconnect(con)
```

5.4 *File compression:

The functions for data input in versions 2.10.0 and later of R are able to accept certain types of compressed files. This extends to `scan()` and to functions such as `read.maimages()` in the *limma* package, that use the standard R data input functions.

By way of illustration, consider the files **coral551.spot**, . . . , **coral556.spot** that are in the subdirectory **doc** of the *DAAGbio* package. In a directory that held the uncompressed files, they were created by typing, on a Unix or Unix-like command line:

Severer compression: replace
`gzip -9`
by
`xz -9e`.

```
gzip -9 coral55?.spot
```

The **.zip** files thus created were renamed back to ***.spot** files.

When saving large objects in image format, specify `compress=TRUE`. Alternatives that may lead to more compact files are `compress="bzip2"` and `compress="xz"`.

Note also the R functions `gzfile()` and `xzfile()` that can be used to create files in a compressed text format. This might for example be text that has been input using `readLines()`.

5.5 Summary

Following input, perform minimal checks that values in the various columns are as expected.

With very large files, it can be helpful to read in the data in chunks (ranges of rows).

Note mechanisms for direct input of web data. Many data archives now offer one or more of several markup formats that facilitate selective access.

6

Data Manipulation and Management

Data analysis has as its end point the use of forms of data summary that will convey, fairly and succinctly, the information that is in the data. The fitting of a model is itself a form of data summary.

Be warned of the opportunities that simple forms of data summary, which seem superficially harmless, can offer for misleading inferences. These issues affect, not just data summary per se, but all modeling. Data analysis is a task that should be undertaken with critical faculties fully engaged.

Data summaries that can lead to misleading inferences arise often, from a unbalance in the data and/or failure to account properly for important variables or factors.

Alternative types of data objects

Column objects: These include (atomic) vectors, factors, and dates.

Date and date-time objects: The creation and manipulations of date objects will be described below.

Data Frames: These are rectangular structures. Columns may be "atomic" vectors, or factors, or other objects (such as dates) that are one-dimensional.

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

Matrices and arrays: Matrices[1] are rectangular arrays in which all elements have the same mode. An array is a generalization of a matrix to allow an arbitrary number of dimensions.

[1] Internally, matrices are one long vector in which the columns follow one after the other.

Tables: A table is a specialized form of array.

Lists: A list is a collection of objects that can be of arbitrary class. List elements are themselves lists. In more technical language, lists are *recursive* data structures.

S3 model objects: These are lists that have a defined structure.

S4 objects: These are specialized data structures with tight control on the structure. Unlike S3 objects, they cannot be manipulated as lists. Modeling functions in certain of the newer packages[2] return S4 objects.

[2] These include *lme4*, the Bioconductor packages, and the spatial analysis packages.

6.1 Manipulations with Lists, Data Frames and Arrays

Recall that data frames are lists of columns that all have the same length. They are thus a specialised form of list. Matrices are two-dimensional arrays. Tables are in essence arrays that hold numeric values.

6.1.1 Tables and arrays

The dataset `UCBAdmissions` is stored as a 3-dimensional table. If we convert it to an array, very little changes:

It changes from a table object to a numeric object, which affects the way that it is handled by some functions. In either case, what we have is a numeric vector of length 24 (= $2 \times 2 \times 6$) that is structured to have dimensions 2 by 2 by 6.

6.1.2 Conversion between data frames and tables

The three-way table `UCBAdmissions` are admission frequencies, by Gender, for the six largest departments at the University of California at Berkeley in 1973. For a reference to a web page that has the details; see the belp page for `UCBAdmissions`. Type

```
help(UCBAdmissions)      # Get details of the data
example(UCBAdmissions)
```

Note the margins of the table:

```
str(UCBAdmissions)
```

```
 table [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
 - attr(*, "dimnames")=List of 3
  ..$ Admit : chr [1:2] "Admitted" "Rejected"
  ..$ Gender: chr [1:2] "Male" "Female"
  ..$ Dept  : chr [1:6] "A" "B" "C" "D" ...
```

In general, operations with a table or array are easiest to conceptualise if the table is first converted to a data frame in which the separate dimensions of the table become columns. Thus, the `UCBAdmissions` table will be converted to a data frame that has columns `Admit`, `Gender` and `Dept`. Either use the `as.data.frame.table()` command from base R, or use the `adply()` function from the *plyr* package.

The following uses the function `as.data.frame.table()` to convert the 3-way table `UCBAdmissions` into a data frame in which the margins are columns:

As `UCBAdmissions` is a table (not an array), `as.data.frame(UCBAdmissions)` will give the same result.

```
UCBdf <- as.data.frame.table(UCBAdmissions)
head(UCBdf, 5)
```

```
     Admit Gender Dept Freq
1 Admitted   Male    A  512
```

```
2 Rejected   Male     A  313
3 Admitted Female     A   89
4 Rejected Female     A   19
5 Admitted   Male     B  353
```

Alternatively, use the function `adply()` from the *plyr* package that is described in Section 6.2. Here the `identity()` function does the manipulation, working with all three dimensions of the array:

```
library(plyr)
UCBdf <- adply(.data=UCBAdmissions,
               .margins=1:3,
               .fun=identity)
names(UCBdf)[4] <- "Freq"
```

First, calculate overall admission percentages for females and males. The following calculates also the total accepted, and the total who applied:

```
library(dplyr)
```

```
gpUCBgender <- dplyr::group_by(UCBdf, Gender)
AdmitRate <- dplyr::summarise(gpUCBgender,
                              Accept=sum(Freq[Admit=="Admitted"]),
                              Total=sum(Freq),
                              pcAccept=100*Accept/Total)
AdmitRate
```

```
# A tibble: 2 x 4
  Gender Accept Total pcAccept
  <fctr>  <dbl> <dbl>    <dbl>
1   Male   1198  2691    44.52
2 Female    557  1835    30.35
```

Now calculate admission rates, total number of females applying, and total number of males applying, for each department:

```
gpUCBgd <- dplyr::group_by(UCBdf, Gender, Dept)
rateDept <- dplyr::summarise(gpUCBgd,
    Total=sum(Freq),
    pcAccept=100*sum(Freq[Admit=="Admitted"])/Total)
```

Results can conveniently be displayed as follows. First show admission rates, for females and males separately:

```
xtabs(pcAccept~Gender+Dept, data=rateDept)
```

```
        Dept
Gender         A      B      C      D      E      F
  Male    62.061 63.036 36.923 33.094 27.749  5.898
  Female  82.407 68.000 34.064 34.933 23.919  7.038
```

Now show total numbers applying:

```
xtabs(Total~Gender+Dept, data=rateDept)
```

```
        Dept
Gender     A   B   C   D   E   F
  Male   825 560 325 417 191 373
  Female 108  25 593 375 393 341
```

As a fraction of those who applied, females were strongly favored in department A, and males somewhat favored in departments C and E. Note however that relatively many males applied to A and B, where admission rates were high. This biased overall male rates upwards. Relatively many females applied to C, D and F, where rates were low. This biased the overall female rates downwards.

The overall bias arose because males favored departments where admission rates were relatively high.

6.1.3 *Table margins*

For working directly on tables, note the function `margin.table()`. The following retains margin 1 (`Admit`) and margin 2 (`Gender`), adding over `Dept` (the remaining margin):

```
## Tabulate by Admit (margin 2) & Gender (margin 1)
(margin21 <- margin.table(UCBAdmissions,
                          margin=2:1))
```

Take margin 2, first, then margin 1, gving a table where rows correspond to levels of `Gender`.

```
        Admit
Gender   Admitted Rejected
  Male       1198     1493
  Female      557     1278
```

Use the function `margin.table()` to turn this into a table that has the proportions in each row:

```
prop.table(margin21, margin=1)
```

```
        Admit
Gender   Admitted Rejected
  Male     0.4452   0.5548
  Female   0.3035   0.6965
```

6.1.4 *Categorization of continuous data*

The data frame `bronchit`, in the *DAAGviz* package, has observations on 212 men in a sample of Cardiff (Wales, UK) enumeration districts. Variables are `r` (1 if respondent suffered from chronic bronchitis and 0 otherwise), `cig` (number of cigarettes smoked per day) and `poll` (the smoke level in the locality).

The dataset `bronchit` may alternatively be found in the *SMIR* package.

It will be convenient to define a function `props` that calculates the proportion of the total in the first (or other nominated element) of a vector:

```
props <- function(x, elem=1)sum(x[elem])/sum(x)
```

Now use the function `cut()` to classify the data into four categories, and form tables:

```
library(DAAGviz)
catcig <- with(bronchit,
               cut(cig, breaks=c(0,1,10,30),
                   include.lowest=TRUE))
tab <- with(bronchit, table(r, catcig))
round(apply(tab, 2, props, elem=2), 3)
```

The argument `breaks` can be either the number of intervals, or it can be a vector of break points such that all data values lie within the range of the breaks. If the smallest of the break points equals the smallest data value, supply the argument `include.lowest=TRUE`.

```
  [0,1]   (1,10] (10,30]
  0.072    0.281   0.538
```

There is a clear increase in the risk of bronchitis with the number of cigarettes smoked.

This categorization was purely for purposes of preliminary analysis. Categorization for purposes of analysis is, with the methodology and software that are now available, usually undesirable. Tables that are based on categorization can nevertheless be useful in data exploration.

It was at one time common practice to categorize continuous data, in order to allow analysis methods for multi-way tables. There is a loss of information, which can at worst be serious.

6.1.5 *Matrix Computations

Let `X` (n by p), `Y` (n by p) and B (p by k) be numeric matrices. Some of the possibilities are:

```
X + Y              # Elementwise addition
X * Y              # Elementwise multiplication
X %*% B            # Matrix multiplication
solve(X, Y)        # Solve X B = Y for B
svd(X)             # Singular value decomposition
qr(X)              # QR decomposition
t(X)               # Transpose of X
```

Note that if `t()` is used with a data frame, a matrix is returned. If necessary, all values are coerced to the same mode.

Calculations with data frames that are slow and time consuming will often be much faster if they can be formulated as matrix calculations. This is in general become an issue only for very large datasets, with perhaps millions of observations. Section 6.4 has examples. For small or modest-sized datasets, convenience in formulating the calculations is likely to be more important than calculation efficiency.

Section 4.3.7 will discuss the use of `apply()` for operations with matrices, arrays and tables.

6.2 plyr, dplyr & reshape2 *Data Manipulation*

The *plyr* package has functions that together:

- provide a systematic approach to computations that perform a desired operation across one or more dimensions of an array, or of a data frame, or of a list;
- allow the user to choose whether results will be returned as an array, or as a data frame, or as a list.

The *dplyr* package has functions for performing various summary and other operations on data frames. For many purposes, it supersedes the *plyr* package.

The *reshape2* package is, as its name suggests, designed for moving between alternative data layouts.

6.2.1 plyr

The *plyr* package has a separate function for each of the nine possible mappings. The first letter of the function name (one of `a` = array,

d = data frame, l = list) denotes the class of the input object, while the second letter (the same choice of one of three letters) denotes the class of output object that is required. This pair of letters is then followed by ply.

Here is the choice of functions:

	Class of Output Object		
	a (array)	d (data frame)	l (list)
Class of Input Object			
a (array)	aaply	adply	alply
d (data frame)	daply	ddply	dlply
l (list)	laply	ldply	llply

First observe how the function adply can be used to change from a tabular form of representation to a data frame. The dimension names will become columns in the data frame.

```
detach("package:dplyr")
library(plyr)
```

```
dreamMoves <-
   matrix(c(5,3,17,85), ncol=2,
          dimnames=list("Dreamer"=c("Yes","No"),
                        "Object"=c("Yes","No")))
(dfdream <- plyr::adply(dreamMoves, 1:2,
                        .fun=identity))
```

```
  Dreamer Object V1
1     Yes    Yes  5
2      No    Yes  3
3     Yes     No 17
4      No     No 85
```

To get the table back, do:

```
plyr::daply(dfdream, 1:2, function(df)df[,3])
```

```
       Object
Dreamer Yes No
    Yes   5 17
    No    3 85
```

The following calculates sums over the first two dimensions of the table UCBAdmissions:

```
plyr::aaply(UCBAdmissions, 1:2, sum)
```

Here, aaply() behaves exactly like apply().

```
          Gender
Admit      Male Female
  Admitted 1198    557
  Rejected 1493   1278
```

The following calculates, for each level of the column trt in the data frame nswdemo, the number of values of re74 that are zero:

```
library(DAAG, quietly=TRUE)
plyr::daply(nswdemo, .(trt),
      function(df)sum(df[,"re74"]==0, na.rm=TRUE))
```

```
  0   1
195 131
```

To calculate the proportion that are zero, for each of control and treatment and for each of non-black and black, do:

Notice the use of the syntax `.(trt, black)` to identify the columns `trt` and `black`. This is an alternative to `c("trt", "black")`.

```
options(digits=3)
plyr::daply(nswdemo, .(trt, black),
      function(df)sum(df[,"re75"]==0)/nrow(df))
```

```
   black
trt     0     1
  0 0.353 0.435
  1 0.254 0.403
```

The function `colwise()` takes as argument a function that operates on a column of data, returning a function that operates on all nominated columns of a data frame. To get information on the proportion of zeros for both of the columns `re75` and `re78`, and for each of non-black and black, do:

Here, `colwise()` operates on the objects that are returned by splitting up the data frame `nswdemo` according to levels of `trt` and `black`. Note the use of `ddply()`, not `daply()`.

```
plyr::ddply(nswdemo, .(trt, black),
      colwise(function(x)sum(x==0)/length(x),
              .cols=.(re75, re78)))
```

```
  trt black  re75   re78
1   0     0 0.353 0.1529
2   0     1 0.435 0.3412
3   1     0 0.254 0.0847
4   1     1 0.403 0.2605
```

6.2.2 *Use of* dplyr *with Word War 1 cricketer data*

Data in the data frame `cricketer`, extracted by John Aggleton (now at Univ of Cardiff), are from records of UK first class cricketers born 1840 – 1960. Variables are

- Year of birth
- Years of life (as of 1990)
- 1990 status (dead or alive)
- Cause of death: killed in action / accident / in bed
- Bowling hand – right or left

The following creates a data frame in which the first column has the year, the second the number of right-handers born in that year, and the third the number of left-handers born in that year. .

Both *plyr* and *dplyr* have functions `summarise()`. As in the code shown, detach *plyr* before proceeding. Alternatively, or additionally, specify `dplyr::summarise()` rather than `summarise()`

```
library(DAAG)
detach("package:plyr")
library(dplyr)
```

```
names(cricketer)[1] <- "hand"
gpByYear <- group_by(cricketer, year)
```

```
lefrt <- dplyr::summarise(gpByYear,
                          left=sum(hand=='left'),
                          right=sum(hand=='right'))
## Check first few rows
lefrt[1:4, ]
```

```
# A tibble: 4 x 3
   year  left right
  <int> <int> <int>
1  1840     1     6
2  1841     4    16
3  1842     5    16
4  1843     3    25
```

The data frame is split by values of `year`. Numbers of left and right handers are then tabulated.

From the data frame `cricketer`, we determine the range of birth years for players who died in World War 1. We then extract data for all cricketers, whether dying or surviving until at least the final year of Workd War 1, whose birth year was within this range of years. The following code extracts the relevant range of birth years.

Note that a cricketer who was born in 1869 would be 45 in 1914, while a cricketer who was born in 1896 would be 18 in 1914.

```
## Use subset() from base R
ww1kia <- subset(cricketer,
                 kia==1 & (year+life)%in% 1914:1918)
range(ww1kia$year)
```

```
[1] 1869 1896
```

Alternatively, use `filter()` from *dplyr*:

```
ww1kia <- filter(cricketer,
                 kia==1, (year+life)%in% 1914:1918)
```

For each year of birth between 1869 and 1896, the following expresses the number of cricketers killed in action as a fraction of the total number of cricketers (in action or not) who were born in that year:

```
## Use filter(), group_by() and summarise() from dplyr
crickChoose <- filter(cricketer,
                      year%in%(1869:1896), ((kia==1)|(year+life)>1918))
gpByYearKIA <- group_by(crickChoose, year)
crickKIAyrs <- dplyr::summarise(gpByYearKIA,
                                kia=sum(kia), all=length(year), prop=kia/all)
crickKIAyrs[1:4, ]
```

```
# A tibble: 4 x 4
   year   kia   all   prop
  <int> <int> <int>  <dbl>
1  1869     1    37 0.0270
2  1870     2    36 0.0556
3  1871     1    45 0.0222
4  1872     0    39 0.0000
```

For an introduction to *dplyr*, enter:

```
vignette("introduction", package="dplyr")
```

6.2.3 reshape2: `melt()`, `acast()` & `dcast()`

The *reshape2* package has functions that move between a dataframe layout where selected columns are unstacked, and a layout where they are stacked. In moving from an unstacked to a stacked layout, column names become levels of a factor. In the move back from stacked to unstacked, factor levels become column names.

Here is an example of the use of `melt()`:

```
## Create dataset Crimean, for use in later calculations
library(HistData)   # Nightingale is from this package
library(reshape2)   # Has the function melt()
Crimean <- melt(Nightingale[,c(1,8:10)], "Date")
names(Crimean) <- c("Date", "Cause", "Deaths")
Crimean$Cause <- factor(sub("\\.rate", "", Crimean$Cause))
Crimean$Regime <- ordered(rep(c(rep('Before', 12), rep('After', 12)), 3),
                          levels=c('Before', 'After'))
formdat <- format.Date(sort(unique(Crimean$Date)), format="%d %b %y")
Crimean$Date <- ordered(format.Date(Crimean$Date,
                        format="%b %y"), levels=formdat)
```

The dataset is now in a suitable form for creating a Florence Nightingale style wedge plot, in Figure C.3.

The dataset `Crimean` has been included in the *DAAGviz* package.

Reshaping data for Motion Chart display – an example

The following inputs and displays World Bank Development Indicator data that has been included with the package *DAAGviz*:

```
## DAAGviz must be installed, need not be loaded
path2file <- system.file("datasets/wdiEx.csv", package="DAAGviz")
wdiEx <- read.csv(path2file)
print(wdiEx, row.names=FALSE)
```

```
 Country.Name Country.Code     Indicator.Name Indicator.Code    X2010    X2000
    Australia          AUS Labor force, total SL.TLF.TOTL.IN 1.17e+07 9.62e+06
    Australia          AUS  Population, total    SP.POP.TOTL 2.21e+07 1.92e+07
        China          CHN Labor force, total SL.TLF.TOTL.IN 8.12e+08 7.23e+08
        China          CHN  Population, total    SP.POP.TOTL 1.34e+09 1.26e+09
```

A *googleVis* Motion Chart does not make much sense for this dataset as it stands, with data for just two countries and two years. Motion charts are designed for showing how scatterplot relationships, here between forest area and population, have changed over a number of years. The dataset will however serve for demonstrating the reshaping that is needed.

For input to Motion Charts, we want indicators to be columns, and years to be rows. The `melt()` and `dcast()`[3] functions from the *reshape2* package can be used to achieve the desired result. First, create a single column of data, indexed by classifying factors:

[3] Note also `acast()`, which outputs an array or a matrix.

```
library(reshape2)
wdiLong <- melt(wdiEx, id.vars=c("Country.Code",
                "Indicator.Name"),
                measure.vars=c("X2000", "X2010"))
## More simply: wdiLong <- melt(wdiEx[, -c(2,4)])
wdiLong
```

```
  Country.Code     Indicator.Name variable    value
1          AUS Labor force, total    X2000 9.62e+06
2          AUS  Population, total    X2000 1.92e+07
3          CHN Labor force, total    X2000 7.23e+08
4          CHN  Population, total    X2000 1.26e+09
5          AUS Labor force, total    X2010 1.17e+07
6          AUS  Population, total    X2010 2.21e+07
7          CHN Labor force, total    X2010 8.12e+08
8          CHN  Population, total    X2010 1.34e+09
```

Now use `dcast()` to "cast" the data frame into a form where the indicator variables are columns:

If a matrix or array is required, use `acast()` in place of `dcast()`.

```
names(wdiLong)[3] <- "Year"
wdiData <- dcast(wdiLong,
                 Country.Code+Year ~ Indicator.Name,
                 value.var="value")
wdiData
```

```
  Country.Code  Year Labor force, total Population, total
1          AUS X2000           9.62e+06          1.92e+07
2          AUS X2010           1.17e+07          2.21e+07
3          CHN X2000           7.23e+08          1.26e+09
4          CHN X2010           8.12e+08          1.34e+09
```

A final step is to replace the factor `Year` by a variable that has the values 2000 and 2010.

```
wdiData <- within(wdiData, {
   levels(Year) <- substring(levels(Year),2)
   Year <- as.numeric(as.character(Year))
})
wdiData
```

```
  Country.Code Year Labor force, total Population, total
1          AUS 2000           9.62e+06          1.92e+07
2          AUS 2010           1.17e+07          2.21e+07
3          CHN 2000           7.23e+08          1.26e+09
4          CHN 2010           8.12e+08          1.34e+09
```

6.3 Session and Workspace Management

6.3.1 Keep a record of your work

A recommended procedure is to type commands into an editor window, then sending them across to the command line. This makes it possible to recover work on those hopefully rare occasions when the session aborts.

Be sure to save the script file from time to time during the session, and upon quitting the session.

6.3.2 Workspace management

For tasks that make heavy memory demands, it may be important to ensure that large data objects do not remain in memory once they are no longer needed. There are two complementary strategies:

- Objects that cannot easily be reconstructed or copied from elsewhere, but are not for the time being required, are conveniently saved to an image file, using the `save()` function.
- Use a separate working directory for each major project.

Note the utility function `dir()` (get the names of files, by default in the current working directory).

Use `getwd()` to check the name and path of the current working directory. Use `setwd()` to change to a new working directory, while leaving the workspace contents unchanged.

Several image files ("workspaces") that have distinct names can live in the one working directory. The image file, if any, that is called **.RData** is the file whose contents will be loaded at the beginning of a new session in the directory.

The removal of clutter: Use a command of the form `rm(x, y, tmp)` to remove objects (here `x`, `y`, `tmp`) that are no longer required.

As noted in Section 2.2.2, a good precaution can be to make an archive of the workspace before such removal.

Movement of files between computers: Files that are saved in the default binary save file format, as above, can be moved between different computer systems.

Further possibilities – saving objects in text form: An alternative to saving objects[4] in an image file is to dump them, in a text format, as dump files, e.g.

```
volume <- c(351, 955, 662, 1203, 557, 460)
weight <- c(250, 840, 550, 1360, 640, 420)
dump(c("volume", "weight"), file="books.R")
```

[4] Dumps of S4 objects and environments, amongs others, cannot currently be retrieved using `source()`. See `help(dump)`.

The objects can be recreated [5] from this "dump" file by inputting the lines of **books.R** one by one at the command line. This is what, effectively, the command `source()` does.

```
source("books.R")
```

[5] The same checks are performed on dump files as if the text had been entered at the command line. These can slow down entry of the data or other object. Checks on dependencies can be a problem. These can usually be resolved by editing the R source file to change or remove offending code.

For long-term archival storage, dump (**.R**) files may be preferable to image files. For added security, retain a printed version. If a problem arises (from a system change, or because the file has been corrupted), it is then possible to check through the file line by line to find what is wrong.

6.4 Computer Intensive Computations

Computations may be computer intensive because of the size of datasets. Or the computations may themselves be demanding, even for data sets that are of modest size.

Note that using all of the data for an analysis or for a plot is not always the optimal strategy. Running calculations separately on different subsets may afford insights that are not otherwise available. The subsets may be randomly chosen, or they may be chosen to reflect, e.g., differences in time or place.

Computation will be slow where computationally intensive calculations are implemented directly in R code, rather than passed to

The computationally intensive parts of regression calculations with `lm()` work with matrices, making these relatively efficient.

efficient compiled code that is called from R. Matrix calculations are passed to highly efficient compiled code.

Where it is necessary to look for ways to speed up computations, it is important to profile computations to find which parts of the code are taking the major time. Really big improvements will come from implementing key parts of the calculation in C or Fortran rather than in an application oriented language such as R or Python. Python may do somewhat better than R.

The relatively new Julia language appears to offer spectacular improvements on both R and Python, with times that are within a factor of 2 of the Fortran or C times. See http://julialang.org/.

There can be big differences between the alternatives that may be available in R for handling a calculation. Some broad guidelines will now be provided, with examples of how differences in the handling of calculations can affect timings.

Use matrices, where possible, in preference to data frames: Most of R's modeling functions (regression, smoothing, discriminant analysis, etc.) are designed to work with data frames. Where an alternative available that works with matrices, this will be faster.

Biological expression array applications are among those that are commonly designed to work with data that is in a matrix format. The matrix or matrices may be components of a more complex data structure.

Matrix operations can be more efficient even for such a simple operation as adding a constant quantity to each element of the array, or taking logarithms of all elements. Here is an example:

```
xy <- matrix(rnorm(5*10^7), ncol=100)
dim(xy)
```

```
[1] 500000    100
```

Timings are on a mid 2012 1.8 Ghz Intel i5 Macbook Air laptop with 8 gigabytes of random access memory.

```
system.time(xy+1)
```

```
   user  system elapsed
  0.155   0.144   0.305
```

```
xy.df <- data.frame(xy)
system.time(xy.df+1)
```

```
   user  system elapsed
  0.425   0.145   0.584
```

Use efficient coding: Matrix arithmetic can be faster than the equivalent computations that use `apply()`. Here are timings for some alternatives that find the sums of rows of the matrix `xy` above:

	user	system	elapsed
`apply(xy,1,sum)`	0.528	0.087	0.617
`xy %*% rep(1,100)`	0.019	0.001	0.019
`rowSums(xy)`	0.034	0.001	0.035

The bigmemory project: For details, go to http://www.bigmemory.org/. The *bigmemory* package for R "supports the creation, storage, access, and manipulation of massive matrices". Note also the associated packages *biganalytics*, *bigtabulate*, *synchronicity*, and *bigalgebra*.

The data.table package: This allows the creation of `data.table` objects from which information can be quickly extracted, often in a fraction of the time required for extracting the same information from a data frame. The package has an accompanying vignette. To display it (assuming that the package has been installed), type

```
vignette("datatable-intro", package="data.table")
```

On 64-bit systems, massive data sets, e.g., with tens or hundreds of millions of rows, are possible. For such large data objects, the time saving can be huge.

6.5 Summary

`apply()`, and `sapply()` can be useful for manipulations with data frames and matrices. Note also the functions `melt()`, `dcast()` and `acast()` from the *reshape2* package.

Careful workspace management is important when files are large. It pays to use separate working directories for each different project, and to save important data objects as image files when they are, for the time being, no longer required.

In computations with large datasets, operations that are formally equivalent can differ greatly in their use of computational resources.

7

Graphics – base, lattice, ggplot2, rgl, googleVis...

Base Graphics (mostly 2-D):

Base graphics implements a "traditional" style of graphics

Functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`,`identify()` etc. form a suite that plot points, lines, text, etc.

The function `plot()` accepts a `data` argument, while `lines()`, `points()` and `text()` do not.

Other Graphics

(i) lattice (trellis) graphics, using the *lattice* package,
(ii) *ggplot2*, implementing Wilkinson's *Grammar of Graphics*
(iii) For 3-D graphics (Section 7.5), note *rgl*, *misc3d* & *tkrplot*
(iv) *Motion Charts* (Section 7.5.1), show a scatterplot changing with movement forward or backward in time.

lattice and *ggplot2* are built on the low-level graphics package *grid*.

Note also various special types of graph. For example word clouds, as in the *wordcloud* package, list words with size proportional to frequency.

Consider first base graphics. Relative to *lattice* and to *ggplot2*, the more traditional style of base graphics is less consistent and less structured. Each system however has its own strengths and uses.

