The R System – An Introduction and Overview

J H Maindonald

Centre for Mathematics and Its Applications
Australian National University.

Languages shape the way we think, and determine what we can think about.
[Benjamin Whorf.]

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.]

John H. Maindonald, Centre for Mathematics & Its Applications, Mathematical Sciences Institute, Australian National University, Canberra ACT 0200, Australia, john.maindonald@anu.edu.au

http://www.maths.anu.edu.au/~johnm

July 18, 2006

This document aims to cover the basics of use of R as briefly as possible.

Contents

1 Introduction

There will be references to the following web sites:
CRAN (Comprehensive R Archive Network): `http://cran.r-project.org`
Go to the nearest mirror. Australian mirrors are:
`http://mirror.aarnet.edu.au/pub/CRAN` and `http://cran.ms.unimelb.edu.au/`.
R homepage: `http://www.r-project.org/`
`http://www.maths.anu.edu.au/~johnm/r/misc-data/`

1.1 Commentary on R

General

R runs on many types of system – Windows, Mac, Unix and Linux. It is free. Obtain it from a CRAN site (see above).

R has extensive graphical abilities that are tightly linked with its analytic abilities.

Much of the power of R for statistical analysis and for specialist graphics comes from the extensive enhancements that the packages build on top of the base system.

Although now relatively mature, the system gets continuing scrutiny, with improvements and enhancements appearing with each new release, i.e., every few months.

Though not perfect in this respect (!), the system has been developed with a keen regard to notions of good statistical practice.

Users should expect to encounter demands to improve their statistical knowledge, in order to use R effectively. The R community expects users to be serious about data analysis, to want more than a quick cook-book fix!

As noted, the R code that is in the base system and in the recommended packages gets unusually careful scrutiny. Nevertheless, there are traps. Take particular care with newer abilities, which may not have been much tested in regular use. Note also that some of the contributed packages may not have been much tested, except by their developers. [Such warnings apply, of course to any statistical system.]

At this time, R primarily serves two groups: statistical and allied professionals who wish to develop or require access to cutting edge tools, and working scientists who have such substantial and continuing data analysis problems that they justify time spent in the mastery of R.

Getting help

Although there is no official support for R, its informal support network, accessible from the r-help mailing list, can be highly effective. Details of this and other lists can be found on the home page for the R project: `http://www.r-project.org`. Be sure to check the available documentation before posting to r-help. Archives are available that can be searched for questions that have been previously answered.

Use of an editor as a run-time environment

The Windows implementation, and the Cocoa based GUI for Mac OS X, now offer a simple script editor that has a Run Line or Selection feature. There are various editors and associated interfaces to R that allow editing of code, again offering a single click Run Line or Selection. On Windows systems, the Tinn-R editor (`http://www.sciviews.org/Tinn-R/`) is an excellent option. ESS (Emacs Speaks Statistics), now fully operational for Windows as well as for Unix, is attractive for users who relish the power of the Emacs editor.

Data set size, and databases

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 that R's memory model allows does however have a cost for some types of operation, relative to systems that are highly efficient in the processing of data from file to file. An important step was the move, with the release of version 1.2, to a dynamic memory model.

The R system has limited database abilities which are unlikely, at present, to be much extended. Instead, the emphasis will be on extending and improving connections into widely used database systems.

The development model, and development strategies

An impressive developer skill base, and the use of the open source development model has been highly effective in ensuring the quality and correctness of the base system and of the recommended packages. The development model for R is broadly similar to that for the Linux operating system. Observe that, whereas Linux competes in the shadow of Microsoft, R is not obviously in the shadow of any other system!

Connectivity has been an important development focus. Better than duplicating abilities that are handled well in other systems is, often, the provision of interfaces into those systems. Systems for which there are interfaces to R include Python, SQL and other databases, parallel computing using MPI, and Excel using the DCOM software.

Unifying Ideas

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.

Expressions can be:

evaluated (of course)

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

Language structures can be manipulated, just like any other object (Manipulate formulae, expressions, argument lists for functions, . . .)

Trellis (lattice) graphics – graphs whose layout reflects data structure

There are many unifying computational features, e.g.

Any 'linear' model (lm, lme, etc) can use spline basis functions to fit spline terms. This extends to any other system of basis functions.

These ideas are not uniformly implemented right through R, reflecting the incremental manner in which R has developed.

Retrospect, prospect and alternatives to R

Ross Ihaka and Robert Gentleman, both at that time from the University of Auckland, developed the initial version of R, for use in teaching tool. It implements a dialect of the S language that was developed at AT&T Bell Laboratories for use as a general purpose scientific language, but with especial strengths in data manipulation, graphical presentation and statistical analysis. Since mid-1997, development has been overseen by a 'core team' of about a dozen people, drawn from many different institutions worldwide.

The commercial S-PLUS implementation of S had popularized the use of S as a language for scientific and statistical computation, and for graphics. The S language had, in the two decades up to

2000, a large user base among professionals and others. The R system tapped into this existing large user base. There were a number of roughly comparable systems – including Matlab, Scilab, Gauss, Python and Lisp-Stat – that might potentially have supplanted R. However these had much smaller existing bases in the institutions and groups that in due course drove the development of R. Note however the popularity of Matlab in the signal and image processing community.

Although with a syntax that looks superficially like that of C, the implementation of R has been heavily influenced by LISP. The R interpreter uses a model that is based on the Scheme dialect of LISP. Luke Tierney, and several others who had previously had a heavy involvement with Luke Tierney's Lisp=Stat system, are now actively involved in the ongoing development of R. See Tierney (2005), and other papers in the same volume of the *Journal of Statistical Software*

With the release of version 1.0 in early 2000, R became a serious tool for professional use. Since that time, the pace of development has been frenetic, with a new package appearing every week or two. There are now more than 400 packages available through the CRAN (Comprehensive R Archive Network) sites. Books that were specifically devoted to R began to appear in 2002.

Novice users will notice small but occasionally important differences between R and S-PLUS. Writers of substantial functions and (especially) packages will find larger differences. R's packages are now more wide-ranging in scope as S-PLUS libraries. Some specialised S-PLUS abilities may not be available in R or in R packages.

The R project has pushed boundaries and shown what is possible when experts in statistical computing work co-operatively to push boundaries. Its language model is however now dated. While many of those who have been involved in the development of R are pondering what might lie beyond R, this has not as yet led to the development of credible prototypes for an alternative. What will eventually overtake and in large part encompass R is anyone's guess.

The Statistics of Data Collection

The scientific context, which includes available statistical methodology, has crucial implications for the experiments that it is useful to do, and for the analyses that are meaningful. There are, in addition, constraints and opportunities that arise from computing software and hardware.

Statistics of data collection encompasses statistical *experimental design*, sampling design, and more besides. At base, the same issues arise in field, industrial, medical, biological and laboratory experimentation. The aim, as always, is to get maximum value from the use of all resources. The planning that is required will be most effective if based on sound knowledge of the materials and procedures used by experimenters. As we learn more about these issues, we gain the knowledge needed to design better experiments.

1.2 Installation and Updates

Versions of R are available, at no cost, for Windows 95 and later versions of Microsoft Windows for Linux, for Unix and for Macintosh systems 8.6 or later. It is available through the Comprehensive R Archive Network (CRAN). Australian users should obtain files that can be used for R installation from one of the sites:
`http://mirror.aarnet.edu.au/pub/CRAN`, `http://cran.ms.unimelb.edu.au/`.

Installation details vary between operating systems. A fresh install is typically required to take advantage of new major releases (e.g. moving from a 2.2 series release to a 2.3 series release) when they appear. For working through these notes, version 2.2.0 or later should be installed.

Once R has been installed, functions are available that will, from within R, install additional packages or update packages that are already installed, via an internet connection. At several points in these notes, the *DAAG* package will be required. If there must be a live internet connection that R can access, then you can install it by entering, from the R command line:

```
install.packages("DAAG")
```

Alternatively, on systems where a menu is available, do this via the menu.

For further information on `install.packages()`, enter `help(install.packages)` from within an R session. Note also `download.packages()` (this takes a list of package names and a destination directory, downloads the newest versions of the package sources and saves them in 'destdir') and `update.packages()` (outdated packages are reported and for each outdated package the user can specify if it should be automatically updated).

On Unix and Linux systems, the relevant gzipped tar files can be downloaded or copied across from a CD collection. They can then be installed using the shell command

R CMD INSTALL <package (.tar.gz file)>

On Windows systems, zip binaries that have been downloaded separately from CRAN can be installed by navigating to the Packages menu item Install package(s) from local zip file. Then navigate to the directory that holds the zip file(s), and select those that are to be installed.

For installation from the command line, call `install.packages()` with `pkgs` giving the files (with path, if necessary), and with the argument `repos=NULL`. If for example the binary **DAAG_0.79.zip** has been downloaded to **D:\tmp\**, it can be installed thus

```
install.packages(pkgs="D:/DAAG_0.79.zip", repos=NULL)
```

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

Help for installation under Windows

Windows users will find a great deal of helpful information on the web page
`http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/installation.html`

Data sets, additional to those in the DAAG package

Several data sets that relate to the present workshop are included in the R image file **misc.RData** that can be obtained from the url:
`http://www.maths.anu.edu.au/~johnm/r/misc-data/`
Other files that are available from this same directory include **travelbooks.txt**, **MouseArray.RData**, and the R scripts for these notes.

1.3 Documentation

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.

R News: Successive issues of *R News* contain much useful information. These can be copied down from one of the CRAN sites.

Contributed Documentation: There is an extensive collection of user-written documents on R that can be accessed by going to this same mirror site, and clicking (under Documentation) on **Contributed**. See also the links that John Fox gives on the web page for his book that is noted under the reference for his book.

Books: Subsection 11.1 includes references to a number of books. Recently, a number of new books on R have appeared. See `http://www.R-project.org/doc/bib/R.bib` for a list that is updated regularly.

2 An Overview of R

Command prompt (`>`)	Enter commands following the prompt, e.g.
	`> 2 + 2        # Calculate 2 + 2`
Quitting	To quit from R type
	`q()            # NB q(), not q`
Case matters	`volume` is different from `Volume`
Help	Use it often. For example
	`help()         # Information on the help function`
	`help(plot)     # help on the plot function`
Demonstrations	Type `demo()` to get details of demonstrations.
Examples	Many of the help pages include examples. To run them, type, e.g.
	`example(plot)  # run all the examples for plot()`
Assignment	The assignment symbol is `<-`, e.g.
	`volume <- c(351, 955, 662, 1203, 557) # c = concatenate`
	`# Store the column of numbers in volume`
Other topics	Simple arithmetic operations; simple plots.

2.1 Use of the console (i.e., command line) window

The command line prompt, i.e. the `>`, is an invitation to start entering commands. For example, type `2+2` and press the Enter key. The following appears on the screen:

```
> 2+2
[1] 4
>
```

The result is 4. The `[1]` says, a little strangely, "first requested element will follow". Here, there is just one element. The `>` indicates that R is ready for another command.

The exit or quit command is

```
> q()
```

Depending on the platform, alternatives may be to click on the File menu and then on Exit, or to click on the **X** in the top right hand corner of the R window. There will be a message asking whether to save the workspace image. Clicking Yes (the safe option) will save the objects that remain in the workspace – any that were there at the start of the session and any that have been added since.

Commands may continue over more than one line. By default, the continuation prompt is

`+`

As with the `>` prompt, this is generated by R. Any attempt to include it in the code that is entered will generate an error!

For the names of R objects or commands, case is significant. Thus `Myr` (millions of years), which we will use below, is different from `myr`. For file names on Windows systems, the Microsoft Windows conventions apply, and case does not distinguish file names. On Unix systems (the Mac OS X version of Unix is an exception) case in file names is significant.

Further points are:

- o The quit command ("quit from the R session") is the function call `q()`. Typing `q` on its own, without the parentheses, displays the text of the function on the screen.
- o Multiple commands may appear on a line, with the semicolon (`;`) as the separator.
- o The `#` symbol indicates that what follows, on that line, is comment.

Practice with R commands

Try the following

```
1:5          # The numbers 1, 2, 3, 4, 5
mean(1:5)
sum(1:5)     # Apply the sum function to the
             # the vector of numbers 1, 2, 3, 4, 5
(2:5)^10     # 2 to the power of 10, 3 to the power of 10, ...
log2(c(0.5, 1, 2, 4, 8)) # Values that differ by a factor of 2
                         # are, on this scale, one unit apart.
```

The R language has the abilities for evaluating arithmetic and logical expressions that are available in most languages. It uses functions to extend these basic arithmetic and logical abilities.

2.2 Demonstrations

There are a number of demonstrations that give useful indications of R's abilities, especially for graphics. To get a list of available demonstration, type:

```
demo()
```

Visually interesting demonstrations are:

```
demo(image)
demo(graphics)
demo(persp)
demo(plotmath)       # Mathematical symbols can be visually interesting
library(lattice)
demo(lattice)        # Demonstrates lattice graphics
```

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.

Try also

```
demo(package = .packages(all.available = TRUE))
```

Also interesting is:

```
library(vcd)         # The vcd package must of course be installed.
demo(mosaic)
```

2.3 Help, and examples

All built-in functions have help files, which can be accessed using the help() command. Try typing

```
help(help)           # Get help on help()
help(mean)
```

Often, a good way to learn how to use a function is to run the examples that are included in the help file. The function `example()` checks the help page for examples, and runs them. Be warned that the examples for relatively simple functions can be non-trivial. Or they may be extensive. Try:

```
example(mean)
example(plot)        # This gives a species of movie show
par(ask=TRUE)        # Ask before displaying the next graph
example(plot)        # Now the plots are displayed one at a time.
example(lowess)       # Fit a smooth curve to scatterplot data
example(image)
example(contour)
example(filled.contour)
par(ask=FALSE)        # From here on, plot without asking
```

Typically, several examples will run one after the other. The code appears on the screen. To re-run an example, look on the screen for the code that was used, and copy or type it following the command line prompt. The examples sometimes illustrate technical details that may puzzle novices.

Use `help.search()` to look for functions that include a specific word in their alias or title. For example, in order to look for a function for bar plots, try

```
help.search("bar")
```

This draws attention to the function `barplot()`. As a first step in investigating the function, try

```
par(ask=TRUE)
example(barplot)
par(ask=FALSE)
```

Several of fhe examples focus on sophisticated barplot abilities. Finally, type in `help(barplot)`, and read the information on the help page.

The function `apropos()` lists all functions (or other R objects) whose names include the text string that is given as the function argument. For example

```
> apropos("str")
[1] "R.version.string" "ls.str" "lsf.str"
[4] "str" "str.POSIXt" "str.data.frame"
[7] "str.default" "str.logLik" "strftime"
[10] "strheight" "stripchart" "strptime"
[13] "strsplit" "structure" "strwidth"
[16] "strwrap" "substr" "substr<-"
[19] "substring" "substring<-"
>
```

Finally, note that `help.start()` should start a browser window that gives access to a variety of help information and documentation.

Vignettes

Vignettes are pdf documents that describe the abilities in packages for R. To get a list of vignettes in all installed packages type:

```
vignette()
```

To get a name(s) of vignette(s), if any, for specific packages, type, e.g.:

```
vignette(package="graph")
vignette(package="e1071")   # e701 includes functions for
                            # Support Vector Machines (SVMs)
vignette(package="mcmc")    # Markov Chain Monte Carlo
```

The package *graph* has two vignettes, **clusterGraph** and **graph**, the package *e1071* has the vignette **svmdoc**, the package *mcmc* has the vignette **demo**. To use the default pdf viewer to display a specific vignette, type, e.g.:

```
vignette("graph")     # Equivalent to vignette(topic="graph")
```

2.4 A Short R Session

We will work with the data set shown in Table 1:

	Volume (mm^3)	Weight (g)	type
Aird's Guide to Sydney	351.00	250.00	Guide
Moon's Australia handbook	955.00	840.00	Guide
Explore Australia Road Atlas	662.00	550.00	Roadmaps
Australian Motoring Guide	1203.00	1360.00	Roadmaps
Penguin Touring Atlas	557.00	640.00	Roadmaps
Canberra - The Guide	460.00	420.00	Guide

Table 1: Weights and volumes, for six Australian travel books.

Entry of vector elements from the command line

Data may be entered from the command line, thus:

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

Now enter the descriptions:

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

Notes:

- The assignment symbol is `<-`
- Read the symbol `c` as "concatenate". The function `c()` joins elements together into a vector. (For `volume` and `weight` the elements were numbers, while for `description` the elements were text strings.
- Typing the name of an object causes the printing of its contents. Try typing `volume`. This applies to functions as well as data objects. For example, try typing `q`, or `mean`.