```
## DAAG has several datasets that will be required
library(DAAG, quietly=TRUE)
```

7.1 Base Graphics

The function `plot()` is the most basic of several functions that create an initial graph. Other functions can be used to add to an existing graph. Note in particular `points()` `lines()` and `text()`.

To see a variety of base (or traditional) graphics plots, enter

```
demo(graphics)
```

Press Enter to see each new graph.

7.1.1 `plot()` and allied base graphics functions

The following are alternative ways to plot `y` against `x` (obviously `x` and `y` must be the same length):

```
plot(y ~ x)    # Use a formula to specify the graph
plot(x, y)     # Horizontal ordinate, then vertical
```

Plot `height` vs `weight` –

```
## Older syntax:
with(women,
     plot(height, weight))
```

```
## Graphics formula:
plot(weight ~ height,
     data=women)
```

The following use the argument `data` to supply the name of a data frame whose column names appear in the graphics formula:

```
plot(distance ~ stretch, data=elasticband)
plot(ACT ~ year, data=austpop, type="l")
plot(ACT ~ year, data=austpop, type="b")
```

The `points()` function adds points, while `lines()` adds lines[1] to a plot. The `text()` function adds text at specified locations. The `mtext()` function places text in one of the margins. The `axis()` function gives fine control over axis ticks and labels.

Here is a further possibility

```
with(austpop, plot(spline(year, ACT), type="l"))
  # Fit smooth curve through points
```

[1] These functions differ only in the default setting for the parameter `type`. Explicitly setting `type = "p"` causes either function to plot points.

Adding text – an example

Here is a simple example (Figure 7.1) that uses the function `text()` to label the points on a plot. Data is from the dataset `primates` (*DAAG*). The first two lines of data are:

```
## Data (1st 2 lines)
head(primates, 2)
```

```
              Bodywt Brainwt
Potar monkey      10     115
Gorilla          207     406
```

Code for a simplified version of the plot is:

```
plot(Brainwt ~ Bodywt, xlim=c(0, 300),
     ylim=c(0,1500), data=primates, fg="gray",
     xlab="Body weight (kg)",
     ylab="Brain weight (g)")
# Specify xlim to allow room for the labels
with(primates,
     text(Brainwt ~ Bodywt, cex=0.8,
          labels=rownames(primates), pos=4))
# pos: pos=1 (below), 2 (left), 3 (above)
```

Figure 7.1: Plot of brain weight against body weight, for selected primates.

Identification and Location on the Figure Region

Draw the graph first, then call the required function.

- `identify()`, discussed in Subsection 1.5, labels points.
- `locator()` prints out the co-ordinates of points. Position the cursor at the location for which coordinates are required, and click the left mouse button.

Section 1.5 described how to terminate the plot, if the limit `n` is not reached first. For `locator()`, `n` is by default set to 500.

7.1.2 Fine control – Parameter settings

In most (not all) instances, parameters can be set either using `par()`, or in a call to a plotting function (`plot()`, `points()`, ...). Changes

To store existing settings for later restoration, proceed thus:

```
oldpar <- par(cex=1.25)
  # par(oldpar) to restore
```

made using `par()` remain in place until changed again, or until a new device is opened. If made in a call to a plotting function, the change applies only to that call.

Some of the more common settings are:

- Plotting symbols: `pch` (choice of symbol); `cex` ("character expansion")[2]; `col` (color).
- Lines: `lty` (line type); `lwd` (line width); `col` (color).
- Axis limits: `xlim`; `ylim`.
- Closeness of fit to the axis limits: `xaxs`, `yaxs`.[3] Specify `xaxs="i"` for an exact fit to the data limits.
- Axis annotation and labels: `cex.axis` (for axis annotation, independently of `cex`); `cex.labels` (for axis labels).
- Margins and positioning within margin:[4] `mar` (inner margins clockwise from bottom, out of box default `mar=c(5.1, 4.1, 4.1, 2.1)`); `oma` (outer margins, use when there are multiple plots on the one graphics page); positioning within margin: `mgp` (margin line for the axis title, axis labels, and axis line, default `mgp=c(3, 1, 0)`).
- Plot shape: `pty="s"` gives a square plot.[5] (default is `pty="m"`)
- Multiple graphs on the one graphics page: `par(mfrow=c(m,n))` gives an *m* rows by *n* columns layout.

[2] Thus `par(cex=1.2)` increases plot symbol size 20% above the default.

[3] The default is `xaxs="r"`; *x*-axis limits are extended by 4% relative to data or `xlim` limits.

[4] Parameters such as `mar`, `mgp` and `oma` specify distances in 'lines' out from the relevant boundary of the figure region. Lines are in units of `mex`, where by default `mex=1`.

[5] This must be set using `par()`

For a 1 by 2 layout of plots; specify `par(mfrow=c(1,2))`. Subsection 3.2.1 has an example.

Type `help(par)` to get a (very extensive) complete list. Figure C.4 demonstrates some of the possibilities.

7.1.3 Color and Opacity

The function `colors()` gives access to 657 different color names, some of them repeats of the same colour. The function `palette()` can be used to show or set colors that will by default be used for base graphics. Thus

- `palette()` lists the colors in the current palette;
- as an example, `palette(rainbow(6))` sets the current palette to a 6-color rainbow palette;
- `palette("default")` resets back to the default.

Run the following code to show the default palette, three sequential palettes from *grDevices*, a color ramp palette given by the function `colorRampPalette()`, and two quantitative palettes from the *RColorBrewer* package.

See `help(palette)` for palettes in the base R *grDevices* package.

```
## Load to run code for Supplementary Figure 1
library(RColorBrewer)   # Required for Set1 and Dark2 RColorBrewer palettes
```

```
colpal <- rev(list(
    "Default palette" = palette()[1:8],  cm.colors = cm.colors(12),
    terrain.colors = terrain.colors(12), heat.colors = heat.colors(12),
    blueRamp = colorRampPalette(c(blues9, "white"))(12),
    "Brewer-Set1" = brewer.pal(8, "Set1"),
    "Brewer-Dark2" = brewer.pal(8, "Dark2")))
palnam <- names(colpal)
plot(1, 1, xlim=c(0.5,12.5), ylim=c(0,length(palnam)+0.5), type="n",
     axes=FALSE, xlab="", ylab="")
for(i in 1:length(palnam)){
    len <- length(colpal[[i]])
    points(1:len, rep(i,len), pch=15, col=colpal[[i]], cex=5.5)
    legend(1, i+0.025, palnam[i], adj=0, box.col="white", bg="white",
           x.intersp=0, y.intersp=0, yjust=0)
```

Each of these palettes, except the default, allows variation in the number of colors, up to a maximum. The palettes available from *RColorBrewer* include other qualitative palettes, sequential palettes, and diverging (light in the middle; dark at the extremes) palettes. To see the full range of *RColorBrewer* possibilities, type:

```
display.brewer.all()
```

Stretch the graphics window vertically (pull on an edge) so that rows do not overlap.

Qualitative schemes that may be suited for use in plots are "Set1" with yellow (the 6th color out of nine) omitted, or "Dark2", or "Accent" with the 4th color (out of 8) omitted. To extract these, do for example:

While limited use of light colors is fine for coloring regions on a map, light colors do not show up well when coloring points on a graph.

```
Set1 <- brewer.pal(8, "Set1")[-6]
## Check out the palette
plot(1:7, pch=16, cex=2, col=Set1)
```

Opacity, and graphs with many points

Figure 7.2: In Panel A, points are plotted with the 100% opacity, i.e., no transparency. In Panel B, `alpha=0.4`, i.e., 40% opacity. Panel C uses the function `smoothScatter()` to show a smoothed color density representation of the data.

An opacity of 0.4 has the effect that, for an isolated point, 60% of the white background shows through.

```
## Sample from the 15992 rows
dfsamp <- cps1[sample(nrow(cps1), 3000), ]
plot(re78 ~ re75, data=dfsamp, pch=20, cex=0.5,
     col="black", las=0, fg="gray")
mtext(side=3, line=0.5, "A: 100% opacity", adj=0)
plot(re78 ~ re75, data=dfsamp, pch=20, cex=0.5, las=0,
     col=adjustcolor("black", alpha=0.4), fg="gray")
mtext(side=3, line=0.5, "B: 40% opacity", adj=0)
```

```
blueRamp <- colorRampPalette(c("white", blues9))
with(dfsamp, smoothScatter(re75~re74, , fg="gray",
                           las=0, colramp=blueRamp))
mtext(side=3, line=0.5, "C: Color density plot",
      adj=0)
```

With `alpha=0.4`, two overlapping points have a combined opacity of 80%, so that 20% of the white background shows through. Three or more overlapping points appear as completely black.

Compare three plots shown in Figure 7.2. Points overlap to such an extent that Panel A, gives very limited information about the density of points. Panel B, where the color opacity is 40%, gives a better indication of variation in the density of points. Panel C uses the function `smoothScatter()` to provide a color density representation of the scatterplot. This is a more nuanced way to show the density of points.

The plots show a sample of 3000 of the points. Plotting all the points gives an incoveniently large graphics file, while not giving a more informative graph.

7.1.4 The shape of the graph sheet

Aspect ratio, i.e., the ratio of *x*-distance to *y*-distance, has a large say in what is visually obvious. Figures 7.3A and 7.3B show the same data: Features that are at an angle that is close to the horizontal or

Figure 7.3: Figures A and B show the same data, but with widely different aspect ratios.

the vertical are hard to detect visually. Patterns of change or other features should, to be visually obvious, be offset by an angle of at least 20° from both the horizontal and the vertical.
For each of Figures 7.3A and 7.3B the code, after setting the dimensions of the figure page, is:

```
plot((1:30)*0.92, sin((1:30)*0.92),
     xlab="", ylab="")
```

The dimensions of the graphics display can be specified when a graphics window is opened. Once opened, the shape and size of a screen device can be changed by clicking and dragging on one corner.

The R for Windows functions `win.graph()` or `x11()` that set up the Windows screen take the parameters `width` (in inches), `height` (in inches) and `pointsize` (in 1/72 of an inch). The setting of `pointsize` (default =12) determines character heights. It is

the relative sizes that matter for screen display or for incorporation into Word and similar programs.

7.1.5 Multiple plots on the one page

The parameter `mfrow` can be used to configure the graphics sheet so that subsequent plots appear row by row, one after the other in a rectangular layout, on the one page. The following presents four different transformations of data from the dataset `Animals` (*MASS*), in a two by two layout:

For a layout in which columns are filled before moving to a new row, use `mfcol` in place of `mfrow`.

```
## Supplementary figure 9.2
library(MASS)
oldpar <- par(pch=16, pty="s", mfrow=c(2,2))
with(Animals, {       # bracket several R statements
  plot(body, brain)
  plot(sqrt(body), sqrt(brain))
  plot(body^0.1, brain^0.1)
  plot(log(body), log(brain))
})                    # close both sets of brackets
par(oldpar)           # Restore former settings
```

A more flexible alternative is to use the graphics parameter `fig` to mark out the part of the graphics page on which the next graph will appear. The following marks out, successively, a plot region that occupies the upper 62% of the plot region, then the lower 38%.

```
par(fig = c(0, 1, 0.38, 1), mgp=c(3,0.5,0))
          # xleft, xright, ybottom, ytop
## Panel A
par(fig = c(0, 1, 0, 0.38), new=TRUE)
## Plot graph B
par(fig = c(0, 1, 0, 1))     # Restore settings
```

`par(fig = c(0,1,0.38,1))` marks out a plot region that is the total width, starts 38% of the way up, and extends to the top. `par(fig=c(0,1,0,0.38), new=TRUE)` marks out the lower 38% of the page.

The effect of `new=TRUE` is, somewhat counter-intuitively, "assume a new page is already open; do not open a new page".

7.1.6 Plots that show the distribution of data values

We discuss histograms, density plots, boxplots and normal probability plots. Normal probability plots are a specialised form of cumulative density plot.

Density plots are much preferable, for most purposes, to histograms. Both have limitations.

Histograms and density plots

The shapes of histograms depend on the placement of the breaks, as illustrated by Figure 7.4. The following code plots the histograms and superimposes the density plots.

```
par(mgp=c(3,0.5,0))
ftotlen <- subset(possum, sex=="f")[, "totlngth"]
## Left panel: breaks at 72.5, 77.5,..
hist(ftotlen, breaks = 72.5 + (0:5)*5, freq=FALSE,
     xlab="Total length", ylim=c(0,0.11),
     main ="A: Breaks at 72.5, 77.5,...")
```

The argument `freq=FALSE` gives a vertical scale that is the number of points per unit interval, i.e., it is the "density" estimate that is given by the upper bar of each rectangle. This is needed for the superposition of a density curve onto the histogram.

Figure 7.4: The two panels show the same data, but with a different choice of breakpoints.

```
## Now superimpose a density curve, as in Fig. 7.3
lines(density(ftotlen))
##
## Panel B: breaks at 75, 80, ...
hist(ftotlen, breaks = 75 + (0:5)*5, freq=FALSE,
     xlab="Total length", ylim=c(0,0.11),
     main="B: Breaks at 75, 80, ...")
```

The height of each rectangle of a histogram provides a crude density estimate. These estimates change in jumps, at breakpoints that are inevitably chosen somewhat arbitrarily. A smoothly changing density estimate, such as given by the superimposed density curves in the panels of Figure 7.4, makes better sense than an estimate that changes in jumps.

Unless samples are very large, the shape of both histograms and density plots will show large statistical variability. Density plots are helpful for showing the mode, i.e., the density maximum.

Neither histograms nor density plots are effective for checking normality. For that, use a normal probability plot.

The following gives a density plot, separately from the histograms that are shown in 7.4.

```
## Supplementary figure 9.3
with(subset(possum, sex=="f"),
     plot(density(totlngth), type="l"))
```

For use of density plots with data that have sharp lower and/or upper cutoff limits, it may be necessary to specify the x-axis limit or limits.[6] Use the parameters `from` and/or `to` for this purpose. This issue most commonly arises with a lower cutoff at 0.

[6] Thus, a failure time distribution will have a sharp cutoff at zero, which may also be the mode.

Boxplots

Boxplots use a small number of characteristics of a distribution to characterize it. Look up `help(boxplot)` for details. It can be insightful to add a "rug" that shows the individual values, by default along the horizontal axis (`side=1`). Figure 7.5 is an example. Code for the plot is:

```
## Code
with(subset(possum, sex=="f"),
     {boxplot(totlngth, horizontal=TRUE)
      rug(totlngth)} )
```

Figure 7.5: Distribution of lengths of female possums. The vertical bars along the x-axis (together making up a 'rug') show actual data values.

Normal probability plots

The function `qqnorm(y)` gives a normal probability plot of the values of `y`. In such a plot, data from a normal distribution will be scattered about a line. To calibrate the eye to recognise plots that indicate non-normal variation, it helps to compare the plot for the data in hand with several normal probability plots that use `rnorm()` to generate random values. Figure 7.6 is an example.

A point pattern that is not consistent with random deviation from a line indicates a non-normal distribution.

Figure 7.6: Normal probability plots. The top left panel shows the 43 lengths of female possums. Other panels are for independent normal random samples of size 43.

```
## Q-Q plot for the data (top left panel)
ftotlen <- subset(possum, sex == "f")[, "totlngth"]
qqnorm(ftotlen, xlab="",
       ylab=expression(bold("Data")))
## Code for a plot with random normal data
qqnorm(rnorm(43), xlab="", ylab="Simulated")
```

There is one unusually small value. Otherwise the points for the female possum lengths are as close to a straight line as in many of the plots for random normal data.

*7.1.7 *Plotting Text that Includes Technical Symbols*

The functions `expression()` and substitute() can be used to create mathematical expressions, for later evaluation or for printing onto a graph. For example, `expression(x^2)` will print, when supplied to `text()` or `mtext()` or another such function (this includes *lattice* and *ggplot2* functions), as x^2.

Axis labels can be expressions, in *lattice* and *ggplot2* as well as in base graphics. Tick labels can for example be vectors of expressions.

For purposes of adding text that includes mathematical and other technical symbols, the notion of expression is generalized, to allow "expressions" that it does not make sense to try to evaluate. For example, `expression("Temperature (" * degree * "C)")` prints as: Temperature (°C).

Items that are separated by an asterisk (*) are juxtaposed side by side. The initial text is followed by a degree symbol, and then by the final text.

The following indicate some of the possibilities:

- Letters such as `a`, `b`, `c`, `x`, . . . are printed literally.

- `alpha` denotes the Greek letter α, while `Alpha` denotes the upper case symbol. Similarly for other Greek letters.
- `hat(x)` denotes $\hat{x}$.
- `italic(x)`, `bold(x)`, `bolditalic(x)`, and `plain(x)` have the obvious meaning.
- `frac(a,b)` denotes $\frac{a}{b}$.

Figure 7.7 demonstrates the use of an expression to provide *y*-axis labeling. The code is:

```
yl <- expression("Area = " * pi * r^~2)
plot(1:5, pi*(1:5)^2, xlab="Radius (r)", ylab=yl)
```

Figure 7.7: A mathematical expression is included as part of the *y*-axis label..

The tilde (~) in `r^~2` is used to insert a small space.

Use `substitute()` in place of `expression()` when symbols in the expression are to be replaced by values that will be provided at the time of forming the expression.

See `help(plotmath)` for further details of the conventions, and of the symbols that are available. Type `demo(plotmath)` to see a wide range of examples of what is possible.

7.2 *Lattice Graphics*

Lattice Graphics:	
Lattice	Lattice is a flavour of trellis graphics (the S-PLUS flavour was the original)
Lattice vs base	Lattice is more structured, automated and stylized. For standard purposes, much is automatic.
Lattice syntax	Lattice syntax is consistent and tightly regulated For lattice, graphics formulae are mandatory.

The *lattice* package is included in all R binary distributions that are available from a CRAN (Comprehensive R Archive Network) mirror. It implements a *trellis* style of graphics, as in the S-PLUS implementation of the S language. It is built on the *grid* low-level graphics system, described in Part II of Paul Murrell's *R Graphics*

To see some of the possibilities that lattice graphics offers, enter

```
demo(lattice)
```

Lattice (trellis) graphics functions allow

the use of the layout on the page to reflect meaningful aspects of data structure. Groups can be readily distinguished within data, either using different colors and/or symbols and/or line types within panels or using different panels. Multiple columns of data can be plotted, either distinguished within panels or using different panels.

Functions that give styles of graph that are additional to those described here include `contourplot()`, `levelplot()`, `cloud()`, `wireframe()`, `parallel()`, `qqmath()` and `tmd()`.

Functions in the *latticeExtra* package further extend what is available.

In the discusssion that follows, there will be use of the layering abilities provided by functions in *latticeExtra*. Note that loading *latticeExtra* will at the same time load *lattice*, which *latticeExtra* has as a dependency.

These abilities, due to Felix Andrews, make it possible to build up lattice graphics objects layer by layer.

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

7.2.1 Lattice graphics – basic ideas

Figure 7.8 was obtained using the lattice function `xyplot()`. In this simple case, the syntax closely matches that of the base graphics function `plot()`. Code is:

```
## On the command line: Create and print object
xyplot(Brainwt ~ Bodywt, data=primates)
```

Figure 7.8: Use of lattice function `xyplot()` to give a graph.

Lattice graphics functions return graphics objects

Note an important difference between lattice and base graphics. Lattice graphics functions do not print graphs.[7] Instead they return trellis graphics objects. The graph appears when the object is printed (use `print()` or `plot()`). Sending the output from a *lattice* graphics function to the command line invokes `print()` and the graph is plotted, as was done for Figure 7.8.

[7] This applies also to *ggplot2*.

A `Brainwt` versus `Bodywt` scatterplot for the `primates` data, such as was given earlier, might alternatively have been obtained using the function the function `xyplot()` from the *lattice* package.

```
## Save the result as a trellis graphics object
# [For plot(), this is not possible.]
## Create trellis object
gph <- xyplot(Brainwt ~ Bodywt, data=primates)
## Print graph; a graphics device must now be open
print(gph)
```

The object `gph` need not be printed at this point. It can be kept for printing at some later time. Or it can be updated, using the function `update()`, and then printed, thus:

```
gph <- xyplot(Brainwt ~ Bodywt, data=primates)
gph2 <- update(gph, xlab="Body wt (kg)",
               ylab="Brain wt (g)")
print(gph2)  # Or it is enough to type 'gph2'
```

Inside a function or in a file that is sourced, `print()` must ordinarily be used to give a graph, thus:

The graph will however be printed if `xyplot(...)` is the final statement in a function that returns its result to the command line.

```
print(xyplot(ACT ~ year, data=austpop))
```

Addition of points, lines, text, ...

For adding[8] to a plot that has been created using a *lattice* function, use `panel.points()`, `panel.text()`, and other such functions, as will be described in Subsection 7.2.8.

[8] Do not try to use `points()` and other such base graphics functions with lattice graphs.

Mechanisms for the control of a wide variety of stylistic features are best discussed in the context of multi-panel graphs, which we now consider.

7.2.2 *Panels of scatterplots*

Graphics functions in the *lattice* package, are designed to allow row by column layouts of panels. Different panels are for different subsets of the data. Additionally, points can be distinguished, within panels, according to some further grouping within the data.

The `ais` dataset (*DAAG*) has data from elite Australian athletes who trained at the Australian Institute of Sport. These included height, weight, and other morphometric measurements, as well as several types of blood cell counts. A breakdown of the total of 202 athletes by sex and sport gives:

See the help page for `ais` for details.

```
with(ais, table(sex,sport))
```

```
   sport
sex B_Ball Field Gym Netball Row Swim T_400m T_Sprnt Tennis W_Polo
  f     13     7   4      23  22    9     11       4      7      0
  m     12    12   0       0  15   13     18      11      4     17
```

Figure 7.9 demonstrates the use of `xyplot()`, for the rower and swimmer subset os the `ais` dataset. The two panels distinguish the two sports, while different plotting symbols (on a color device, different colors will be used) distinguish females from males.

Figure 7.9: Height (`ht`) versus Weight (`wt`), for rowers (`Row`) and swimmers (`Swim`). Different plotting symbols are used to distinguish males from females.

Suitable code is:

```
xyplot(ht ~ wt | sport, groups=sex, data=ais,
       par.settings=simpleTheme(pch=c(4,1)),
       scales=list(tck=0.5),
       auto.key=list(space="right"),
       subset=sport%in%c("Row","Swim"))
```

Use `auto.key=list(columns=2)` to generate a simple key, with items side by side in two columns rather than stacked in a single column as is the default `columns=1`.

In the graphics formula `ht ~ wt | sport`, the vertical bar indicates that what follows, in this case `sport`, is a conditioning variable or factor. The graphical information is broken down by levels of the factor `sport`. The parameter `aspect` controls the ratio of dimensions in the y and x directions.

Subsection 7.2.3, which now follows, explains the use of the argument `par.settings`, and its call to `simpleTheme()`.

7.2.3 Setting stylistic features

The function `simpleTheme()` creates a "theme", i.e., a list of settings, that can be supplied via the argument `par.settings` in the graphics function call. Use of the argument `par.settings` to a lattice function makes the settings locally, for the specific graphics object that results.

The function `simpleTheme()` accepts arguments `col`, `alpha`, `cex`, `pch`, `lty`, `lwd`, `font`, `fill`, `border`, plus `col.points`, `col.line`, `alpha.points` and `alpha.line`. These allow separate control (of color and of opacity) for points and lines.

Settings that are not available using `simpleTheme()` can if required be added to the theme object that `simpleTheme()` returns. See Subsection 7.2.6 has details.

The function `trellis.device()` opens a new graphics device, with settings that have in mind the use of *lattice* functions. The function `trellis.par.set()` sets or changes stylistic features for the current device. Both these functions accept an argument `theme`.[9] Settings made by `trellis.device()` or `trellis.par.set()` will be over-written by any local settings that are stored as part of the graphics object.

[9] Simple variations on the default theme can be created by a call to `simpleTheme()`.

7.2.4 Groups within data, and/or columns in parallel

Table 7.1 shows selected rows from the data set `grog` (*DAAG* package). Each of three liquor products (drinks) has its own column. Rows are indexed by the factor `Country`.

	Beer	Wine	Spirit	Country	Year
1	5.24	2.86	1.81	Australia	1998
2	5.15	2.87	1.77	Australia	1999
....					
9	4.57	3.11	2.15	Australia	2006
10	4.50	2.59	1.77	NewZealand	1998
11	4.28	2.65	1.64	NewZealand	1999
....					
18	3.96	3.09	2.20	NewZealand	2006

Table 7.1: Apparent annual alcohol consumptiom values, obtained by dividing estimates of total available alcohol by number of persons aged 15 or more. These are based on Australian Bureau of Statistics and Statistics New Zealand figures.

Figure 7.10 is one of several possible displays that might be used to summarize the information in Table 7.1. It has been created by updating the following simplified code:

```
## Simple version of plot
grogplot <- xyplot(
              Beer+Spirit+Wine ~ Year | Country,
              data=grog, outer=FALSE,
              auto.key=list(space="right"))
```

Observe that:

- Use of `Beer+Spirit+Wine` gives plots for each of `Beer`, `Spirit` and `Wine`. The effect of `outer=FALSE` is that these appear in the same panel.

Figure 7.10: Australian and New Zealand apparent per person annual consumption (in liters) of the pure alcohol content of liquor products, for 1998 to 2006.

- Conditioning by country (| `Country`) gives separate panels for separate countries.

The following updates the object to give Figure 7.10:

```
## Update trellis object, then print
ylab <- "Amount consumed (per person)"
parset <- simpleTheme(pch=c(1,3,4))
finalplot <- update(grogplot, ylim=c(0,5.5),
                    xlab="", ylab=ylab,
                    par.settings=parset)
print(finalplot)
```

Notice the use of the function `simpleTheme()` to set up a "theme" that was used to control point and line settings.

Figure 7.10 used different symbols, in the one panel, to distinguish drinks, with different countries in different panels. For separate panels for the three liquor products (different levels of `Country` can then use the same panel), specify `outer=TRUE`:

```
xyplot(Beer+Spirit+Wine ~ Year,
       groups=Country, outer=TRUE,
       data=grog, auto.key=list(columns=2) )
```

Where plots are superposed in the one panel and, e.g., regression lines or smooth curves are fitted, this is done separately for each different set of points. Different colors, and/or by different symbols and/or line styles, can be used to make the necessary distinctions.

Here is a summary:

Break data down a/c to levels of the factor `Country`:

Overplot (a single panel):	Separate panels:
`Beer ~ Year, groups=Country`	`Beer ~ Year	Country`

Plot columns in parallel, as in `Beer+Wine+Spirit ~ Year`:

Overplot (a single panel):	Separate panels:
`outer=FALSE`	`outer=TRUE`

7.2.5 *Keys – `auto.key`, `key` and `legend`*

The argument `auto.key=TRUE` gives a basic key. If not otherwise

The argument `auto.key` sets up a call `key=simpleKey()`. If necessary, use `legend=NULL` when updating, to remove an existing key and allow the addition of a new key.

specified, colors, plotting symbols, and line type use the current settings for the device. The argument `text` has `levels(groups)` as its default. that identifies colors, plotting symbols and names for the groups. For greater flexibility, `auto.key` can be a list. Settings that are often useful are:

- `points`, `lines`: in each case set to `TRUE` or `FALSE`.
- `columns`: number of columns of keys.
- `x` and `y`, which are coordinates for the whole display area. Use with `corner` set to one of `c(0,0)`, `c(1,0)`, `c(1,1)` and `c(0,1)`.
- `space`: one of `"top"`, `"bottom"`, `"left"`, `"right"`.

`c(0,0)` is the bottom left corner of the legend, etc.

*Use of `textGrob()` to add legends

The function `textGrob()` (*grid*) creates a text object which can then be supplied to the lattice function. This mechanism for supplying legends can be used when multiple legends are required.

The following code adds an initial legend, as in Figure 7.11:

```
plotnam <- "Stripplot of cuckoo data"
stripplot(species ~ length, xlab="", data=cuckoos,
  legend=list(top=list(fun=grid::textGrob,
                       args=list(label=plotnam,
                                 x=0))))
# x=0 is equivalent to x=unit(0,"npc")
# npc units are on a scale from 0 to 1
```

Figure 7.11: The argument `legend` has been used to add text, supplied as a 'grob'.. Here, it would be easier to use of the argument `main`.

Additional legends are supplied by adding further list elements, for example a list element `bottom` as well as a list element `top`.

7.2.6 Lattice settings – further notes

In general, use `themes` to make point, line and fill color settings. Use the `scales` argument, in the call to the lattice function, for axes, tick marks, and tick labels.

For changes that go beyond what `simpleTheme()` allows, first identify the names under which settings are stored. Type:

```
> names(trellis.par.get())
[1] "fontsize"          "background"        "clip"
. . .
[28] "par.sub.text"
```

For a visual display that shows default settings for points, lines and fill color, enter:

```
trellis.device(color=FALSE)
show.settings()
trellis.device(color=TRUE)
show.settings()
```

The following sets the fontsize, separately for `text` and `points`:

```
trellis.par.set(fontsize = list(text = 7,
                                points = 4))
```

Parameters that affect axes, tick marks, and axis labels

These are manipulated by use of the `scales` argument to the lattice function. The code for Figure 7.12 provides an example.

Figure 7.12: Jobs growth in Canadian provinces, between January 1995 and December 1996.

The following gives a basic graph, which will then be updated:

```
## 1. Create a basic version of the graphics object
jobsB.xyplot <-
  xyplot(Ontario+Quebec+BC+Alberta+Prairies+Atlantic ~ Date,
         data=jobs, type="b", layout=c(3,2), outer=TRUE,
         ylab="Number of jobs",
         scales=list(y=list(relation="sliced", log=TRUE)))
```

Now make several enhancements:

- Change the y-axis labels to show number of jobs, with log(number) in parentheses underneath.
- Use dates of the form `Jan95` to label the x-axis.[10]
- Reduce tick marks in length (`tck=0.6`, i.e., 60% of the default).
- The argument `between=list(x=0.5, y=0.5)` adds horizontal and vertical space between the panels.[11]

[10] Refer back to Subsection 4.3.9.

[11] This avoids overlap of tick labels.

```
## 2. Code for the enhancements to jobsB.xyplot
ylabpos <- exp(pretty(log(unlist(jobs[,-7])), 100))
ylabels <- paste0(round(ylabpos),"\n(", log(ylabpos), ")")
## Create a date object 'startofmonth'; use instead of 'Date'
atdates <- seq(from=95, by=0.5, length=5)
datelabs <- format(seq(from=as.Date("1Jan1995", format="%d%b%Y"),
                       by="6 month", length=5), "%b%y")
update(jobsB.xyplot, xlab="", between=list(x=0.5, y=0.5),
       scales=list(x=list(at=atdates, labels=datelabs),
                   y=list(at=ylabpos, labels=ylabels), tck=0.6) )
```

7.2.7 *Lattice plots that show distributions*

Stripplots, dotplots and boxplots

Because the syntax for `stripplot()` and `boxplot()` are very similar, we demonstrate suitable code side by side. Figure 7.13 summarizes cuckoo egg length data, from the dataset `cuckoos` from *DAAG*:

Differences between `dotplot()` and `stripplot()` are mainly cosmetic.

Figure 7.13: A stripplot and a dotplot appear side by side.

```
stripplot(species ~ length, data=cuckoos,
          xlab="Cuckoo egg length (mm)")
bwplot(species ~ length, data=cuckoos,
       xlab="Cuckoo egg length (mm)")
```

For slightly improved labeling, precede the code with:

```
levels(cuckoos$species) <-
 sub(".", " ",
  levels(cuckoos$species),
  fixed=TRUE)
```

The `aspect` argument determines the ratio of distance in the y-direction to distance in the x-direction.

Lattice style density plots

Here is a density plot (Figure 7.14), for data from the `possum` data set (*DAAG*), that compares `sex`es and `Vic/other` populations.

Figure 7.14: Lattice style density plot comparing possum earconch measurements, separately for males and females, between Victorian and other populations. Observe that the scatter of data values is shown along the horizontal axis.

```
## Code
colset <- c("gray","black")
densityplot(~ earconch | sex, groups=Pop,
            data=possum,
            par.settings=simpleTheme(col=colset),
            auto.key=list(space="right"))
```

The functions `densityplot()` and `histogram()` do not allow a name on the left of the ~ symbol. The function `histogram()`,

which is otherwise similar to `densityplot()`, does not accept a `groups` argument.

7.2.8 *Panel functions*

Each lattice function that creates a graphics object has its own panel function. Creation of one's own panel function allows detailed control of panel contents. Or `update()` can be used to modify the panel or panels.

A user panel function will typically include, or consist of, calls to several of the variety of panel functions that are provided in *lattice*. The function `xyplot()` has the panel function `panel.xyplot()`.[12] The following are equivalent:

```
xyplot(species ~ length, xlab="", data=cuckoos)
xyplot(species ~ length, xlab="", data=cuckoos,
       panel=panel.xyplot)
```

A user function, used in place of `panel.xyplot()`, might for example call `panel.superpose()`, followed or preceded by other available panel functions.

Available panel functions include:

- `panel.points()`, `panel.lines()`, `panel.text()`, `panel.rect()`, `panel.arrows()`, `panel.segments()`, `panel.polygon()`
  (all documented on the same help page as `panel.points()`);
- `panel.abline()`, `panel.curve()`, `panel.rug()`, `panel.fill()`, `panel.average()`, `panel.mathdensity()`, `panel.refline()`, `panel.loess()`, `panel.lmline()`
  (all documented on the same help page as `panel.abline()`).

The following graphics object `gph` will be used as a starting point, in the discussion that now follows:

```
gph <- xyplot(Brainwt ~ Bodywt,  data=primates,
              xlim=c(0,300))
```

Now create a panel function that both plots the points and adds labels. The graphics object can then be updated, as in the code that now follows, to use this panel function:

```
my.panel <- function(x,y){
  panel.xyplot(x,y)
  panel.text(x,y, labels=rownames(primates),
             cex=0.65, pos=4)
}
update(gph, panel=my.panel,
       scales=list(tck=0.6))
```

Note that we could have supplied a panel function that plots the points and adds the labels in the initial function call, thus:

Subsection 7.2.9 will describe a radical extension of this basic scheme. Further layers, created using `layer()` and allied functions in the *latticeExtra* package can be "added" (the operator is "+") to a trellis graphics object.

[12] When a `groups` argument is supplied, `panel.xyplot()` calls the function `panel.superpose()`.

Note that an alternative to `panel.points()` is `lpoints()`. Similarly for the other functions.

Figure 7.15: Addition of labels, as here, can be done by updating a graph that has the points, by use of a panel function that both plots points and and adds labels, or by adding a new layer.

```
xyplot(Brainwt ~ Bodywt, data=primates,
       xlim=c(0,300), panel=my.panel)
```

A further possibility is to add a new layer that has the labels, as in Subsection 7.2.9 which now follows. However the plot is obtained, Figure 7.15 shows the result.

7.2.9 *The addition of new layers*

The layering mechanism greatly extends the range of possibilities. The code that follows gives a simple and somewhat trivial example of its use – an alternative the use of a panel function for adding labeling to Figure 7.15.

Note again the graphics object gph, created above:

```
gph <- xyplot(Brainwt ~ Bodywt,  data=primates,
              xlim=c(0,300))
```

The following uses the function `layer()`, from the *latticeExtra* package, to create a second layer that has the labels. The layer that is thus created is added to the graphics object gph:

```
gph + latticeExtra::layer(panel.text(x,y,
                       labels=rownames(primates),
                       pos=4))
```

Note also `layer_()`, which reverses the order of the layers, equivalent to using `layer()` with `under=TRUE`.

The function `layer()` allows as arguments, passed via the `...` argument, any sequence of statements that might appear in a panel function. Such statements can refer to panel function arguments, including 'x', 'y' and 'subscripts'. Additionally, named column objects can be passed through an optional `data` argument.

The function `as.layer()` creates a layer from a trellis graphics object. This can then be "added" in the usual way.

Other convenience functions are `glayer()` and `glayer_()`. These are equivalent, respectively, to calling `layer()` and `layer_()` with `superpose=TRUE`. The layer is drawn once for each level of any group in the plot.

7.2.10 *Interaction with plots – latticist and playwith*

Here will be noted the abilities in the *latticist* and *playwith* packages, for interaction with lattice plots.[13]

For using *playwith*, the GTK+ toolkit must be installed. For details, go to the website `http://playwith.googlecode.com/`.

[13] More limited abilities are available to interact with other plots.

`latticist()`: When called with a data frame as argument, the function `latticist()` (in the *latticist* package) opens a window that has graphical summary information on the columns of the data frame. Additionally, it opens a GUI interface to the *lattice* and *vcd* packages, allowing rapid creation of plots that may be useful in their own right, or may be a first step in creating more carefully crafted plots. Various annotation features are available from the GUI.

`playwith()`: The `playwith()` function (from the *playwith* package) was used for Figure 7.16. The menu that appears to the left of the graph can be used to initiate single click identification, to add

annotation or arrows, or to mark out a rectangle on the graph for zooming in or out. If labels are not specified, row names are used.

Figure 7.16: This playwith GUI window was generated by wrapping the call to `xyplot()` in the function `playwith()`, then clicking on Identify. Click near to a point to see its label. A second click adds the label to the graph. A color version appears as C.2.

```
## Code that initiates interactive display
library(playwith)
playwith(xyplot(age ~ distance, data=hotspots),
         labels=hotspots$name)
```

Alternative code for Figure 7.16:

```
gph <-
    xyplot(age ~ distance,
           data=hotspots)
library(playwith)
playwith(update(gph),
    labels=hotspots$name)
```

Note that `playwith()` can be used, also, for more limited interaction with plots created using base graphics, or using *ggplot2*.

7.3 *ggplot2 – A Grammar of Graphics*

The *ggplot2* syntax is consistent, but less stylized than the *lattice* syntax. As with *lattice*, *ggplot2* functions return a graphics object. The graphics objects that *ggplot2* functions return can be saved for later use, or updated, or printed directly on to the graphics page. Each different type of *ggplot2* graphic display – scatterplot, histogram, density plot, histogram, etc. – is a different plot `geom`, or "geometry". These can be overlaid.

The following loads the *ggplot2* package:

The *ggplot2* syntax is a variant of Wilkinson's "Grammar of Graphics" (Springer, 2nd edn, 2005).

```
library(ggplot2)
```

7.3.1 Examples that demonstrate ggplot2 abilities

Brain weight versus body weight

Figure 7.17 repeats Figure 3.1B from Chapter 3, now using *ggplot2* abilities:

Figure 7.17: Plot of brain weight (gm) versus body weight (kg). Log scales have been used on both axes. The function `coord_equal()`, used with a logarithmic scale, ensures that a given distance (e.g., 1cm) on either axis represents the same relative change.

Code is:

```
library(MASS)
quickplot(body, brain, data=mammals, log="xy") +
  coord_fixed()
```

Notice that `quickplot()` has been used to create an initial plot, with `coord_equal()` then used ("added") to specify that a given distance will represent the same change on both axes. As a logarithmic scale is used, this implies that the same relative change will be given by the same distance. Here, observe that grid lines in both directions are the same distance apart, with the distance representing a change by a factor of 100,

In subsequent discussion, the abbreviated name `qplot` will be used in place of `quickplot`.

The following adds a regression line:

```
quickplot(body, brain, data=mammals, log="xy") +
  coord_fixed() +
  geom_smooth(method=lm)
```

This "addition" of new functions that add to or modify the initial graph can in principle proceed without limit.

As the name hints, the function `qplot()` (or `quickplot()`) shortcuts the more detailed *ggplot2* syntax. The call

```
quickplot((body, brain, data=mammals, log="xy")
```

when written out using the detailed syntactic steps, becomes:

```
ggplot(mammals, aes(body, brain)) +
  geom_point() +
  scale_x_continuous(trans="log") +
  scale_y_continuous(trans="log")
```

The successive "+" operators combine output from function calls to create a single graphics object. In the detailed syntactic steps, the call to `geom_point()` plots the points, while the subsequent calls change the *x*- and *y*-axis scales to logarithmic scales.

In the call to `ggplot()`, the `data` argument is the only mandatory argument. It can be repeated in the call(s) to one or more of the later `geom` functions. This allows different `geom`s, if required, to take their data from different data frames.

Changes to `color` or `size` or `shape` settings can be made separately for each different `geom`. Changing `geom_point()` to `geom_point(size=2.5)` affects only the points.

Note that `cex` and `size` are synonyms, as are `color` and `colour`. Also `type` is a synonym for `geom`.

Aesthetic mappings vs settings

Distinguish between *settings* and *aesthetic mappings*:

	Use of `quickplot()`	Plots based on `ggplot()`
Settings	`size=I(3)` or `cex=3`	`size=3`
Aesthetic mappings	`size=3` or `size=sport`	`aes(size=3)` or `aes(size=sport)`

The function `aes()` maps variables in the data to visual properties ("aesthetics") of `geom`s. In `aes(body, brain` above, the mappings are to the *x*– and *y*–axes of the plot. Other possible mappings are to `color` (use color to distinguish groups within the data), `shape`[14] (distinguish by shape), `size` and `fill`.

[14] Where base graphics has `pch`, *ggplot2* has `shape`.

Use of the argument `size=3` in a call to `quickplot()` does change the point size, but it adds an extraneous key. The same happens if the argument `mapping=aes(size=3)` is supplied to `ggplot()` or to `geom_point()` or to another such function.

A further possibility is to use `quickplot()` (or `qplot()`) to create an initial graphics object, then adding to this object. The following code uses this approach to create Figure 7.20:

```
qplot(Year, mdbRain, data=bomregions2015,
      geom="point",
      xlab="", ylab="Av. rainfall, M-D basin") +
  geom_smooth(span=0.5, se=TRUE)
```

In all cases, a `ggplot` object is created. This can be `print`ed immediately, or it can be saved as a named object. The graph is created using the `print` method for a `ggplot` object.

7.3.2 *An overview of* ggplot2 *technicalities*

Available geometries and settings

Table 7.2 has details of a number of the geometries that are available for `ggplot` objects. Table 7.3 lists some of the settings, in addition to those already noted, that are available:

Table 7.2: Available geoms.

quickplot()	ggplot()	Available arguments to the geom function
geom=		(data, mapping, color, fill, alpha, plus ...)
"point"	+ geom_point()	size, shape, etc.
"line"	+ geom_line()	size, linetype
"path"	+ geom_path()[1]	size, linetype
"smooth"	+ geom_smooth()	linetype, weight, se (TRUE or FALSE).
"histogram"	+ geom_histogram()	linetype, weight
"density"	+ geom_density()	weight, linetype, size
"density2d"	+ geom_density2d()	weight, linetype, size

[1] Use geom_path() to connect observations, in the original order.

Table 7.3: Control of ggplot2 graphics features. Functions such as xlab() and scale_x_continuous() that relate to the *x*-axis all have counterparts with y in place of x. .

	Argument to qplot()	ggplot() or qplot()[1]
Title	main="mytitle"	+ labs(title="mytitle")
Axes	see help(qplot)	+ scale_x_continuous()[2] [or scale_x_discrete() or scale_x_date()]
Axis labels	e.g., xlab="myxlab"	+ xlab("myxlab")[3]
log axes	log="x", (or "y", or "xy")	+ scale_x_log10()[4]
Facets[5]	facets=sex ~ sport	+ facet_grid(sex ~ sport)
Aspect ratio	e.g., asp=1	+ coord_equal()[6]
Theme	—	
Graph title	e.g., main="maintitle"	+ ggtitle("mytitle")

[1] Recall that quickplot() (or qplot()) returns a ggplot object. Functions such as xlab() or scale_x_continuous() can be used, just as for any other ggplot2 object, to update objects returned by quickplot().
[2] Available arguments include limits, breaks (locations for the ticks), labels (labels for the breaks), and trans (e.g., trans="log").
[3] This is an alternative to using name (e.g., name="myxlab") as an argument to scale_x_continuous() or scale_x_discrete().
[4] This is an alternative to using trans="log10" as an argument to scale_x_continuous() or scale_x_discrete(). Note also trans="log" and trans="log2").
[5] Facets give Lattice style *conditioning*.
[6] By default (ratio=1), a given distance, e.g., 1cm, represents the same range along both *x*− and *y*−axes.
[7] Themes control such graphical attributes as background color, gridlines, and size and color of fonts. See help(ggtheme) for details of other available themes.

Example — Measurements on Australian athletes

Figure 7.18 plots height against weight, by sex, for the `ais` data. Additionally, boxplots show the distributions of heights, and there are two-dimensional density contours estimates. The graph is a tad crowded.

Figure 7.18: Height versus weight, by sex, for Australian athletes in the `ais` data set. Boxplots that show the distributions of heights, and two-dimensional density contours have been added.

The following gives a simplified version of the plot:

```
## Overlay with boxplots and density contours
quickplot(wt, ht, data=ais,
          geom=c("boxplot", "point", "density2d"),
          facets = . ~ sex)
```

To set axis labels, show the boxplot outline in gray, show contour lines in gray (the default is blue), and make various other changes as in Figure 7.18, specify:

```
quickplot(wt, ht, xlab="Weight (kg)",
          ylab="Height (cm)", data=ais,
          facets = . ~ sex) +
  geom_boxplot(aes(group=sex),
               outlier.size=1.75,
               outlier.colour="gray",
               color="gray") +
  geom_point(shape=2, size=1) +
  geom_density2d(color="gray")
```

The `facets` argument has the form `row.var ~ col.var`, where `row.var` indexes rows of panels, `col.var` indexes columns, and "`.`" serves as a placeholder when there is one row or one column only.

Code for the next plot will work with a subset of the `ais` data, limiting attention to rowers and swimmers:

```
## Extract from ais data for rowers and swimmers
aisRS <- subset(ais, sport %in% c("Row","Swim"))
aisRS$sport <- droplevels(aisRS$sport)
```

Here are alternative code fragments that can be used to create Figure 7.19, one using `quickplot()` and the other using successive

Figure 7.19: Use `color` for distinguishing `sexes`, `shapes` for `sports`.

calls to `ggplot()` and to `geom_point()`:

1: Use `quickplot()`:

```
quickplot(wt, ht,
          data=aisRS,
          geom="point",
          size=I(2),
          colour=sex,
          shape=sport)
```

2: `ggplot() + geom_point()`:

```
ggplot(aisRS) +
  geom_point(aes(wt, ht,
                 color=sex,
                 shape=sport),
             size=2)
```

Multiple aesthetics can be used for the one distinction, here between `sex`es:

```
## Distinguish sex by color & shape
## Different sports have different panels
quickplot(wt, ht, data=aisRS, geom="point",
          size=I(2.5), color=sex, shape=sex,
          facets = . ~ sport)
```

Here are further possibilities:

```
## Identify sex by color, sport by shape (1 panel)
quickplot(wt, ht, data=aisRS, geom="point",
           color=sex, shape=sport, size=I(2.5))
## Identify sex by color, sport by size (1 panel)
quickplot(wt, ht, data=aisRS, geom="point",
          color=sex, size=sport)
```

Australian rain data

Figure 7.20 plots annual rainfall for Australia's Murray-Darling basin region. The following code uses the function `quickplot()`:

```
library(DAAG)
library(ggplot2)
## Default loess smooth, with SE bands added.
quickplot(Year, mdbRain, data=bomregions2015,
          geom=c("point","smooth"), xlab="",
          ylab="Av. rainfall, M-D basin")
```

Arguments `size` (e.g., `size=I(2.5)`), `color` (e.g., `color=I("red")`, etc, can be supplied, affecting both points and the added smooth curve. NB: `size=I(2.5)`, not `size=2.5`.

Code that shows the detailed syntactic steps is:

```
ggplot(bomregions2015, aes(x=Year, y=mdbRain)) +
  geom_point() +                        # Scatterplot
  geom_smooth(span=0.5, se=TRUE) +      # Add smooth
  xlab("") +                 # Blank out x-axis label
  ylab("Av. rainfall, M-D basin")
## NB: aes() has supplied x- and y-axis variables
```

As before, the successive "+" operators combine output from function calls to create a single graphics object.

Try also the following. This requires both the *quantreg* package and the *splines* package:

```
library(quantreg)
library(splines)
## Supplementary figure 4
quickplot(Year, mdbRain, data=bomregions2015) +
```

The normal spline basis `ns(x,5)` is supplied to the function that estimates the quantile curves, so that 5 d.f. spline curves are fitted at the 20%, 50% and 80% quantiles.

Figure 7.20: Annual rainfall, from 1901 to 2012, for the Murray-Darling basin region of Australia. The curve is fitted using the default loess smoother. The pointwise standard error bands assume that errors about the curve are independent; this is unlikely to be strictly true. To suppress these bands, specify `se=FALSE`.

```
        geom_quantile(formula = y ~ ns(x,5),
        quantiles=c(0.2,0.5,0.8) )
```

Florence Nightingale's Wedge Plot

Florence Nightingale's Crimean War experience prepared her for later major work in the reform of army and civilian hospitals and public health administration, and to wider social reform. Her influence extended to the army and civilian administration in India.

Figure C.3 in Appendix C is a "wedge" plot, showing the mortality of British troops according to cause in the Crimean war over 1854–1856. It shows the abilities of the *ggplot2* package to spectacular effect. The plot is obtained by using polar coordinates for plotting a stacked bar chart! Use of areas to convey numerical information is however not ideal, especially when as here the areas overlap.

7.4 Static graphics – additional notes

7.4.1 Multiple graphs on a single graphics page

For *base* graphics, refer back to Subsection 7.1.5. The following demonstrates use of the `fig` argument to `par()` to select a part of the display region for plotting:

```
par(fig = c(0, 1, 0.38, 1))
          # xleft, xright, ylow, yhigh
## Plot graph A
par(fig = c(0, 1, 0, 0.38), new=TRUE)
## Plot graph B
par(fig = c(0, 1, 0, 1))      # Resets to default
```

For lattice graphs, the location of the graph can be determined by the argument `position`, when `print()` is called. The following demonstrates its use:

```
cuckoos.strip <- stripplot(species ~ length, xlab="", data=cuckoos)
print(cuckoos.strip, position=c(0,0.5,1,1))
                   # xleft, ybottom, xright, ytop
cuckoos.bw <- bwplot(species ~ length, xlab="", data=cuckoos)
print(cuckoos.bw, position=c(0,0,1,0.5), newpage=FALSE)
```

Note the use of `newpage=FALSE` for the second plot.

Base and trellis plots on the same graphics page

The following uses the base graphics command `mtext()` to label a lattice plot:

```
plot(0:1, 0:1, type="n", bty="n", axes=FALSE,
     xlab="", ylab="")
lab <- "Lattice bwplot (i.e., boxplot)"
mtext(side=3, line=3, lab)
cuckoos.bw <- bwplot(species~length, data=cuckoos)
print(cuckoos.bw, newpage=FALSE)
```

Inclusion of graphs in Microsoft Word

Graphs may not import well from the clipboard into Word on the Macintosh under OS X. On Windows systems, an effective option is to use `win.metafile()` to write graphics output to a Windows metafile format that should import without problem into a Word or Power Point document.

7.5 Dynamic Graphics – rgl

This section will describe a range of abilities that create displays which the user can then manipulate dynamically. Note in particular the *rgl* and *googleVis* packages, designed for interactive exploration of dynamic changes in relationships with time. The *googleVis* package reproduces most of the abilities of Google's Public Data Explorer, which can be accessed at `http://www.google.com/publicdata/home`

Figure 7.21: Snapshot of a 3-D dynamic display, for the `nihills` data from the *DAAG* package. The display has been dragged to a position where points very nearly fall on a line.

The *rgl* package provides three-dimensional dynamic graphics. Use of the functions `scatter3d()` and `identify3d()` from the *car*

package may be more convenient for novices than the *rgl* functions that they call. Figure 7.21 shows a snapshot of the plot obtained for the `nihills` data from the *DAAG* package.
The following loads the needed packages:

```
## The car and rgl packages must be installed
library(rgl, quietly=TRUE)
library(car, quietly=TRUE)
rgl::setupKnitr()
knit_hooks$set(rgl=hook_rgl)
```

Code for the figure is:

```
library(DAAG, quietly=TRUE)
open3d()            # Precedes the call to par3d()
par3d(cex=0.75)     # Optional
                    # Other params: see help(par3d)
with(nihills, scatter3d(x=log(dist), y=log(climb),
                        z=log(time),
                        grid=FALSE,
                        surface=FALSE,
                        point.col="black",
                        axis.scales=FALSE))
## NB: Use middle or right mouse button to drag a
## rectangle around a point that is to be labeled.
```

Use `rgl.snapshot()` to save the current plot into a file.

Use the function `identify3d()` to start the identification of points. Following the call to `identify3d()`, use the middle (or maybe right) mouse button to drag a rectangle around any point that is to be labeled. To cease identifying points, make a middle (or right) click on an empty region of the plot. The labels may appear only at this point. Here, for identification of points shown in Figure 7.21, is suitable code:

```
with(nihills, identify3d(x=log(dist), y=log(climb),
                         z=log(time),
                         labels=row.names(nihills),
                         col="gray"))
```

Such a plot can be helpful in identifying high leverage points, e.g., in the regression of `log(time)` on `log(dist)` and `log(climb)`. The plot needs to be rotated to give a view in which the leverage is apparent.

The `rggobi` package

The *rggobi* package offers a wider range of features, via an interface to the GGobi system. For installation details go to `http://www.ggobi.org/`. Windows users can use the following to install all the required files from an R session that has access to a live internet connection:

```
source("http://www.ggobi.org/downloads/install.r")
```

7.5.1 *The* googleVis *Package*

This provides an interface to the abilities of Google's Public Data Explorer. While these abilities can be accessed from the web page noted below, there are obvious advantages in setting up the display from one's own computer.[15]

[15] An internet connection is needed to access Google's API (Application Program Interface) when the chart is displayed.

Google's Public Data Explorer

Upon accessing Google's web page `http://www.google.com/publicdata/home`, the display will cycle through examples of the use of Motion Charts, and other related charts. Click on Explore Data to go to the interactive version of the relevant display.[16] These charts, which are interesting in themselves, show the abilities that *googleVis* is designed to emulate. See the annotations in Figure C.1 below for details that should be enough to get started.

A slider below the graph can be moved to show how the relationships that are plotted change over the available timespan, commonly 1960 through to 2010 (but note that not all data will be available for all years). Click on the solid right-pointing triangle on the left of the slider scale to see the graph changing dynamically in moving from the currently shown year through to 2010.

[16] Various controls are placed in the margins of the graph. Move the pointer over one or other control feature to get information on its purpose, or over a point to display information about that point. For changing the *x*- and/or *y*- variables, or for changing the variable that determines point size, click on the relevant downward pointing selector arrow. Scales, separately for the two axes, can be either linear or logarithmic.

Use of googleVis *to create motion charts*

The details given here should be supplemented with examination of the vignette that accompanies the *googleVis* package. Note especially Figure 1 on page 5 of the vignette.

To display the vignette, type:
`vignette("googleVis")`

Creation of a motion chart, once the data is in place, is remarkably straightforward. The starting point is a data frame in that has a column (e.g. Countries) that can be used as an `id` variable, a column (e.g. Year) that has a `timevar` variable, and columns that can be used to supply *x*- and *y*- variables. Optionally, columns may be identified for use as a `colorvar` and/or a `sizevar`.

The dataset `grog` from the *DAAG* package has a suitable structure. One can create a motion chart thus:

```
library(googleVis)
M <- gvisMotionChart(grog, id="Country", timevar="Year")
## This next line requires a live internet connection,
## and Adobe Flash must be installed.
plot(M)
```

If the browser window that appears displays 'Flash' in gray in the middle of the screen, click there to proceed. A browser window with a gray display region should appear.

For the `grog` dataset, the Motion Chart does a less satisfactory job than Figure 7.10 in Section 7.2.4. Motion charts come into their own for the examination of steady changes over time in a bivariate relationship, with different patterns of relationship for different

subgroups of the data.

A plot that allows the display of various World Bank development indicators can be obtained by typing:

```
demo(WorldBank)
```

The code used to download the data and display the motion chart will appear on your screen.

This can take a while to start up – data has to be downloaded from the World Bank web site. Hover the mouse pointer over features that appear in the margins of the display to see annotation that indicates how you can change or manipulate various aspects of the display.

The data from the World Bank site is stored into a data frame `WorldBank`. The command that creates a gvis object M is:[17]

```
M <- gvisMotionChart(WorldBank, idvar="country",
          timevar="year",
          xvar="life.expectancy",
          yvar="fertility.rate",
          colorvar="region", sizevar="population",
          options=list(width=700, height=600))
## Now display the motion chart
plot(M)
```

[17] Both `WorldBank` and `M` should be in your workspace after running `demo(WorldBank)`. The data are also alternatively available from the image file `WorldBank.RData` at the url noted on the reverse of the title page.

Change `width` and `height` as needed to make better use of the screen display.

If arguments are supplied, security setting issues on the user computer can result in an initial assignment of columns that does not accord with the supplied arguments.[18] The drop-down menus should however function correctly, and can be used to obtain a display that accords with any choice of arguments that the user may want.

[18] The *gvis* object M comprises Javascript code that can be included on a web page. This should display correctly when the web page is accessed.

For a further example, load the image file **wdiSel.RData**, available from the url noted on the reverse of the title page. This will make available the data frame `wdiSel`. This has a larger number of indicators, but for 26 countries only. Figure C.1 (with the figures that are shown in color) shows an annotated version of a motion chart that was created from this dataset.

The following code generated the initial chart. The change to a log scale on the vertical axis was made interactively:

```
xnam <- "Electric power consumption (kWh per capita)"
ynam <- "Mobile cellular subscriptions (per 100 people)"
M <- gvisMotionChart(wdiSel, idvar="Country.Name", timevar="Year",
                     xvar=xnam, yvar=ynam,
                     colorvar="region", sizevar="Population, total",
                     options=list(width=600, height=500),
                     chartid="wbMotionChartSel")
plot(M)
```

7.6 Summary

Base graphics functions plot a graph. Lattice and ggplot2 functions return a graphics object. which can then stored or updated or plotted (printed).

A powerful feature, both of *ggplot2* graphics and of *lattice* graphics when the layering abilities of the *latticeExtra* package is used, is the ability to build a graph up layer by layer.

The R system makes available, via its various package, a wide variety of other graphics abilities. This includes dynamic and other 3-dimensional graphics.

7.7 Exercises

In the following exercises, if there is no indication of whether to use `base` or `lattice` graphics, use whichever seems most suitable.

1. Exercise 3 in Section 2.6.2 showed how to create the data frame `molclock`. Plot `AvRate` against `Myr`. Use `abbreviate()` to create abbreviated versions of the row names, and use these to label the points.

2. Compare the following graphs that show the distribution of head lengths (`hdlngth`) in the `possum` data set. What are the advantages and disadvantages of these different forms of display?

   a) a histogram (`hist(possum$hdlngth)`);

   b) a stem and leaf plot (`stem(qqnorm(possum$hdlngth))`;

   c) a normal probability plot (`qqnorm(possum$hdlngth)`); and

   d) a density plot (`plot(density(possum$hdlngth))`.

3. This exercise uses the data set `hotspots` (*DAAG* package).

   Plot `age` against `distance`. Use `identify()` to determine which years correspond to the two highest mean levels. That is, type

   ```
   plot(age ~ distance, data=hotspots)
   with(hotspots, identify(age ~ distance, labels=name))
   ```

   Use the left mouse button to click on the highest two points on the plot. (Right click in the figure region to terminate labeling.)

4. Use `mfrow()` to set up the layout for a 3 by 4 array of plots. In the top 4 rows, show normal probability plots for four separate 'random' samples of size 10, all from a normal distribution. In the middle 4 rows, display plots for samples of size 100. In the bottom four rows, display plots for samples of size 1000. Comment on how the appearance of the plots changes as the sample size changes.

5. The function `runif()` can be used to generate a sample from a uniform distribution, by default on the interval 0 to 1. Print out the numbers you get from `x <- runif(10)`. Then repeat exercise 6 above, but taking samples from a uniform distribution rather than from a normal distribution. What shape do the points follow?

6. The data frame `airquality` that is in the base package has columns `Ozone`, `Solar.R`, `Wind`, `Temp`, `Month` and `Day`. Plot `Ozone` against `Solar.R` for each of three temperature ranges, and each of three wind ranges.

7. Create a version of the data frame `Pima.tr2` that has `anymiss` as an additional column:

```
missIND <- complete.cases(Pima.tr2)
Pima.tr2$anymiss <- c("miss","nomiss")[missIND+1]
```

   (a) Use strip plots to compare values of the various measures for the levels of `anymiss`, for each of the levels of `type`. Are there any columns where the distribution of differences seems shifted for the rows that have one or more missing values, relative to rows where there are no missing values?

   Hint: The following indicates how this might be done efficiently:

```
library(lattice)
stripplot(anymiss ~ npreg + glu | type, data=Pima.tr2, outer=TRUE,
          scales=list(relation="free"), xlab="Measure")
```

   (b) Density plots are in general better than strip plots for comparing the distributions. Try the following, first with the variable `npreg` as shown, and then with each of the other columns except `type`. Note that for `skin`, the comparison makes sense only for `type=="No"`. Why?