Operations with `vectors`

Here are the values of `volume`

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

Here are various arithmetic operations:

```
> # Final element of volume
> volume[6]
[1] 460
> ## Ratio of weight to volume, i.e., density
> round(weight/volume,2)
[1] 0.71 0.88 0.83 1.13 1.15 0.91
```

Notice the use of `#` to preface the comment, which will be ignored by the command line interpreter.

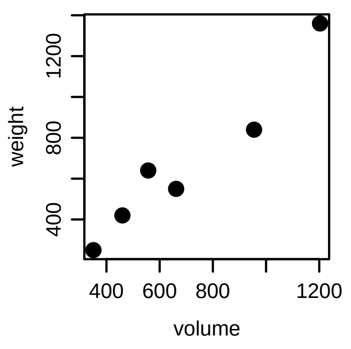


Figure 1: Weight versus volume, for six Australian travel books.

A simple plot

To plot (`weight` against `volume` (Figure 1), type one of the following:

```
plot(weight ~ volume, pch=16, cex=1.5)
  # pch=16 gives a solid blob as plotting symbol
  # cex=1.5 makes points 1.5 times the default size
## Alternative
plot(volume, weight, pch=16, cex=1.5)
```

The parameter `weight ~ volume` is a graphics formula. The "formulae" that are used in specifying models, and in the functions `xtabs()` and `unstack()`, take a similar form.

The axes can be labeled:

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

Labeling of the points (e.g., with the species names) can be done interactively, with the `identify()` command. Type:

```
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 you want labelled.

Depending on the computer system, either click outside the graphics area to terminate the labelling, or click the right mouse button outside the figure area.

Alternatively, use `text()` to place labels on all the points.

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 and so on. A later section (Section 5) is devoted to graphics.

2.5 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.

Use `q()` to quit from R. If newly created objects are to be retained, save the workspace upon quitting.

Useful help functions are `help()` (for getting information on a known function), `help.search()` (for searching for a word that is used in the header for the help file), and `apropos()` (for identifying functions that include a particular text string as part of their names). Note also the use of `help.start()`, to start a browser window from which R help information can be accessed.

3 The Working Environment of an R Session

The Working Environment:

Working directory	R will by default read files from this directory, or write files to it
Object	A data structure or function that R recognizes
Workspace	This holds objects that relate to the user's current session
`read.table()`	Use this function to read data, from a file, into the workspace
Image files	These store a collection of R objects, e.g., the workspace conents. (The expected file extension is **.RData** or **.rda**)
`save.image()`	Use, at any time in an R session, to store all or some workspace contents. Or use the relevant menu item to achive the same effect.
Packages	Packages are collections of R functions and/or data.
`library()`	Use this function to attach a package, e.g. `library(DAAG)` (Recommended packages are included with binary distributions of R. Other packages must be installed, before they can be attached)
Objects in R	Both data objects and functions are referred to as objects Use `ls()` to list the objects in the current workspace.
Search path	The search path determines where R looks for objects that are internal to R, or to one of its packages. The items on the search path have the name "databases".

A system such as R requires access to directories where relevant files can be stored or accessed. Files that belong to the R system are by default placed somewhere that is (usually) sensible. Novice users should not need to concern themselves with the details.

It may be necessary to know:

- the location of the current working directory, where R will by default look for files or store files that are external to R;
- the search path, which is important for determining where R looks for objects that are required in an R session. Here we are talking about objects that are internal to R.

3.1 The Working Directory and the Workspace

The *working directory* is the directory in which R will by default look for files, and save files. It pays to have a separate working directory, and associated workspace, for each major project.

Listing Workspace Contents

To see a list of the objects or of selected objects that are in the workspace, type a command such as the following:

```
> ls()
[1] "volume" "weight"
> ls(pattern="^w")
[1] "weight"
```

In a long session in a working directory, cautious users will from time to time save the current workspace image as a backup, perhaps first using `rm()` to remove objects that are no longer required. The command `save.image()`) will save everything in the workspace, by default into a file with the name **.RData** in the working directory. On implementations that offer a menu, this can alternatively be done by clicking on the relevant menu item.

Before saving the workspace, consider use of `rm()` to remove objects that are no longer required. Saving the workspace image will then save everything that remains, by default into a file called **.Rdata** in the working directory.

Upon quitting from R (type `q()`, or on Macs and Windows use the relevant item on the menu), users are asked whether they wish to save the current workspace.[1] The workspace is reloaded next time an R session is started in the directory.

The workspace is at the base of a search list that gives access to packages, objects in other directories, etc.

Setting the Working Directory

When working from a Unix or Linux command line, the working directory is the directory in which you start. When an session is started by clicking on an icon, the session will start in a working directory that is associated with the icon. The default choice, usually an R installation directory, is not a good choice for long-term use, and should be changed. On Windows PC systems, change the Start In directory that is specified in the Preferences setting for the icon.

It is good practice to use a separate working directory 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.

It is also possible to change the working directory once a session has been started. This can be done either from the menu (if available) or from the command line. Before making such a change, be sure to save the existing workspace, if it is to be kept. Then, once the working directory has been changed, load the new workspace.

3.2 Saving and retrieving R objects

Image files, created using `save()` or `save.image()`, may contain arbitrary R objects. One or more objects can be saved to an image file at any time during a session. Upon quitting a session, the user is offered the option of saving the workspace in the default image file. The following demonstrate the explicit use of the `save()` and `load()` commands:

```
save(volume, weight, file="books.RData")
  # Can save many objects in the same file
load("books.RData")          # Recover the saved objects
```

The function `save.image()` is a variation on `save()` that saves the contents of the workspace, by default in the file **.RData**. The contents of any **.RData** file in the working directory are automatically loaded when a new session is started.

An alternative to saving the objects in an image file is to save them, in a text format, as dump files:the above use of `save()` is:

```
dump(c("volume", "weight"), file="books.R")
```

The objects can be recreated from this “dump” file by inputting the lines of **books.R** one by one at the command line. The following command restores both objects to the workspace:

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

For day to day use, image **.RData** files are in general preferable to dump files. The same checks are performed on dump files as if the text had been entered at the command line. These may be unwanted, and they slow down entry of the data or other object.

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

[1]Users of Linux should however note that clicking on the × in the upper right of the Window in order to quit may not, depending on the window manager, give the option to save the workspace.

3.3 Input of Data

For input into data frames, the most important function is `read.table()`. The following assumes that the file **travelbooks.txt** (on my web page) is in the working directory. Here is a listing of the first of these files:

	thickness	width	height	weight	volume	kind
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

Notice that the first column has no header information. A suitable way to read in these data is:

```
# Row 1 of the file gives column names. Column 1 gives row names
travelbooks <- read.table("travelbooks.txt")
# The following is safer and more explicit
travelbooks <- read.table("travelbooks.txt", header=TRUE, row.names=1)
```

The object `travelbooks` is a data frame. Data frames are pervasive in R. Most datasets that are included with R packages are supplied as data frames.

This data frame has column and row names. The first seven columns are numeric. The final column is a factor, though for present purposes it can be treated as a character vector. There are various ways to access the columns. The following will do for now:

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

3.4 Writing of data frames to text files

Data frames can be stored as text files. Use the function `write.table()` to write a data frame to a text file.

More generally, to save several objects (data frames or any other R object) in the one file, use `dump()` (to save in a text format) or `save.image()`, as noted above.

3.5 Installations, packages and sessions

3.5.1 The architecture of an R installation

An R installation is structured as a library of packages.

- All installations should have the base packages (one of them is called *base*) , which provide the superstructure for other packages.
- Binaries that are available from CRAN sites include, also, all the recommended packages.
- Other packages can be installed as required.

A number of packages are by default attached at the start of a session. Other packages can be attached (use `library()`) as required. To discover which packages have been attached, enter:

```
sessionInfo()
```

3.5.2 The search list: attach() and library()

The R system has a search path where it looks for objects. This can be changed in the course of a session. To get a snapshot of the search list, type:

```
> search()
 [1] ".GlobalEnv"        "package:xtable"    "package:MASS"      "package:vcd"
 [5] "package:methods"   "package:stats"     "package:graphics"  "package:utils"
 [9] "Autoloads"         "package:base"
```

Technically, these are called "databases". The search list is used to structure the search for R objects.

The database ".GlobalEnv" is the user's workspace, which is at the base of the search path. The `attach()` and `library()` commands extend the search list.

3.5.3 R packages

The use of `library()` to load a new R package extends the search list. The system then looks in the package database for objects that are not in the user workspace.

If at some point (often the end of the session) the workspace is saved, and objects that were added have not been explicitly removed, they will be saved as part of the workspace. If saved in the default **.RData** image file in the working directory, the workspace will be automatically loaded when a new session is next started in that working directory.

Use the function `.path.package()` to get the path of a currently attached package. By default, this information is given for all loaded packages.

3.5.4 The attachment of image files

As noted earlier, the function `attach()` extends the search list, by simplifying access to the columns of data frames or to the elements of lists, or by giving access to an image file that is stored somewhere.

Additionally, any R image file can be attached, either from the current working directory, or from any other directory. For example:

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

The workspace then has access to objects in the file **books.RData**. The file becomes a further "database" on the search list, separate from the workspace. If however the object is modified, the modified copy does become part of the workspace.

In order to detach such a database, proceed thus:

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

3.6 Summary

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

User-created objects appear appear in the workspace. At the end of a session (and perhaps from time to time during the session), an image of the workspace will typically be saved into the working directory.

It is usually best to keep a separate workspace and associated working directory, for each major project.

4 Objects and Functions

Key notions and language structures:

Vector	Collects together elements that are all of the one mode. (Possible modes are "logical", "integer", "numeric", "complex", "character") and "raw")
Factors	Use for categorical data. They are often invaluable for specifying models. (Challenge: Explain why factors are not vectors, even though they behave pretty much like vectors! Columns of data frames can be factors)
Data frame	A list of columns, all of the same length, but perhaps of different modes. (Data frames are often the best way to supply data to Modeling functions)
Lists	Lists group together an arbitrary collection of objects (These have the recursive list structure that computer buffs might expect.)
Missing values	The symbol is `NA`. The handling of `NA`s can be tricky.
Generic functions	They examine the object given as argument, before deciding what action is needed. Examples include `print()`, `plot()` & `summary()`
Modeling functions	These fit statistical models. Thus note `lm()`, for *linear* modeling
Model objects	These store information from a call to a modeling function, such as `lm()` (They serve as a repository from which to extract output information)
Packages	Packages are collections of R functions and/or data.
`library()`	Use this function to attach a package, e.g. `library(DAAG)` (Recommended packages are included with binary distributions of R. Other packages must be installed, before they can be attached)
The Search List	The current list specifies "databases" where R looks, in turn, for objects. Specify `search()` to display the search list.

4.1 Data Frames – Lists of Vectors

The following demonstrates the use of a data frame to group together, under the name `travelbooks`, the several columns of Table 1.

```
## NB, the row names will now be shortened
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,  # Include values of weight, entered earlier
   volume = volume,  # Include values of volume, entered earlier
   kind = c("Guide", "Guide", "Roadmaps", "Roadmaps", "Roadmaps", "Guide"),
   row.names = description
)
## Remove objects that are not now needed.
rm(volume, weight, description)
```

The vectors `volume`, `weight` and `description` had already been entered, and it was not necessary to re-enter them. 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 different columns of a data frame can be any of the modes logical, or numeric or character; there are other possibilities also. All elements in a column must of course have the same mode.

The function `read.table()` takes data from a text file (or from the clipboard, specify `file="clipboard"`) and places them in a data frame.

The Use of Data Frames

Data frames can be used in the following ways:

o They can be manipulated directly; see below for some of the possibilities

- o They can be attached (use the `attach()` function), making their columns available by name. Alternatively, use `with()`, which makes the data frame available for the duration of the command.
- o In modelling and graphics functions, there is often a `data` parameter that can be used to specify the data frame from which variable and factor names will be taken. In effect, the data frame is attached for the duration of the analysis, though in the environment of the function.

The command **attach(travelbooks)** extends R's search list to include the columns of the data frame, thus:

```
> volume

Error: Object "volume" not found
> attach(travelbooks)
> volume
[1]  351  955  662 1203  557  460
```

Once `travelbooks` is for the time being no longer required, it is advisable to detach it, thus:

```
detach(travelbooks)
```

This reduces the risk of ambiguity because two data sets that are attached at the same time have one or more column names in common, or because the name of column in a data frame is the same as the name of an object in the workspace.

The following, which displays the values of `volume`, illustrates the use of `with()` to make the data frame available for the duration of the command:

```
## Display the column volume, from the travelbooks data frame
with(travelbooks, volume)
```

Data frames as lists

Data frames are lists of column vectors. One consequence is that use of the subscript notation to extract a row from a data frame gives a different data structure from use of the subscript notation to extract a column. Specifically:

The result of `travelbooks$volume` or `travelbooks[,"volume"]` or `travelbooks[,1]` is a vector.

The result of `travelbooks["robin", ]` or `travelbooks[3, ]` is a data frame, i.e., a special form of list. The syntax `unlist(travelbooks["robin", ])` can be used to turn such a list into a vector.

Data Frames versus Matrices

Matrices are rectangular arrays in which all elements have the same class. Internally, matrices are one long vector in which the columns are strung out one after the other. For a regression calculation, a data frame is necessary. Depending on the calculation that is to be performed, matrices and data frames may require a different syntax, or even explicit conversion from one to the other.

Matrices will be discussed in Subsection 4.4.

4.2 Input of Data – some further comments

Recall the earlier use of `read.table()`:

```
travelbooks <- read.table("travelbooks.txt", header=TRUE, row.names=1)
```

Points to note are:

> If there is one less column heading than there are columns, the first column of input is, by default, used for row names. Use of the argument `heading=FALSE` overrides this default.
>
> If all rows have the same number of values, the default is to take the first row of the file as the first row of data, i.e., it is assumed that there is no header row.
>
> The final column (`nest`) will, by default, be a factor. To prevent such type conversions, specify `as.is=TRUE`.
>
> If the first row of the file has the same number of fields as later rows, it is assumed that this is the first row of data, not a header row. Alternatively, specify `header=FALSE` to ensure this. The default is then to give the columns the names `V1`, `V2`, ...

4.3 The Use of Data Frames in Correlation and Regression

The purpose here is to indicate how R uses data frames in statistical calculations, using the data in the data frame `travelbooks`.

We will first add, to the data frame `travelbooks`, new columns `area` (area of page), and `density` (`weight` to `volume` ratio):

```
> travelbooks$area <- with(travelbooks, width*height)
> travelbooks$density <- with(travelbooks, weight/volume)
> names(travelbooks)   # Check full set of column names
[1] "thickness" "width"     "height"    "weight"    "volume"    "kind"
[7] "area"      "density"
```

A first step is to call the generic function plot with `travelbooks` as argument:

```
pairs(travelbooks[, -6])  # Omit column 6
## Alternative
plot(travelbooks[,-6])
```

Not unexpectedly, there are relationships between `width` and `height`, and between `weight` and `volume`. Less expected, perhaps, is the indication of a relationship between `density` and `area`.

It is possible, in R, to "plot" just about any data object.

Correlation calculations

The function `cor()`, with a matrix or data frame as argument, calculates the correlation matrix for the columns. By default, it gives the Pearson product-moment linear correlation. For example:

```
> round(cor(travelbooks[, -6]),2)
          thickness width height weight volume  area density
thickness      1.00 -0.47  -0.77   0.40   0.58 -0.59   -0.16
width         -0.47  1.00   0.91   0.44   0.28  0.98    0.80
height        -0.77  0.91   1.00   0.13  -0.08  0.97    0.67
weight         0.40  0.44   0.13   1.00   0.97  0.30    0.69
volume         0.58  0.28  -0.08   0.97   1.00  0.12    0.51
area          -0.59  0.98   0.97   0.30   0.12  1.00    0.77
density       -0.16  0.80   0.67   0.69   0.51  0.77    1.00
```

Dependence of `weight` on `volume`

One way to investigate is to examine the regression relationship:

```
> wtvol.lm <- lm(weight ~ volume, data=travelbooks)
> wtvol.lm

Call:
lm(formula = weight ~ volume, data = travelbooks)