```
## Exercise 7b
library(lattice)
## npreg & glu side by side (add other variables, as convenient)
densityplot( ~ npreg + glu | type, groups=anymiss, data=Pima.tr2,
            auto.key=list(columns=2), scales=list(relation="free"))
```

8

Regression with Linear Terms and Factors

Linear Models, in the style of `lm()`:	
Linear model	Any model that `lm()` will fit is a "linear" model. `lm()` can fit highly non-linear forms of response!
Diagnostic plots	Use `plot()` with the model object as argument, to get a basic set of diagnostic plots.
`termplot()`	If there are no interaction terms, use `termplot()` to visualize the contributions of the different terms.
Factors	In model formulae, factors model qualitative effects.
Model matrices	The model matrix shows how coefficients should be interpreted. (This is an especial issue for factors.)
GLMs	Generalized Linear Models are an extension of linear models, commonly used for analyzing counts.
Modern regession	This can use smoothers – spline and other functions of explanatory variables that adapt to suit the data.
[NB: `lm()` assumes independently & identically distributed (iid) errors, perhaps after applying a weighting function.]	

In this chapter, the chief focus will be on the `lm()` (*linear model*) function, discussed earlier in Section 3.2. The `lm()` function is the most widely used of a huge range of model fitting abilities, available in the various R packages.

Linear models are linear in the model parameters, not necessarily in the variables. A linear model can perfectly well fit a combination of *basis* curves.

Thus spline fits are formed as a linear combination from a kitset of curves.

8.1 Linear Models in R – Basic Ideas

Here, we fit a straight line, which is very obviously a linear model! This simple starting point gives little hint of the range of models that can be fitted using R's linear model `lm()` function.

The `lm()` function returns, as well as estimates, standard errors for parameters and for predictions. The standard error and p-value information provided by the `lm()` function assumes that the random term is i.i.d. (independently and identically distributed) normal. The independence assumption can be crucial.

The standard errors assume, also, a single model that is known from the start and on which the analysis is based.[1] If this assumption is incorrect, it can be important to resort to the use of empirical methods for assessing model performance – perhaps some variation of training/test methodology, or the bootstrap.

The symbolic notation[2] that is available in R for describing linear models makes it straightforward to set up quite elaborate and intricate models.

[1] The standard errors become increasingly unrealistic as the number of possible choices of model terms (variables, factors and interactions) increases.

[2] Wilkinson, GN and Rogers, CE, 1973. Symbolic description of models in analysis of variance, *Applied Statistics* 22: 392-399.

Scatterplot with fitted line – an example

The following plots data from the data frame `roller` (as in Figure 8.1) from the *DAAG* package.

```
library(DAAG)
plot(depression ~ weight, data=roller, fg="gray")
```

Figure 8.1: Plot of `depression` versus `weight`, using data from the data frame `roller` in the *DAAG* package.

The formula `depression ~ weight` can be used either as a graphics formula or as a model formula. The following fits a straight line, then adding it to the above plot:

```
plot(depression ~ weight, data=roller, fg="gray")
roller.lm <- lm(depression ~ weight, data=roller)
# For a line through the origin, specify
# depression ~ 0 + weight
abline(roller.lm)
```

Figure 8.2: This repeats Figure 8.1, now adding a fitted line.

Figure 8.2 repeats the plot, now with a fitted line added.

The different explanatory variables in the model are called `terms`. In the above, there is one explicit term only on the right, i.e., `weight`. This is in addition to the intercept, which is included by default.

8.1.1 Straight line regression – algebraic details

The standard form of simple straight line model can be written

$$\text{depression} = \alpha + \beta \times \text{weight} + \text{noise}.$$

Now write y in place of `depression` and x in place of `weight`, and add subscripts, so that the observations are: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Then the model can be written:

$$y_i = \alpha + \beta x_i + \varepsilon_i.$$

The $\alpha + \beta x_i$ term is the "fixed" component of the model, and ε_i is the random noise.

The line is chosen so that the sum of squares of residuals is as small as possible, i.e., the intercept α and the slope β chosen to minimize

$$\sum_{i=1}^{n}(y_i - \alpha - \beta_i x_i)^2$$

The R function `lm()` will provide estimates a of α and b of β. The straight line

$$\widehat{y} = a + bx$$

can then be added to the scatterplot.

Fitted or predicted values, calculated so that they lie on the estimated line, are obtained using the formula:

$$\widehat{y}_1 = a + bx_1,\ \widehat{y}_2 = a + bx_2, \ldots .$$

The residuals, which are the differences between the observed and fitted values, give information about the noise.

$$e_1 = y_1 - \widehat{y}_1,\ e_2 = y_2 - \widehat{y}_2, \ldots . \tag{8.1}$$

8.1.2 Syntax – model, graphics and table formulae:

The syntax for `lm()` models that will be demonstrated here is used, with modification, throughout the modeling functions in R. A very similar syntax can be used for specifying graphs and for certain types of tables.

Model objects

The following code returns a model object to the command line.

```
lm(depression ~ weight, data=roller)
```

```
Call:
lm(formula = depression ~ weight, data = roller)

Coefficients:
(Intercept)       weight
      -2.09         2.67
```

When returned to the command line in this way, a printed summary is returned.

Alternatively, the result can be saved as a named object, which is a form of list.

```
roller.lm <- lm(depression ~ weight, data=roller)
```

The names of the list elements are:

```
names(roller.lm)
```

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

Components of model objects can be accessed directly, as list objects. But it is usually better to use an extractor function. Note in particular `residuals()` (can be abbreviated to `resid()`), `coefficients()` (`coef()`), and `fitted.values()` (`fitted()`). For example:

```
coef(roller.lm)
```

8.1.3 Matrix algebra – straight line regression example

In order to write the quantity

$$\sum_{i=1}^{10}(y_i - a - bx_i)^2$$

that is to be minimized in matrix form, set:

$$\mathbf{X} = \begin{pmatrix} 1 & 1.9 \\ 1 & 3.1 \\ 1 & 3.3 \\ 1 & 4.8 \\ 1 & 5.3 \\ 1 & 6.1 \\ 1 & 6.4 \\ 1 & 7.6 \\ 1 & 9.8 \\ 1 & 12.4 \end{pmatrix}; \quad \mathbf{y} = \begin{pmatrix} 2 \\ 1 \\ 5 \\ 5 \\ 20 \\ 20 \\ 23 \\ 10 \\ 30 \\ 25 \end{pmatrix}; \quad \mathbf{e} = \mathbf{y} - \mathbf{Xb} = \begin{pmatrix} 2-(a+1.9b) \\ 1-(a+3.1b) \\ 5-(a+3.3b) \\ 5-(a+4.8b) \\ 20-(a+5.3b) \\ 20-(a+6.1b) \\ 23-(a+6.4b) \\ 10-(a+7.6b) \\ 30-(a+9.8b) \\ 25-(a+12.4b) \end{pmatrix} \quad \text{where} \quad \mathbf{b} = \begin{pmatrix} a \\ b \end{pmatrix}$$

Here a and b are chosen to minimize the sum of squares of elements of $\mathbf{e} = \mathbf{y} - \mathbf{Xb}$, i.e., to minimize

$$\mathbf{e}'\mathbf{e} = (\mathbf{y} - \mathbf{Xb})'(\mathbf{y} - \mathbf{Xb})$$

The least squares equations can be solved using matrix arithmetic.

Recap, and Next Steps in Linear Modeling

For this very simple model, the model matrix had two columns only. Omission of the intercept term will give an even simpler model matrix, with just one column.

Regression calculations in which there are several explanatory variables are handled in the obvious way, by adding further columns as necessary to the model matrix. This is however just the start to the rich range of possibilities that model matrices open up.

8.1.4 A note on the least squares methodology

More fundamental than least squares is the maximum likelihood principle. If the "error" terms are independently and identically normally distributed, then least squares and maximum likelihood are equivalent.

The assumptions of independence and identical distribution (iid) are crucial. The role of normality is commonly over-stated.

Least squares will not in general yield maximum likelihood estimates, and the SEs returned by `lm()` or by `predict()` from an `lm` model will be problematic or wrong if:

- Variances are not homogeneous[3];
- Observations are not independent;
- The sampling distributions of parameter estimates are noticeably non-normal.

[3] Weighted least squares is however justified by maximum likelihood if it is known how the variances change with x_i, or if the pattern of change can be inferred with some reasonable confidence.

- Model terms (variables, factors and/or interactions) have been chosen from some wider set of possibilities (the theory assumes a specfic known model).

Simplifying the model, in ways that do not much affect coefficients that remain in the model, may be acceptable.

Normality of the model 'errors' is more than is in practice required. Outliers, and skewness in the distribution, do often mean that the theory cannot be satisfactorily used as a good approxation.

8.2 *Checks — Before and After Fitting a Line*

Consider here a female versus male comparison of record times for Northern Island hill races.

Figure 8.3: Graphs compare female with male record times, for Northern Ireland hill races. Least squares lines are added, and marginal boxplots are shown on the horizontal axis. Panel A has untransformed scales, while Panel B has log transformed scales. For the code, see the script file for this chapter.

Untransformed vs transformed scales: Figure 8.3 shows two alternative views of the data. Least squares line have in each case been added.

In Panel A, a single data point at the top right lies well away from the main body of data. In Panel B, points are more evenly spread out, though still with a tail out to long times.

The following fits a regression line on the untransformed scale:

```
mftime.lm <- lm(timef ~ time, data=nihills)
```

The line appears to fit the data quite reasonably well. Is this an effective way to represent the relationship? An obvious problem is that data values become increasingly sparse as values increase, with one point widely separated from other data. That one data point, widely separated from other points, stands to have a disproportionate effect in determining the fitted line.

The coefficients for the line that is fitted on a logarithmic scale is:

```
mflogtime.lm <- lm(log(timef) ~ log(time),
                   data=nihills)
round(coef(mflogtime.lm), 3)
```

```
(Intercept)   log(time)
      0.267       1.042
```

The coefficient of 1.042 for `log(time)` implies that the relative rate of increase of female times is 4.2% greater than the relative rate of increase of male times.

The use of residuals for checking the fitted line:

In Figure 8.3, departures from the line do not stand out well relative to the line. To make residuals stand out, Figures 8.4A and 8.4B rotate the lines, for the untransformed and transformed data respectively, ~45° clockwise about its mid-point, to the horizontal.

Figure 8.4: In Panel A, residuals from the line for the unlogged data have been plotted against male times. Panel B repeats the same type of plot, now for the regression for the logged data.

Notice that, in Figure 8.4A, all residuals except that for the largest time lie very nearly on a line. The point with the largest fitted value, with the largest male (and female) time, is pulling the line out of whack. There is a mismatch between the data and the model. The picture in Figure 8.4B is much improved, though there still is an issue with the point for the longest time.

Residuals on the vertical scale of Figure 8.4B are on a scale of natural logarithms (logarithms to base *e*). As the range is small (roughly between -0.1 and 0.1), the values can be interpreted as relative differences on the scale of times.

A residual of -0.1 denotes a time that is about 10% (more accurately 9.5%) less than the fitted value on the line. A residual of 0.1 denotes a time that is about 10% (more accurately 10.5%) more than the fitted value.

The common benefits of a logarithmic transformation: Where measurement data have a long tail out to the right, it commonly makes sense to work with logarithms of data values, as in Figure 8.3B. Often, working with data on a logarithmic scale has several useful consequences:

- The skewness is reduced
- The variation at the high end of the range of values is reduced, relative to variation at the low end of the range of values.
- Working on a logarithmic scale is equivalent to working with relative, rather than absolute, change. Thus a change from 10 to

20 is equivalent to a change from 20 to 40, or from 40 to 80. On a logarithmic scale, these are all changes by an amount of $\log(2)$.

- By default, the function `log()`) returns natural logarithms, i.e., logarithms to base e. On this scale, a change of 0.05 is very close to a change of 5%. A change of 0.15 is very roughly a change of 15%. [A decrease of 0.15 is a decrease of $\simeq$ 13.9%, while an increase of 0.15 is a increase of $\simeq$ 16.2%]

Once the model is fitted, checks can and should be made on the extent and manner of differences between observations and fitted model values. Graphical checks are the most effective,[4] at least as a starting point. Mostly, such checks are designed to highlight common types of departure from the model.

[4] Statistics that try to provide an overall evaluation focus too much on a specific form of departure, and do a poor job at indicating whether the departure from assumptions matters.

Figure 8.4B provided a simple form of diagnostic check. This is one of several checks that are desirable when models have been fitted.

8.2.1 *Diagnostics – checks on the fitted model

For `lm` models, the R system has a standard set of diagnostic plots that users are encouraged to examine. These are a starting point for investigation. Are apparent departures real, or may they be a result of statistical variation? For the intended use of the model output, do apparent departures from model assumptions matter.

Section 8.3 demonstrates the use of simulation to help in judging between genuine indications of model departures and features of the plots that may well reflect statistical variation.

For drawing attention to differences between the data and what might be expected given the model, plots that show residuals are in general much more effective than plots that show outcome (y) variable values. Additionally, plots are needed that focus on common specific types of departure from the model.

All four diagnostic plots

Figure 8.5 shows the default diagnostic plots for the regression with the untransformed data:

Figure 8.5: Diagnostic plots from the regession of `timef` on `time`.

Simplified code is:

```
mftime.lm <- lm(timef ~ time, data=nihills)
plot(mftime.lm, cex.caption=0.8)
```

The first of these plots has similar information to Figure 8.4A above. A difference is that residuals are now plotted against fitted values. It is immaterial, where there is just one explanatory variable, whether residuals are plotted against fitted values or against x-values – the difference between plotting against $a + bx$ and plotting against x amounts only to a change of labeling on the x-axis.

Figure 8.6 shows the default diagnostic plots for the transformed data. Simplified code is:

```
plot(mflogtime.lm, cex.caption=0.8)
par(opar)
```

Two further plots are available; specify `which=4` or `which=6`, e.g. `plot(mftime.lm, which=4)`. These give a different slant on what is shown in the fourth default plot (`which=5`).

Figure 8.6: Diagnostic plots from the regression of `log(timef)` on `log(time)`.

The point for the largest time still has a very large leverage and, as indicated by its position relative to the Cook's distance contours, a large influence on the fitted regression line. This may, in part or entirely, be a result of a variance that, as suggested by the scale-location plot in Panel 3, tends to increase with increasing fitted value. The normal Q-Q plot (Panel 2) suggests an overall distribution of residuals that is acceptably normal. The large residual associated with the largest time in Panel 1 would look much less out of place if there was an adjustment that allowed for a variance that increases with increasing fitted value.

These differences from the assumed model are small enough that, for many purposes, the line serves as a good summary of the data. The equation is:

```
mflogtime.lm <- lm(log(timef) ~ log(time),
                   data=nihills)
round(coef(mflogtime.lm), 3)
```

```
(Intercept)   log(time)
      0.267       1.042
```

The coefficient that equals 1.042 can be interpreted as a relative rate of increase, for female time relative to male time. Consider a race for which the male record time is 100 minutes. The predicted female time is:

$$\exp(0.267 + 1.042\log(100/60)) = 2.223\text{h} = 133.4\text{m}.$$

An increase of one minute, or 1%, in the male time, is predicted to lead to an increase of close to $1.042 \times 1\%$ in the female time. The predicted increase is 133.4×1.042m.

Panel 2 — A check for normality: The second panel in Figure 8.5 identifies two large negative residuals and one large positive residual. This seems inconsistent with the assumption that residuals have a normal dsitribution. Again, Figure 8.6 shows an improvement.

Modest departures from normality are not a problem per se. Heterogeneity of variance, and outliers in the data, are likely to be the more serious issues.

To obtain this second plot only, without the others, type:

```
plot(mftime.lm, which=2)
```

Panel 3 — Is the variance constant?: The third panel is designed to check whether variation about the fitted line, as measured by the variance, is constant. For this, there should be no trend up or down in the points. The large upward trend in the third panel of 8.5 has largely disappeared in the third panel of Figure 8.6.

To obtain this third plot only, without the others, type:

```
plot(mftime.lm, which=3)
```

Panel 4 — a check for high leverage points: The fourth panel is designed to check on points with large leverage and/or large influence. In straight line regression, the most extreme leverage points are points that are separated from the main body of points, and are at the high or (less commonly) low end of the range of x-values.

To obtain this fourth plot only, without the others, type:

```
plot(mftime.lm, which=5)
```

The combined effect of leverage and magnitude of residual determines what *influence* a point has. Large leverage translates into large influence, as shown by a large Cook's distance, when the residual is also large. Points that lie within the region marked out by the 0.5 or (especially) the 1.0 contour for Cook's distance have a noticeable influence on the fitted regression equation. Even with the logged data, the point for the largest time ('Seven Sevens') is skewing the regression line noticeably.

For working within the main range of the data values, we might prefer to use the regression line that is obtained when 'Seven Sevens' is omitted. If 'Seven Sevens' is omitted in estimating the regression line, this should be made clear in any report, and the reason explained. The large residual for this point does hint that extrapolation much beyond the upper range of data values is hazardous.

Note that there is no reason to suspect any error in this value. Possibly the point is taking us into a part of the range where the relationship is no longer quite linear. Or this race may be untypical for more reasons than that it is an unusually long race.

The following shows the change when 'Seven Sevens' is omitted:

```
round(coef(mflogtime.lm), 4)
```

```
(Intercept)   log(time)
     0.2667      1.0417
```

```
omitrow <- rownames(nihills)!="Seven Sevens"
update(mflogtime.lm, data=subset(nihills, omitrow))
```

```

Call:
lm(formula = log(timef) ~ log(time), data = subset(nihills, omitrow))

Coefficients:
(Intercept)    log(time)
      0.239        0.991
```

8.2.2 *The independence assumption is crucial*

A key assumption is that observations are independent. The independence assumption is an assumption about the process that generated the data, about the way that it should be modeled. There is no one standard check that is relevant in all circumstances. Rather the question should be: "Are there aspects of the way that the data were generated that might lead to some form of dependence?" Thus, when data are collected over time, there may be a time series correlation between points that are close together in time.

Another possibility is some kind of clustering in the data, where observations in the same cluster are correlated. In medical applications, it is common to have multiple observations on the one individual. Where clusters may be present, but there is no way to identify them, dependence is hard or impossible to detect.

Issues of dependence can arise in an engineering maintenance context. If the same mechanic services two aircraft engines at the same time using replacement parts from the same batch, this greatly increases the chances that the same mistake will be made on the two engines, or the same faulty part used. Maintenance faults are then not independent. Independence is not the harmless assumption that it is often made out to be!

There may be further checks and tests that should be applied. These may be specific to the particular model.

8.3 *Simulation Based Checks

A good way to check whether indications of departures from the model may be a result or random variation is to compare the plot with similar plots for several sets of simulated data values, as a means of verifying that the mismatch is, if residuals from the line are independent and normally distributed, real. This is the motivation for Figure 8.7.

If the assumption of independent random errors is wrong, patterns in the diagnostic plots that call for an explanation may be more common than suggested by the simulations.

Figure 8.7: The plots are four simulations of residuals, from the model that is fitted to the unlogged data. The coefficients used, and the standard deviation, are from the fitted least squares line.

```
## Code
gph <- plotSimScat(obj=mftime.lm, show="residuals",
                   type=c("p","smooth"),
                   layout=c(4,1))
```

```
update(gph, xlab="Time (h) for males",
      ylab="Residuals")
```

The simulations indicate that there can be a pattern in the smooth curve that is largely due to the one point that is widely separated from other data. On the other hand, the very large residual seen in the actual data is not matched in any of the simulations.

This type of check can be repeated for the other diagnostic plots:

- A check for normality (Panel 2: `which=2`. Type:

```
plotSimDiags(obj=mftime.lm, which=2, layout=c(4,1))
```

- Is the variance constant? (Panel 3: `which=3`. Type:

```
plotSimDiags(obj=mftime.lm, which=3, layout=c(4,1))
```

- Are there issues of leverage and influence? (Panel 4: `which=5`. Type:

```
plotSimDiags(obj=mftime.lm, which=5, layout=c(4,1))
```

Scatterplots that are derived from the simulation process

To see scatterplots that are derived from the simulation process, use the function `plotsimscat()`, from the *DAAG* package. For example, try, working with the untransformed data:

```
gph <- plotSimScat(mftime.lm, layout=c(4,1))
update(gph, xlab="Male record times (h)",
       ylab="Female record times (h)")
```

Observe that the largest simulated value lies consistently above the data value. Other simulated values, for male times of more than around one hour, tend to lie below the actual data. This is much easier to see in the plots of residuals. A scatterplot that shows the actual data values is not a good tool for making these difference visually obvious.

8.4 Key questions for the use of models

Key questions are:

- Modeling and analysis
  - Which model?
  - Do we want to make predictions? Or is the interest in getting parameter estimates that are interpretable?
  - How will model performance be measured?
  - How close can we get to measuring the performance that matters?
- Interpretation

- The task is easier if the aim is prediction, rather than interpretation of model parameters.
- Can model parameters be interpreted in scientifically meaningful ways?

  [This is a minefield, with huge scope for getting it wrong.]

More detailed comments will now follow on some of the issues raised above.

The choice of method: Note the use of the word "method", not algorithm. Algorithms specify a sequence of computational steps. Something more than an algorithm is needed, if results are to have some use that generalizes beyond the specific data used.

There are many different methods. How should the analyst choose between them? What are good ways to assess the performance of one or other algorithm? A credible measure of model performance is needed, evaluated on test data that closely reflects the context in which the model will be applied.

Which are the important variables? Often, the analyst would like to know which data columns (variables, or features) were important for, e.g., a classification. Could some of them be omitted without loss?

The analyst may wish to attach an interpretation to one or more coefficients? Does the risk of heart attack increase with the amount that a person smokes? For a meaningful interpretation of model parameters, it is necessary to be sure that:

- All major variables or factors that affect the outcome have been accounted for.
- Those variables and factors operate, at least to a first order of approximation, independently.

In some cases, a different but equivalent choice of parameters will be more meaningful. For working with the Northern Ireland hillrace data in Subsection 8.6, the parameters `dist` and `climb` clearly do not exercise their effects independently, making their coefficients difficult to interpret. It is better to work with `log(dist)` and `log(dist/climb)`, which are very nearly independent.

See Rosenbaum's *Observational Studies*[5] for comments on approaches that are often useful in the attempt to give meaningful interpretations to coefficients that are derived from observational data.

[5] Rosenbaum, P.R, 2002. *Observational Studies*, 2nd edn. Springer-Verlag.

8.5 Factor Terms – Contrasts

Here, we show how regression models can be adapted to fit terms involving factors.

Another special type of term is one that allows smooth functions of explanatory variables. Again, linear models can be adapted to handle such terms.

Water (Water only)	A (Additive 1)	B (Additive 2)	C (Additive 3)
1.50	1.50	1.90	1.00
1.90	1.20	1.60	1.20
1.30	1.20	0.80	1.30
1.50	2.10	1.15	0.90
2.40	2.90	0.90	0.70
1.50	1.60	1.60	0.80
Mean = 1.683	0.983	1.75	0.983

Table 8.1: Root weights (*weight*) (g) of tomato plants, grown with water only and grown wth three different treatments. Data are in the data frame `tomato` (`DAAG` 1.17 or later).

Additive A is `conc nutrient`, B is `3x conc nutrient`, and C is `2-4-D + conc nutrient`. For convenience, we label the factor levels `Water`, `A`, `B`, and `C`, in that order.

```
lev <- c("Water", "A", "B", "C")
tomato[, "trt"] <- factor(rep(lev, rep(6,4)),
                          levels=lev)
```

Taking `Water` as the initial level the effect, in the first analysis that is given below, that it is treated as a reference level.

8.5.1 Example – tomato root weight

The model can be fitted either using the function `lm()` or using the function `aov()`. The two functions give different default output. The main part of the calculations is the same whether `lm()` or `aov()` is used.

For model terms that involve factor(s), there are several different ways to set up the relevant columns of the model matrix. The default, for R and for many other computer programs, is to take one of the treatment levels as a baseline or reference, with the effects of other treatment levels then measured from the baseline. Here it makes sense to set `Water` as the baseline.

Table 8.2 shows the model matrix when `Water` is taken as the baseline. Values of the response (`tomato$weight`) have been added in the final column. Also included, in the column headers, is information from the least squares fit.

The following uses `aov()` for the calculations:

```
## Analysis of variance: tomato data (from DAAG)
tomato.aov <- aov(weight ~ trt, data=tomato)
```

Figure 8.8A is a useful summary of what the analysis has achieved. The values are called *partial*s because the overall mean has been subtracted off. Figure 8.8B that is shown alongside shows the effect of working with the logarithms of weights. The scatter about the mean for the treatment still appears much larger for the controls than for other treatments.
Code for Figures 8.8A and 8.8B is:

```
## Panel A: Use weight as outcome variable
```

Figure 8.8: Termplot summary of the one-way analysis of variance result — A: for the analysis that uses weights as the outcome variable, and B: for the analysis that works with `log(weight)`

```
tomato.aov <- aov(weight ~ trt, data=tomato)
termplot(tomato.aov, xlab="Treatment",
         ylab="Partial for treatment",
         partial.resid=TRUE, se=TRUE, pch=16)
mtext(side=3, line=0.5, "A: weight", adj=0, cex=1.2)
## Panel B: Use log(weight) as outcome variable
logtomato.aov <- aov(log(weight) ~ trt, data=tomato)
termplot(logtomato.aov, xlab="Treatment",
         ylab="Partial for treatment",
         partial.resid=TRUE, se=TRUE, pch=16)
mtext(side=3, line=0.5, "B: log(weight)", adj=0,
      cex=1.2)
```

Residuals, if required, can be obtained by subtracting the fitted values in Table 8.2 from the observed values (y) in Table 8.1.

Coefficient estimates for the model that uses `weight` as the dependent variable, taken from the output summary from R, are:

```
round(coef(summary.lm(tomato.aov)),3)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.683      0.187   9.019    0.000
trtA           0.067      0.264   0.253    0.803
trtB          -0.358      0.264  -1.358    0.190
trtC          -0.700      0.264  -2.652    0.015
```

The row labeled `(Intercept)` gives the estimate (= 1.683) for the baseline, i.e., `Water`. The remaining coefficients (differences from the baseline) are:

A: weight differs by 0.067.

B: weight differs by −0.358.

C: weight differs by −0.700.

Regression calculations have given us a relatively complicated way to calculate the treatment means! The methodology shows its power to better effect in more complex forms of model, where there is no such simple alternative.

Examination of the model matrix can settle any doubt about how to interpret the coefficient estimates, The first four columns of Table 8.2 comprise the model matrix, given by:

```
model.matrix(tomato.aov)
```

The multipliers determined by least squares calculations are shown above each column. Also shown is the fitted value, which can be calculated either as `fitted(tomato.aov)` or as `predict(tomato.aov)`.

Water: 1.683	A: +0.067	B: −0.358	C: −0.700	Fitted value
1	0	0	0	1.683
1	0	0	0	1.683
....				
1	0	0	0	1.683
1	1	0	0	1.750
1	1	0	0	1.750
....				
1	1	0	0	1.750
1	0	1	0	1.325
1	0	1	0	1.325
....				
1	0	1	0	1.325
1	0	0	1	0.983
1	0	0	1	0.983
....				
1	0	0	1	0.983

Table 8.2: The model matrix for the analysis of variance calculation for the data in Table 8.1 is shown in gray. A fourth column has been added that shows the fitted values. At the head of each column is the multiple, as determined by least squares, that is taken in forming the fitted values.

8.5.2 *Factor terms – different choices of model matrix*

In the language used in the R help pages, different choices of *contrasts* are available, with each different choice leading to a different model matrix and to different regression parameters. The fitted values remain the same, the termplot in Figure fig:tomatotermA is unchanged, and the analysis of variance table is unchanged.

Where there is just one factor, the constant term can be omitted, i.e., it is effectively forced to equal zero. The parameters are then the estimated treatment means. Specify:

```
## Omit constant term from fit;
## force parameters to estimate treatment means
tomatoM.aov <- aov(weight ~ 0 + trt, data=tomato)
```

The first nine rows of the model matrix are:

```
mmat <- model.matrix(tomatoM.aov)
mmat[1:9, ]
```

```
  trtWater trtA trtB trtC
1        1    0    0    0
2        1    0    0    0
3        1    0    0    0
4        1    0    0    0
5        1    0    0    0
```

```
6          1     0     0     0
7          0     1     0     0
8          0     1     0     0
9          0     1     0     0
```

```
## ...    ...     ...     ...
```

Observe that there is now not an initial column of ones. This is fine when there is just one factor, but does not generalize to handle more than one factor and/or factor interaction.

The default (*treatment*) choice of *contrasts* uses the initial factor level as baseline, as we have noted. Different choices of the baseline or reference level lead to different versions of the model matrix. The other common choice, i.e., *sum* contrasts, uses the average of treatment effects as the baseline.

Be sure to choose the contrasts that give the output that will be most helpful for the problem in hand. Or, more than one run of the analysis may be necessary, in order to gain information on all effects that are of interest.

The sum contrasts

Here is the output when the baseline is the average of the treatment effects, i.e., from using the *sum* contrasts:

```
oldoptions <- options(contrasts=c("contr.sum",
                                  "contr.poly"))
tomatoS.aov <- aov(weight ~ trt, data=tomato)
round(coef(summary.lm(tomatoS.aov)),3)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.435      0.093  15.381    0.000
trt1           0.248      0.162   1.534    0.141
trt2           0.315      0.162   1.946    0.066
trt3          -0.110      0.162  -0.683    0.502
```

```
options(oldoptions)  # Restore default contrasts
```

The baseline, labeled `(Intercept)`, is now the treatment mean. This equals 1.435. Remaining coefficients are differences, for Water and for treatment levels A and B, from this mean. The sum of the differences for all three treatments is zero. Thus the difference for C is (rounding up)

$$-(0.2479 + 0.3146 - 0.1104) = -0.4521.$$

Yet other choices of contrasts are possible; see `help(contrasts)`.

The estimates (means) are:

Water: $1.435 + 0.248 = 1.683$.

A: $1.435 + 0.315 = 1.750$.

B: $1.435 - 0.110 = 1.325$.

C: $1.435 - 0.452 = 0.983$.

Interaction terms

The data frame `cuckoos` has the lengths and breadths of cuckoo eggs that were laid in the nexts of one of six different bird species. The following compares a model where the regression line of breadth against length is the same for all species, with a model that fits a different line for each different cuckoo species:

```
cuckoos.lm <- lm(breadth ~ species + length, data=cuckoos)
cuckoosI.lm <- lm(breadth ~ species + length + species:length, data=cuckoos)
print(anova(cuckoos.lm, cuckoosI.lm), digits=3)
```

```
Analysis of Variance Table

Model 1: breadth ~ species + length
Model 2: breadth ~ species + length + species:length
  Res.Df  RSS Df Sum of Sq    F Pr(>F)
1    113 18.4
2    108 17.2  5      1.24 1.56   0.18
```

Here, the model `cuckoos.lm`, where the regression lines are parallel (the same slope for each species), appears adequate.

An alternative way to compare the two models is:

```
anova(cuckoos.lm, cuckoosI.lm, test="Cp")
```

The `Cp` statistic (smaller is better) compares models on the basis of an assessment of their predictive power. Note the use of the argument `test="cp"`, even though this is not a comparison that is based on a significance text.

8.6 Regression with two explanatory variables

Data exploration

The dataset `nihills` in the *DAAG* package has record times for Northern Ireland mountain races. First, get a few details of the data:

```
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 ...
```

The following Figure 8.9 repeats Figure 3.6 from Chapter 3.3. The left panel shows the unlogged data, while the right panel shows the logged data:

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, the point with the largest unlogged values (the same for all variables) will have a leverage, influencing the fitted regression, that is enormously larger than that of other points.

The log transformed data are consistent with a form of parsimony that is well-designed to lead to a simple form of model. We will see that this also leads to more readily interpretable results. Also the distributions for individual variables are more symmetric.

Here again is the code:

```
## Unlogged data
library(lattice)
## Scatterplot matrix; unlogged data
splom(~nihills)
```

Figure 8.9: Scatterplot matrices for the Northern Ireland mountain racing data. In the right panel, code has been added that shows the correlations. This repeats Figure 3.6 from Chapter 3.3.

The right panel requires a data frame that has the logged data

```
## Logged data
lognihills <- log(nihills)
names(lognihills) <- paste0("l", names(nihills))
## Scatterplot matrix; log scales
splom(~ lognihills)
```

8.6.1 *The regression fit*

The following regression fit uses logarithmic scales for all variables:

```
lognihills <- log(nihills)
lognam <- paste0("l", names(nihills))
names(lognihills) <- lognam
lognihills.lm <- lm(ltime ~ ldist + lclimb,
                    data=lognihills)
round(coef(lognihills.lm),3)
```

```
(Intercept)       ldist      lclimb
     -4.961       0.681       0.466
```

Thus for constant `climb`, the prediction is that time per mile will decrease with increasing distance. Shorter races with the same climb will involve steeper ascents and descents.

The fitted equation gives predicted times:

$$\begin{aligned} & e^{3.205} \times \text{dist}^{0.686} \times \text{climb}^{0.502} \\ = \; & 24.7 e^{3.205} \times \text{dist}^{0.686} \times \text{climb}^{0.502} \end{aligned}$$

A result that is easier to interpret can be obtained by regressing `log(time)` on `log(dist)` and `log(gradient)`, where `gradient` is `dist/climb`.

```
nihills$gradient <- with(nihills, climb/dist)
lognihills <- log(nihills)
lognam <- paste0("l", names(nihills))
names(lognihills) <- lognam
lognigrad.lm <- lm(ltime ~ ldist + lgradient,
```

```
                    data=lognihills)
round(coef(lognigrad.lm),3)
```

```
(Intercept)       ldist   lgradient
     -4.961       1.147       0.466
```

Thus, with `gradient` held constant, the prediction is that `time` will increase at the rate of dist$^{1.147}$. This makes good intuitive sense.

Figure 8.10: The vertical scales in both "term plot" panel show log(time), centered to a mean of zero. The partial residuals in the left panel are for `ldist`, while those in the right panel are for `lgradient`, i.e., `log(climb/dist)`. Smooth curves (dashes) have been passed through the points.

We pause to look more closely at the model that has been fitted. Does `log(time)` really depend linearly on the terms `ldist` and `log(lclimb)`? The function `termplot()` gives a good graphical indication (Figure 8.10).

```
## Plot the terms in the model
termplot(lognigrad.lm, col.term="gray", partial=TRUE,
         col.res="black", smooth=panel.smooth)
```

The vertical scales show changes in `ltime`, about the mean of `ltime`. The lines show the estimated effect of each explanatory variable when the other variable is held at its mean value. The lines, which are the contributions of the individual linear terms ("effects") in this model, are shown in gray so that they do not obtrude unduly. The dashed curves, which are smooth curves that are passed through the residuals, are the primary features of interest.

Notice that, in the plot for `ldist`, the smooth dashed line does not quite track the fitted line; there is a small but noticeable indication of curvature that can be very adequately modeled with a quadratic curve. Note also that until we have modeled effectively the clear trend that seems evident in this plot, there is not much point in worrying about possible outliers.

8.7 *Variable Selection – Stepwise and Other*

Common variable selection methods include various versions of forward and backward selection, and exhaustive (best subset) selection. These or other variable selection methods invalidate standard model assumptions, which assume a single known model.

There are (at least) three inter-related issues for the use of results from a variable selection process:

The points made here can have highly damaging implications for analyses where it is important to obtain interpretable regression coefficients. In such analyses, changes to the initial model should be limited to simplifications that do not modify the model in any substantial manner. Following the selection process, check coefficients against those from the full model. Any large changes should ring a warning bell.

The implications for prediction are, relatively, much more manageable.

(i) Use of standard theoretically based model fitting procedures, applied to the model that results from the model selection process, will lead to a spuriously small error variance, and to a spuriously large model F-statistic. Coefficient estimates will be inflated, have spuriously small standard errors, and spuriously large t-statistics. (Or to put the point another way, it is inappropriate to refer such statistics to a standard t-distribution.)

(ii) Commonly used stepwise and other model selection processes are likely to over-fit, i.e., the model will not be optimal for prediction on test data that are distinct from the data used to train the model. The selected model may in some instances be inferior, judged by this standard, to a model that uses all candidate explanatory variables. (There are alternative ways to use all variables. Should low order interactions be included? Should some variables be transformed?)

(iii) Coefficients may change, even to changing in sign, depending on what else is included in the model. With a different total set of coefficients, one has a different model, and the coefficients that are common across the two models may be accordingly different. (They will be exactly the same only in the unusual case where "variables" are uncorrelated.) There is a risk that variable selection will remove variables on whose values (individually, or in total effect) other coefficient estimates should be conditioned. This adds uncertainty beyond what arises from sampling variation.

Note that these points apply to pretty much any type of regression modelling, including generalized linear models and classification (discriminant) models.

Where observations are independent, items (i) and (ii) can be addressed, for any given selection process, by splitting the data into training, validation and test sets. Training data select the model, with the validation data used to tune the selection process. Model performance is then checked against the test data.

Somewhat casual approaches to the use of backward (or other) stepwise selection may be a holdover from hand calculator days, or from times when computers grunted somewhat to handle even modest sized calculations. This may be one of the murky dark alleys of statistical practice, where magic incantations and hope too often prevail over hard evidence.

Appropriate forms of variable selection process can however be effective in cases where a few only of the coefficients have predictive power, and the relevant t-statistics are large – too large to be substantially inflated by selection effects.

8.7.1 Use of simulation to check out selection effects:

The function `bestsetNoise()` (*DAAG*) can be used to experiment with the behaviour of various variable selection techniques with data

that is purely noise. For example, try:[6]

```
bestsetNoise(m=100, n=40, nvmax=3)
bestsetNoise(m=100, n=40, method="backward",
             nvmax=3)
```

The analyses will typically yield a model that appears to have highly (but spuriously) statistically significant explanatory power, with one or more coefficients that appear (again spuriously) significant at a level of around p=0.01 or less.

[6] See also Section 6.5, pp. 197-198, in: Maindonald, JH and Braun, WJ, 2010. *Data Analysis and Graphics Using R – An Example-Based Approach*, 3rd edition. Cambridge University Press.

The extent of selection effects – a detailed simulation: As above, datasets of random normal data were created, always with 100 observations and with the number of variables varying between 3 and 50. For three variables, there was no selection, while in other cases the "best" three variables were selected, by exhaustive search.

Figure 8.11 plots the p-values for the 3 variables that were selected against the total number of variables. The fitted line estimates the median p-value. Code is:

```
library(DAAG)
library(quantreg)
library(splines)
set.seed(37)    # Use to reproduce graph shown
bsnVaryNvar(m=100, nvar=3:50, nvmax=3, fg="gray")
```

Figure 8.11: p-values, versus number of variables available for selection, when the "best" 3 variables were selected by exhaustive search. The fitted line estimates the median p-value.

When all 3 variables are taken, the p-values are expected to average 0.5. Notice that, for selection of the best 3 variables out of 10, the median p-value has reduced to about 0.1.

Examples from the literature The paper cited in the sidenote[7] gives several examples of published spurious results, all for the use of discriminant methods with microarray data. The same effects can arise from model tuning.

[7] Ambroise, C and McLachlan, GJ, 2001. Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the National Academy of Sciences USA*, **99**: 6562-6566.

8.7.2 *Variable and model selection – strategies*

Several alternative mechanisms are available that can yield reasonable standard errors and other accuracy measures. These include:

a) Fit the model to test data that have played no part in the model selection and tuning process;

b) use cross-validation. The model selection and fitting process must be repeated at each cross-validation fold;

c) repeat the whole analysis, selection and all, with repeated bootstrap samples, using variation between the different sample results to assess the accuracy of one or other statistic;

d) simulate, including all selection and tuning steps, from the fitted model.

For b) and c), there will be somewhat different selections for each different cross-validation fold or bootstrap sample. This is itself instructive.

One possibility, following stepwise or other selection, is that the p-values of one or more coefficients may be so small that they are very unlikely to be an artefact of the selection process. In general, a simulation will be required, in order to be sure.

If however the coefficients are not themselves very meaningful, what is the point?

Model selection more generally:

More generally, the model may be chosen from a wide class of models. Again, model selection biases standard errors to be smaller than indicated by the theory, and coefficients and t-statistics larger. The resulting anti-conservative estimates of standard errors and other statistics should be regarded sceptically.

Use of test data that are separate from data used to develop the model deals with this issue.

A further issue, which use of separate test data does not address, is that none of the models on offer is likely to be strictly correct. Mis-specification of the fixed effects will bias model estimates, at the same time inflating the error variance or variances. Thus it will to an extent work in the opposite direction to selection effects.

8.8 1970 cost for US electricity producers

There is a wide range of possible choices of model terms. Figure 8.12 shows the scatterplot matrices of the variables. Code is:

```
library(car)
library(Ecdat)
data(Electricity)
spm(Electricity, smooth=TRUE, reg.line=NA,
    col=adjustcolor(rep("black",3), alpha.f=0.3))
```

8.8.1 Model fitting strategy

The analysis will start by checking for clearly desirable transformations to variables. Then, for obtaining a model whose whose parameters are as far as possible interpretable, a strategy is:

(i) Start with a model that includes all plausible main effects (variables and factors). Ensure that the model is parameterised in a way that makes parameters of interest as far as possible interpretable (e.g., in Subsection 3.3 above, work with `distance` and `gradient`, not `distance` and `climb`)

(ii) [21pt] Model simplification may be acceptable, if it does not change the parameters of interest to an extent that affects interpretation. The common $p > 0.05$ is too severe; try instead $p = 0.15$ (remove terms with $p > 0.15$) or $p = 0.20$.

Removal of terms with $p > 0.15$ or $p > 0.2$ rather than $p > 0.05$ greatly reduces the risk that estimates of other parameters, and their standard errors, will change in ways that affect the interpretation of model results.

(iii) Variables and/or factors that have no detectable main effect are in general unlikely to show up in interactions. Limiting attention to the main effects that were identified in (ii) above, we then compare a model which has only main effects with a model that inludes all

`cost:` total cost
`q:` total output
`pl:` wage rate
`sl:` cost share, labor
`pk:` capital price index
`sk:` cost share, capital
`pf:` fuel price
`sf:` cost share, fuel

Figure 8.12: Scatterplot matrix, for the variables in the data set `Electricity`, in the *Ecdat* package. Density plots are shown in the diagonal.

2-way interactions. Then, using $p \simeq 0.15$ or $p \simeq 0.2$ as the cutoff, remove interaction terms that seem unimportant, and check that there are no changes of consequence in terms that remain.

(iv) In principle, the process may be repeated for order 3 interactions.

(v) Use the function `add1()` to check for individual highly significant terms that should be included. For this purpose, we might set $p = 0.01$ or perhaps $p = 0.001$.

The strategy is to be cautious (hence the cutoff of $p = 0.2$) in removing terms, whether main effects or first order interactions. In a final check whether there is a case for adding in terms that had been omitted, we include a term only if it is highly statistically significant. This limits the scope for selection effects.

Distributions of variables

The distributions of `cost` and `q` are highly skew. The relationship between these two variables is also very close to linear. We might try taking logarithms of both these variables.

Figure 8.13 examines the scatterplot matrix for the logarithms of the variables `cost` and `q`. Code is:

Figure 8.13: Scatterplot matrix for the logarithms of the variables `cost` and `q`. Density plots are shown in the diagonal.

```
varlabs <- c("log(cost)", "log(q)")
spm(log(Electricity[,1:2]), var.labels=varlabs,
    smooth=TRUE, reg.line=NA,
    col=adjustcolor(rep("black",3), alpha.f=0.5))
```

We start with a model that has main effects only:

```
elec.lm <- lm(log(cost) ~ log(q)+pl+sl+pk+sk+pf+sf,
              data=Electricity)
```

Now examine the termplot (Figure 8.14): Code is:

Figure 8.14: Termplot summary for the model that has been fitted to the `Electricity` dataset.

```
termplot(elec.lm, partial=T, smooth=panel.smooth,
         transform.x=TRUE)
```

Notice that in the partial plot for `q`, the dashed curve that is fitted to the residuals closely tracks the fitted effect (linear on a scale of `log(q)`. This confirms the use of `log(q)`, rather than `q`, as explanatory variable.

Now examine the model output:

```
round(coef(summary(elec.lm)),5)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.41328    0.70720 -7.6545  0.00000
log(q)       0.89250    0.00994 89.8326  0.00000
pl          -0.00002    0.00001 -1.9341  0.05499
sl           2.48020    0.74898  3.3114  0.00116
pk           0.00083    0.00127  0.6562  0.51272
sk           0.62272    0.70837  0.8791  0.38076
pf           0.03042    0.00228 13.3338  0.00000
sf          -0.30965    0.69091 -0.4482  0.65467
```

The p-values suggest that `pk`, `sk`, and `sf` can be dropped from the model. Omission of these terms makes only minor differences to the coefficients of terms that remain.

```
elec2.lm <- lm(log(cost) ~ log(q)+pl+sl+pf,
               data=Electricity)
round(coef(summary(elec2.lm)),5)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.28641    0.13701 -38.585  0.00000
log(q)       0.88901    0.00986  90.167  0.00000
pl          -0.00002    0.00001  -2.072  0.03994
sl           2.69722    0.32464   8.308  0.00000
pf           0.02659    0.00191  13.934  0.00000
```

Now check whether interaction terms should be included:

```
elec2x.lm <- lm(log(cost) ~ (log(q)+pl+sl+pf)^2,
                data=Electricity)
anova(elec2.lm, elec2x.lm)
```

```
Analysis of Variance Table

Model 1: log(cost) ~ log(q) + pl + sl + pf
Model 2: log(cost) ~ (log(q) + pl + sl + pf)^2
  Res.Df  RSS Df Sum of Sq    F  Pr(>F)
1    153 5.00
2    147 2.81  6      2.19 19.1 2.3e-16
```

The case for including first order interactions seems strong. The coefficients and SEs are:

```
round(coef(summary(elec2x.lm)),5)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -3.63931    0.67803 -5.3675  0.00000
log(q)       0.71481    0.05455 13.1039  0.00000
pl          -0.00031    0.00007 -4.2508  0.00004
sl           6.06389    1.64592  3.6842  0.00032
pf           0.01592    0.01499  1.0623  0.28985
log(q):pl    0.00003    0.00001  6.2902  0.00000
log(q):sl   -0.67829    0.10229 -6.6308  0.00000
log(q):pf    0.00080    0.00113  0.7133  0.47680
pl:sl        0.00007    0.00018  0.4144  0.67916
pl:pf        0.00000    0.00000  0.1421  0.88722
sl:pf        0.01680    0.03092  0.5432  0.58780
```

This suggests omitting the terms `pf`, and all interactions except `log(q):pl` and `log(q):sl`. We check that omission of these terms makes little difference to the terms that remain:

```
elec2xx.lm <- lm(log(cost) ~ log(q)+pl+sl+pf+
                 log(q):pl+log(q):sl,
                 data=Electricity)
round(coef(summary(elec2xx.lm)),5)
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -4.12003    0.33312 -12.368        0
log(q)       0.74902    0.03852  19.445        0
pl          -0.00029    0.00004  -7.755        0
sl           7.29642    0.70758  10.312        0
pf           0.02611    0.00145  18.017        0
log(q):pl    0.00003    0.00000   7.657        0
log(q):sl   -0.68969    0.09435  -7.310        0
```

Now check whether there is a strong case for adding in any further individual terms:

```
add1(elec2xx.lm, scope=~(log(q)+pl+sl+pk+sk+pf+sf)^2, test="F")
```

```
Single term additions

Model:
```

```
log(cost) ~ log(q) + pl + sl + pf + log(q):pl + log(q):sl
          Df Sum of Sq  RSS  AIC F value Pr(>F)
<none>                 2.83 -622
pk         1    0.0041 2.82 -620    0.22   0.64
sk         1    0.0329 2.79 -621    1.76   0.19
sf         1    0.0294 2.80 -621    1.58   0.21
log(q):pf  1    0.0040 2.82 -620    0.21   0.64
pl:sl      1    0.0060 2.82 -620    0.32   0.57
pl:pf      1    0.0004 2.83 -620    0.02   0.88
sl:pf      1    0.0016 2.83 -620    0.09   0.77
```

8.9 *An introduction to logistic regression*

The data that will be used for illustration are from the data frame `bronchit` in the *DAAGviz* package. The following loads packages that will be needed:

The dataset `bronchit` may alternatively be found in the *SMIR* package.

```
library(DAAGviz, quietly=TRUE)
library(KernSmooth, quietly=TRUE)
```

Figure 8.15 shows two plots – one of `poll` (pollution level) against `cig` (number of cigarettes per day), and the other of `poll` against `log(poll)`. In each case, points are identified as with or without bronchitis.

Figure 8.15: Panel A plots `poll` (pollution level) against `cig` (number of cigarettes per day). In panel B, the *x*-scale shows the logarithm of the number of cigarettes per day.

```
## Panel A
colr <- adjustcolor(c("red","blue"), alpha=0.5)
plot(poll ~ cig,
     xlab="# cigarettes per day", ylab="Pollution",
     col=colr[r+1], pch=(3:2)[r+1], data=bronchit,
     ylim=ylim)
legend(x="topright",
       legend=c("Non-sufferer","Sufferer"),
       ncol=2, pch=c(3,2), col=c(2,4), cex=0.8)
```

```
## Panel B
plot(poll ~ log(cig+1), col=c(2,4)[r+1], pch=(3:2)[r+1],
     xlab="log(# cigarettes per day + 1)", ylab="", data=bronchit, ylim=ylim)
xy1 <- with(subset(bronchit, r==0), cbind(x=log(cig+1), y=poll))
xy2 <- with(subset(bronchit, r==1), cbind(x=log(cig+1), y=poll))
est1 <- bkde2D(xy1, bandwidth=c(0.7, 3))
est2 <- bkde2D(xy2, bandwidth=c(0.7, 3))
lev <- pretty(c(est1$fhat, est2$fhat),4)
contour(est1$x1, est1$x2, est1$fhat, levels=lev, add=TRUE, col=2)
contour(est2$x1, est2$x2, est2$fhat, levels=lev, add=TRUE, col=4, lty=2)
legend(x="topright", legend=c("Non-sufferer","Sufferer"), ncol=2, lty=1:2,
```

The logarithmic transformation spreads the points out in the x-direction, in a manner that is much more helpful for prediction than the untransformed values in panel A. The contours for non-sufferer and sufferer in panel B have a similar shape. The separation between non-sufferer and sufferer is stronger in the x-direction than in the y-direction. As one indication of this, the contours at a density of 0.02 overlap slightly in the x-direction, but strongly in the y-direction.

Logistic regression calculations

Figure 8.15 made it clear that the distribution of number of cigarettes had a strong positive skew. Thus, we might fit the model:

```
cig2.glm <- glm(r ~ log(cig+1) + poll, family=binomial, data=bronchit)
summary(cig2.glm)
```

```
Call:
glm(formula = r ~ log(cig + 1) + poll, family = binomial, data = bronchit)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-1.611  -0.586  -0.362  -0.239   2.653

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  -10.7877     2.9885   -3.61  0.00031
log(cig + 1)   1.2882     0.2208    5.83  5.4e-09
poll           0.1306     0.0494    2.64  0.00817

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 221.78  on 211  degrees of freedom
Residual deviance: 168.76  on 209  degrees of freedom
AIC: 174.8

Number of Fisher Scoring iterations: 5
```

Termplots (Figure 8.16) provide a useful summary of the contributions of the covariates. For binary (0/1) data such as here, including the data values provides no visually useful information. Code is:

```
termplot(cig2.glm)
```

Figure 8.16: The panels show the contributions that the respective terms make to the fitted values (logit of probability of bronchitis), when the other term is held constant.

8.10 Exercises

1. Exercise 3 in Section 2.6.2 involved reading data into a data frame `molclock1`. Plot `AvRate` against `Myr`. Fit a regression line (with intercept, or without intercept?), and add the regression line to the plot. What interpretation can be placed upon the regression slope?

2. Attach the *DAAG* package. Type `help(elasticband)` to see the help page for the data frame `elasticband`. Plot `distance` against `stretch`. Regress `distance` against `stretch` and explain how to interpret the coefficient.

3. Repeat the calculations in Section 8.6, now examining the regression of `time` on `dist` and `climb`. Does this regression adequately model the data. Comment on the results.

4(a) Investigate the pairwise relationships between variables in the data frame `oddbooks` (*DAAG*).

(b) Fit the models

```
volume <- apply(oddbooks[, 1:3], 1, prod)
area <- apply(oddbooks[, 2:3], 1, prod)
lob1.lm <- lm(log(weight) ~ log(volume), data=oddbooks)
lob2.lm <- lm(log(weight) ~ log(thick)+log(area), data=oddbooks)
lob3.lm <- lm(log(weight) ~ log(thick)+log(breadth)+log(height),
              data=oddbooks)
```

Comment on what you find, i.e., comment both on the estimates and on the standard errors.

(c) Can `weight` be completely explained as a function of `volume`? Is there another relevant variable?

5. Repeat the calculations in Section 3.3, now with the dataset `hills2000` (*DAAG*). Do any of the points stand out as outliers? Use `predict()`, with `newdata = hills200`, to obtain predictions from the `hills2000` model for the `nihills` data. Compare with the predictions from the `nihills` model.

9

*A Miscellany of Models & Methods

This chapter is a tour through models and methods that are straightforward to fit using R. Some of these lend themselves to relatively automated use. There is some limited attention to the traps that can catch users who are unwary, or who have ventured too easily into areas that call for some greater level of statistical sophistication than their training and experience has given them.

In each case, comments with be introductory and brief. Firstly, there are brief comments on the fitting of smooth curves. The second and third topics highlight specific types of departure from the iid (independently and identically distributed) assumption.

9.1 Regression with Fitted Smooth Curves

Load the *DAAG* package:

```
library(DAAG)
```

Commentary on Smoothing Methods

Two types of methods will be described – those where the user controls the choice of smoothing parameter, and statistical learning type methods where the amount of smoothing is chosen automatically:

- The first class of methods rely on the user to make a suitable choice of a parameter that controls the smoothness. The default choice is often a good first approximation. Note here:
  - Smoothing using a "locally weighted regression smoother". Functions that use this approach include `lowess()`, `loess()`, `loess.smooth()`, and `scatter.smooth()`.
  - Use of a regression spline basis in a linear model. Here the smoothness is usually controlled by the choice of number of spline basis terms.

Smoothness is controlled by the width of the smoothing window. The default is `f=2/3` for `lowess()`, or `span=0.75` for `loess()`. For other functions that rely on this methodology, check the relevant help page.

- A second class of methods use a "statistical learning" approach in which the amount of smoothing is chosen automatically. The

approach of the *mgcv* package extends and adapts the regression spline approach.[1] The methodology generalizes to handle more general types of outcome variables, including proportions and counts. These extensions will not be further discussed here.

[1] Strong assumptions are required, notably that observations are independent. Normality assumptions are, often, less critical.

9.1.1 Locally weighted scatterplot smoothers

Locally weighted scatterplot smoothers pass a window across the data, centering the window in turn at each of a number points that are equally spaced through the data. The smooth at an x-value where the window has been centred is the predicted value from a line (or sometimes a quadratic or other curve) that is fitted to values that lie within the window. A weighted fit is used, so that neighbouring points get greater weight than points out towards the edge of the window.

Figure 9.1 shows a smooth that has been fitted using the `lowess` (locally weighted scatterplot smoothing) methodology. The default choice of with of smoothing window (a fraction $f = \frac{2}{3}$ of the total range of x) gives a result that, for these data, looks about right. The curve does however trend slightly upwards at the upper end of its range. A monotonic response might seem more appropriate.

The code used to plot the graph is:

```
## Plot points
plot(ohms ~ juice, data=fruitohms, fg="gray")
## Add smooth curve, using default
## smoothing window
with(fruitohms,
     lines(lowess(ohms ~ juice), col="gray", lwd=2))
```

Figure 9.1: Resistance in ohms is plotted against apparent juice content. A smooth curve (in gray) has been added, using the `lowess` smoother. The width of the smoothing window was the default fraction $f = \frac{2}{3}$ of the range of values of the x-variable.

A more sophisticated approach uses the `gam()` function in the *mgcv* package. This allows automatic determination of the amount of smoothing, providing the assumption of independent residuals from the curve is reasonable. We now demonstrate the use of a GAM model for a two-dimensional smooth.

9.1.2 Contours from 2-dimensional Smooths

Data are the amplitudes of responses to a visual stimulus, for each of 20 individuals, at different regions of the left eye. We use the function `gam()` to create smooth surfaces, for males and females separately. Figure 9.2 then uses the function `vis.gam()` to plot heatmaps that show the contours:

The GAM fit will as far as possible use the smooth surface to account for the pattern of variation across the eye, with residuals from the surface treated as random normal noise.