Coefficients:
(Intercept)       volume
   -137.768        1.167
```

Notice that we saved the regression (`lm`) object, with the name `wtvol.lm`. This can be printed or summarized or used to extract other information, or be used as the basis for various graphs.

The large intercept is perhaps unexpected. We might expect `weight` to be proportional to `volume`. To force the line through the origin, do the following:

```
wtvol0.lm <- lm(weight ~ -1 + volume, data=travelbooks)
wtvol0.lm
```

Summary information – print(), summary() and plot()

All three functions are *generic*. The effect depends on the class of object that is printed (on the screen) or summarized (again, on the screen), or plotted.

The function `print()` typically displays relatively terse output, while `summary()` may display more extensive output. When used with `lm` objects, `print()` calls `print.lm()`, while `summary()` calls `summary.lm()`.

Actually `summary()` calls `Usemethod("summary")`, which notes that `myr.lm` is an lm object and therefore calls `summary.lm()`. The extent of information given by `print()` and `summary()` varies from one type of model object to another.

Compare the outputs from the following:

```
print(wtvol.lm)
  # Equivalent to typing wtvol.lm at the command line
summary(wtvol.lm)
## The following gives diagnostic plots
par(mfrow=c(2,2))  # Subsequent plots appear in a 2 x 2 layout
plot(wtvol.lm)
par(mfrow=c(1,1))  # Reset to 1 plot per page, for any later plots
```

Note also the generic extractor functions residuals(), `coefficients()`, and `fitted.values()`. These can be abbreviated to `resid()`, `coef()`, and `fitted()`.

4.4 Data Objects

Although it is often convenient to distinguish data objects from functions, there is not a rigid distinction. For, example, data that will be analyzed may be stored along with functions in the same object.

4.4.1 Vectors

Examples of vectors are

```
c(2,3,5,2,7,1)
3:10 # The numbers 3, 4,.., 10
c(T,F,F,F,T,T,F)
c("cherry","mango","apple","prune")
```

As noted in Chapter 1, vectors may have mode logical, numeric or character. The first two vectors above are numeric, the third is logical (i.e. a vector with elements of mode logical), and the fourth is a string vector (i.e. a vector with elements of mode character).

The `c` in `c(2, 3, 5, 7, 1)` has the meaning is: "Join (concatenate) these numbers together into a vector". Existing vectors may supply some or all of the elements that are to be concatenated.

The missing value symbol, which is `NA`, can be included as an element of a vector.

Factors will be discussed in Subsection 4.4.6. There is one important respect (the use of `c()`) in which they differ from vector.

Subsets of Vectors

There are three common ways to extract subsets of vectors.

1. Specify the subscripts of the elements that are to be extracted, e.g.

```
> x <- c(3,11,8,15,12)      # Assign to x the values
3, 11, 8, 15, 12
> x[c(2,4)]          # Extract elements (rows) 2 and 4
[1] 11 15
```

Negative numbers may be used to omit elements:

```
> 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.

```
> x>10 # This generates a vector of logical
(TRUE or FALSE)
[1] F T F T T
> x[x > 10]
[1] 11 15 12
```

Arithmetic relations that may be used in the extraction of subsets `>=`, `==`, and `!=`. The first four compare magnitudes, `==` tests for equality, and `!=` tests for inequality.

3. For vectors of named elements, the elements may be identified by name:

```
> library(DAAG)
> cuckooEgglengths <- cuckoohosts[,1]
> cuckooEgglengths
[1] 22.3 23.1 22.5 22.6 23.1 21.1 22.6
>
> ## Assign names to the vector elements
> names(cuckooEgglengths) <- rownames(travelbooks)
> cuckooEgglengths
 meadow pipit hedge sparrow         robin      wagtails
         22.3          23.1          22.5          22.6
   tree pipit          wren  yellow ammer
         23.1          21.1          22.6
>
```

```
> ## Names can be used to extract elements
> cuckooEgglengths[c("hedge.sparrow", "robin", "wren")]
hedge sparrow          robin           wren
         23.1           22.5           21.1
```

4.4.2 Matrices as vectors

A matrix is a vector that is laid out in a two-dimensional tabular format and has a dimension attribute. The dimension attribute can be examined thus:

```
> travelmat <- as.matrix(travelbooks[, 1:4])
> dim(travelmat)          # Equivalent to attr(molmat, "dim")
[1] 6 4
> attr(travelmat, "dim")
[1] 7 4
```

The dimension attribute can be changed or removed, thus:

```
> dim(travelmat) <- NULL  # Columns of travelmat become one long
                          # vector, stacked in order of columns.
> travelmat
 [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
> # Use of as.vector(travelmat) is however preferable to the above
> # For these data, why would one do this?
```

Matrix Manipulations

Let X, Y and B be numeric matrices. Some of the possibilities are:

```
X + Y        # Elementwise addition (The dimensions must agree)
X * Y        # Elementwise multiplication (The dimensions must agree)
X %*% B      # Matrix multiplication (X is n by k; B k by m)
solve(X, Y) # Solve X B = Y (X is n by k, Y n by m, B k by m)
svd(X)       # Singular value decomposition of X
qr(X)        # QR decomposition of X.
```

Use of matrices for efficient computation For working with large numerical arrays, the matrix structure can allow much faster computations than are possible with data frames. See Section 9.4

4.4.3 Data frames versus matrices

Data frames that consist entirely of numeric data are in some (but not all) contexts interchangeable with numeric matrices. There are however important differences. In some applications, and notably in the analysis of microarray data, it is likely to be necessary to work with both of these types of object.

One way to check whether an object is a data frame or matrix is to see what value the function `length()` returns:

```
> length(travelbooks)        # travelbooks is a list of 7 vectors
[1] 5
> length(as.matrix(travelbooks[,1:4]))   # matrix has 28 elements
[1] 28
```

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
```

Tables may similarly be converted into data frames. Use `as.data.frame.table()`.

Subsets of data frames and matrices

To extract the first three rows of the data frame `travelbooks`, specify `travelbooks[1:3,]`. For columns 1 to 3, specify `travelbooks[1:3,]`. Additionally, a vector of logicals (`TRUE`s and `FALSE`s) may be used to extract rows and/or columns, thus:

```
> travelbooks$weight > 500 ## Result is a vector of logicals
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> travelbooks[travelbooks$weight > 500, ]
                             thickness width height weight volume
kind
Moon's Australia handbook          3.9  13.1   18.7    840    955    Guide
Explore Australia Road Atlas       1.2  20.0   27.6    550    662 Roadmaps
Australian Motoring Guide          2.0  21.1   28.5   1360   1203 Roadmaps
Penguin Touring Atlas              0.6  25.8   36.0    640    557 Roadmaps
```

4.4.4 Lists

A list is a collection of arbitrary objects. Above, we encountered data frames. A data frame is a specialised form of list. The list elements hold the columns of the data frame, which must all be of the same length. Here we note two other common types of list, the first of importance for microarray work, and the second of general importance.

Lists of matrices

Included with the data in the *sma* package are the objects `mouse.data` and `mouse.lratio`. These hold data from a microarray experiment, in which gene expression intensities for knockout mice were compared with expression intensities for normal mice. Assuming that the *sma* package is installed, the following will bring it into the workspace:

```
library(sma)
data(MouseArray)      # MouseArray holds several data objects
ls()                  # Check contents of workspace
```

The object `mouse.data` is a list. It has four elements, with names `R`, `G`, `Rb` and `Gb`. Each of these elements is a matrix, with dimensions 6384 spots by 6 slides. The object `mouse.lratio` is a list with elements `A` and `M`. Again each element is a matrix with dimensions 6384 spots by 6 slides. [The values in `M` are the logarithms to base 2 of the ratios of “red” to “green” intensities, while the values in `M` are averages of the logarithms of the “red” and “green” intensities.]

4.4.5 Model objects are lists

Model objects are typically lists, consisting of elements that can be of very different types. We will use the output from the regression calculation with data in `travelbooks`, stored in the `lm` object `wtvol.lm`, as an example. Note first that `lm` is, in effect, a mnemonic for linear model.

Regression objects hold output structures from the regression output that can be used for further calculations. It stores the information in a list, with named elements. To see the names of the list elements in `clock.lm`, type:

```
> names(wtvol.lm)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

To extract individual list elements type, e.g.:

```
> wtvol.lm$call
lm(formula = weight ~ volume, data = travelbooks)
```

Where an extractor function is available, use this in preference to accessing the specific list elements. Thus use:

```
> coef(wtvol.lm)
(Intercept)      volume
-137.767923    1.166812
> resid(wtvol.lm)
      Aird's Guide to Sydney    Moon's Australia handbook
                   -21.78300                   -136.53728
Explore Australia Road Atlas    Australian Motoring Guide
                   -84.66144                     94.09341
       Penguin Touring Atlas          Canberra - The Guide
                   127.85379                     21.03453
```

Notice that the residuals carry the row names. We might prefer shorter row names.

The various R modeling functions all return their own particular type of model object, always stored as a list. Note also that data frames are a specialized form of list, with the restriction that all columns must be vectors that all have the same length.

4.4.6 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. They provide, in data frames, the default way to store text string information.

The character vector `c("cherry","mango","apple","prune","cherry","prune")` might equally well be stored as a factor, thus:

```
> fruit <- c("cherry","mango","apple","prune","cherry")
> fruitfac <- factor(fruit)
> as.numeric(fruitfac)
[1] 2 3 1 4 2 4
> levels(fruitfac)
[1] "apple"  "cherry" "mango"  "prune"
```

Notice that, by default, the levels are taken in alphanumeric order.

Internally, the factor is stored as the integer vector 2, 3, 1, 4, 2, 4. It has stored with it the table:

1	2	3	4
"apple"	"cherry"	"mango"	"prune"

Thus, the numeric values are codes for text strings, with information that matches the codes to a unique set of text strings stored in the separate table. Once stored as a factor, the space required for storage can, depending on the lengths of the text strings and their frequencies of occurrence, be greatly reduced.

The order can be specified. For example:

```
> ## Take fruit in order of stated glycemic index (15, 22, 38, 55)
> fruitfac <- factor(fruit, levels=c("prune","cherry","apple","mango"))
> levels(fruitfac)
[1] "prune"  "cherry" "apple"  "mango"
```

Changing the levels in this way can be useful where factor levels are used, as in some *lattice* graphs that have one panel for each factor level, to determine the order in which plots appear.

Note that 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.

Incorrect spelling of the level names generates missing values, for the level that was mis-spelled. Try:

```
fruitfac <- factor(fruitfac, levels=c("prune","cherry","Apple","mango"))
table(fruitfac)      # The number of Apples is given as 0
```

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

```
> fruitfac <- factor(c("cherry","mango","apple","prune","cherry"))
> fruitfac == "cherry"
[1]  TRUE FALSE FALSE FALSE  TRUE
```

Subsection 7.2 has detailed examples of the use of factors in model formulae and the resultant model matrices.

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
```

4.4.7 Missing Values, Infinite Values and NaNs

Problems with missing values, infinite values and `NaN`s are a common reason why functions fail. An understanding of the conventions for arithmetic with NAs will reduce the scope for unwelcome surprises.

The missing value symbol is `NA`. 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:

```
> is.na(c(1, NA, 3, 0, NA))
[1] FALSE TRUE FALSE FALSE TRUE
```

The expression `NA == NA` returns `NA`, and cannot be used to test for a missing value.

```
> NA == NA
[1] NA
```

Note that the attempt to assign values to an expression whose subscripts include missing values generates an error.

```
> y <- c(1, NA, 3, 0, NA)
> y[y > 0]
[1]  1 NA  3 NA
> y[y > 0] <- c(11, 12)
Error: NAs are not allowed in subscripted assignments
```

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

Identifying and processing 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(test.df)
[1] TRUE TRUE FALSE
> na.omit(test.df)
x y
1 1 1
2 2 2
```

4.4.8 Information on Data Objects

We will make later work with the `possum` data frame. The function `str()` gives basic information on the object that is given as argument.

```
> str(possum)
‘data.frame’: 104 obs. of 14 variables:
$ case : num 1 2 3 4 5 6 7 8 9 10...
$ site : num 1 1 1 1 1 1 1 1 1 1...
$ Pop : Factor w/ 2 levels "Vic","other": 1 1 1 1 1 1 1 1 1 1...
$ sex : Factor w/ 2 levels "f","m": 2 1 1 1 1 1 2 1 1 1...
$ age : num 8 6 6 6 2 1 2 6 9 6...
$ hdlngth : num 94.1 92.5 94 93.2 91.5 93.1 95.3 94.8 93.4 91.8...
$ skullw : num 60.4 57.6 60 57.1 56.3 54.8 58.2 57.6 56.3 58...
$ totlngth: num 89 91.5 95.5 92 85.5 90.5 89.5 91 91.5 89.5...
$ taill : num 36 36.5 39 38 36 35.5 36 37 37 37.5...
$ footlgth: num 74.5 72.5 75.4 76.1 71 73.2 71.5 72.7 72.4 70.9...
$ earconch: num 54.5 51.2 51.9 52.2 53.2 53.6 52 53.9 52.9 53.4...
$ eye : num 15.2 16 15.5 15.2 15.1 14.2 14.2 14.5 15.5 14.4...
$ chest : num 28 28.5 30 28 28.5 30 30 29 28 27.5...
$ belly : num 36 33 34 34 33 32 34.5 34 33 32...
```

4.5 Functions

4.5.1 Built-In Functions

Common useful functions, for use with vectors

Common Useful Functions are

```
print()            # Prints a single R object
cat()              # Prints multiple objects, one after the other
length()           # Number of elements in a vector or of a list
mean()
```

```
median()
range()
unique()          # Gives the vector of distinct values
diff()            # Vector of first differences
                  # N. B. diff(x) has one less element than x
sort()            # Sort elements into order, but omitting NAs
order()           # x[order(x)] orders elements of x, with NAs last
cumsum()
cumprod()
rev()             # reverse the order of vector elements
any()             # Returns TRUE if there are any missing values
is.factor()       # Returns TRUE if the argument is a factor
is.na()           # Returns TRUE if the argument is an NA
                  # NB also is.logical(), is.matrix(), etc.
str()             # Information on an R object
args()            # Information on arguments to a function
```

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

```
> mean(c(1, NA, 3, 0, NA), na.rm=T)
[1] 1.3
```

4.5.2 User-defined functions

The R language is a functional language. 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:

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

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.[2]

```
> hetero <- c(.43,.25,.53,.47,.81,.42,.61)
> mean.and.sd(hetero)
     mean        sd
0.5028571 0.1749966
```

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. The function body has been reduced to a single line, so that the curly braces are not needed.

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

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

```
0.05013396 0.68567644
> mean.and.sd()
     mean         sd
0.1067355 1.1383779
```

We get a different set of random numbers, and hence a different mean and SD, each time that the function is run with its default argument.

Information on Functions

A useful function is `args()`. It lists function arguments, together with default values, if any.

4.6 Tables and Cross-Tabulation

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

table()

The function `table()` makes a table of counts. Specify one vector of values (often a factor) for each table margin that is required. A simple example is:

```
library(DAAG)
> table(possum$Pop, possum$sex)

        f  m
  Vic   24 22
  other 19 39
```

Here is a use of `table()` to count the number of `NA`s in the column `bp` (blood pressure) from the data frame `Pima.tr2` that is in the *MASS* package.

```
> library(MASS)
> table(is.na(Pima.tr2$bp))

FALSE  TRUE
  287    13
```

By default, `table()` ignores NAs. Hence the need for a check of the following type, here using `sapply()` function. This can be used with a data frame as its first argument to apply the function specified as its second argument in parallel to all columns of the data frame, to give the number of NAs in each column of the data frame `Pima.tr2`

```
> sapply(Pima.tr2, function(x)sum(is.na(x)))
npreg   glu    bp  skin   bmi   ped   age  type
    0     0    13    98     3     0     0     0
```

The action needed to get NAs tabulated under a separate NA category depends, annoyingly, on whether or not the vector is a factor. If the vector is not a factor, specify `exclude=NULL`. If the vector is a factor then it is necessary to generate a new factor that includes `NA` as a level. Specify

```
x <- factor(x, exclude=NULL)
```

Creating Groups

The following uses the `cut()` function to group data in the column `bp` in the data frame `Pima.tr2`, into four categories, then tabulating the numbers in the four categories:

```
> catBP <- cut(Pima.tr2$bp, breaks=4)
> table(catBP)
catBP
(37.9,57]   (57,76]   (76,95]  (95,114]
       22       170        87         8
> sum(table(catBP))
[1] 287
```

Notice that the 13 missing values, out of the total of 300, are not included in the table. This can be a trap. They are in catBP, as we can easily verify:

```
> table(is.na(catBP))

FALSE  TRUE
  287    13
```

Thus, above, we can do the following

```
> catBP <- factor(catBP, exclude=NULL)
> table(catBP)
catBP
(37.9,57]   (57,76]   (76,95]  (95,114]      <NA>
       22       170        87         8        13
> sum(table(catBP))
```

4.7 Option Settings

Setting the number of significant digits in output

Often, the printed result of calculations will, unless the default is changed (as has sometimes been done for the output in this document) show more decimal places of output than are useful. The `options()` function can be used to change to make a global (until further notice) change to the number of significant digits that are printed. For example:

```
> sqrt(10)
[1] 3.162278
> options(digits=2) # Change until further notice,
> # or until end of session.
> sqrt(10)
[1] 3.2
```

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

Rounding will sometimes introduce small inconsistencies. For example, with results rounded to two decimal places

```
> round(sqrt(372/12), 2)
[1] 5.57
> round(sqrt(2) * sqrt(372/12), 2)
[1] 7.87
> round(sqrt(2) * 5.57, 2)
[1] 7.88
```

Other option settings

Type in `help(options)` to get further details. I prefer to use the setting `options(show.signif.stars=FALSE)`. (The default is `TRUE`.)