```
## Code
library(DAAGviz)
library(mgcv)
eyeAmpM.gam <- gam(amp ~ s(x,y), data=subset(eyeAmp, Sex=="m"))
eyeAmpF.gam <- gam(amp ~ s(x,y), data=subset(eyeAmp, Sex=="f"))
```

Figure 9.2: Estimated contours of left eye responses to visual stimulae,

```
lims <- range(c(predict(eyeAmpF.gam), predict(eyeAmpM.gam)))
vis.gam(eyeAmpM.gam, plot.type='contour', color="cm", zlim=lims, main="")
mtext(side=3, line=0.5, adj=0, "A: Response amplitudes, Males")
vis.gam(eyeAmpF.gam, plot.type='contour', color="cm", zlim=lims, main="")
mtext(side=3, line=0.5, adj=0, "B: Response amplitudes, Females")
```

9.2 *Hierarchical Multi-level Models*

Models with Non-iid Errors – Multi-level models:	
Error Term	Errors do not have to be (and often are not) iid
Multi-level models	Multi-level models are a (relatively) simple type of non-iid model, implemented using `lme()` (*nlme*) or `lmer()` (*lme4* package). Such models allow different errors of prediction, depending on the intended prediction.

Figure 9.3: Yields from 4 packages of land on each of eight sites on the Caribbean island of Antigua. Data are a summarized version of a subset of data given in Andrews and Herzberg 1985, pp.339-353.

Figure 9.3 shows corn yield data from the Caribbean island of Antigua, as in the second column ("Yields") of Table 9.1. Each value is for one package of land. The code for the figure is:

```
# ant111b is in DAAG
Site <- with(ant111b, reorder(site, harvwt,
                              FUN=mean))
stripplot(Site ~ harvwt, data=ant111b, fg="gray",
          scales=list(tck=0.5),
          xlab="Harvest weight of corn")
```

Depending on the use that will be made of the results, it may be essential to correctly model the structure of the random part of the model. In comparing yields from different packages of land, there are two sorts of comparison. Packages on the same location should be relatively similar, while packages on different locations should

Location	Yields	Location effect		Residuals from location mean
DBAN	5.16, 4.8, 5.07, 4.51	(4.29)	+0.59	0.28, −0.08, 0.18, −0.38
LFAN	2.93, 4.77, 4.33, 4.8		−0.08	−1.28, 0.56, 0.12, 0.59
NSAN	1.73, 3.17, 1.49, 1.97		−2.2	−0.36, 1.08, −0.6, −0.12
ORAN	6.79, 7.37, 6.44, 7.07		+2.62	−0.13, 0.45, −0.48, 0.15
OVAN	3.25, 4.28, 5.56, 6.24		+0.54	−1.58, −0.56, 0.73, 1.4
TEAN	2.65, 3.19, 2.79, 3.51		−1.26	−0.39, 0.15, −0.25, 0.48
WEAN	5.04, 4.6, 6.34, 6.12		+1.23	−0.49, −0.93, 0.81, 0.6
WLAN	2.02, 2.66, 3.16, 3.52		−1.45	−0.82, −0.18, 0.32, 0.68

Table 9.1: The leftmost column has harvest weights (`harvwt`), for the packages of land in each location, for the Antiguan corn data. Each of these harvest weights can be expressed as the sum of the overall mean (= 4.29), location effect (third column), and residual from the location effect (final column).

be relatively more different, as Figure 9.3 suggests. A prediction for a new package at one of the existing locations is likely to be more accurate than a prediction for a totally new location.

Multi-level models are able to account for such differences in predictive accuracy. For the Antiguan corn yield data, it is necessary to account both for variation within sites and for variation between sites. The R packages `nlme` and `lme4` are both able to handle such data.

Because of the balance the corn yield data, an analysis of variance that specifies a formal `Error` term is an alternative to the fitting of a multi-level model.

9.3 Regular Time Series in R

Models with Non-iid Errors – Time Series:

Time sequential	Points that are close together in time commonly show a (usually, +ve) correlation. R's `acf()` and `arima()` functions are powerful tools for use with time series.

Any process that evolves in time is likely to have a sequential correlation structure. The value at the current time is likely to be correlated with the value at the previous time, and perhaps with values several time points back. The discussion that follows will explore implications for data analysis.

9.3.1 Example – the Lake Erie data

Figure 9.4: Lake Erie levels (m).

```
Erie <- greatLakes[,"Erie"]
plot(Erie, xlab="", fg="gray",
     ylab="Level (m)")
```

The data series `Erie`, giving levels of Lake Erie from 1918 to 2009, will be used as an example from which to start the discussion.[2] The series is available in the *DAAG* package, as the column `Erie` in the multivariate time series object `greatLakes`.

Figure 9.4 shows a plot of the series.

[2] Data are from `http://www.lre.usace.army.mil/greatlakes/hh/greatlakeswaterlevels/historicdata/greatlakeshydrographs/`

Figure 9.5: Panel A plots Lake Erie levels vs levels at lags 1, 2 and 3 respectively. Panel B shows a consistent pattern of decreasing autocorrelation at successive lags.

```
## Panel A
lag.plot(Erie, lags=3,
         do.lines=FALSE,
         layout=c(1,3), fg="gray",
```

```
## Panel B
acf(Erie, main="", fg="gray")
```

The plots in Figure 9.5 are a good starting point for investigation of the correlation structure. Panel A shows lag plots, up to a lag of 3. Panel B shows estimates of the successive correlations, in this context are called autocorrelations.

There is a strong correlation at lag 1, a strong but weaker correlation at lag 2, and a noticeable correlation at lag 3. Such a correlation pattern is typical of an autoregressive process where most of the sequential dependence can be explained as a flow-on effect from a dependence at lag 1.

If possible, the analyst will want to find covariates that largely or partly explain that dependence. At best, such covariates will commonly explain part only of the dependence, and there will remain dependence that requires to be modeled.

In an autoregressive time series, an independent error component, or "innovation" is associated with each time point. For an order p autoregressive time series, the error for any time point is obtained by taking the innovation for that time point, and adding a linear combination of the innovations at the p previous time points. (For the present time series, initial indications are that $p = 1$ might capture most of the correlation structure.)

An autoregressive model is a special case of an Autoregressive Moving Average (ARMA) model.

9.3.2 *Patterns that are repeatable*

What sorts of patterns may then be repeatable? Indications that a pattern may be repeatable include:

- A straight line trend is a good starting point for some limited extrapolation. But think: Is it plausible that the trend will continue more than a short distance into the future?
- There may be a clear pattern of seasonal change, e.g., with seasons of the year or (as happens with airborne pollution) with days of the week. If yearly seasonal changes persist over different years, or weekly day-of-the-week changes persist over different weeks, these effects can perhaps be extrapolated with some reasonable confidence.
- There is a regression relationship that seems likely to explain future as well as current data.

An ideal would be to find a covariate or covariates than can largely explain the year to year changes. For this series, this does not seem a possibility. In the absence of identifiable direct cause for the year to year changes, a reasonable recourse is to look for a correlation structure that largely accounts for the pattern of the year to year change.

Smoothing terms can be fitted to the pattern apparent in serially correlated data, leaving *errors* that are pretty much uncorrelated. Such a pattern is in general, however, unrepeatable. It gives little clue of what may happen the future. A re-run of the process (a new *realization*) will produce a different series, albeit one that shows the same general tendency to move up and down.

Smooth, with automatic choice of smoothing parameter

Figure 9.6: GAM smoothing term, fitted to the Lake Erie Data. Most of the autocorrelation structure has been removed, leaving residuals that are very nearly independent.

```
## Code
library(mgcv)
df <-  data.frame(
  height=as.vector(Erie),
  year=time(Erie))
obj <- gam(height ~ s(year),
           data=df)
plot(obj, fg="gray",
     shift=mean(df$height),
     residuals=TRUE, pch=1,
     xlab="",
     ylab="Height of lake")
```

While smoothing methods that asssume independent errors can be used, as in Figure 9.6, to fit a curve to such data, the curve will not be repeatable. Figure 9.6 does not separate systematic effects from effects due to processes that evolve in time. Figure 9.6 uses the abilities of the *mgcv* package, assuming independently and identically distributed data (hence, no serial correlation!) to make an automatic choice of the smoothing parameter. As the curve is conditional on a particular realization of the process that generated it, its usefulness is limited.

The pointwise confidence limits are similarly conditioned, relevant perhaps for interpolalation given this particular realization. All that is repeatable, given another realization, is the process that generated the curve, not the curve itself.

9.3.3 *Fitting and use of an autoregressive model*

There are several different types of time series models that may be used to model the correlatios structure, allowing realistic estimates of the lake level a short time ahead, with realistic confidence bounds around those estimates. For the Lake Erie data, an autoregressive correlation structure does a good job of accounting for the pattern of change around a mean that stays constant.

Figure 9.5 suggested that a correlation between each year and the previous year accounted for the main part of the autocorrelation structure in Figure 9.4. An AR1 model (autoregressive with a correlation at lag 1 only), which we now fit, formalizes this.

```
ar(Erie, order.max=1)
```

```
Call:
ar(x = Erie, order.max = 1)

Coefficients:
    1
0.851

Order selected 1  sigma^2 estimated as  0.0291
```

The one coefficient that is now given is the lag 1 correlation, equalling 0.851.

Figure 9.7: The plots are from repeated simulations of an AR1 process with a lag 1 correlation of 0.85. Smooth curves, assuming independent errors, have been fitted.

```
for (i in 1:6){
ysim <-
  arima.sim(list(ar=0.85),
            n=200)
df <- data.frame(x=1:200,
                 y=ysim)
df.gam <- gam(y ~ s(x),
              data=df)
plot(df.gam, fg="gray",
     ylab=paste("Sim", i),
     residuals=TRUE)
}
```

Figure 9.7 then investigates how repeated simulations of this process, with a lag 1 correlation of 0.0.85, compare with Figure 9.4. This illustrates the point that a GAM smooth will extract, from an autoregressive process with mean 0, a pattern that is not repeatable when the process is re-run.

The curves are different on each occasion. For generalization beyond the particular realization that generated them, they serve no useful purpose.

Once an autoregressive model has been fitted, the function `forecast()` in the *forecast* package can be used to predict future levels, albeit with very wide confidence bounds. For this, it is necessary to refit the model using the function `arima()`. An arima model with order (1,0,0) is an autoregressive model with order 1.

Figure 9.8: Predictions, 15 years into the future, of lake levels (m). The shaded areas give 80% and 95% confidence bounds.

```
erie.ar <- arima(Erie,
            order=c(1,0,0))
library(forecast)
fc <- forecast(erie.ar,
               h=15)
plot(fc, main="", fg="gray",
     ylab="Lake level (m)")
  # 15 time points ahead
```

This brief excursion into a simple form of time series model is intended only to indicate the limitations of automatic smooths. and to give a sense of the broad style of time series modeling. The list of references at the end of the chapter has details of several books on time series.

9.3.4 Regression with time series errors

Figure 9.9 fits annual rainfall, in the Murray-Darling basin of Australia, as a sum of smooth functions of `Year` and `SOI`. Figure 3.9 shows the estimated contributions of the two model terms.

```
## Code
mdbRain.gam <- gam(mdbRain ~ s(Year) + s(SOI),
                   data=bomregions)
plot(mdbRain.gam, residuals=TRUE, se=2, fg="gray",
     pch=1, select=1, cex=1.35, ylab="Partial, Year")
mtext(side=3, line=0.75, "A: Effect of Year", adj=0)
plot(mdbRain.gam, residuals=TRUE, se=2, fg="gray",
     pch=1, select=2, cex=1.35, ylab="Partial, SOI")
mtext(side=3, line=0.75, "B: Effect of SOI", adj=0)
```

The left panel indicates a consistent pattern of increase of rainfall with succeeding years, given an adjustment for the effect of SOI. Errors from the fitted model are consistent with the independent errors assumption. The model has then identified a pattern of increase of rainfall with time, given SOI, that does seem real. It is necessary to warn against reliance on extrapolation more than a few time points into the future. While the result is consistent with expected effects from global warming, those effects are known to play out very differently in different parts of the globe.

Figure 9.9: Estimated contributions of model terms to `mdbRain`, in a GAM model that adds smooth terms in `Year` and `Rain`. The dashed curves show pointwise 2-SE limits, for the fitted curve.

Investigation of the residual error structure

Sequential correlation structures are often effective, with data collected over time, for use in modeling departure from iid errors. Where there is such structure structure in the data, the methodology will if possible use a smooth curve to account for it.

The residuals can be checked to determine whether the fitted curve has removed most of the correlation structure in the data. Figure 9.10 shows the autocorrelation function of the residuals, followed by autocorrelation functions for several series of independent random normal numbers. Apart from the weakly attested correlation at a lag of 12 years, which is a commonplace of weather data, the pattern of sequential correlation is not much different from what can be expected in a sequence of independent random normal numbers.

Figure 9.10: The top left panel shows the autocorrelations of the residuals from the model `mdbRain.gam`. The five remaining panels are the equivalent plots for sequences of independent random normal numbers.

Code is:

```
mdbRain.gam <- gam(mdbRain ~ s(Year) + s(SOI),
                   data=bomregions)
n <-  dim(bomregions)[1]
acf(resid(mdbRain.gam), ylab="MDB series")
```

```
for(i in 1:5)acf(rnorm(n), ylab=paste("Sim",i),
                 fg="gray", col="gray40")
```

9.3.5 *Box-Jenkins ARIMA Time Series Modeling

From the perspective of the Box-Jenkins ARIMA (Autoregressive Integrated Moving Average) approach to time series models, autoregressive models are a special case. Many standard types of time series can be modeled very satisfactorily as ARIMA processes.

Models that are closely analagous to ARIMA models had been used earlier in control theory. ARIMA models are feedback systems!

Exercise

The simulations in Figure 9.7 show a pattern of variation that seems not too different from that in the actual series. Modeling of the process as an ARMA or ARIMA process (i.e., allow for a moving average term) may do even better. Use the `auto.arima()` function in the *forecast* package to fit an ARIMA process:

9.3.6 Count Data with Poisson Errors

Data is for aircraft accidents, from the website `http://www.planecrashinfo.com/`. The 1920 file has accidents starting from 1908. The full data are in the dataset `gamclass::airAccs`. Such issues as there are with sequential correlation can be ameliorated by working with weekly, rather than daily, counts.

Data are a time series. Serious accidents are however sufficently uncommon that occasions where events occur together, or where one event changes the probability of the next event, seem likely to be uncommon.

Figure 9.11: Estimated number of events (aircraft crashes) per week, versus time. The yearly tick marks are for January 1 of the stated year. See Section 4.3.9 for further details on the function `eventCounts()`.

Figure 9.11 shows a fitted smooth curve, with pointwise confidence bounds, from a GAM smoothing model that was fitted to the weekly counts.

The function `gamclass::eventCounts()` was used to create weekly counts of accidents from January 1, 2006:

```
## Code
airAccs <- gamclass::airAccs
fromDate <- as.Date("2006-01-01")
dfWeek06 <- gamclass::eventCounts(airAccs, dateCol="Date",
                                  from=fromDate,
                                 by="1 week", prefix="num")
dfWeek06$day <- julian(dfWeek06$Date, origin=fromDate)
```

Code for Figure 9.11 is then.

```
## Code
library(mgcv)
year <- seq(from=fromDate, to=max(dfWeek06$Date), by="1 year")
at6 <- julian(seq(from=fromDate, to=max(dfWeek06$Date), by="6 months"), origin=fromDate)
atyear <- julian(year, origin=fromDate)
dfWeek06.gam <- gam(num~s(day, k=200), data=dfWeek06, family=quasipoisson)
avWk <- mean(predict(dfWeek06.gam))
plot(dfWeek06.gam, xaxt="n", shift=avWk, trans=exp, rug=FALSE,
     xlab="", ylab="Estimated rate per week", fg="gray")
axis(1, at=atyear, labels=format(year, "%Y"), lwd=0, lwd.ticks=1)
abline(h=0.5+(1:4)*0.5, v=at6, col="gray", lty=3, lwd=0.5)
```

The argument 'k' to the function `s()` that sets up the smooth controls the temporal resolution. A large `k` allows, if the data seem to justify it, for fine resolution. A penalty is applied that discriminates against curves that are overly "wiggly".

Not all count data is suitable for modeling assuming a Poisson type rare event distribution. For example, the dataset `http://maths-people.anu.edu.au/~johnm/stats-issues/data/hurric2014.csv` has details, for the years 1950-2012, of US deaths from Atlantic hurricanes. For any given hurricane, deaths are not at all independent rare events.

9.4 Classification

Classification models have the character of regression models where the outcome is categorical, one of g classes. The `fgl` (forensic glass) dataset that will be used as an example has measurements of each on nine physical properties, for 214 samples of glass that are classified into $g = 6$ different glass types.

For the special case $g = 2$, logistic regression models are an alternative.

This section will describe a very limited range of available approaches. For details on how and why these methods work, it will be necessary to look elsewhere.[3]

[3] Limited further details and references are provided in Maindonald and Braun: *Data Analysis and Graphics Using R*, Cambridge University Press, 3rd edn 2010.

Linear discriminant analysis (LDA), and quadratic discriminant analysis (QDA) which slightly generalizes LDA, both use linear functions of the explanatory variables in the modeling of the probabilities of group membership. These methods will be contrasted with the strongly non-parameteric approaches of tree-based classification and of random forests.

Linear and quadratic discriminant analysis

The functions that will be used are `lda()` and `qda()`, from the *MASS* package. The function `lda()` implements linear discriminant analysis, while `qda()` implements quadratic discriminant analysis.[4]

[4] Quadratic discriminant analysis is an adaptation of linear discriminant analysis to handle data where the variance-covariance matrices of the different classes are markedly different.

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

Results from use of `lda()` lead very naturally to useful and informative plots. Where `lda()` gives results that are a sub-optimal fit to the data, the plots may hint at what type of alternative method

may be preferable. They may identify subgroups of the orginal g groups, and/or identify points that seem misclassified.

An attractive feature of `lda()` is that the discriminant rule that is obtained has a natural representation r-dimensional space. Providing that there is sufficient independent covariate data, $r = g - 1$. The analysis leads[5] to r sets of scores, where each set of scores explains a successively smaller (or at least, not larger) proportion of the sum of squares of differences of group means from the overall mean. The r sets of scores can be examined using a pairs plot. With larger numbers of groups, it will often happen that two or at most three dimensions will account for most of the variation.

[5] This is based on a *spectral* decomposition of the model matrix.

With three groups, two dimensions will account for all the variation. A scatterplot is then a geometrically complete representation of what the analysis has achieved.

Use of `lda()` to analyse the forensic glass data

As noted above, the data frame `fgl` has 10 measured physical characteristics for each of 214 glass fragments that are classified into 6 different types. First, fit a linear discriminant analysis, and use leave-one-out cross-validation to check the accuracy, thus:

```
fglCV.lda <- lda(type ~ ., data=fgl, CV=TRUE)
tab <- table(fgl$type, fglCV.lda$class)
## Confusion matrix
print(round(apply(tab, 1, function(x)x/sum(x)),
            digits=3))
```

```
        WinF WinNF   Veh   Con  Tabl  Head
  WinF  0.729 0.237 0.647 0.000 0.111 0.034
  WinNF 0.229 0.684 0.353 0.462 0.222 0.069
  Veh   0.043 0.000 0.000 0.000 0.000 0.000
  Con   0.000 0.039 0.000 0.462 0.000 0.034
  Tabl  0.000 0.026 0.000 0.000 0.556 0.000
  Head  0.000 0.013 0.000 0.077 0.111 0.862
```

The function `confusion()` (*DAAG*) makes it easy to get all the above output. Enter:

```
library(DAAG)
confusion(fgl$type, fglCV.lda$class)
```

Two-dimensional representation

Now fit the model with `CV=FALSE`, which is the default:

```
fgl.lda <- lda(type ~ ., data=fgl)
```

The final three lines of the output, obtained by entering `fgl.lda` at the command line, are:

```
Proportion of trace:
  LD1   LD2   LD3   LD4   LD5
0.815 0.117 0.041 0.016 0.011
```

The numbers show the successive proportions of a measure of the

Observe that most of the discriminatory power is in the first two dimensions.

Figure 9.12: Visual representation of scores from a *linear discriminant analysis*, for the forensic glass data. A six-dimensional pattern of separation between the categories has been collapsed to two dimensions. Some categories may therefore be better distinguished than is evident from this figure.

variation that are accounted for by projections onto spaces with successively larger numbers of dimensions.

Figure 9.12 shows the two-dimensional representation.

```
library(lattice)
scores <- predict(fgl.lda)$x
xyplot(scores[,2] ~ scores[,1], groups=fgl$type,
       xlab="Discriminant 1",
       ylab="Discriminant 2",
       aspect=1, scales=list(tck=0.4),
       auto.key=list(space="right"),
```

Additionally, it may be useful to examine the plot of the third versus the second discriminant. Better still, use the abilites of the *rgl* package to examine a 3-dimensional dynamic representation. With most other methods, a low-dimensional representation does not arise so directly from the analysis.

See Figure 9.15 in Subsection 9.5.1, for an example of the type of low-dimensional representation that is possible for results from a randdom forest classification.

Two groups – comparison with logistic regression

The approach is to model the probability of class membership given the covariates, using the same logistic fixed part of the model as for linear and quadratic discriminant analysis. With π equal to the probability of membership in the second class, the model assumes that

$$\log(\pi/(1-\pi) = \beta'\mathbf{x}$$

where β is a vector of coefficients that are to be estimated, and $\mathbf{x}$ is a vector of covariate values.

A logistic regression model is a special case of a Generalized Linear Model (GLM), as implemented by R's function `glm()`. There is no provision to adjust predictions to take account of prior probabilities, though this can be done as an add-on to the analysis. Other points of difference from linear discriminant analysis are:

More technical points, as they apply to the use of R's function `glm()` for logistic regression, are:

- The fitting procedure minimizes the *deviance*. This equals 2 (*loglikelihood* for fitted model, minus the loglikelihood for the 'saturated' model). The 'saturated model has predicted values equal to observed values.
- Standard errors and Wald statistics (roughly comparable to t-statistics) are given for parameter estimates. These depend on approximations that may fail if predicted proportions are close to 0 or 1 and/or the sample size is small.

- Inference is conditional on the observed covariate values. A model for the probability of covariate values $\mathbf{x}$ given the class c, as for linear discriminant analysis, is not required. (Linear discriminant analysis assumes a multivariate normal distribution assumptions for $\mathbf{x}$, given the class c. In practice, results seem relatively robust against failure of this assumption.)
- The logit model uses the *link* function $f(\pi) = \log(\pi/(1-\pi))$. Other choices of link function are available. Where there are sufficient data to check whether one of these other links may be more appropriate, this should be checked. Or there may be previous experience with comparable data that suggests use of a link other than the logit.
- Observations can be given prior weights.

9.5 *Tree-based methods and random forests*

On a scale in which highly parametric methods lie at one end and highly non-parametric methods at the other, linear discriminant methods lie at the parametric end, and tree-based methods and random forests at the non-parametric extreme. An attraction of tree-based methods and random forests is that model choice can be pretty much automated.

We begin by loading the *rpart* package:

```
library(rpart)
```

For the calculations that follow, data are columns in the data frame `bronchit`, in the *DAAGviz* package.

The dataset `bronchit` may alternatively be found in the *SMIR* package.

```
head(bronchit, 3)
```

Here `r=1` denotes bronchitis, while `r=0` indicates that bronchitis is absent.

```
  r  cig poll
1 0 5.15 67.1
2 0 6.75 64.4
3 0 0.00 65.9
```

In place of the variable `r` with values 0 and 1, we use a factor with levels `abs` and `pres`. Labels that appear in the output are then more meaningful.

```
## Now make the outcome variable a factor
bronchit <-
  within(bronchit,
         rfac <- factor(r, labels=c("abs","pres")))
```

The following fits a tree-based model:

With a factor (`rfac`) as outcome, `method="class"` is the default. Setting `method="class"`, to make it quite clear that we are using a splitting rule that is appropriate to a categorical (rather than continuous) outcome, is good practice.

```
set.seed(47)   # Reproduce tree shown
b.rpart <- rpart(rfac ~ cig+poll, data=bronchit,
                 method="class")
```

The "complexity" paremeter `cp`, by default set to 0.01, controls how far splitting continues. In practice, it is usual to set `cp` small

enough that splitting continues further than is optimal, then pruning the tree back. Cross-validation based accuracies are calculated at each split, and can be used to determine the optimal depth of tree. Details will not be given at this point, as the interest is in trees as a lead-in to random forests. For random forests, the depth of the splits in individual trees is not of great consequence — it is not important for individual trees to be optimal.

Figure 9.13 is a visual summary of results from the tree-based classification, designed to predict the probability that a miner will have bronchitis. Where the condition at a node is satisfied, the left branch is taken. Thus, at the initial node, `cig<4.385` takes the branch to the left. In general (no random number seed), the tree may be different for each different run of the calculations.

Tree-based classification proceeds by constructing a sequence of decision steps. At each node, the split is used that best separates the data into two groups. Here (Figure 9.13) tree-based regression does unusually well (CV accuracy = 97.2%), perhaps because it is well designed to reproduce a simple form of sequential decision rule that has been used by the clinicians.

How is 'best' defined? Splits are chosen so that the Gini index of "impurity" is minimized. Other criteria are possible, but this is how `randomForest()` constructs its trees.

Figure 9.13: Decision tree for predicting whether a miner has bronchitis.

Code for Figure 9.13 is:

```
plot(b.rpart)
text(b.rpart, xpd=TRUE)
```

9.5.1 *Random forests*

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

Figure 9.14 shows trees that have been fitted to different bootstrap samples of the bronchitis data. Typically 500 or more trees are fitted, without a stopping rule. Individual trees are likely to overfit. As each tree is for a different random sample of the data, there is no overfitting overall.

Figure 9.14: Each tree is for a different bootstrap sample of observations. The final classification is determined by a random vote over all trees. Where there are > 2 explanatory variables (but not here) a different random sample of variables is typically used for each different split. The final classification is determined by a random vote over all trees.

For each bootstrap sample, predictions are made for the observations that were not included – i.e., for the out-of-bag data. Com-

parison with the actual group assignments then provides an unbiased estimate of accuracy.

For the `bronchit` data, here is the `randomForest()` result.

```
(bronchit.rf <- randomForest(rfac ~ cig+poll,
                             data=bronchit))
```

```
Call:
 randomForest(formula = rfac ~ cig + poll, data = bronchit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 22.64%
Confusion matrix:
     abs pres class.error
abs  146   20      0.1205
pres  28   18      0.6087
```

The accuracy is much better than the `rpart()` accuracy. The random forest methodology will often improve, sometimes quite dramatically, on tree-based classification.

Figure 9.15 is a visual summary of the random forest classification result. The proportion of trees in which any pair of points appear together at the same node may be used as a measure of the "proximity" between that pair of points. Then, subtracting proximity from one to obtain a measure of distance, an ordination method is used to find an approximates representation of those points in a low-dimensional space.

There is a tuning parameter `mtry` which controls the number of randomly chosen variables considered for each tree. This is not too much of an issue for the present data, where there are only two explanatory variables.

Code for Figure 9.15 is:

```
parset <- simpleTheme(pch=1:2)
bronchit.rf <- randomForest(rfac ~ cig+poll,
                            proximity=TRUE,
                            data=bronchit)
points <- cmdscale(1-bronchit.rf$proximity)
xyplot(points[,2] ~ points[,1],
       groups=bronchit$rfac,
       xlab="Axis 1", ylab="Axis 2",
       par.settings=parset, aspect=1,
       auto.key=list(columns=2))
```

Figure 9.15: The plot is designed to represent, in two dimensions, the random forest result. It aims to reflect probabilities of group membership given by the analysis. It is not derived by a 'scaling' of the feature space.

A random forest fit to the forensic glass data

The algorithm can be used in a highly automatic manner. Here then is the random forest analysis for the forensic glass data, leaving the tuning parameter (`mtry`) at its default[6]:

```
(fgl.rf <- randomForest(type ~ ., data=fgl))
```

[6] The default is to set `mtry` to the square root of the total number of variables, rounded up to an integral value.

```
Call:
 randomForest(formula = type ~ ., data = fgl)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 20.09%
Confusion matrix:
      WinF WinNF Veh Con Tabl Head class.error
WinF    63     6   1   0    0    0      0.1000
WinNF    9    59   1   4    2    1      0.2237
Veh      7     3   7   0    0    0      0.5882
Con      0     3   0   9    0    1      0.3077
Tabl     0     2   0   0    7    0      0.2222
Head     1     2   0   0    0   26      0.1034
```

This improves substantially on the linear discriminant result. This may happen because the explanatory variables have effects that are non-linear on a logit scale. The more likely reason is that there are interaction effects, perhaps of a relatively complicated kind, for which the `lda()` analysis has not accounted.

The predictive accuracy might be further improved by varying the tuning parameter `mtry` from its default. See `help(tuneRF)` for details of the function `tuneRF()` that is designed to assist in findind=g the optimum choice of `mtry`.

9.6 *Ordination

From Australian road travel distances between cities and larger towns, can we derive a plausible "map", or "ordination", showing the relative locations? The resulting "map" would give a better indication than a geographical map of the road travel effort involved in getting from one place to another.

An ordination might alternatively be based on road travel times, or on air travel times.

Genomic data provides another example. Various methods are available for calculating genomic "distances " between, e.g., different insect species. The distance measures are based on evolutionary models that aim to give distances between pairs of species that are a monotone function of the time since the two species separated.

The ordination methods described here are all versions of multi-dimensional scaling (MDS). If distances are not already given, a first tasl is to calculate 'distances' between points. Or if similarities are given, they must be first be transformed into 'distances'.

Ordination is a generic name for methods for providing a low-dimensional view of points in multi-dimensional space, such that "similar" objects are near each other and dissimilar objects are separated. The plot(s) from an ordination in 2 or 3 dimensions may provide useful visual clues on clusters in the data and/or on outliers.

One standard type of problem starts from a matrix $\mathbf{X}$ of n observations by p variables, then seeking a low-dimensional representation. A first step is then to calculate distances between observations.[7] The hope is that a major part of the information content in the p variables, as it relates to the separation between observations, can be pretty much summarized in a small number of constructed

[7] Principal components analysis circumvents the calculation of distances, for the commonly used Euclidean distance measure. See below.

variables.

There is typically no good model, equivalent to the evolutionary models used by molecular biologists, that can be used to motivate distance calculations. There is then a large element of arbitrariness in the distance measure used. Results may depend strongly on the distance measure used. Unless measurements are comparable (e.g., relative growth, as measured perhaps on a logarithmic scale, for different body measurements), it is usually desirable to calculate distances from standardized variable values. This is done by subtracting the mean and dividing by the standard deviation.

If data can be separated into known classes that should be reflected in any ordination, then the scores from classification using `lda()` may be a good basis for an ordination. Plots in 2 or perhaps 3 dimensions may then reveal additional classes and/or identify points that may be misclassified and/or are in some sense outliers. They give an indication of the effectiveness of the discrimination method in choosing the boundaries between classes.

Figure 9.15 demonstrated the use of "proximities" that are available from `randomForest()` as measures of the closeness of any pair of points. These were then turned into rough distance measures that then formed the basis for an ordination. With Support Vector Machines, distance measures can be derived from the 'decision values' and used for ordination.

9.6.1 Distance measures

Euclidean distances

Treating the rows of $\mathbf{X}$ (n by p) as points in a p-dimensional space, the squared Euclidean distance d_{ij}^2 between points i and j is

$$d_{ij}^2 = \sum_{k=1}^{p} (x_{ik} - x_{jk})^2$$

The distances satisfy the triangle inequality[8]

$$d_{ij} \leq d_{ik} + d_{kj}$$

[8] This says that a straight line is the shortest distance between two points!

The columns may be weighted differently.[9] Use of an unweighted measure with all columns scaled to a standard deviation of one is equivalent to working with the unscaled columns and calculating d_{ij}^2 as

[9] More generally, they can be arbitrarily transformed before calculating the d_{ij}.

$$d_{ij}^2 = \sum_{k=1}^{p} w_{ij}(x_{ik} - x_{jk})^2$$

where $w_{ij} = (s_i s_j)^{-1}$ is the inverse of the product of the standard deviations for columns i and j.

Where all elements of a column are positive, use of the logarithmic transformation is common. A logarithmic scale makes sense for

biological morphometric data, and for other data with similar characteristics. For morphometric data, the effect is to focus attention on relative changes in the various body proportions, rather than on absolute magnitudes.

Non-Euclidean distance measures

Euclidean distance is one of many possible choices of distance measures, still satisfying the triangle inequality. As an example of a non-Euclidean measure, consider the Manhattan distance. The Manhattan distance is the shortest distance for a journey that always proceeds along one of the co-ordinate axes. In Manhattan in New York, streets are laid out in a rectangular grid. This is then (with $k = 2$) the walking distance along one or other street. For other choices, see the help page for the function `dist()`.[10]

For the Manhattan distance:

$$d_{ij} = \sum_{k=1}^{p} | x_{ik} - x_{jk} |$$

[10] The function `daisy()` in the *cluster* package offers a wider choice, including distance measures for factor or ordinal data. Its argument `stand` causes prior standardization of data.

From distances to a representation in Euclidean space

Irrespective of the method of calculation of the distance measure, ordination methods yield a representation in Euclidean space. It is always possible to find a configuration X in Euclidean space in which the "distances" are approximated, perhaps rather poorly.[11] It will become apparent in the course of seeking the configuration whether an exact embedding (matrix X) is possible, and how accurate this embedding is. The representation is not unique. The matrices X and $\mathsf{X}\mathbf{P}$, where $\mathbf{P}$ is an orthonormal matrix, give exactly the same distances.

[11] This is true whether ot not the triangle inequality is satisfied.

The connection with principal components

Let $\mathbf{X}$ be an n by p matrix that is used for the calculation of Euclidean distances, after any transformations and/or weighting. Then metric p-dimensional ordination, applied to Euclidean distances between the rows of $\mathbf{X}$, yields a representation in p-dimensional space that is formally equivalent to that derived from the use of principal components. The function `cmdscale()` yields, by a different set of matrix manipulations, what is essentially a principal components decomposition. Principal components circumvents the calculation of distances.

We assume that none of the columns can be written as a linear combination of other columns.

Semi-metric and non-metric scaling

Semi-metric and non-metric methods all start from "distances", but allow greater flexibility in their use to create an ordination. The aim is to represent the "distances" in some specified number of dimensions, typically two dimensions. As described here, a first step is to treat the distances as Euclidean, and determine a configuration in Euclidean space. These Euclidean distances are then used as a

The assumption of a Euclidean distance scale is a convenient starting point for calculations. An ordination that preserves relative rather than absolute distances can often be more appropriate. Additionally, small distances may be measured more accurately than large distances.

starting point for a representation in which the requirement that these are Euclidean distances, all determined with equal accuracy, is relaxed. The methods that will be noted here are Sammon scaling and Kruskal's non-metric multidimensional scaling.

Example – Australian road distances

The distance matrix that will be used is in the matrix `audists`, from the `DAAG` package. Figure 9.16 is from the use of classical multi-dimensional scaling, as implemented in the function `cmdscale()`: An alternative way to add names of cities or other labels is to use `identify()` to add labels interactively, thus:

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

Code is:

```
aupts <- cmdscale(audists)
plot(aupts, axes=FALSE, ann=FALSE, fg="gray",
     frame.plot=TRUE)
city <- rownames(aupts)
pos <- rep(1,length(city))
pos[city=="Melbourne"]<- 3
pos[city=="Canberra"] <- 4
par(xpd=TRUE)
text(aupts, labels=city, pos=pos)
par(xpd=FALSE)
```

Classical multi-dimensional scaling, as implemented by `cmdscale()`, gives long distances the same weight as short distances. It is just as prepared to shift Canberra around relative to Melbourne and Sydney, as to move Perth. It makes more sense to give reduced weight to long distances, as is done by `sammon()` (*MASS*).

Figure 9.16: Relative locations of Australian cities, derived from road map distances, using metric scaling.

Figure 9.17 shows side by side the overlays of the "maps" that result from the different ordinations, onto a physical map of Australia. Panel A shows the result for classical multi-dimensional scaling.

Figure 9.17: In Panel A, Figure 9.16 has been linearly transformed, to give a best fit to a map of Australia. Each city moves as shown by the line that radiates out from it. Panel B is the equivalent plot for Sammon scaling.

Panel B does the same, now for the result from Sammon scaling. Notice how Brisbane, Sydney, Canberra and Melbourne now maintain their relative positions better.

To see the code for Figure 9.17, source the file that has the code for the figures of this chapter, and type:

```
fig12.15A
fig12.15B
```

The exercise can be repeated for multidimensional scaling (MDS). MDS preserves only, as far as possible, the relative distances. A starting configuration of points is required. This might come from the configuration given by `cmdscale()`. For the supplementary figure `supp12.1()` that shows the MDS configuration, however, we use the physical latitudes and longitudes.

To show this figure, source the file, if this has not been done previously, that has the code for the figures of this chapter. Then type:

```
supp12.1()
```

References

Cowpertwait P. S. P. and Metcalfe A. V. 2009. *Introductory Time Series with R.* Springer.

Hyndman, R. J.; Koehler, A. B.; Od, J. K.; and Snyder, R. D. 2008. *Forecasting with Exponential Smoothing: The State Space Approach*, 2nd edn, Springer.

Taleb, Naseem. 2004. *Fooled By Randomness: The Hidden Role Of Chance In Life And In The Markets.* Random House, 2ed.

[Has insightful comments about the over-interpretation of phenomena in which randomness is likely to have a large role.]

10
Map Overlays and Spatial Modeling

In the past several years, there have been spectacular advances in R's mapping and spatial analysis abilities. These have used R as a unfied framework both for abilities that were developed within R and for abilities that were designed to run independently of R. Use of R in this way can have huge benefits.

For an overview of what is available under the R umbrella, see the CRAN Task View: http://cran.csiro.au/web/views/Spatial.html.

From the R command line, the relevant R packages can be installed thus: [1]

```
install.packages(c("rgdal","gstat","sp"),
                 dependencies=TRUE)
```

[1] Alternatively, install from the GUI menu.

The *rgdal* binaries for Windows and for MacOS X include GDAL, PROJ.4 and Expat. This avoids any need to install this software outside of R. Ensure also that you have `rJava`.

10.1 Static Overlay onto Maps

```
library(oz)
library(DAAG)
```

10.1.1 Overlay onto country and regional outlines

Figure 10.1 uses the function `oz()` in the *oz* package to plot an outline of the Australian coast and state boundaries.

Labeling information has been added, using the functions `points()` and `text()`, that identifies seven sites where studies of possums were conducted. Names of sites, and latitude and longitude information, were taken from the `possumsites` data set.

Code used for plot is:

```
oz(sections=c(3:5, 11:16), col="gray")
chh <- par()$cxy[2]
with(possumsites, {
  points(Longitude,
         Latitude+c(0,0,0,.2,-.2, 0,0)*chh,
         col="blue")
  text(Latitude ~ Longitude,
       labels=rownames(possumsites),
       col="red", pos=c(2,4,2,1,3,2,2), xpd=TRUE)
```

Figure 10.1: Sites at which possums were collected.

```
  # pos = 1:below, 2:left, 3:above, 4:right
  # xpd=TRUE allows plotting outside figure region
})
```

10.1.2 *The* dismo *package's Interface to Google maps*

The function `gmap()` (*dismo* package) is designed to access and download Google map data.

Loading the package *dismo*, with the default `dependencies=TRUE`, will at the same time load the *sp* and *raster* packages.

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

```
  ## The raster and sp packages are dependencies
```

Basic syntax, accepting defaults: The following is a simple example of what is possible:

The argument `type` to `gmap()` can be set to `'roadmap'`, `'satellite'`, `'hybrid'` or `'terrain'`. Use `scale=2` to double the number of pixels (default is 1.

```
cbr <- gmap("Canberra, ACT")
plot(cbr)
acton <- gmap("Acton, ACT", type="satellite")
plot(acton)
```

Road addresses can be specified. Try, e.g.

The argument `zoom` takes values between 0 and 21.

```
TBhouse <- gmap('11 Bowen St, Wellington, NZ',
                type='satellite', scale=2, zoom=20)
plot(TBhouse)
```

Specify map longitude/latitude extent; overlay onto map: The data frame `possumsites` (*DAAG*) holds latitudes and longitudes of possum study sites. In the following, a map is created that takes in all the sites. Site names are then overlaid on to the map:

Extents are specified in a longitude/latitude coordinate system. The function `gmap()` will, unless called with `lonlat=TRUE`, return a raster object that uses a Mercator projection.

```
## Extend longitude & latitude ranges slightly
lonlat <- with(possumsites,
               c(range(Longitude)+c(-3,3),
                 range(Latitude)+c(-2,2))
)
## Obtain map, as a ``RasterLayer'' object
googmap <- gmap(extent(lonlat))
plot(googmap, inter=TRUE)
## From latitude/longitude to Mercator projection
xy <- Mercator(with(possumsites,
                    cbind(Longitude, Latitude)))
## Points show location of sites on the map
points(xy)
## Add labels that give the names
text(xy, labels=row.names(possumsites))
```

Interactive selection of a map extent from a screen display: Type:

```
newlims <- drawExtent()
```

Now click at opposite corners[2] of the rectangular region that is to be selected. Red '+' symbols will appear at the locations clicked, and a bounding box will be marked out in red. Then enter:

[2] An initial click on the map may be required to initiate the process.

The new area is likely to be somwhat larger than was marked out by the bounding box.

```
googmap2 <- gmap(newlims)
plot(googmap2)
```

10.1.3 *The* plotKML *package*

```
require(plotKML)
```

The `plotKML()` function overlays any of points, lines, contours and images onto a Google earth display. Just as with a standard Google Earth display, this can be manipulated with a mouse or trackpad. The following brings the data into R:

```
library(DAAGviz)
fullpath <- system.file('datasets/nzquakes.CSV',
                        package='DAAGviz')
quakes <- read.csv(fullpath)
quakes$Date <- as.Date(quakes$Date)
quakes$Energy <- 10^quakes$magnitude/1000000
```

The range of dates is from 2009-08-02 to 2014-01-20, for quakes of magnitude 4.5 or greater. Subsection |refss:markup has the code that was used to retrieve the data from the NZ GeoNet website.

Now prepare the data for plotting:

```
## Prepare data for plotting
coordinates(quakes) <- ~ longitude+latitude
proj4string(quakes) <-
  CRS("+proj=longlat +datum=WGS84")
```

Now create the display:

```
plotKML(quakes['Energy'], points_names="")
  # Makes circle area proportional to Energy
```

If the image does not appear, look in the working directory for **quakes__Energy__.kml**, and click on it.

10.1.4 *Specifying projections*

Note that we had to specify a coordinate system. With longitude/latitude coordinates, as above, we can specify `"WGS84"`. Projections become important when spatial data, e.g., on metal concentrations, is overlaid on map data, e.g., from Google maps. See `help(proj4string)` and `help(CRS)` for further details.

10.2 *Working with Raster (Image) Files*

The function `readGDAL()` in the *rgdal* package is intended for reading GDAL grid maps. The following, loosely based on the example code in the help pages for `readGDAL`, requires the packages *rgdal*, *sp*, and *grid*. We begin by using the function `readGDAL()` to input, as a grid, the R logo file that is supplied with the R package `rgdal`:

Among the many raster formats are several that are widely used for image files more generally: BMP, various JPEG formats, GIF, PNG, XPM, etc. Georeferencing, allowing inclusion of spatial reference data, is available for BMP, JPEG 2000 formats, and TIFF.

```
library(rgdal)
```

```
logofile <- system.file("pictures/Rlogo.jpg",
                        package = "rgdal")[1]
rlogo <- readGDAL(logofile, silent=TRUE)
```

Typing these commands from the command line, but with `silent=FALSE`, will result in the warning message "GeoTransform values not available". This is not surprising. The pixels in the image do not have associated geographical spatial coordinates.

Now examine the input object:

```
class(rlogo)
```

```
[1] "SpatialGridDataFrame"
attr(,"package")
[1] "sp"
```

```
names(rlogo)
```

```
[1] "band1" "band2" "band3"
```

Observe that the file has been input as a `SpatialGridDataFrame`. The function `image` has a method for objects of this class.

The command `image()`, specifying the red, green and blue channels, can be used to show the figure, using the folowing code:

```
image(rlogo, red="band1",
      green="band2",
      blue="band3")
```

The function `spplot.grid()` is called to do the plotting. In turn, it calls function `levelplot()` from the *lattice* package.

Another possibility is to use the function `spplot()` to examine the red green and blue layers separately, as in Figure 10.2:

Figure 10.2: Red, green and blue layers from the R logo image.

```
## Code
col3 <- c("red","green","blue")
spplot(rlogo, zcol=1:3, names.attr=col3,
       col.regions=grey(0:100/100), as.table=TRUE,
       layout=c(3,1), main=paste("3-layer (RGB)",
       "raster image - example"))
```

Note also the functions `spplot.polygons()` and `spplot.points()`. These are all documented on the same page as the generic function `spplot()`.

*A genuine spatial image

The following locates the file that will be input from the package installation directory:

```
sp27 <- system.file("pictures/SP27GTIF.TIF",
                    package = "rgdal")[1]
```

Now use `readGDAL()` to create a GDAL grid map image:

```
SP27GTIF <- readGDAL(sp27, output.dim=c(100,100),
                     silent=TRUE)
class(SP27GTIF)
```

```
[1] "SpatialGridDataFrame"
attr(,"package")
[1] "sp"
```

The following can be used to get information about the file, before inputting it:

```
GDALinfo(sp27)
```

Then to plot the image, enter:

```
spplot(SP27GTIF)
```

10.2.1 *Overlaying onto a bubble plot*

Figure 4.2 in Section 4.4.1 showed how to use a bubble plot to display the `meuse` data from the `sp` package. Figure 10.3 adds river boundaries, using data from the dataset `meuse.riv`. (This is a matrix, with Eastings in column 1 and Northings in column 2.)

The function `bubble()` uses the abilities of the *lattice* package. As a consequence, the layering abilities of the *latticeExtra* package (see Subsection 7.2.8) can be used to add to the plot. Code is:

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

```
library(sp)
data(meuse); data(meuse.riv)
coordinates(meuse) <- ~ x + y
gph <- bubble(meuse, "zinc", pch=1, key.entries =
100 * 2^(0:4),
              main = "Zinc(ppm)", scales=list(axes=TRUE, tck=0.4))
add <- latticeExtra::layer(panel.lines(meuse.riv[,1], meuse.riv[,2],
                      col="gray"))
gph+add
```

Figure 10.3: Bubble plot for `zinc`, with area of bubbles proportional to concentration. River Meuse boundaries are in gray.

10.3 *Reading and Processing Shapefiles

Shapefiles (ESRI Shapefiles) are a popular geospatial vector data format. For detailed comments, see `http://en.wikipedia.org/wiki/Shapefile`. They describe geometries – points, lines and polygons.

The subdirectory **vectors**, stored as part of the *rgdal* package, has a number of shapefile collections. The following extracts and stores the path (dsn = "dataset name") to this directory:

```
dsn <- system.file("vectors", package = "rgdal")[1]
dir(dsn, pattern="shp$")
```

```
[1] "cities.shp"
[2] "kiritimati_primary_roads.shp"
[3] "scot_BNG.shp"
[4] "trin_inca_pl03.shp"
```

There are thus three shapefile collections, or "layers".

The files in the `cities` shapefile collection are:

The following can be used to get information on the `"cities"` layer, prior to input:

```
ogrInfo(dsn=dsn,
        layer="cities")
```

```
## Get names of files in shapefile collection
dir(dsn, pattern="cities")
```

```
[1] "cities.dbf" "cities.htm" "cities.prj"
[4] "cities.sbn" "cities.sbx" "cities.shp"
[7] "cities.shx"
```

The mandatory files are the **.shp** (shape format), the **.shx** (shape index format) and **.dbf** (attribute format) files. For detailed information, see http://en.wikipedia.org/wiki/Shapefile.

Note also **.prj** files, which hold coordinate system and projection information.

Now input the shapefile collection **cities**:

```
cities <- readOGR(dsn=dsn, layer="cities", verbose=FALSE)
```

The `readOGR()` function combines information from the separate shapefiles into a `SpatialPointsDataFrame`

The following gives summary information:

```
summary(cities)
```

```
Object of class SpatialPointsDataFrame
Coordinates:
              min   max
coords.x1 -165.27 177.1
coords.x2  -53.15  78.2
Is projected: FALSE
proj4string :
[+proj=longlat +datum=WGS84 +no_defs
+ellps=WGS84 +towgs84=0,0,0]
Number of points: 606
Data attributes:
        NAME        COUNTRY      POPULATION  CAPITAL
 Hyderabad:  2   US     : 49   -99    : 30   N:442
 San Jose :  2   Russia : 37   1270000:  5   Y:164
 Tripoli  :  2   China  : 36   1550000:  4
 Valencia :  2   Canada : 22   1140000:  3
 Abidjan  :  1   India  : 20   1190000:  3
 Abu Zaby :  1   Brazil : 19   1225000:  3
 (Other)  :596   (Other):423   (Other):558
```

The four "data attributes" are columns of the `data` slot. Use for example `cities$COUNTRY` or `cities[,"COUNTRY"]` to extract the column `COUNTRY`. Further columns can be added as required, or existing columns removed or modified.

For example, omne might add, using data on the World Bank website, a column giving the percentage of the population under 20 years of age.

To get information obout the countries represented, type:

```
slotNames(cities)
```

```
[1] "data"        "coords.nrs"  "coords"
[4] "bbox"        "proj4string"
```

```
names(cities)
```

```
[1] "NAME"       "COUNTRY"    "POPULATION"
[4] "CAPITAL"
```

```
  ## Returns the names in the data slot
length(levels(cities$COUNTRY))
```

```
[1] 165
```

There are 165 countries.

The following extracts and plots the shapefile details for Canada:

```
canada <- subset(cities, COUNTRY=="Canada")
trellis.par.set(sp.theme())
spplot(canada, zcol="POPULATION")
```

ASGC Digital Boundaries

The following downloads, to the working directory, a zipfile for a directory that holds 2011 statistical region boundaries in ESRI shapefile format:

```
url <- paste0("http://www.abs.gov.au/ausstats/subscriber.nsf/",
  "log?openagent&1259030001_sr11aaust_shape.zip&1259.0.30.001",
  "&Data%20Cubes&B8003880BC09FA5BCA2578CC00124E25&0",
  "&July%202011&14.07.2011&Latest")
downloadTo <- "../downloads/1259030001_sr11aaust_shape.zip"
download.file(url, destfile=downloadTo)
```

The file is 25.6MB and, depending on the speed of the connection, will take a modest time to download.

Alternatively, go to the web page http://www.abs.gov.au/AUSSTATS/abs@.nsf/DetailsPage/1259.0.30.001July%202011?OpenDocument, locate the zip file identified as containing "Statistical Region ASGC Ed 2011 Digital Boundaries in ESRI Shapefile Format", and download it. Click on SUMMARY to get information that describes the codes that are used to identify states and territories, the field headers, ans the nature of the included data.

The following then unzips the files into the subdirectory **au-srs**:

If the file is not in the working directory, precede the file name with the path to the file.

```
unzip("../downloads/1259030001_sr11aaust_shape.zip", exdir="au-srs")
dir("../downloads/au-srs")
```

```
[1] "SR11aAust.cpg" "SR11aAust.dbf" "SR11aAust.prj"
[4] "SR11aAust.shp" "SR11aAust.shx"
```

Now input the information from the shapefiles, and combine it to create a `SpatialPolygonsDataFrame`, named `auSRS`, that has statistical region boundaries:

```
auSRS <- readOGR("../downloads/au-srs", layer="SR11aAust")
```

```
OGR data source with driver: ESRI Shapefile
Source: "../downloads/au-srs", layer: "SR11aAust"
with 66 features
It has 3 fields
```

States or territories are 1:New South Wales, 2:Victoria, 3:Queensland, 4:South Australia, 5:Western Australia, 6:Tasmania, 7:Northern Territory, 8:Australian Capital Territory, 9:Other Territories. The following extracts the `SpatialPolygonsDataFrame` for Victoria.

```
vicSRS <- subset(auSRS, STATE_CODE==2)
unique(vicSRS@data[,"SR_NAME11"])
```

```
 [1] Outer Western Melbourne
 [2] North Western Melbourne
 [3] Inner Melbourne
 [4] North Eastern Melbourne
 [5] Inner Eastern Melbourne
 [6] Southern Melbourne
 [7] Outer Eastern Melbourne
 [8] South Eastern Melbourne
 [9] Mornington Peninsula
[10] Barwon-Western District
[11] Central Highlands-Wimmera
[12] Loddon-Mallee
[13] Goulburn-Ovens-Murray
[14] All Gippsland
66 Levels: All Gippsland ...
```

Notice that all 66 factor levels from `auSRS` have been retained, even though only 14 of these levels are present in this dataset.

Functions are available for combining files of this type, or for removing boundaries such as between the `SR_NAME11` regions.

Further information

The R geo website, at http://www.r-project.org/Rgeo/ has extensive information. The Wiki page http://spatial-analyst.net/wiki/index.php?title=Software has extensive information about installation of relevant geostatistical software.

Table 3.1 in Hengl(2011) compares the spatio-temporal abilities of some popular statistics and GIS packages. There are columns for R+*gstat* and R+*geoR*.

The web page http://www.nceas.ucsb.edu/scicomp/usecases/ReadWriteESRIShapeFiles has several examples that demonstrate the reading and plotting of shapefiles, comparing abilities in the *rgdal* package with those in *maptools* and *PBSmapping*. Note also the package *shapefiles*.

The website http://info.geonet.org.nz/display/appdata/Earthquake+Web+Feature+Service has information on how New Zealand earthquake data may be retrieved in a variety of formats. The site http://www.christchurchquakemap.co.nz/ has impressive dynamic visualizations of Christchurch (NZ) and other quake data.

A carefully documented example of the use of shapefiles of New Zealand data can be found at
http://www.r-bloggers.com/simplifying-polygon-shapefiles-in-r/.

10.4 Other software – QGIS

Note in particular QGIS, which has an interface via *manageR* to R; go to http://www.ftools.ca/plugins.html.

To obtain Windows and Linux installers for QGIS, go to http://www.qgis.org/wiki/Download. The standalone installer for Windows includes GRASS. For MacOSX, go to http://www.kyngchaos.com/software/qgis. For Leopard and Snow Leopard installations, the QGIS 1.7 developer builds seem relatively stable. GRASS must be installed separately.

10.5 References

Bivand R, Pebesma E J, Gomez-Rubio, V. 2008. Applied Spatial Data Analysis with R. Springer.

Diggle, Peter J. & Ribeiro Jr, Paulo J 2007. Model-Based Geostatistics. Springer.

Hengl, T. 2011, A Practical Guide to Geostatistical Mapping. 2nd edn.

[To download (free) or purchase ($US15.19), go to: http://www.lulu.com/ and search for 'Hengl']

Hijmans, R J. 2011. Introduction to the 'raster' package.

[With the R package *raster* attached, type `vignette("Raster")`.

Hijmans, R J and Elith J. 2011. Species distribution modeling with R.

[With the R package *dismo* attached, type `vignette("sdm")`. The vignette appears to be an outline for a book. Later chapters are very incomplete.]

Lamigueiro, O P 2012. Maps with R (I) http://procomun.wordpress.com/2012/02/18/maps_with_r_1/

Maindonald, J H 2011. Generalized Additive Models in Spatial Statistics – Linear Models with a Twist (slides). maths.anu.edu.au//~johnm/r/spatial/

[This offers a perspective on spatial interpolation.]

Quantum GIS Development Team 2010. Quantum GIS User Guide(Version 1.6.0 'Copiapó'). Obtain from http://www.qgis.org/en/documentation/manuals.html

See also the vignettes that accompany the package *sp*, describing classes and methods for spatial data, and overlay and aggregation.

11
Brief Notes on Text Mining

A first step is to load the *tm* package. This is designed for working with a *corpus* — *corpus* is the name for a collection of documents.