4.8 Common sources of difficulty

In the use of `read.table()` for the entry of data that has a rectangular layout, it is important to tune the parameter settings to the input data set. See the discussion of `read.table()` in Subsection 9.1 for comments on common issues.

The function `count.fields()` can be a useful way to determine how many fields `read.table()` thinks it has found in each record. Alternatively, use `read.table()` with the parameter setting `fill=TRUE`, and carefully check the input data frame. Blank fields will be implicitly added, as needed, in order to ensure that all records have an equal number of fields.

Character vectors that are included as columns in data frames become, by default, factors. There are implications for the use of `read.table()`.

Factors can often be treated as vectors of text strings, with values given by the factor levels. There are several, potentially annoying, exceptions. If a factor is used to provide labels when using the function `text()`, use `as.character()` to turn the factor into a text string whose elements are the factor levels.

The handling of missing values is a common source of difficulty. Refer back to Subsection 4.4.7.

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. The structure is different, depending on whether a column or a row is extracted. Specify `unlist(elasticband[2, ])` to obtain the vector of numeric values in the second row of the data frame. For another instance (use of `sapply()`) where the difference between a numeric data frame and a numeric matrix is important, see Subsection 4.4.3.

It is inadvisable to assign new values to a data frame, thus creating a new local data frame with the same name, while it is attached. Use of the name of the data frame accesses the new local copy, while the column names that are in the search path are for the original data frame. There is obvious potential for confusion and erroneous calculations. The new local copy replaces the original when the data frame is detached.

Data objects that individually or in combination occupy a large part of the available computer memory can slow down all memory intensive computations. See further Subsection 9.4 for comment on associated workspace management issues. Note that most of the data objects that are used for our examples are small and thus will not, except where memory is very small, make much individual contribution to demands on memory.

4.9 Summary

Important R data structures are vectors, factors, data frames and lists. Vectors may be of mode numeric, logical, character or complex.

Factors, used for categorical data, are fundamental to the use of many of the R modeling functions. Ordered factors are appropriate for use with ordered categorical data.

The function `c()` (concatenate) joins vector elements together into vectors. It may be used for logical and character vectors, as well as for numeric vectors.
[It is in fact more general than this. It may be used to join lists, which are non-atomic vectors]

Missing Values, Infinite Values and `NaN`s can require special care.

Data frames group columns that all have the same length, together as a single R object. A data frame may have columns that are any mix of logical, numeric, character, factor or complex.

For simple forms of scatterplot, note the `plot()` function. There is a wide range of plotting abilities, beyond those offered by `plot()`.

Matrices, and data frames whose elements are all of one type (typically all numeric) are in some contexts handled similarly. In other contexts, they are handled quite differently. A matrix is a vector (with matrix elements stacked column upon column) that has a dimension attribute.

`read.table()` is the function of first recourse for inputting rectangular files.

Where access is required to columns of a data frame for one or two lines of code only, it is often convenient to use the function `with()`.

Attachment of a data frame (use the function `attach()`) can be where there are a number of lines of code that require access to its columns. Give the name of the data frame, i.e., no quotes.

Note also the use of `attach()` to give access to objects in an image (**.RData**) file. Include the name of the file in quotes.

The search path determines the order of search for objects that are accessed from the command line, or that are not found in the environment of a function that accesses them.

The R system has a wide range of generic functions, including `print()`, `plot()` and `summary()`, For such functions, the result depends on the class of object that is given as argument.

Modeling functions typically output a list, known as a *model object*, that holds key output information from the analysis.

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

Various options settings control such matters as the number of significant digits that will be displayed in output.

4.10 Exercises

Table 2: Estimates of amino acid replacements per 100 million years, for the three genes GPDH, SOD, and XDH. The column Ave is a weighted average of these three, with weights proportional to sequence length. The final column (Myr) gives time since divergence in millions of years.

	Gpdh	Sod	Xdh	AvRate	Myr
Drosophila subgroups	1.50	25.70	30.40	22.40	55.00
Drosophila subgenera	2.00	30.70	29.20	22.30	60.00
Drosophial genera	4.40	34.90	31.70	24.90	65.00
dipteran families	9.25	33.70	25.30	22.00	120.00
mammalian orders	11.60	46.00	17.10	18.70	70.00
animal phyla	13.20	19.20	19.20	17.50	600.00
fungi	40.00	24.90	13.70	21.40	300.00
kingdoms	13.00	12.60	11.50	11.90	1100.00

1. Read the data that is stored in the file **molclock.txt** (shown in Table 2), into the data frame `molclock`. Modify the arguments to `read.table()` so that:
   - The file **molclock1.txt** is input correctly

- The file **molclock2.txt** is input correctly.

2. For the data in Table 2, plot graphs of `Sod` against `Myr`, and of `Xdh` against `Myr`. Use `abbreviate()` to create abbreviated versions of the row names, and use these to label the points.
[Hint: Use `readLines("molclock.txt", n=-1)` to inspect, from the R command line, the contents of **molclock.txt**.]

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

4. Find an R function that will sort a vector. Give an example of its use.

5. Type `library()` and check that the *DAAG* package is installed. Attach the *DAAG* package. Type `library(DAAG)` 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. Examine the diagnostic plot and check whether there is anything that calls for special attention.

6. Find out what the function `substring()` does.

7. Find two ways to split a text string into a vector that holds its individual characters, using `substring()`, and using `strsplit()`. Use the function on the text strings

```
gi14786865 <- "LNLFFAGTETVSTTLRYGFLLLMKHPEVEAKVHEEI"

o4cka3 <- "LNIMVAGRDTTAGLLSFAMFELARNPKIWNKLREEV"
```

5 Base Graphics

Base graphics implements a relatively "traditional" style of graphics

Functions	`plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`, `identify()` etc. form a suite that plots points, lines and text.
Plot `y` vs `x`	Either: `with(women, plot(height, weight))` (older syntax)
	Or: `plot(weight ∼ height, data=women)` (uses graphics formula)
Caveat	Some base graphics functions do not take a `data` parameter

Note also (i) the lattice flavour of trellis graphics, implemented in the *lattice* package, and (ii) the low-level *grid* package on which *lattice* is built.
Lattice graphics will be the subject of the next chapter.

To see some of the possibilities that traditional (or base) R graphics offers, enter

```
demo(graphics)
```

Press the Enter key to move to each new graph.
For lattice graphics, enter

```
library(lattice)
demo(lattice)
```

Our discussion will start with an exposition of traditional graphics.

5.1 `plot()` and allied functions

The following both plot `y` against `x`:

```
plot(y ~ x)    # Use a formula to specify the graph
plot(x, y)     # Horizontal ordinate, then verhills20-0tical
```

Obviously `x` and `y` must be the same length.
Try

```
plot((0:20)*pi/10, sin((0:20)*pi/10))
plot((1:30)*0.92, sin((1:30)*0.92))
```

Is it obvious that these points lie on a sine curve? (To make this obvious, place the cursor over the lower border of the graph sheet, until it becomes a double-sided arror. Drag the border in towards the top border, making the graph sheet short and wide.)
Here are two further examples.

```
library(DAAG)
attach(elasticband)  # R can now find distance & stretch
plot(distance ~ stretch)
plot(ACT ~ year, data=austpop, type="l")
plot(ACT ~ year, data=austpop, type="b")
detach(elasticband)
```

The `points()` function adds points to a plot. The `lines()` function adds lines to a plot[3]. 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

[3]Actually these functions differ only in the default setting for the parameter `type`. The default setting for `points()` is `type = "p"`, and for `lines()` is `type = "l"`. Explicitly setting `type = "p"` causes either function to plot points, `type = "l"` gives lines.

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

Newer plot methods

Above, I described the default plot method. The plot function is a generic function that has special methods for "plotting" various different classes of object. For example, as we saw in chapter 2, plotting an `lm` object (created by the use of the `lm()` modelling function) gives diagnostic and other information that can help in the interpretation of regression results.

Recall also

```
plot(travelbooks[,1:5])
  # Has the same effect as pairs(travelbooks[, 1:5])
```

Note also the function splom() from the *lattice* package. For this, specify

```
library(lattice)
splom(~ travelbooks[, 1:5])    # Lattice alternative to pairs plot
```

5.2 Fine control – Parameter settings

For `plot()`, `points()` and `text()`, the parameter `cex` ("character expansion") controls the size, while `col` ("color") controls the colour. The parameter `pch` controls the choice of plotting symbol.

Figure 2 shows some of the possibilities:

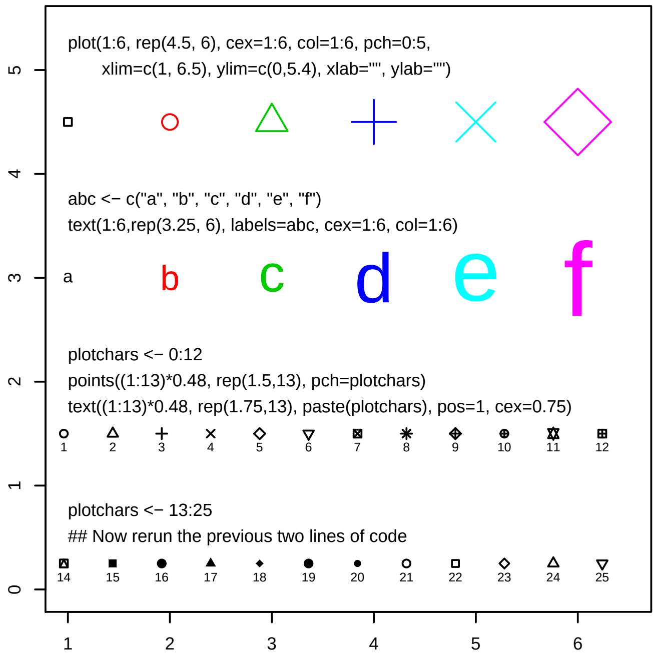


Figure 2: Different plot symbols, colours and sizes.

The default settings of parameters, such as character size, are often adequate. When it is necessary to change parameter settings, alternatives may be to supply the values required as parameters to the plotting function, or to use `par()` to make the change, prior to calling the plotting function. For some parameters, the first method must be used, while for others the second method is necessary. In some case, the reasons for this are obvious. One that is not obvious is `pty="s"` (specify using `par()`), which gives a square plotting region.

On the first use of `par()` to make changes to the current device, it is often useful to store existing settings, so that they can be restored later. For this, specify, e.g.:

```
oldpar <- par(cex=1.25, col="red")
```

This stores the existing settings in `oldpar`, then changes parameters (here `cex` and `col`) as requested. To restore the original parameter settings at some later time, enter `par(oldpar)`. See below ("Multiple plots on the one page") for an example

Type `help(par)` to get details of all the parameter settings that are available.

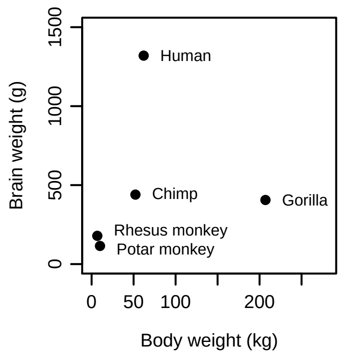
Adding Text in the Margin

`mtext(side, line, text,..)` adds text in the margin of the current plot. The sides are numbered 1(x-axis), 2(y-axis), 3(top) and 4.

5.3 Adding points, lines and text – examples

Here is a simple example that uses the function `text()` to label the points on a plot.

Figure 3 has the plot:



```
> ## Data used in plot
> primates    # DAAG package
               Bodywt Brainwt
Potar monkey     10.0     115
Gorilla         207.0     406
Human            62.0    1320
Rhesus monkey     6.8     179
Chimp            52.2     440
```

Figure 3: Plot of brain weight against body weight, for selected primates.

Code that gives the above plot is:

```
attach(primates)
plot(Bodywt, Brainwt, xlim=c(0, 250),
     xlab="Body weight (kg)", ylab="Brain weight (g)")
  # Specify xlim so that there is room for the labels
text(x=Bodywt, y=Brainwt, labels=rownames(primates), pos=4)
  # Alternatives are pos=1 (below), 2 (left), 3 (above)
detach(primates)
```

Example – Labels that locate possum study sites

Where are the possums? The *oz* package plots an outline of the Australian coast and state boundaries. As it uses standard plot functions, we can use `text()` to add information about specific locations.

```
> possumsites         # DAAG package
            latitude longitude altitude
Cambarville    145.9    -37.55      800
```

```
Bellbird        148.8    -37.62      300
Allyn River     151.5    -32.12      300
Whian Whian     153.3    -28.62      400
Byrangery       153.4    -28.62      200
Conondale       152.6    -26.43      400
Bulburin        151.5    -24.55      600
```

A plot that shows the sites, on a map of the Eastern coast of Australia, may be obtained thus:

```
attach(possumsites)
## Ensure that the oz package is installed
library(oz)
oz()
text(longitude ~ latitude, labels=rownames(possumsites), adj=1, col="red")
```

Figure 4 improves somewhat on this simple code:

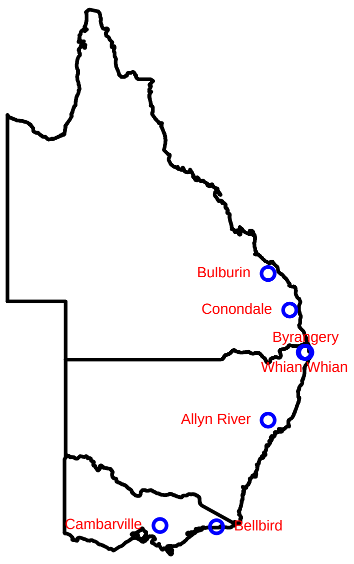


Figure 4: Sites at which possums were collected.

```
## Code used for plot:
oz(sections=c(3:5, 11:16), col="gray", xlim=c(130, 165))
chh <- par()$cxy[2]
points(latitude, longitude+c(0,0,0,.2,-.2, 0,0)*chh, col="blue")
text(longitude ~ latitude, labels=rownames(possumsites),
     pos=c(2,4,2,1,3,2,2), col="red", xpd=TRUE)
  # Settings for pos are 1: below, 2: left, 3:above and 4: right
  # With xpd=TRUE plotting is allowed outside the figure region
```

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. For a column by column layout, use `mfcol` instead. The following presents four different transformations of the primates data, in a two by two layout:

```
oldpar <- par(mfrow=c(2,2), pch=16)
library(MASS)
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 to 1 figure per page, and pch=1
```

Color palettes

A variety of color palettes are available. Here is a function that displays some of the possibilities:

```
view.colours <- function(){
    plot(1, 1, xlim=c(0,14), ylim=c(0,3), type="n", axes=F, xlab="",ylab="")
    text(1:6, rep(2.5,6), paste(1:6), col=palette()[1:6], cex=2.5)
    text(10, 2.5, "Default palette", adj=0)
    rainchars <- c("R","O","Y","G","B","I","V")
    text(1:7, rep(1.5,7), rainchars, col=rainbow(7), cex=2.5)
    text(10, 1.5, "rainbow(7)", adj=0)
    cmtxt <- substring("cm.colors", 1:9,1:9)
    # Split "cm.colors" into its 9 characters
    text(1:9, rep(0.5,9), cmtxt, col=cm.colors(9), cex=3)
    text(10, 0.5, "cm.colors(9)", adj=0)
}
```

To run the function, enter

```
view.colours()
```

The shape of the graph sheet

There is provision to specify the size and shape of the graph page, e.g. so that the individual plots are rectangular rather than square. 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 of these parameters that matter for screen display or for incorporation into Word and similar programs. Graphs can be enlarged or shrunk by pointing at one corner, holding down the left mouse button, and pulling.

5.4 Identification and Location on the Figure Region

Two functions are available for this purpose. Draw the graph first, then call one or other of these functions.

o `identify()` labels points. Position the cursor near the point that is to be identified, and click the left mouse button.

o `locator()` prints out the co-ordinates of points. One positions the cursor at the location for which coordinates are required, and clicks the left mouse button.

A click with the right mouse button signifies that the identification or location task is complete, unless the setting of the parameter `n` is reached first. For `identify()` the default setting of `n` is the number of data points, while for `locator()` the default setting is `n` = 500.

An example of the use of `identify()` was given in Subsection 2.4. The use of `identify()` is even simpler.

```
with(travelbooks, plot(weight ~ volume)
locator()
```

Now, left click at the locations whose coordinates are required, terminating as for `identify()`. Alternatively or additionally, specify the number of points to be located as an argument to the function.

5.5 Plots that show the distribution of data values

We discuss histograms, density plots, boxplots and normal probability plots.

Histograms and density plots

The shapes of histograms depend on the placement of the breaks, as Figure 5 illustrates:

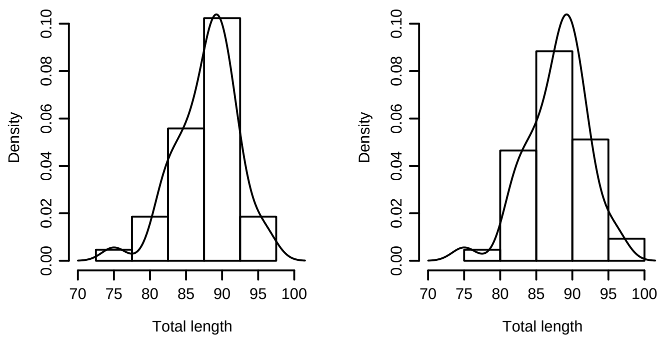


Figure 5: The two graphs show the same data, but with a different choice of breakpoints.

Here is the code used to plot the histograms:

```
par(mfrow = c(1, 2), pty="s")    # pty="s"; square plots
attach(possum)
here <- sex == "f"
hist(totlngth[here], breaks = 72.5 + (0:5) * 5, ylim = c(0, 22),
     xlab="Total length", main ="A: Breaks at 72.5, 77.5,...")
hist(totlngth[here], breaks = 75 + (0:5) * 5, ylim = c(0, 22),
     xlab="Total length", main="B: Breaks at 75, 80,...")
par(mfrow=c(1,1))
detach(possum)
```

Below, I argue against placing much reliance on histograms. Some may prefer them on the grounds that they are derived very directly and simply from the actual data. In that case, why not show the actual distribution of data values. The following uses the function `rug()` to show the distribution of the data values along the horizontal axis:

```
with(subset(possum, sex=="f"), hist(totlngth))
with(subset(possum, sex=="f"), rug(totlngth))
```

One of the following can sometimes be insightful:

```
with(subset(possum, sex=="f"), dotchart(totlngth))
with(subset(possum, sex=="f"), stripchart(totlngth))
```

Density Plots

The density at each point is an estimate of the number of points per unit interval. A histogram is a crude form of density estimate, one in which the density estimate changes discretely at class boundaries. Density plots are in general preferable to histograms, because they give a density estimate that changes smoothly. Density plots do not depend on a choice of breakpoints. They do require the choice of a bandwidth parameter and a choice between different types of window; often the default choices are acceptable.

The following gives a density plot:

```
with(possum, plot(density(totlngth[here]),type="l"))
```

Note that in the overlaid density plot in 5 the y-axis for the histogram is labelled so that the area of a rectangle is the frequency for that rectangle. To get the plot on the left, specify:

```
attach(possum)
here <- sex == "f"
dens <- density(totlngth[here])
xlim <- range(dens$x)
ylim <- range(dens$y)
hist(totlngth[here], breaks = 72.5 + (0:5) * 5,
     probability = T, xlim = xlim, ylim = ylim,
     xlab="Total length", main="")
lines(dens)
detach(possum)
```

With data that have sharp lower and/or upper cutoff limits, it may be necessary to specify the limit or limits. For example, a failure time distribution may have a mode close to zero, with a sharp cutoff at zero. Use the parameters `from` and/or `to` for this purpose. This issue most commonly arises with a lower cutoff at 0.

Boxplots

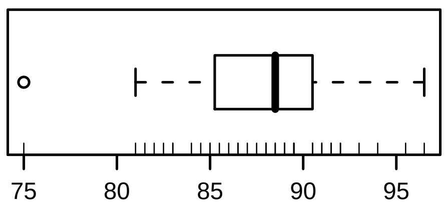


Figure 6: Distribution of lengths of female possums. The bars (rugs) show actual data values.

Boxplots use a small number of characteristics of a distribution to characterize it. Look up `help(boxplot)` for details. Here is code that gives a boxplot of the above possum data:

```
with(subset(possum, sex=="f"), boxplot(totlngth, horizontal=TRUE))
```

It can be insightful to add a “rug” that shows the individual values, by default along the horizontal axis (`side=1`). To add a rug to the above plot, type

```
with(subset(possum, sex=="f"), rug(totlngth))
```

Figure 6 shows the result.

Side by side boxplots

Boxplots allow convenient side-by-side comparisons of different groups, as in the cuckoo egg data that we now present (from Latter1902; Tippett 1931 presents them in a summarized form.) Cuckoos lay eggs in the nests of other birds. The eggs are then unwittingly adopted and hatched by the host birds. Figure7 shows side by side boxplots of these data.

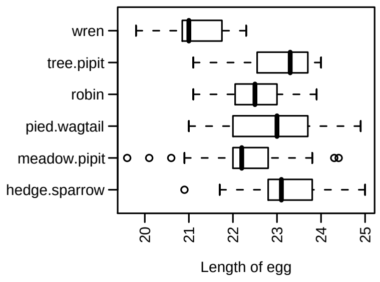


Figure 7: Side by side boxplots of egg length data.

```
## Code used for plot:
boxplot(length ~ species, data=cuckoos,
        xlab="Length of egg", horizontal=TRUE,
        las=2)    # las=2 => Plot labels perpendicular to axis
```

Normal probability plots

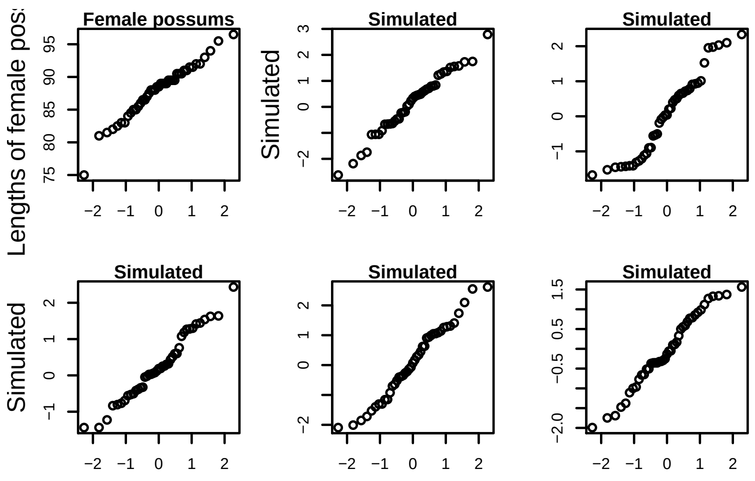


Figure 8: Normal probability plots. For data from a normal distribution points should fall, approximately, along a line. The top left panel shows the 43 lengths of female possums. Other panels are for independent normal random samples of size 43.

The function `qqnorm(y)` gives a normal probability plot of the elements of `y`. The points of this plot will lie approximately on a straight line if the distribution is Normal. In order to calibrate the eye to recognise plots that indicate non-normal variation, it is helpful to do several normal probability plots for random samples of the relevant size from a normal distribution, obtained using the function `rnorm()`.

```
x11(width=8, height=6) # This is a better shape for this plot
attach(possum)
here <- sex == "f"
par(mfrow=c(3,4))    # 3 by 4 layout of plots
```

```
y <- totlngth[here]
qqnorm(y,xlab="", ylab="Length", main="Possums")
for(i in 1:11)qqnorm(rnorm(43), xlab="", ylab="Simulated lengths",
                     main="Simulated")
par(mfrow=c(1,1))
# By default, rnorm() generates random samples from a normal
# distribution with mean 0 and standard deviation equal to 1.
detach(possum)
```

Figure 8 shows the plots. 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.

The idea is an important one. In order to judge whether data are normally distributed, examine a number of randomly generated samples of the same size from a normal distribution. It is a way to train the eye.

5.6 Scatterplot Smoothing

Figure 9 shows miles per gallon (`mpg`), plotted against piston displacement (`disp`), for 32 cars whose detailed characteristics were described in the US 1974 magazine *Motor Trends*.

It can be useful to make a comparison with a curve provided by a data smoothing routine that is not restricted to using a particular mathematical form of curve.
[R has both a `lowess()` function and the more general `loess()` function. The `lowess()` function does smoothing in one dimension only, while `loess()` will handle multi-dimensional smoothing. For technical details of `lowess()` and `loess()`, see Cleveland (1981), and references given in that paper.]

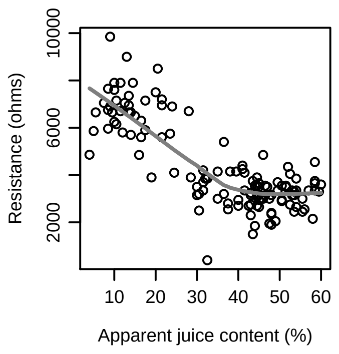


Figure 9: Plot showing change in fuel usage (miles per gallon) with displacement (`disp`). Code is:

```
attach(mtcars)        # From the datasets package
plot(mpg ~ disp,
     xlab=expression("Displacement ("*"in"^3*")"),
     ylab="Miles per gallon")
lines(lowess(mpg ~ disp, f=0.5))
detach(mtcars)
```

A smooth trend curve that has been superimposed on a scatterplot can be a useful aid to interpretation. When the data appear to scatter about a simple mathematical curve, the curve-fitting methods that are discussed in other sections can be used to obtain a ‘best-fit’ or regression line or curve to pass through the points.

The female athletes (AIS) data

Here we use data on the heights of 100 female athletes[4]. First load the data frame `ais`, if necessary from `http://www.maths.anu.edu.au/~johnm/r/workshop`.

The function `panel.smooth()` plots points, then adds a smooth curve. For example:

```
attach(ais)
here<- sex=="f"
plot(pcBfat[here]~ht[here], xlab = "Height", ylab = "% Body fat")
panel.smooth(ht[here],pcBfat[here])
abline(lm(pcBfat[here] ~ ht[here]))
detach(ais)
```

The least squares regression line (`abline()`) has been added for comparison.

5.7 Plotting Mathematical Symbols

Both `text()` and `mtext()` will take an expression rather than a text string. In `plot()`, either or both of `xlab` and `ylab` can be an expression. Figure 10 is an example.

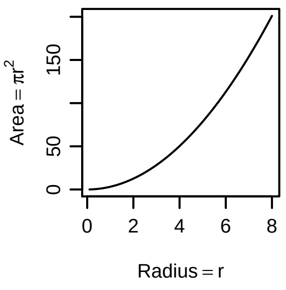


Figure 10: The x-axis and y-axis labels are both mathematical expressions.

```
## Code used for plot:
r <- seq(0.1, 8.0, by=0.1)
plot(r, pi * r^2, xlab=expression(Radius == r),
     ylab=expression(Area == pi*r^2), type="l")
 # NB: Use ==, within an expression, to print =
```

See Subsection 8.5 (Figure 15) for another and much more complicated example of the plotting of mathematical expressions.

See `help(plotmath)` for detailed information. The final plot from

```
demo(graphics)
```

shows some of the possibilities for plotting mathematical symbols.

5.8 Multi-way Tables – Mosaic Plots

Here is a more interesting example, using the multi-way table `UCBAdmissions` that is in the *base* package. This holds admission data, from University of California at Berkeley in 1973. First, we will get information about the data:

[4]Data relate to the paper: Telford, R.D. and Cunningham, R.B. 1991: Sex, sport and body-size dependency of hematology in highly trained athletes. *Medicine and Science in Sports and Exercise* 23: 788-794.

```
> class(UCBAdmissions)
[1] "table"
> dim(UCBAdmissions)
[1] 2 2 6
> dimnames(UCBAdmissions)
$Admit
[1] "Admitted" "Rejected"
$Gender
[1] "Male" "Female"

$Dept
[1] "A" "B" "C" "D" "E" "F"
```

Finally while we explore around this table, here is the table for department A:

```
> UCBAdmissions[, , "A"]   # Equivalent to UCBAdmissions[, , 1]
 Gender
Admit Male Female
Admitted 512 89
Rejected 313 19
```

To get a mosaic plot, type

```
mosaicplot(UCBAdmissions, color=TRUE)
```

For a direct comparison between male and female admission rates, it is helpful to permute the dimensions:

```
mosaicplot(aperm(UCBAdmissions, c(2,1,3)), color=TRUE)
```

Note also the functionmosaic(), from the *vcd* package.

5.9 Regular Graphics Functions – Additional Points

The variety of R graphics functions

For other plotting possibilities, look under the help for the functions `hist()`, `coplot()`, `contour()`, `filled.contour()`, `image()`, etc. Use of `example()` with these functions gives some impressive plots.

Functions that handle smoothing and density estimation can be useful adjuncts to the graphics functions. These include `lowess()` and `density()` and `scatter.smooth()`, and `bkde2D()` in the *KernSmooth* package.

Additionally, there are several specialist graphics packages. Assuming that it is installed, examine the help information for the package *scatterplot3d.*

Plots with Large Numbers of Points

When the number of points is large, such standard forms of presentation as scatterplots can be problematic. Graphs may appear as a dense uninformative mass of black ink. The graphics files may be so large that they take an inordinate time to print. There are ways to address these issues, but they require the use of a different form of graphic.

Histograms, density curves and boxplots present highly summarized information. The density of points on the graph is small, so that there should not be a problem. Normal probability plots can be problematic, because they try to present each point individually. The central part of the plot, e.g., in the region where points overlap, might, in most instances be replaced by a curve through the data. There need not be any loss of visual information.

For two-dimensional data, one approach is to use a two-dimensional analogue of a density plot. The function `bkde2D()` in the *KernSmooth* package may be used. Here is an example of its use. We take the range of each of the variables to be about a tenth of the range of the data:

```
library(KernSmooth)
attach(mouse.lratio)
M13 <- bkde2D(na.omit(M[,c(1,3)]), bandwidth=c(1,1))
attach(M13)
filled.contour(x1,x2,fhat, xlab="Slide 1", ylab="Slide 3")
detach(M13)
detach(mouse.lratio)
```

Inclusion of Graphs in Other Documents

Both on PC and Mac systems, writing to a pdf file that is then included in a latex document, or directly inserted into a pdf file, gives a high quality result.

Graphs do not, for some reason, import well from the clipboard into Word on the Macintosh under OS X. I have used `png()` to write to a file that is then inserted as a picture. I have left `width` and `height` at their defaults, except when the shape of the graph sheet needs to be changed. I have mostly used `pointsize=18` or `pointsize=16`. Within Word, graphs are then shrunk to the requisite size.

5.10 Exercises

1. The data set `LakeHuron` (*datasets* package) has mean July average water surface elevations, in feet, IGLD (1955) for Harbor Beach, Michigan, on Lake Huron, Station 5014, for 1875-1972. Use the following to create a data frame that has the same information:

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

   a) Plot `mean.height` against year.

   b) Use the identify function to determine which years correspond to the lowest and highest mean levels. That is, type

```
identify(huron$year, huron$mean.height, labels=huron$year)
```

   and use the left mouse button to click on the lowest point and highest point on the plot. To quit, press both mouse buttons simultaneously.

   c) As in the case of many time series, the mean levels are correlated from year to year. To see how each year's mean level is related to the previous year's mean level, use

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

   This plots the mean level at year i against the mean level at year i-1.

   d) Now explain why the following code achieves the same effect:

```
plot(LakeHuron)
identify(LakeHuron, labels=time(LakeHuron))
```

2. Check the distributions of head lengths (`hdlngth`) in the `possum` data set. Compare the following 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))`.