```
library(tm)
```

11.1 Creation of a Volatile Corpus

The data used is from three text files, stored in the *DAAGviz* directory tree. They hold text from the respective chapter ranges 1 - 5, 6 - 7, and 8 - 9 of an older version of this present document. We show two ways to use it to form a corpus. The first breaks the process down into detailed steps, while the second uses a much terser and summary approach. The resultant corpus is *volatile*, so described because stored in the workspace. Unless saved separately or as part of the workspace, it will disappear at the end of the session.

A pdf to text converter has taken the pdf for this document, and extracted the three chapter ranges into the respective files **ch1-5prelims.txt**, **ch6-7data.txt**, and **ch8-9graphics.txt**.

Detailed steps

First create paths to the files, and check that they seem correct:

```
## Create paths to the text files, stored in the
## subdirectory "texts" of the DAAGviz package.
txdir <- system.file("texts", package="DAAGviz")
dir(txdir, pattern=".txt$")
```

```
[1] "data6-7.txt"     "graphics8-9.txt"
[3] "prelims1-5.txt"
```

```
txfiles <- dir(txdir, pattern=".txt$", full.names=TRUE)
```

```
## Input first file, with one text string per line
tx1 <- readLines(txfiles[1], encoding="UTF-8", warn=FALSE)
## Join the separate text strings end to end
tx1 <- paste(tx1, collapse=" ")
```

Repeat this process for files 2 and 3:

```
tx2 <- readLines(txfiles[2], encoding="UTF-8", warn=FALSE)
tx2 <- paste(tx2, collapse=" ")
tx3 <- readLines(txfiles[3], encoding="UTF-8", warn=FALSE)
tx3 <- paste(tx3, collapse=" ")
```

Now bring the three text strings together into a corpus:

```
txcorp <- Corpus(VectorSource(c(tx1, tx2, tx3)))
```

Creation of a corpus using `DirSource()`

The following creates a directory source:

```
dirSource <- DirSource(directory=txdir,
                       pattern=".txt$")
```

Now create the corpus. The text will be input from the files that were identified, within the specified directory **doc**:

```
toUTF8 <- function(x) iconv(x, to="UTF-8",
                            sub = "byte")
txcorp <- Corpus(dirSource)
txcorp <- tm_map(txcorp,
    content_transformer(toUTF8))
```

The call to `tm_map()` is a mechanism for marking the document as UTF-8. The pdf to text converter creates UTF-8 documents. The tokenizer `scan_takenizer` then calls `scan()`, but without marking the document that results as UTF-8, as required for use of `termDocumentMatrix()` or `termFreq()`.

Next steps

A common starting point for further work is a term by document matrix. For this, use `TermDocumentMatrix()`. Or if a document by term matrix is required, use `DocumentTermMatrix()`

Pre-processing steps prior to creating such a matrix may include stripping away stopwords, elimination of white space, and conversion to lower case. These can be performed in the process of creating a term document matrix, thus:

```
ctl <- list(removePunctuation = list(preserve_intra_word_dashes = FALSE),
            removeNumbers = TRUE,
            stopwords=c(stopwords("SMART"), "[1]"),
            wordLengths=c(3,Inf))
tx.tdm <- TermDocumentMatrix(txcorp, control=ctl)
```

Notice the identification of `[1]`, which appears quite frequently in the R output, as a stopword. This omits it from the list of terms. Closer investigation would reveal other issues, most because the default tokenizer[1] is not designed to handle R code and output.

[1] See `help(termFreq)` for an example of a user-supplied tokenizer.

Stopwords are words that are likely to be uniformative for purposes of comparing text content. The function `tm::stopwords()` accepts the arguments `kind="en"`, which gives a relatively restricted list of English stopwords, or `kind="SMART"` which gives a much more extensive list. See `?tm::stopwords` for details of non-English stopword lists that are immediately available. The following will give an idea of the sorts of words that are in the two lists:

```
## First few stopwords in the "en" set
sort(stopwords())[1:5]
```

```
[1] "a"     "about" "above" "after" "again"
```

```
## First few stopwords in the "SMART" set
stopwords('SMART')[1:5]
```

```
[1] "a"     "a's"   "able"  "about" "above"
```

```
## Stopwords in "SMART" but not in "en"; first 10
stopwords("SMART")[!stopwords("SMART")%in%stopwords()][1:10]
```

```
 [1] "a's"         "able"        "according"
 [4] "accordingly" "across"      "actually"
 [7] "afterwards"  "ain't"       "allow"
[10] "allows"
```

Now list terms that occur 100 or more times:

```
findFreqTerms(tx.tdm, 100)
```

```
 [1] "code"      "data"      "figure"    "file"
 [5] "frame"     "function"  "functions" "graphics"
 [9] "objects"   "package"   "plot"
```

Wordclouds

First load the *wordcloud* package:

```
library(wordcloud)
```

Figure 11.1 shows wordcloud plots for the first (chapters 1-5), second (6-7) and third (8-9) documents in the corpus.

Figure 11.1: Wordcloud plots are A: for the words in Chapters 1 - 5; B: 6 - 7; and C: 8 - 9.

A: Chapters 1 − 5

B: Chapters 6 − 7

C: Chapters 8 − 9

Code for the plots is:

```
pal <- brewer.pal(6, "Dark2")
fnam1 <- as.matrix(tx.tdm)[,1]
wordcloud(names(fnam1), fnam1, max.words=80, colors=pal[-1],
          random.order=FALSE, scale=c(8.5,.5))
mtext(side=3, line=3.5, "A: Chapters 1 - 5", adj=0, cex=1.8)
fnam2 <- as.matrix(tx.tdm)[,2]
```

```
wordcloud(names(fnam2), fnam2, max.words=80, colors=pal[-1],
          random.order=FALSE, scale=c(4.5,.5))
mtext(side=3, line=3.5, "B: Chapters 6 - 7", adj=0, cex=1.8)
fnam3 <- as.matrix(tx.tdm)[,3]
wordcloud(names(fnam3), fnam3, max.words=80, colors=pal[-1],
          random.order=FALSE, scale=c(6.0,.5))
mtext(side=3, line=3.5, "C: Chapters 8 - 9", adj=0, cex=1.8)
```

Less frequent words will be lost off the edge of the plot if the size of the graphics page is too small relative to the pointsize. Note the different scaling ranges used in the three cases, with the large scaling range for Panel B (`scale=c(10,.5)`) used to accommmodate a frequency distribution in which one item ('data') is a marked outlier.

All three panels used a 5in by 5in graphics page, with a pdf pointsize of 12.

11.2 Creation of a Corpus from PDF Files

The *tm* package has functions that can be used to create readers for several different types of files. Type `getReaders()` to get a list. Note in particular `readPDF()` that can be used with pdf files. See `?tm::readPDF` for details of PDF extraction engines that may be used. The default is to use the Poppler PDF rendering library as provided in the **pdftools** package.

The following sets the path to the directory **pdf** in the package **DAAGviz** that holds the pdf files for (possibly, an older version) of the four ranges of chapters of the present text:

```
uri <- system.file("pdf", package="DAAGviz")
## Check names of files in directory
dir(uri)
```

```
[1] "ch1-3.pdf" "ch4-6.pdf" "ch7.pdf"   "ch8-9.pdf"
```

The corpus that has all three documents is, starting with the pdf files, most easily created thus:

```
fromPDF <- Corpus(DirSource(directory=uri, pattern=".pdf$"),
                  readerControl=list(reader=readPDF,
                  PdftotextOptions = "-layout"))
makeChar <- function(x)gsub("[^[:alnum:][:blank:]]","" , x, ignore.case = TRUE)
fromPDF <- tm_map(fromPDF, content_transformer(makeChar))
```

Create the term-document matrix thus:

```
txx.tdm <- TermDocumentMatrix(fromPDF, control=ctl)
```

11.3 Document Collections Supplied With tm

Several document collections are supplied with the package, as text files or as XML files. To get the path to the directories where these document collections are stored, type

```
(pathto <- system.file("texts", package="tm"))
```

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

```
dir(pathto)
```

```
[1] "acq"                "crude"              "custom.xml"
[4] "loremipsum.txt"     "rcv1_2330.xml"      "reuters-21578.xml"
[7] "txt"
```

The subdirectory **acq** has 50 Reuters documents in XML format, **crude** has the first 23 of these, and **txt** has a small collection of 5 text documents from the Roman poet Ovid. These can be accessed and used for experimenation with the abilities provided in *tm*, as required.

The following are the names of the five Ovid documents:

```
dir(paste(pathto, "txt",sep="/"))
```

```
[1] "ovid_1.txt" "ovid_2.txt" "ovid_3.txt" "ovid_4.txt" "ovid_5.txt"
```

The following brings these documents into a volatile corpus, i.e., a corpus that is stored in memory:

```
(ovid <-
   Corpus(DirSource(paste(pathto, "txt", sep="/")),
          readerControl=list(language="lat")))
```

```
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 5
```

12

*Leveraging R Language Abilities

12.1 Manipulation of Language Constructs

Language structures can be manipulated, just like any other object. Below, we will show how formulae, expressions, and argument lists for functions, can be pasted together.

12.1.1 Manipulation of Formulae

Formulae are a key idea in R, though their implementation is incomplete. They are widely available for specifying graphs, models and tables. Details will be given below.

Model, graphics and table formulae

We demonstrate the construction of model or graphics formulae from text strings. The following plots the column `mpg`, from the data frame `mtcars` (*MASS*), against `disp`:

```
plot(mpg ~ disp, data=mtcars)
```

The following gives the same result:

```
yvar <- "mpg"
xvar <- "disp"
form <- as.formula(paste(yvar, "~", xvar))
plot(form, data=mtcars)
```

With this second approach, `yvar` and `xvar` can be arguments to a function, and `xvar` and `yvar` can be any pair of columns. A suitable functionis:

```
plot.mtcars <- function(xvar="disp", yvar="mpg"){
    form <- as.formula(paste(yvar, "~", xvar))
    plot(form, data=mtcars)
}
```

The following calls the function with `xvar="hp"` and `yvar="mpg"`:

```
plot.mtcars(xvar="hp", yvar="mpg", data=mtcars)
```

The data frame `mtcars` has 11 columns from which the two axes for a scatterplot might be chosen:

```
names(mtcars)
```

```
 [1] "mpg"  "cyl"
 [3] "disp" "hp"
 [5] "drat" "wt"
 [7] "qsec" "vs"
 [9] "am"   "gear"
[11] "carb"
```

12.1.2 Extraction of names from a formula

Use the function `all.vars()` to extract the variable names from a formula, thus:

```
all.vars(mpg ~ disp)
```

```
[1] "mpg"  "disp"
```

As well as using a formula to specify the graph, the following gives more informative *x*- and *y*-labels:

```
plot.mtcars <- function(form = mpg ~ disp){
   yvar <- all.vars(form)[1]
   xvar <- all.vars(form)[2]
   ## Include information that allows a meaningful label
   mtcars.info <-
     c(mpg= "Miles/(US) gallon",        cyl= "Number of cylinders",
       disp= "Displacement (cu.in.)",   hp= "Gross horsepower",
       drat= "Rear axle ratio",         wt= "Weight (lb/1000)",
       qsec= "1/4 mile time",           vs= "V/S",
       gear= "Number of forward gears",
       carb= "Number of carburettors",
       am= "Transmission (0 = automatic, 1 = manual)")
   xlab <- mtcars.info[xvar]
   ylab <- mtcars.info[yvar]
   plot(form, xlab=xlab, ylab=ylab)
}
```

12.2 Function Arguments and Environments

12.2.1 Extraction of arguments to functions

A simple use of `substitute()` is to extract a text string representation of a function argument:

```
plot.mtcars‘ <-
  function(x = disp, y = mpg){
    xvar <- deparse(substitute(x))
    yvar <- deparse(substitute(y))
    form <- formula(paste(yvar, "~", xvar))
    plot(form, xlab=xvar, ylab=yvar, data=mtcars)
  }
```

12.2.2 Use of a list to pass parameter values

The following are equivalent:

Use of `do.call()` allows the parameter list to be set up in advance of the call. The following shows the use of `do.call()` to achieve the same effect as `mean(possum$totlngth)`:

```
do.call("mean", list(x=possum$totlngth))
```

This makes more sense in a function, thus:

```
`average` <-
  function(x=possum$chest, FUN=function(x)mean(x)){
    fun <- deparse(substitute(FUN))
    do.call(fun, list(x=x))
  }
```

This allows, e.g., the following:

```
average()
average(FUN=median)
```

Note also `call()`, which sets up an unevaluated expression. The expression can be evaluated at some later time, using `eval()`. Here is an example:

```
mean.call <- call("mean", x=rnorm(5))
eval(mean.call)
```

```
[1] 0.1472
```

```
eval(mean.call)
```

```
[1] 0.1472
```

Notice that the argument `x` was evaluated when `call()` was evoked. The result is therefore unchanged upon repeating the call `eval(mean.call)`. This can be verified by printing out the expression:

```
mean.call
```

```
mean(x = c(-1.6525945681225, -0.00828593746666305, 1.78534257077903,
-0.157454671640352, 0.76915223549876))
```

12.2.3 *Function environments*

Every call to a function creates a frame that contains the local variables created in the function. This combines with the environment in which the function was defined to create a new environment. The global environment, `.Globalenv`, is the workspace. This is frame 0. The frame number increases by 1 with each new function call.[1]

```
[1] "test"
```

Here is code that determines, from within a function, the function name:

```
test <- function(){
 fname <- as(sys.call(sys.parent())[[1]],
             "character")
  fname
}
test()
```

```
[1] "test"
```

[1] Additionally, frames may be referred to by name. Use

`sys.nframe()` to get the number of the current evaluation frame

`sys.frame(sys.nframe())` to identify the frame by name

`sys.parent()` to get the number of the parent frame.

Now change the function name to `newtest()`:

```
newtest <- test
newtest()
```

```
[1] "newtest"
```

When a number of graphs are required, all for the one dociment, a sequential naming system, e.g., `fig1()`, `fig2()`, ..., may be convenient, with matching names **fig1.pdf**, **fig2.pdf**, ... for the respective graphics files. The following function `gf()` generates the file name automatically, for passing to the graphics device that is opened.

```
gf <-
    function(width=2.25, height=2.25, pointsize=8){
        funtxt <- sys.call(1)
        fnam <- paste0(funtxt, ".pdf")
        print(paste0("Output is to the file '",
                     fnam, "'"))
        pdf(file=fnam, width=width, height=height,
            pointsize=pointsize)
    }
```

Now create a function that calls `gf()`:

```
fig1 <- function(){
    gf()              # Call with default parameters
    curve(sin, -pi, 2*pi)
    dev.off()
}
fig1()
```

Output goes to the file **fig1.pdf**. For a function `fig2()` that calls `gf()`, output goes to the file **fig2.pdf**, and so on.

Scoping of object names

Local objects are those that are created within the body of the function. Objects that are not local and not passed as parameters are first searched for in the frame of the function, then in the parent frame, and so on. If they are not found in any of the frames, then they are sought in the search list.

12.3 Creation of R Packages

Much of the functionality of R, for many important tasks, comes from the packages that are built on top of base R. Users who make extenive use of R may soon find a need to document and organize both their own functions and associated data. Packages are the preferred vehicle for making functions and/or data available to others, or for use by posterity.

The RStudio documentation includes a large amount of information on package preparation, testing, and submission to CRAN or other repositories. Click on
Help | RStudio Docs
and look under
PACKAGE DEVELOPMENT.

Organisation of data and functions into a package may have the following benefits:

- Where the package relates to a project, it should be straightforward to return to the project at some later time, and/or to pass the project across to someone else.
- Attaching the packages give immediate access to functions, data and associated documentation.

- Where a package is submitted to CRAN (Comprehensive R Archive Network) and used by others, this extends opportunities for testing and/or getting contributions from other workers. Checks that are required by CRAN ensure that the package (code and documentation) meets certain formal standards. CRAN checks include checks for consistency between code and documentation, e.g., in names of arguments. Code must conform to CRAN standards.

Namespaces

Packages can have their own namespaces, with private functions and classes that are not ordinarily visible from the command line, or from other packages. For example, the function `intervals.lme()` that is part of the *lme* package must be called via the generic function `intervals()`.

12.4 S4 Classes and Methods

There are two implementations of classes and methods – those of version 3 of the S language (S3), and those of version 4 of the S language (S4). The *methods* package supplies the infrastructure for the S4 implementation. This extends the abilities available under S3, builds in checks that are not available with S3, and are is conducive to good software engineering practice. The Bioconductor bundle of packages makes extensive use of S4 style classes and methods. See `help(Methods)` (note the upper case M) for a brief overview of S4 classes and methods.

Where available, extractor functions should be used to extract slot contents. If this is not possible, use the function `slotNames()` to obtain the names of the slots, and either the function `slot()` or the operator `@` to extract or replace a slot. For example:

```
library(DAAG)
library(lme4)
```

```
hp.lmList <- lmList(o2 ~ wattsPerKg | id,
                    data=humanpower1)
slotNames(hp.lmList)
```

```
[1] ".Data"     "call"      "pool"      "groups"
[5] "origOrder"
```

The following are alternative ways to display the contents of the `"call"` slot:

```
hp.lmList@call
```

```
lmList(formula = o2 ~ wattsPerKg | id, data = humanpower1)
```

```
slot(hp.lmList, "call")
```

```
lmList(formula = o2 ~ wattsPerKg | id, data = humanpower1)
```

Where available, use an extractor function to extract some relevant part of the output, thus:

```
coef(hp.lmList)
```

```
  (Intercept) wattsPerKg
1      -1.155      15.35
2       1.916      13.65
3     -12.008      18.81
4       8.029      11.83
5      11.553      10.36
```

For moderately simple examples of the definition and use of S4 classes and methods, see `help(setClass)` and `help(setMethod)`.

How is it possible to identify, for a particular S4 class, the function that implements a method. To identify the function in the `sp` package that implements the `spplot` method for `SpatialGridDataFrame` objects, type:

```
library(sp)
selectMethod("spplot",
             signature="SpatialGridDataFrame")
```

```
Method Definition:

function (obj, ...)
spplot.grid(as(obj, "SpatialPixelsDataFrame"), ...)
<environment: namespace:sp>

Signatures:
        obj
target  "SpatialGridDataFrame"
defined "SpatialGridDataFrame"
```

This makes it clear that the `spplot` method for `SpatialGridDataFrame` objects calls the function `spplot.grid()`. To display the function `spplot.grid()`, type:

```
getFromNamespace("spplot.grid", ns="sp")
```

Alternatively, use the less targeted `getAnywhere("spplot.grid")`.

Use `showMethods()` to show all the methods for one or more classes of object. For example:

```
showMethods(classes='SpatialGridDataFrame')
```

12.5 Summary

Language structures (formulae and expressions) can be manipulated, just like any other object.

R uses formulae to specify models, graphs and (`xtabs()` only) tables.

The expression syntax allows the plotting of juxtaposed text strings, which may include mathematical text.

All evaluations have an environment that determines what objects will be visible. This can be especially important for the writing and testing of functions.

Packages are the preferred vehicle for making substantial collections of functions and/or data available to others, or for use by posterity. They facilitate re-use of code and enforce checks for common inconsistencies. They make it straighforward to enforce high standards of documentation.

Many of R's more recent packages use S4 classes and methods. Extractor functions are available that will extract the most commonly required types of information.

A

*R System Configuration

A.1 Fine tuning the installation

The information in this section is relevant mainly to users of the Windows or Mac GUI, rather than RStudio. With RStudio, work is typically organized by projects, albeit usually with a different working directory for each different project.

Figure A.1: This shows the sequence of clicks needed to display the page from which the Start in: directory can be set. This will then be the working directory in which R will start.

The working directory: This is best set to a directory that is convenient for storing files connected with the project on which you are currently working. When starting a new project, it to start a new working directory.

Under Windows, each R icon has associated with it a working directory. Right click the icon. Then click on Properties (at the bottom of the list), thus displaying the Properties submenu. Make sure that Shortcut is selected. Set the Start in: directory to the working directory in which you want R to start. See Figure A.1.

Under MacOS X, dragging a file onto the R icon will start R in the directory that contains the file. Alternatively, in a terminal window, type for example:

```
open -a R ~/r/course
```

This will start R with **~/r/course** as working directory.

Multiple (MDI) or Single (SDI) display interface for Windows: One way to get R to start in SDI mode is to add `-sdi`, with a preceding space, to the target that is shown in Figure A.1.

A.2 Setting the Default CRAN Mirror

The default CRAN mirror can be set from the Windows or MacOS X R GUI.

Windows: Click on Packages | Set CRAN mirror . . .

Figure A.2: This shows the R Windows GUI menu option that can be used to set the CRAN mirror. If not set, the user is asked to nominate the mirror whenever one or more packages are downloaded or updated from CRAN.

MacOS X: Click on R | Preferences

Click on Preferences. Then click, if necessary, on Startup to display the startup options, shown in the Window on the right.

Figure A.3: In a factory fresh MacOS X installation, packages are downloaded from `http://cran.r-project.org`. The preferences pages allow the setting of a wide variety of other preferences also.

A.3 R system information

If access is needed to files that are in the R installation tree, obtain the path, thus:

```
R.home()
```

```
[1] "C:/PROGRA~1/R/R-32~1.2"
```

When using Microsoft Windows systems, a more intelligible result is obtained by wrapping the function call in `normalizePath()`, thus:

```
normalizePath(R.home(), winslash="/")
```

```
[1] "C:/Program Files/R/R-3.2.2"
```

If the `winslash="/"` argument is omitted, double backslashes are used to separate names in the directory tree.

To see a list of all R system variables, type

```
names(Sys.getenv())
```

These can then be inspected individually.

```
Sys.getenv("R_HOME")
```

```
[1] "C:/PROGRA~1/R/R-32~1.2"
```

See `help(Rprofile)` for details on how to set system variables.

The following can be used to get the path to files that come with an installed package:

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

```
[1] "C:/Users/JohnM/Documents/R/win-library/3.2/DAAG/misc/ViewTemps.RData"
```

A.4 Repositories additional to CRAN

Figure A.4: List of available repositories, as given by the Windows GUI.

Figure A.4 shows a list of available repositories, as given by the Windows GUI, after clicking on: `Packages|Set repositories`. To select more than one repository, hold down the Windows key, or under MacOS X the command key, while left-clicking.

To use the command line to view a list of all repositories known to the R installation, and possibly to select one or more, type:

```
setRepositories()
```

Alternatively, use the argument `ind`, in a call to `setRepositories()`, to specify repositories. For example:

```
setRepositories(ind=1:2)
```

A.5 Running R in Batch Mode

On the command line (for Windows, Unix, Linux, . . .), enter

```
R CMD BATCH infile outfile
```

Here infile is a file that holds R commands. See `help(BATCH)` for further information. The path to the R executable must be included in the system path variable. The R FAQ for Windows has information on setting environment variables.

For Windows 64-bit R (2.12.0 or later), the directory **R_HOME\bin\x64** must be in the system path. For Windows 32-bit R, replace **x64** by **i386**. For determining **R_HOME**, see above.

A.6 The R Windows installation directory tree

The R system can be installed into any directory where the installer has write permission. The hierarchy of directories and files that form the R installation is termed the *directory tree*. Likely defaults for the directory tree of an R-3.2.2 Windows installation are:

C:\PROGRAM FILES\R\R-3.2.2

or, for example,

C:\Documents and Settings\Owner\My Documents\R\R-3.2.2

The directory tree is relocatable. It can be copied to a flash drive or CD or DVD, which can be then be used to run R. Thus, copy

For running R from a DVD (or CD), be sure to change the Start In directory to a directory that is writable.

the installation tree with root at **C:\Program Files\R\R-3.2.2** across to **D:** to become a tree with root **D:\R-3.2.2**. The executable (binary) **D:\R-3.2.2\bin\x64\Rgui.exe** can then be used to start a 64-bit R session. To verify that this works, click on its icon. For 32-bit R, replace **x64**, where it appears in the path, by **i386**.

A.7 Library directories

Packages that the user installs can go anywhere on the system. It is not necessary to install them into the library directory that is in the R directory tree.

An R session that is running from one installation, perhaps the main installation on the hard drive, can in principle access any compatible R library directory that is available. Thus, the following gives access, from an R session that is started (e.g.) from the hard drive, to to a library tree that is stored under **D:\R-3.2.2**:

```
.libPaths("D:/R-3.0.2/library")
```

This might for example be the library tree from an R installation on a DVD that has been placed in the **D:** drive.

To avoid the need to type this each time a new session is started in that working directory, create the following function:[1]

```
.First <- function().libPaths("D:/R-3.0.2/library")
```

[1] This function will then be saved as part of the default workspace, and executed at the start of any new session in that directory.

Alternatively, the function can be placed in a startup profile file, as described in the next section.

In moving to a new major version of R (e.g., from R-3.1.2 to R-3.2.0), the library directory in the R installation tree is replaced. A relatively painless way to update the new installation library directory is to copy packages across from the former installation library directory to the new library directory, being careful not to replace any existing packages in that directory. (If of course packages are in a separate user directory, no moving is required.) Then, from within R, type:

```
update.packages(checkBuilt=TRUE, ask=FALSE)
## Check package status:
summary(packageStatus())
```

Use of a **.Renviron** file in the home directory is a further possibility. This can be conveniently done from within R:[2]

[2] This follows a suggestion from Bill Venables.

```
cat('LIB1="C:/Users/owner/R/win-library/2.15"\n',
    file = "~/.Renviron", append = TRUE)
cat('LIB2="C:/Users/owner/R/win-library/2.14"\n',
    file = "~/.Renviron", append = TRUE)
cat('LIB1="R_LIBS_USER=${LIB1};${LIB2}\n',
    file = "~/.Renviron", append = TRUE)
```

A.8 The Startup mechanism

Various system variables are set at startup, and a number of packages are attached. The details can be controlled at an installation level, at a user level, and at a startup directory level. If started in the standard manner, the 'R_PROFILE' environment variable[3] can be used to give the name of a site-wide profile file. See `help(Startup)` for details.

R next searches at startup for a file called **.Rprofile** in the current directory or in the user's home directory (in that order).[4] Such a **.Rprofile** file can for example define a `.First()` and/or a `.Last()` function.

A user (or site-wide) profile file might for example include the following statements:

```
options(papersize="a4")
options(editor="notepad")          # Preferred editor
options(pager="internal")
options(tab.width = 4)
options(width = 120)               # Wider console line
options(graphics.record=TRUE)
options(show.signif.stars=FALSE)   # Banish the stars
options(prompt="? ")               # Use '?' as prompt
options(continue="  ")             # Blank continuation
.libPaths("C:/my_R_library")       # Add library to path.
```

[3] If this is unset, R searches for a file **R_HOME/etc/Rprofile.site**. 'Factory-fresh' installations will not have such a file. Code is sourced into the *base* package, before other packages are loaded.

[4] Alternatively, the 'R_PROFILE_USER' environment variable can be used to set the name of a user profile file.

B

The R Commander Graphical User Interface

To start the R commander, start R and enter:

```
library(Rcmdr)
```

This opens an R Commander script window, with the output window underneath. This window can be closed by clicking on the × in the top left corner. If thus closed, enter `Commander()` to reopen it again later in the session.

At startup, the R Commander checks whether all packages are available that are needed for the full range of features. If some are missing, the R commander offers to install them. (This requires a live internet connection.)

From GUI to writing code: The R commander displays the code that it generates. Users can take this code, modify it, and re-run it.

The code can be run either from the script window or from the R console window (if open).

The active data set: There is, at any one time, a single "active" data set. Start by clicking on the Data drop-down menu. To select or create or change the active data set, do one of the following:

- Click on Active data set, and pick from among data sets, if any, in the workspace.
- Click on Import data, and follow instructions, to read in data from a file. The data set is read into the workspace, at the same time becoming the active data set.
- Click on New data set . . . , then entering data via a spreadsheet-like interface.
- Click on Data in packages, then Read Data from Package. Then select an attached package and choose a data set from among those included with the package.
- A further possibility is to load data from an R image (`.RData`) file; click on Load data set . . .

Creating graphs: To draw graphs, click on the Graphs drop-down menu. Then, among other possibilities:

- Click on Scatterplot . . . to obtain a scatterplot.[1]
- Click on X Y conditioning plot . . . for *lattice* scatterplots and panels of scatterplots.

[1] This uses `scatterplot()` (*car* package), which in turn uses functions from base graphics.

- Click on 3D graph to obtain a 3D scatterplot.[2]

[2] This uses the R Commander function `scatter3d()` that is an interface to functions in the *rgl* package.

Statistics (& fitting models): Click on the Statistics drop down menu to get submenus that give summary statistics and/or carry out various statistical tests. This includes (under Contingency tables) tables of counts and (under Means) One-way ANOVA. Also, click here to get access to the Fit models submenu.

**Models:* Click here to extract information from model objects once they have been fitted. (NB: To fit a model, go to the Statistics drop down menu, and click on Fit models).

Other GUIs for R

The *rattle* GUI, aimed broadly at "data mining" (data manipulation, regression, classification and clustering) applications, is a powerful and sophisticated system. It has a number of features that make it attractive for use in standard data mining applications. Note also *JGR* (Java Graphics for R) and *pmg* (Poor Man's GUI).

Note also the abilities in *playwith* and *latticist* for interaction with graphs. These are both discussed in Subsection 7.2.10. See also Figure 7.16

C
Color Versions of Selected Graphs

Annotated Motion Chart

Figure C.1: Snapshot of a motion chart, with annotation that identifies selected chart features that can be modified interactively.

A Playwith GUI Window

Figure C.2: This playwith GUI window was generated by wrapping the call to `xyplot()` in the function `playwith()`, then clicking on Identify. Click near to a point to see its label. A second click adds the label to the graph. This is a color version of 7.16.

Florence Nightingale's Wedge Plot

Figure C.3 is a "wedge" plot that shows the mortality of British troops according to cause in the Crimean War over 1853-1853. It has often been called a "coxcomb" plot – a name that suits this imaginative form of graphical presentation.

The name "coxcomb" arose from a misreading of Florence Nightingale's *Mortality of the British Army* that was an annex to a larger official report. See `http://www.york.ac.uk/depts/maths/histstat/small.htm`

Figure C.3: Deaths (per 1000 per annum), up to and after the Sanitary Commission's visit in March 1855. Areas, measured from the centres of the common vertices, are proportional to mortalities..

Figure C.3 is used here to show abilities in *ggplot2*. The wedge plot is not ideal for giving an accurate sense of the data. It is not however easy to suggest an alternative that is clearly better.

Code for the wedge plot

The url `http://maths.anu.edu.au/~johnm/r/functions/wedgeORbubble.R`.[1] has code for the function `wedgeplot()` that was used for this plot. Note also the function `gdot()` that gives a not entirely satisfactory alternative to the wedge plot.

[1] Data are from the data frame `Nightingale` in the *HistData* package. Subsection C.3 showed how to create a dataset `Crimean` that is in a convenient form for creating this plot.

Selected base graphics parameter settings

A: Plot symbols and text; specify colors and/or character expansion; draw rectangle

```
par(fig=c(0, 1, 0.415, 1))
```

```
plot(0, 0, xlim=c(0, 13), ylim=c(0, 19), type="n")
xpos <- rep((0:12)+0.5, 2);  ypos <- rep(c(14.5,12.75), c(13,13))
points(xpos, ypos, cex=2.5, col=1:26, pch=0:25)
text(xpos, ypos, labels=paste(0:25), cex=0.75)
```

```
## Plot characters, vary cex (expansion)
text((0:4)+0.5, rep(9*ht, 5), letters[1:5], cex=c(2.5,2,1,1.5,2))
```

```
## Position label with respect to point
xmid <- 10.5; xoff <- c(0, -0.5, 0, 0.5)
ymid <- 5.8; yoff <- c(-1,0,1,0)
col4 <- colors()[c(52, 116, 547, 610)]
points(xmid+xoff, ymid+yoff, pch=16, cex=1.5, col=col4)
posText <- c("below (pos=1)", "left (2)", "above (3)", "right (4)")
text(xmid+xoff, ymid+yoff, posText, pos=1:4)
rect(xmid-2.3, ymid-2.3, xmid+2.3, ymid+2.3, border="red")
```

B: Triangles or polygons, circles, and mathematical text

```
par(fig=c(0, 1, 0.01, 0.40), new=TRUE)
```

```
plot(0, 0, xlim=c(0, 13), ylim=c(0, 12), type="n")
polygon(x=c(10.7,12.8,12), y=c(7.5,8,11), col="gray", border="red")

## Draw a circle, overlay 2-headed arrow (code=3)
xcenter <- 11.7; ycenter <- 4; r=1.1
symbols(x=xcenter, y=ycenter, circles=r,
        bg="gray", add=TRUE, inches=FALSE)
arrows(x0=xcenter-r, y0=ycenter, x1=xcenter, y1=ycenter,
       length=.05, code=3)

## Use expression() to add labeling information
charht <- strheight("R")
text(x=xcenter-r/2, y=ycenter-charht, expression(italic(r)))
text(xcenter, ycenter+3.5*charht, expression("Area" == pi*italic(r)^2))
```

.

Figure C.4: This figure, intended to accompany Section 7.1.2, demonstrates the use of parameter settings to control various graphical features.

Note that the function `paste()`, used in line 5 of Panel A, turns the vector of numerical values `0:12` into a vector of character strings with elements `"0", "1", ..., "12"`. An alternative to `paste(0:12)` is `as.character(0:12)`.