What are the advantages and disadvantages of these different forms of display?

3. 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.

4. The function `runif()` can be used to generate a sample from a uniform distribution, by default on the interval 0 to 1. Try `x <- runif(10)`, and print out the numbers you get. Then repeat exercise 6 above, but taking samples from a uniform distribution rather than from a normal distribution. What shape do the points follow?

5. Repeat exercise 4, but for other distributions. For example `x <- rchisq(10,1)` will generate 10 random values from a chi-squared distribution with degrees of freedom 1. The statement `x <- rt(10,1)` will generate 10 random values from a t distribution with degrees of freedom one. Make normal probability plots for samples of various sizes from these distributions.

6. For the first two columns of the data frame `hills`, examine the distribution using:

   (a) histograms

   (b) density plots

   (c) normal probability plots.

   Repeat (a), (b) and (c), now working with the logarithms of the data values.

7. The following data gives milk volume (g/day) for smoking and nonsmoking mothers[5]:

   Smoking Mothers: 621, 793, 593, 545, 753, 655, 895, 767, 714, 598, 693

   Nonsmoking Mothers: 947, 945, 1086, 1202, 973, 981, 930, 745, 903, 899, 961

   Present the data (i) in side by side boxplots; (ii) using a dotplot form of display.

8. The 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.

[5]Data are from the paper "Smoking During Pregnancy and Lactation and Its Effects on Breast Milk Volume" (Amer. J. of Clinical Nutrition).

6 Lattice Graphics

Lattice	Lattice is a flavour of trellis graphics (cf the earlier S-PLUS flavour)
Grid	*grid* is the low-level graphics system on which *lattice* is built.
	Part II of Paul Murrell's *R Graphics* has an accessible introduction to *grid*.
Lattice vs base	Lattice graphics is more structured, automated and stylized.
Lattice syntax	Lattice type graphics formulae are mandatory for lattice plots, e.g.,
	`xyplot(csoa ~ it | sex * agegp, groups = target, data = tinting)`
	`csoa ~ it`: Plot `csoa` vs `it`
	`| sex * agegp`: Condition on `sex * agegp`,
	(a different panel for each combination of `sex` and `agegp`)
	`groups`: Within each panel, group by levels (`locon`, `hicon`) of `target`
`auto.key`	Use `auto.key` for a basic key to the group labeling (`groups` parameter).

Lattice (trellis) graphics allow the use of the layout on the page to reflect meaningful aspects of data structure. They offer abilities similar to those in the S-PLUS *trellis* package.

6.1 Panels of Scatterplots – Examples of the Use of xyplot()

The basic function for drawing panels of scatterplots is `xyplot()`. We will use the data frame `tinting` (supplied) to demonstrate its use. These data are from an experiment that investigated the effects of tinting of car windows on visual performance[6]. The authors were mainly interested in visual recognition tasks that would be performed when looking through side windows.

In this data frame, `csoa` (critical stimulus onset asynchrony, i.e. the time in milliseconds required to recognise an alphanumeric target), `it` (inspection time, i. e. the time required for a simple discrimination task) and `age` are variables, `tint` (level of tinting: no, lo, hi) and `target` (contrast: locon, hicon) are ordered factors, `sex` (1 = male, 2 = female) and `agegp` (1 = younger, in the early 20s; 2 = an older participant, in the early 70s) are factors. Attaching the *DAAG* package makes these data available:

```
library(DAAG)
```

A simple graph, that does not distinguish the two different targets, can be obtained with:

```
xyplot(csoa~it|sex*agegp, data=tinting)
```

Figure 11 shows a graph that makes more extensive use of the function's abilities, using different symbols (and, if available, different colors) for different targets. It uses the parameter setting `auto.key=list(columns=2)` to obtain a simple key. (Setting `columns=2` places the two key items in separate columns, i.e., side by side rather than in a single column.) The code is:

```
xyplot(csoa~it|sex*agegp, data=tinting,
       auto.key=list(columns=2), groups=target)
```

Setting `groups=target` automatically invokes the use of `panel.superpose`. If the device supports color, different colors are by default used for the different groups.

A striking feature is that the very high values, for both `csoa` and `it`, occur only for elderly males. It is apparent that the long response times for some of the elderly males occur, as we might have expected, with the low contrast target. The following puts smooth curves through the data, separately for the two target types:

```
xyplot(csoa~it|sex*agegp, data=tinting,
       groups=target, auto.key=list(columns=3),
       type=c("p","smooth"))
```

[6]Data relate to the paper: Burns, N. R., Nettlebeck, T., White, M. and Willson, J. 1999. Effects of car window tinting on visual performance: a comparison of elderly and young drivers. Ergonomics 42: 428-443.

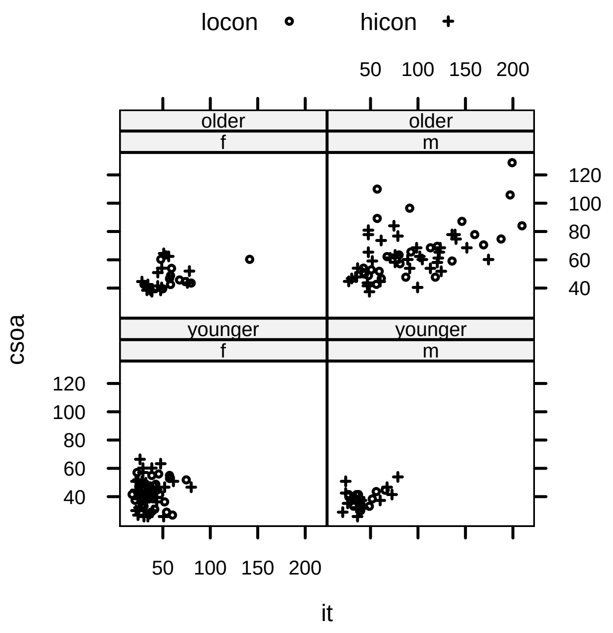


Figure 11: csoa versus it, for each combination of females/males and older/younger. The two targets (o = low, + = high contrast) are shown with different symbols.

The relationship between `csoa` and `it` seems much the same for both levels of contrast.

The following plot uses different symbols (in black and white) or different colours for different levels of tinting. The longest times are for the high level of tinting.

```
xyplot(csoa ~ it|sex*agegp, data=tinting, groups=tint,
       auto.key=list(columns=3))
```

6.2 Annotation – the `auto.key`, `key` and `legend` arguments

For a key is required that identifies the colours, plotting symbols and names for the groups, the easiest mechanism is to use the setting `auto.key=TRUE`. For greater flexibility, `auto.key` can be a list. Settings that are often useful are:

- `x` and `y`, where these are coordinates with respect to the whole display area, together with `corner`, which is one of `c(0,0)` (bottom left corner of legend), `c(1,0)`, `c(1,1)` and `c(0,1)`.
- `space`: one of `"above"`, `"below"`, `"left"`, `"right"`.
- `points`, `lines`: in each case set to `TRUE` or `FALSE`. `columns`: number of columns of keys.

Colours, plotting symbols, line type are then taken from the trellis settings for the device used. Unless `text` is supplied as a parameter, `levels(groups)` provides the legends.

Alternatives are to supply a list of arguments to the parameter `key`, or to use the argument `key=simpleKey()`. If `key` is supplied as a list, at least one of `lines`, `points` and `text` must be supplied. Arguments to `simpleKey()` are the names of list elements when `auto.key` is used. When the parameter `key` is supplied, arguments that are not otherwise specified use the trellis settings that were in place when the trellis object was created.

See `help(xyplot)` for further details.

6.3 Trellis settings

To ensure that changes from defaults affect both the graph and the legend, they can be made by using the function `trellis.par.set()`, which changes the default settings for the current device.

Alternatively, the settings can be specified in the `par.settings` argument to the trellis function. To find what settings might be changed, type:

```
> names(trellis.par.get())
 [1] "fontsize"         "background"       "clip"
 [4] "add.line"         "add.text"         "bar.fill"
 [7] "box.dot"          "box.rectangle"    "box.umbrella"
[10] "dot.line"         "dot.symbol"       "plot.line"
[13] "plot.symbol"      "reference.line"   "strip.background"
[16] "strip.shingle"    "superpose.line"   "regions"
[19] "shade.colors"     "superpose.symbol" "axis.line"
[22] "axis.text"        "box.3d"           "par.xlab.text"
[25] "par.ylab.text"    "par.zlab.text"    "par.main.text"
[28] "par.sub.text"
```

The settings that are of interest can then be inspected individually. When `groups=TRUE`, settings for the symbols are controlled by the `superpose.symbol` list item, and for lines by the `superpose.line` list item. Inspection of the settings for `superpose.symbol` gives:

```
> trellis.par.get("superpose.symbol")
$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8

$col
[1] "grey"      "steelblue" "black"

$font
[1] 1 1 1 1 1 1 1

$pch
[1] 16 16 16
```

For `superpose.line`, the list elements are `col`, `lty` and `lwd`. The `superpose.line` settings apply when a line is used to join the points (specify `type="l"`), or a smooth curve is passed through them (specify `type=c("p", ‘‘smooth’’)` to get points, with a smooth curve added).

6.4 An example

The code that now follows uses the parameter `par.settings` to make changes to these settings. It adds smooth curves, and includes information about the line type and color in the legend. Figure 12 shows the result:

```
trellis.device()
xyplot(csoa ~ it|sex*agegp, data=tinting,
       par.settings=list(superpose.symbol=list(col=c("gray","gray","black"),
                                               pch=c(1,16,16)),
                         superpose.line=list(col=c("grey", "gray", "black"),
                                             lty=c(1,2,4), lwd=c(2,2,1))),
       groups=tint, type=c("p","smooth"), span=0.9,
       auto.key=list(columns=3, lines=TRUE))
  # The parameter "span" controls the extent of smoothing.
```

The different levels of tinting (o=no, +=low, >=high) are shown with different symbols. Annotation is now included on the plot. Smooth curves are fitted, one for each level of tinting.

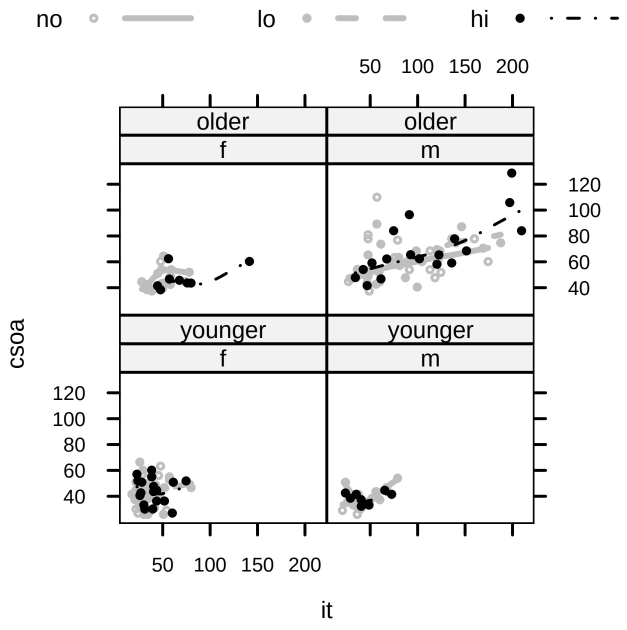


Figure 12: *csoa* versus *it*, for each combination of females/males and elderly/young. Different line and point styles are used for different levels of `tint`. The parameter `par.settings` has been used to make point and line settings.

6.5 Lattice Style Stripplots and Boxplots

The following stripplot (for the cuckoos data set is from *DAAG*) has a different 'strip' for each different host species:

```
## The two lines that follow ensure display of the species name
levnam <- strsplit(levels(cuckoos$species), "\\.")
levels(cuckoos$species) <- sapply(levnam, paste, collapse="")
stripplot(species ~ length, xlab="Length of egg", aspect=0.5,
          data=cuckoos)
```

Compare the above with a boxplot form of presentation

```
## NB. The lattice function is bwplot()
bwplot(species ~ length, xlab="Length of egg", data=cuckoos)
```

The `aspect` argument determines the ratio of distance in the y-direction to distance in the x-direction.

Inclusion of lattice graphics functions in user functions

The function `xyplot()` does not itself print the graph. Instead, it returns an object of class *trellis* which, if the statement is typed on the command line, is then "printed" by the function `print.trellis()`. Thus, typing

```
xyplot(csoa ~ it | sex * agegp, data=tinting)
```

on the command line is equivalent to

```
print(xyplot(csoa ~ it | sex * agegp, data=tinting))
```

In a function, unless the lattice command appears as the final statement of the function, the print statement must be explicit, i.e.

`print(xyplot(csoa ~ it | sex * agegp, data=tinting))`

or equivalently:

```
tinting.trellis <- xyplot(csoa ~ it | sex * agegp, data=tinting)
print(tinting.trellis)
```

6.6 Dotplots

Dotplots are typically used to show the variation in values of a response variable, at each of a number of levels of a factor. Try for example:

```
dotplot(variety ~ yield | site, data = barley, groups = year,
    key = simpleKey(levels(barley$year), space = "right"),
    xlab = "Barley Yield (bushels/acre) ",
    aspect = 0.5, layout = c(1, 6), ylab = NULL)
```

Try stretching the plot vertically so that the labels do not overlap.

Where there is just one point per factor level, the argument `type="h")` gives a line from the origin to the point. Both a line and a point may be given. This can be used to provide a graph that can be quite striking in its effect, as in the following example:

```
deathrate <- c(40.7, 36,27,30.5,27.6,83.5)
ord <- order(deathrate)
hosp <- c("Cliniques of Vienna (1834-63)\n(> 2000 cases pa)",
     "Enfans Trouves at Petersburg\n(1845-59, 1000-2000 cases pa)",
     "Pesth (500-1000 cases pa)",
     "Edinburgh (200-500 cases pa)",
     "Frankfort (100-200 cases pa)", "Lund (< 100 cases pa)")
hosp <- factor(hosp, levels=hosp[order(deathrate)])
dotplot(hosp ~ deathrate, xlim=c(0,110), cex=1.5,
        scale=list(cex=1.25), type=c("h","p"),
        xlab=list("Death rate per 1000 ", cex=1.5),
        sub="From Nightingale (1871) - data from Dr Le Fort")
```

Data are from Nightingale, F. (1871): *Notes on Lying-in Institutions.*

6.7 Lattice Style Density Plots

Here is a density plot, for data from the `possum` data set (*DAAG*), that compares `sex`es and `Vic/other` populations.

```
densityplot(~earconch | Pop, groups=sex, data=possum)
```

Different labeling for different sites would be helpful. For this, merge information from the data frame possum, with information from `possumsites` (*DAAG*), allowing the sites to be labeled by name.

```
possumsites$site <- 1:7
possumsites$Site <- factor(rownames(possumsites))
possums <- merge(possum, possumsites[, 4:5], by="site")
```

Here then is the code for the density plots:

```
densityplot(~earconch | sex, group=Site, data=possums, aspect=1.5)
```

We can include a key, thus:

```
densityplot(~earconch | sex, group=Site, data=possums, aspect=1.5,
            key=simpleKey(text=levels(possums$Site), points=TRUE))
```

Note that `histogram()` ignores the parameter `group`.

6.8 An incomplete list of lattice Functions

```
dotplot(factor ~ numeric,..)   # 1-dim. Display
stripplot(factor ~ numeric,..) # 1-dim. Display
barchart(character ~ numeric,..)
histogram( ~ numeric,..)
densityplot( ~ numeric,..)     # Density plot
bwplot(factor ~ numeric,..)    # Box and whisker plot
qqmath(factor ~ numeric,..)    # normal probability plots
splom( ~ dataframe,..)         # Scatterplot matrix
parallel( ~ dataframe,..)      # Parallel coordinate plots
cloud(numeric ~ numeric * numeric, ...)      # 3D surface
wireframe(numeric ~ numeric * numeric, ...)  # 3D scatterplot
```

In each instance, users can add conditioning variables.
To get a succession of examples of the use of `bwplot()`, type:

```
example(bwplot)
```

Similarly for `qqmath()`, `dotplot()`, etc. The examples are in each instance taken from the relevant help page.

6.9 Summary

The functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`, `identify()` etc. form a suite that plots points, lines and text.

Note the alternatives `plot(x, y)`, `plot(y ~ x)`

7 Fitting Statistical Models

Linear models	In R, any model that can be fitted using `lm()` is a "linear" model. Such models can handle many types of highly non-linear response.
Diagnostic plots	Use `plot()` with the model object as argument, to get a basic set of diagnostic plots. Where interactions are not involved, use `termplot()` to give a visual represenation of the contributions of the different terms. (Why are interactions a problem for `lm()`?
Extractor functions	Where available, these are the preferred way to extract information from model objects.
Factors	Factors are used, in model terms, to model qualitative effects.
Model matrices	What are they, how are they obtained, and why are they important?
The Error Term	Errors do not have to be (and often are not) independently and identically distributed. A simple hierarchical multi=level model will illustrate why attention to the error term can be important.

The various R packages provide, between them, a huge range of model fitting abilities. Perhaps the most widely used model fitting function is `lm()`, where the `lm` stands for linear model. The example of the use of `lm()` that appears below is designed to demonstrate the far reach of models that may be fitted using this function. It is an example that stretches the meaning of the word *linear*. The `lm()` function will fit any model for which the fitted values are a linear combination of basis functions. Each basis function can in principle be an arbitrary transformation of one or more explanatory variables. "Additive models" may be better terminology.

The `lm()` function assumes that the random term is i.i.d. (independently and identically distributed) normal. Various R packages, of which *nlme* is (or was, until *lme4* came along) perhaps the most important, include functions that allow a more general form of random term. In the example in Section 7.3 below, there are i.i.d. random ("error") terms at two levels, between units and between groups to which the units belong. (In this case, the groups were different sites where experiments were conducted.) This simple form of non-i.i.d. random structure has sufficient complication to show how important it can be to take account of random structure in the random term, and to illustrate the possibilities for misleading inference when structure in the random part of the model is ignored.

Among other models that handle non-i.i.d. random structure, note:

- Time series models, where the random term typically has a sequential correlation structure;
- Repeated measures models, which model the time profiles of multiple "individuals".
- Spatial models, where there is a correlation structure that depends on the separation of observations in space.

7.1 A Regression Model

The data that will be used are a subset of the `races2000` data set that is in the `DAAG` package. To make the data available, do the following:

```
library(DAAG)
names(races2000)      # Require dist, climb, time and timef
hills2k <- races2000[races2000$type=="hill", 7:10]
```

An initial step is to obtain scatterplot matrices that shows all of the pairwise scatterplots for the variables that are of interest:

```
plot(hills2k)         # Equivalent to pairs(hills2k)
plot(log(hills2k))    # Check what difference this makes
```

The diagonal panel in the same column as the graph gives the x-axis label. The diagonal panel in the same row gives the y-axis label. Such a graph 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[7] will underestimate the strength of any relationship; (3) Severely skewed distributions, so that the correlation is an unreliable measure of the strength of relationship.

The graph should be used only as an initial coarse screening device. Skewness in the individual distributions is better checked with the use of plots of density estimates.

The row names store the names of the hillraces. I have recently discovered that for the `Caerketton` race, where the time seems anomalously small, `dist` should probably be 1.5mi not 2.5mi. The safest option may be to omit this point. For later reference, note the row number:

```
> match("Caerketton", rownames(hills2k))
[1] 42
> hills2k[42, "dist"]
[1] 2.5
```

The interest is in prediction of `time` as a function of `dist` and `climb`. First examine the scatterplot matrices, for the untransformed variables, and for the log transformed variables. The pattern of relationship between the two explanatory variables – `dist` and `climb` – is much closer to linear for the log transformed data, i.e., the log transformed data are consistent with a form of parsimony that is advantageous if we hope to find a relatively simple form of model. Note also that the graphs of `log(dist)` against `log(time)` and of `log(climb)` against `log(time)` are consistent with approximately linear relationships. Thus, we will work with the logged data:

```
loghills2k <- log(hills2k[-42, ])    # Omit the dubious point
names(loghills2k) <- c("ldist", "lclimb", "ltime", "ltimef")
loghills2k.lm <- lm(ltime ~ ldist + lclimb, data=loghills2k)
par(mfrow=c(2,2))
plot(loghills2k.lm)                  # Diagnostic plot
```

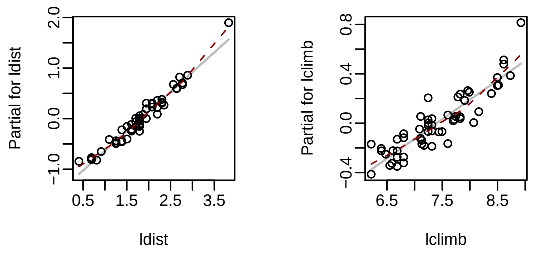


Figure 13: Term plot, showing the partial residuals for log(time) against log(dist) (left panel), and for log(time) against log(climb) (right panel). Smooth curves (dashes) that have been passed through the points. The vertical scale, which is centered on the mean of log(time), shows changes in log(time).

We pause at this point and 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 13).

```
## Plot the terms in the model
termplot(loghills2k.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 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

[7]There are several alternative forms of correlation coefficient. The usual form of correlation coefficient, such as we discuss here, is the "product-moment" correlation, which measures linear association.

residuals, are the primary feature of interest in these plots. In both panels, they show clear indications of curvature. A spline of degree 3 (by default a cubic polynomial) seemed adequate for capturing the curvature in the partial residuals for `ldist`, while a spline of degree 4 seemed adequate for capturing the slightly more complicated pattern of curvature in the partial residuals for `lclimb`. The function `ns()`, used to generate a spline basis, is in the *splines* package.

```
library(splines)      # Attach the splines package
loghills2ks.lm <- lm(ltime ~ ns(ldist,3) + ns(lclimb,4),
                     data=loghills2k)
par(mfrow=c(2,2))
plot(loghills2ks.lm)  # Diagnostic plot
```

Notice that, in the diagnostic plot, one point (row 19: 12 Trig Trog) has a huge Cook's distance. With a time of 8.3h, it is the longest of any of the races.

The following plots the contributions of the individual spline curves ("the effects"), shows the partial residuals, and passes a smooth curve (red dashes) through the partial residuals:

```
termplot(loghills2ks.lm, col.term="gray", partial=TRUE,
         col.res="black", smooth=panel.smooth)
```

Also the fitted curve for `lclimb` is not monotonic for small values of `lclimb`. It would be desirable to constrain it to be monotonic.

The Basis Functions

The basis functions are available for inspection, thus:

```
bases <- model.matrix(loghills2ks.lm)
colnames(bases)
options(digits=3)
bases[1:5,]
```

To see the shape of the basis curves, the best idea is to plot them. The following plots the four basis functions that are formed by `lclimb`

```
par(mfrow=c(2,2))
for (i in 0:3)plot(loghills2k$lclimb, bases[,5+i])
par(mfrow=c(1,1))
```

The contribution of `lclimb` to the fitted values is determined as a linear combination of these three curves.

7.2 Models that Include Factor Terms – Contrasts

A Simple Example

The data frame `cuckoos` (*DAAG*) has egg `length` for eggs laid in the nests of six different species of host bird.

```
> str(cuckoos)
'data.frame':   120 obs. of  4 variables:
 $ length : num  21.7 22.6 20.9 21.6 22.2 22.5 22.2 24.3 22.3 22.6 ...
 $ breadth: num  16.1 17 16.2 16.2 16.9 16.9 17.3 16.8 16.8 17 ...
 $ species: Factor w/ 6 levels "hedge.sparrow",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ id     : num  21 22 23 24 25 26 27 28 29 30 ...
```

The analysis of these data can be handled within the linear model framework, although this may not be the most natural approach for this relatively simple analysis. The factor `species` is a qualitative description of categories ("levels") in the data. The levels are:

```
> levels(cuckoos$species)
[1] "hedge.sparrow" "meadow.pipit"  "pied.wagtail"  "robin"
[5] "tree.pipit"    "wren"
```

For purposes of fitting a model the levels are taken to be 1, 2, . . . , k, where k is the number of levels. For a factor that is not ordered, these are qualitative distinctions. The first level (1) does however have a special role for the default parameterization, as a reference. (The order can easily be changed, so that one of the other species is placed first and thus becomes the reference.)

The following code demonstrates the choice of parameters for coding the factor `species`. It creates a factor that has one element for each of the six levels 1, 2, . . . , 6, and then forms the model matrix. The model matrix for `species` repeats each row from this model matrix as it is required:

```
> alpha <- factor(1:6)
> mm <- model.matrix(~alpha)
> rownames(mm) <- levels(cuckoos$species)
> mm
              (Intercept) alpha2 alpha3 alpha4 alpha5 alpha6
hedge.sparrow           1      0      0      0      0      0
meadow.pipit            1      1      0      0      0      0
pied.wagtail            1      0      1      0      0      0
robin                   1      0      0      1      0      0
tree.pipit              1      0      0      0      1      0
wren                    1      0      0      0      0      1
attr(,"assign")
[1] 0 1 1 1 1 1
attr(,"contrasts")
attr(,"contrasts")$alpha
[1] "contr.treatment"
```

The parameter `(Intercept)` is the expected value for `hedge.sparrow`. The parameter `alpha2` is the amount added to this to get the expected value for `meadow.pipit`. Similarly for each of the other species. For each level, the reference is `hedge.sparrow`.

The columns of the model matrix are known as “contrasts”. The values in columns of the model matrix after the first give contrasts with the reference. The specific form of contrasts used here, which are the default for qualitative factors, are called the “treatment” contrasts.

The mechanism just described allows the automatic creation of the model matrix for any qualitative factor. Extensions and variants will be described later. The following demonstrates its use to fit a simple one-way analysis of variance model, in which there is a different mean for each different species:

```
> cuckoos.lm <- lm(length ~ species, data=cuckoos)
> coef(cuckoos.lm)
        (Intercept)  speciesmeadow.pipit  speciespied.wagtail
            23.1143              -0.8210              -0.2276
       speciesrobin    speciestree.pipit          specieswren
            -0.5580              -0.0343              -1.9943
```

Much more information can be gleaned from the analysis, including an analysis of variance table, but the above will do for now. The estimated expected values for the different species are:

```
      (Intercept)        meadow.pipit        pied.wagtail
          23.1143      23.1143-0.8210      23.1143-0.2276
            robin          tree.pipit                wren
   23.1143-0.5580      23.1143-0.0343      23.1143-1.9943
```

Recall that `(Intercept)` is the reference, which is `hedge.sparrow`.

There are other ways to choose model parameters. Here are two of the possibilities:

```
> ## 1. Leave out the intercept term, i.e. reference = 0
> cuckoos.lm <- lm(length ~ -1+species, data=cuckoos)
> coef(cuckoos.lm)
specieshedge.sparrow   speciesmeadow.pipit   speciespied.wagtail
               23.11                 22.29                 22.89
        speciesrobin      speciestree.pipit           specieswren
               22.56                 23.08                 21.12
```

As the estimated values are the same as before, this is the same model, but with a different parameterization.

It may seem preferable to make the overall mean the reference. For this, specify the "sum" contrasts:

```
> ## 2. Make the overall mean the reference
> cuckoos.lm <- lm(length ~ C(species, sum), data=cuckoos)
> coef(cuckoos.lm)
     (Intercept) C(species, sum)1 C(species, sum)2 C(species, sum)3
         22.5084           0.6059          -0.2151           0.3782
C(species, sum)4 C(species, sum)5
          0.0478           0.5716
```

Now it is the final level that is omitted. The `C(species, sum)6` term, if it were given, would be calculated so that the sum of the six such terms is zero. Here are two ways that it might be calculated:

```
> coef(cuckoos.lm)[1] - sum(coef(cuckoos.lm)[-1])
(Intercept)
      21.12
> coef(cuckoos.lm) %*% c(1,-1,-1,-1,-1,-1)    # Scalar product
      [,1]
[1,] 21.12
```

The operator `%*%` is matrix multiplication. When the arguments are vectors, it treats the second argument as a row vector and thus forms the scalar product.

The inclusion of interaction terms

Extensions of the above ideas allow the creation of a model matrix, and thus the fitting of an `lm` model, for terms that include any combination of variables, spline terms, factors and interactions between such terms. Here is a simple example:

```
> x <- 1:6
> f1 <- factor(rep(1:2, 3))           # 1,2,1,2,1,2
> f2 <- factor(rep(1:3, rep(2,3)))    # 1,1,1,2,2,2
> model.matrix(~ x + f1 + f2 + f1:f2 + f1:x + f2:x)
  (Intercept) x f12 f22 f23 f12:f22 f12:f23 x:f12 x:f22 x:f23
1           1 1   0   0   0       0       0     0     0     0
2           1 2   1   0   0       0       0     2     0     0
3           1 3   0   1   0       0       0     0     3     0
4           1 4   1   1   0       1       0     4     4     0
5           1 5   0   0   1       0       0     0     0     5
6           1 6   1   0   1       0       1     6     0     6
attr(,"assign")
 [1] 0 1 2 3 3 4 4 5 6 6
attr(,"contrasts")
attr(,"contrasts")$f1
[1] "contr.treatment"
```

```
attr(,"contrasts")$f2
[1] "contr.treatment"
```

The model formula could have been specified more simply, as in:

```
model.matrix(~(x + f1 + f2)^2)
```

The effect of the `^2` is to include the main effects and pairwise interactions involving all terms that are specified within the round brackets.

Contrasts for ordered factors

Here the default is to use "polynomial" contrasts, which are successive orthogonal polynomial terms in the numeric factor values.

```
> tint <- ordered(c("lo", "med", "hi"), levels=c("lo", "med", "hi"))
> model.matrix(~tint)
  (Intercept)    tint.L tint.Q
1           1 -7.07e-01  0.408
2           1 -9.07e-17 -0.816
3           1  7.07e-01  0.408
attr(,"assign")
[1] 0 1 1
attr(,"contrasts")
attr(,"contrasts")$tint
[1] "contr.poly"
```

The values in `tint.L` are, to a close approximation, (-1/$\sqrt{2}$, 0, 1/$\sqrt{2}$). The values in `tint.Q` are obtained by dividing the vector (1, -2, 1) by $\sqrt{1^2 + 2^2 + 1^2}$.

7.3 Hierarchical Multi-level Models

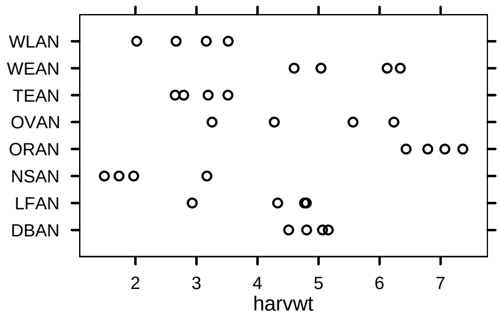


Figure 14: Yields from blocks on eight sites on the Caribbean island of Antigua. Data are a summarized version of a subset of data given in Andrews and Herzberg 1985, pp.$\tilde{3}$39-353.

Figure 14 shows corn yield data from the Caribbean island of Antigua. The figure can be obtained using the code:

```
# ant111b is in DAAG
library(lattice)
stripplot(site ~ harvwt, data=ant111b)
```

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. The analysis will use the abilities of the `lme()` function in the *nlme* package, though the example is one where it is easy to get the needed sums of squares from a linear model calculation. The data give results for each of several blocks at a number of different locations (sites). A prediction for a new block 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.

7.3.1 Analysis of the Antiguan corn yield data

The data that will be analyzed are in the second column of Table 3, which has the block means for the Antiguan data. In comparing yields from different blocks, there are two sorts of comparison. Blocks on the same location should be relatively similar, while blocks on different locations should be relatively more different. Figure 14 suggests that this is indeed the case.

Location	Location means	Location effect		Residuals from location mean
DBAN	5.16, 4.8, 5.07, 4.51		$+0.59$	$0.28, -0.08, 0.18, -0.38$
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	(4.29)	$+2.62$	$-0.13, 0.45, -0.48, 0.15$
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	$-0.82, -0.18, 0.32, 0.68$
		square, add, multiply by 4, divide by d.f.=7, to give ms		square, add, divide by d.f.=24, to give ms

Table 3: The leftmost column has harvest weights (`harvwt`), for the blocks 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). This information that can be used to create the analysis of variance table.)

Comparison with analysis of variance: In an analysis of variance formalization, the two-level structure of variation is handled by splitting variation, as measured by the total sum of squares about the grand mean, into two parts – variation within locations, and variation between location means. The final two columns in Table 3 indicate how to calculate the relevant sums of squares and (by dividing by degrees of freedom) mean squares. The division of the sum of squares into two parts mirrors two different types of predictions that can be based on these data.

Variance components – a multi-level model

The model that is used is:

$$\text{yield} = \text{overall mean} + \underset{\text{(random)}}{\text{location effect}} + \underset{\text{(random)}}{\text{block effect}}$$

In formal mathematical language:

$$y_{ij} = \mu + \underset{\text{(location, random)}}{\alpha_i} + \underset{\text{(block, random)}}{\beta_{ij}} \qquad (i = 1, \ldots, 8; j = 1, \ldots, 4)$$

with $\text{var}[\alpha_\text{i}] = \sigma_\text{L}^2$, $\text{var}[\beta_\text{ij}] = \sigma_\text{B}^2$.

The quantities σ_L^2 and σ_B^2 are known, technically, as *variance components.* The variance components analysis allows inferences that are not immediately available from the breakdown of the sums of squares in the analysis of variance table. Importantly, the variance components provide information that can help design another experiment.

Analysis using *lme*

The modeling command takes the form:

```
library(nlme)
ant111b.lme <- lme(fixed=harvwt ~ 1, random = ~1 | site,
                   data=ant111b)
```

The only fixed effect is the overall mean. The parameter `random = ~1 | site` fits random variation between locations. Variation between the individual units that are nested within locations, i.e., between blocks, are by default treated as random. Here is the default output:

```
> ant111b.lme
Linear mixed-effects model fit by REML
  Data: ant111b
  Log-restricted-likelihood: -47.208
  Fixed: harvwt ~ 1
(Intercept)
     4.2917

Random effects:
 Formula: ~1 | site
        (Intercept) Residual
StdDev:      1.5387  0.75996

Number of Observations: 32
Number of Groups: 8
```

Notice that *lme* gives, not the components of variance, but the standard deviations (`StdDev`) which are their square roots. Observe that, according to *lme*, $\widehat{\sigma_B^2}/n = 0.75996^2 = 0.57754$, and $\widehat{\sigma_L^2} = 1.5387^2 = 2.3676$.

Those who are familiar with an analysis of variance table for such data should note that *lme* does not give the mean square at any level higher than level 0, not even in this balanced case.

The take-home message from this analysis is:

- For prediction for a new block at one of the existing sites, the standard error is 0.76
- For prediction for a new block at a new site, the standard error is $\sqrt{1.5387^2 + .76^2} = 1.72$
- For prediction of the mean of n blocks at a new site, the standard error is $\sqrt{1.5387^2 + 0.76^2/n}$

Where there are multiple levels of variation, the predictive accuracy can be dramatically different, depending on what is to be predicted. Similar issues are arise in repeated measures contexts, and in time series. Repeated measures data has multiple profiles, i.e., many small time series.

Additional Calculations

Is variability between blocks similar at all locations?

```
library(DAAG)
vars <- sapply(split(ant111b$harvwt, ant111b$site), var)
vars <- vars/mean(vars)  # Standardize to a variance of one
qreference(3*vars, distribution=function(x)qchisq(x, df=3))
```

Does variation within locations follow a normal distribution?

```
qqnorm(residuals(ant111b.lme))
```

What is the pattern of variation between sites?

```
sitemeans <- sapply(split(ant111b$harvwt, ant111b$site), mean)
qqnorm(sitemeans)
```

The distribution seems remarkably close to normal.

Fitted values and residuals in *lme*

By default fitted values account for all random effects, except those at level 0. In the example under discussion `fitted(ant111b.lme)` calculates fitted values at level 1, which can be regarded as estimates of the location means. They are not however the location means, as the graph given by the following calculation demonstrates:

```
hat.lm <- fitted(lm(harvwt ~ site, data=ant111b))
hat.lme <- fitted(ant111b.lme)   # By default, level=1)
plot(hat.lme ~ hat.lm, xlab="Location means",
     ylab="Fitted values (BLUPS) from lme")
abline(0,1,col="red")   # Show the line y=x
```

The fitted values are known as BLUPs (Best Linear Unbiased Predictors). Relative to the location means, they are pulled in toward the overall mean. The most extreme location means will on average, because of random variation, be more extreme than the corresponding "true" means for those locations. There is a theory that gives the factor by which they should be shrunk in towards the true mean.

Residuals are by default the residuals from the block means, i.e., they are residuals from the fitted values at the highest level available. To get fitted values and residuals at level 0, enter:

```
hat0.lme <- fitted(ant111b.lme, level=0)
res0.lme <- resid(ant111b.lme, level=0)
plot(res0.lme, ant111b$harvwt - hat0.lme)  # Points lie on y=x
abline(0,1,col="red")
```

8 Common Uses for Key Language Ideas

Key notions and language structures:

Classes & methods	R's implementation (or, rather, implementations) of classes and methods makes generic functions possible (There are two implementations of classes and methods, the original S3 implementation, & the newer S4 implementation of the *methods* package.)
Formulae	They can be manipulated, like other objects. (Currently, there can be model, graphics and table formulae)
Expressions	It's not surprising that formulae can be evaluated. Perhaps more surprising is that they can be printed (on a graph)
Argument lists	Argument lists for functions can be constucted in advance, as a list of named values, with `do.call()` then used to pass the argument list to the function
Environments	Environments hold quite a number of subtleties. There are a few basic matters that it helps to know about.

While there is much that users can do without understanding in too much detail what happens "under the hood", a knowledge of important aspects of R's "construction kit" can be a great help, both in understanding why some things work the way they do, and in getting R to do things that otherwise would be impossible.

8.1 Generic Functions (Classes and Methods)

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 consider `print()`. For a factor, `print.factor()` is used, for a data frame `print.data.frame()` is used, and so on. Ordered factors "inherit" the print method for factors. For objects (such as numeric vectors) that do not otherwise have a print method, `print.default()` handles the printing.

Generic functions do not call the specific method, such as `print.factor()`, directly. Instead they call the `UseMethod()` function, which then calls the relevant method for that class of object, e.g., the factor method (such as `print.factor()`) for a factor object.

Thus, here is the function `print()`.

```
> print
function (x,...)
UseMethod("print")
```

The function `UseMethod()` notes the class of the object, now identified as `x`, and calls the print function for that class.

To get a list of the S3 methods that are available for a generic function such as `plot()`, type, e.g.:

```
methods(plot)
```

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()`.

The function `intervals.lme()` does however have its own help page that can be accessed with `help(intervals.lme)`. For accessing the code for this function, specify `getAnywhere(intervals.lme)`.

S4 Classes and Methods – the *methods* package

There are two sets of conventions and mechanisms – those of version 3 of the S language (S3), and those of version 4 of the S language (S4). The *methods* package makes available S4 style methods, which offer various advantages 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. 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.

Users of packages (e.g., *lme4*, or Bioconductor packages) may need to access the slots of S4 objects. 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, consider the following example

```
> ## ant111b is in the DAAG package
> library(lme4)       # lme4 must be installed
> ant111b.lmer <- lmer(harvwt ~ 1 + (1 | site), data = ant111b)
> slotNames(ant111b.lmer)
 [1] "assign"    "call"      "family"    "fitted"    "fixed"     "frame"
 [7] "logLik"    "residuals" "terms"     "flist"     "perm"      "Parent"
[13] "D"         "bVar"      "L"         "ZZpO"      "Omega"     "method"
[19] "RXX"       "RZX"       "XtX"       "ZtZ"       "ZtX"       "cnames"
[25] "devComp"   "deviance"  "nc"        "Gp"        "status"
> slot(ant111b.lmer, "assign")
[1] 0
> slot(ant111b.lmer, "call")
lmer(formula = harvwt ~ 1 + (1 | site), data = ant111b)
> ant111b.lmer@call
lmer(formula = harvwt ~ 1 + (1 | site), data = ant111b)
```

Most often, an extractor function will be used to extract some relevant part of the output. For example:

```
> VarCorr(ant111b.lmer)
 Groups   Name        Variance Std.Dev.
 site     (Intercept) 2.368    1.54
 Residual             0.578    0.76
```

For moderately simple examples of the definition and use of S4 classes and methods, see `help(setClass)` and `help(setMethod)`.

Model, Graphics and Table 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.

Expressions

Expressions can be:

evaluated (of course)

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

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.

8.2 Manipulation of Formulae

Model and Graphics Formulae

We demonstrate the construction of model or graphics formulae from text strings. For example, here is a function that takes two named columns from the data frame `mtcars` (in the *datasets* package that is part of the default installation), plotting them one against another:

```
plot.mtcars <- function(xvar="disp", yvar="mpg"){
      mt.txt <- paste(yvar, "~", xvar)
      plot(formula(mt.txt), xlab=xvar, ylab=yvar)
}
```

We can, when we call the function, set `xvar` and `yvar` to be any columns we choose:

```
> attach(mtcars)
> names(mtcars)
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
[11] "carb"
> plot.mtcars(xvar="hp", yvar="mpg")
```

Extraction of Variable Names from Formula Objects

The function `all.vars()` takes a formula as argument, and returns the names of the variables that appear in the formula. For example:

```
> all.vars(mpg ~ disp)
[1] "mpg" "disp"
```

As well as allowing the use of 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",
                    am= "Transmission (0 = automatic, 1 = manual)",
                    gear= "Number of forward gears",
                    carb= "Number of carburettors")
   xlab <- mtcars.info[xvar]
   ylab <- mtcars.info[yvar]
   plot(form, xlab=xlab, ylab=ylab)
}
```

8.3 Expressions

An expression is anything that can be evaluated. Thus `x^2` Is an expression, and `y == x^2` is an expression.

A simple use of `expression()` is to extract a text string representation of a function argument.

```
plot.mtcars <- function(x = disp, y = mpg){
    xvar <- deparse(expression(x))
    yvar <- deparse(expression(y))
    plot(y ~ x, xlab=xvar, ylab=yvar)
}
```

Formatting and plotting of text and equations

The construction

`expr <- expression(x^2)`

creates an expression that can be evaluated later. For example

```
> expr <- expression(x^2)
> x <- 5
> eval(expr)
[1] 25
> eval(expr, list(x=7))
[1] 49
```

Moreover, such an expression can be plotted:

```
x <- 1:10
plot(x, x^2, ylab=expression(x^2))
```

Some expressions that cannot be evaluated can nevertheless be plotted. For example the following is a legitimate expression:

`"Graph is for" * phantom(0) * italic("Acmena smithii")`

The construct phantom(0) inserts a space. The construct `italic("Acmena smithii")` gives italic text. The asterisks are left out. The reasoning is that `expression(x*y)` is written *xy*, whether *x* and *y* are quoted or unquoted. Such "expressions" can be included in any of the functions `text()` or `mtext()` or `title()`.

The following, which follows normal text with italic text, demonstrates the mixing of different text styles in a single line of text:

```
library(DAAG)    # rainforest is from DAAG
plot(wood ~ dbh, data = rainforest, subset = species=="Acmena smithii")
mtext(side=3, line=1.25, expression(italic("Acmena smithii") * ": " *
      "wood vs dbh"))
# phantom(0) inserts a space that is the width of a zero
```

Here is an example of the use of mathematical expression in an annotation for a graph. It uses both `expression()` and `substitute()`. This latter function is a generalization of `expression()` that allows the substitution of selected symbols in the expression:

```
## First plot the "angular" transformation
curve(asin(sqrt(x)), from = 0, to = 1.0, n=101,
xlab = expression(italic(p)),
ylab = expression(asin(sqrt(italic(p)))))
# Place a title above the graph
mtext(side = 3, line = 1,
      substitute(tx * italic(y) == asin(sqrt(italic(p))),
                 list(tx = "Plot of the angular function ")))
# Use '==' where '=' will appear in the plotted text
```

The following generates random numbers from the Weibull distribution, and gives a plot of the density:

```
random.weibull <- function(n=100, shape=1.75){
    x <- rweibull(n=100, shape=shape)
    simden <- density(x, from=0)
    xval <- pretty(x,50)
    theoryden <- list(x=xval, y=dweibull(pretty(x,50), shape=shape))
    plot(simden, ylim=range(c(simden$y, theoryden$y)))
lines(theoryden, col="red")
topright <- par()$usr[c(2,4)] - c(0, 1.5*par()$cxy[2])
text(topright[1], topright[2],
substitute(atop("Density is" * phantom(0) *
        f(x) == (a/b) (x/b)^(a-1) * exp(- (x/b)^a),
        "with"*phantom(0)*list(a==z, b==1)),
        list(z=shape)),  adj=1)
}
## Run function
random.weibull()
```

See help(plotmath) for further details.

8.4 The use of a list to pass parameter values

The following are equivalent:

```
mean(rnorm(20))
do.call("mean", args=list(x=rnorm(20)))
```

Use of do.call() allows the parameter list to be set up in advance of the call. The following function demonstrates a use for this.

```
simple.simulate.distribution <-
function(distn="weibull", parms=NULL, n=100){
## Simulates one of the distributions: weibull, gaussian,
## logistic, exponential
if(is.null(parms))
parms <- switch(distn, weibull=, list(shape=1), NULL)
## weibull requires a default shape parameter.
## ===================================================
## Choose the function that will generate the random sample
rfun <- switch(distn,
               weibull= rweibull,
               gaussian= rnorm,
               logistic= rlogis,
               exponential= rexp)

## Call the function. Use of do.call() allows giving
## the parameter list as a list of named values.
## (If the list is null, no parameters are passed.)
x <- do.call("rfun", args=c(parms, list(n=n)))
## Notice the use of c() to add another list item
invisible(x)
}
```

Note also call(), which sets up an unevaluated expression. The expression can be evaluated at some later time, using eval().

```
> mean.call <- call("mean", x=rnorm(5))
> eval(mean.call)
[1] -0.6276536
> eval(mean.call)
[1] -0.6276536
```

Notice that the argument `x` was evaluated when `call()` was evoked. Hence the result is unchanged upon repeating the use of `eval()`. This can be verified by printing out the expression:

```
> mean.call
mean(x = c(-0.68467334794551, -0.376091734366091, -0.289459988631994,

-3.04694266628697, 1.25889972957396))
```

8.5 *Symbolic substitution in parallel

This requires the use of a list of "quoted" unevaluated expressions. The list is passed, using `do.call()` to `expression()`, which returns a list of expressions that can then be plotted. The functions `quote()` returns a quoted unevaluated expression, while `bquote()` is an extension of `quote()` that allows symbolic substitution. These functions are used to provide the legend in Figure 15.

```
## Plot Acmena smithii data (subset of rainforest, from DAAG)
Acmena <- subset(rainforest, species="Acmena")
plot(wood~dbh, data=Acmena)
b <- coef(Acmena.lm <- lm(log(wood) ~ log(dbh), data=Acmena))
curve(exp(b[1])*x^b[2], add=TRUE)
b <- round(b,3)
arg1 <- bquote(italic(y) == .(A) * italic(x)^.(B),
                    list(A=b[1], B=b[2]))
arg2 <- quote("(" * italic(y) * "=wood; " *
                           italic(x) * "=dbh)")
legend("topleft", legend=do.call("expression", c(arg1, arg2)),
       bty="n")
```

Figure 15 has further refinements that are unrelated to the subject of this subsection. The code is:

```
plot(wood~dbh, data=Acmena, xlim=c(0, max(Acmena$dbh)))
## In fitting the curve, omit the point where dbh is largest
b <- round(lm(log(wood) ~ log(dbh), data=Acmena,
               subset=Acmena$dbh < max(Acmena$dbh)))
largest2 <- sort(Acmena$dbh, decreasing=TRUE)[1:2]
curve(exp(b[1])*x^b[2], from=min(Acmena$dbh), to=largest2[2], add=TRUE)
curve(exp(b[1])*x^b[2], from=largest2[2], to=largest2[1],
      lty=2, add=TRUE)
## Finally, use the code given above to add the legend and a title
```

The object `arg2` that is created by the call to `quote()` has class call; it can be included as a list element in the `args` parameter to `do.call()`, for passing to the function that is specified by the `what` argument to `do.call()`

The call to `bquote()` that creates `arg1` adds a further feature – the substitution of values for the variables `A` and `B`. The values of `A` in `.(A)`, and of `B` in `.(B)`, are each taken from the environment, here specified by `list(A=b[1], B=b[2])`.

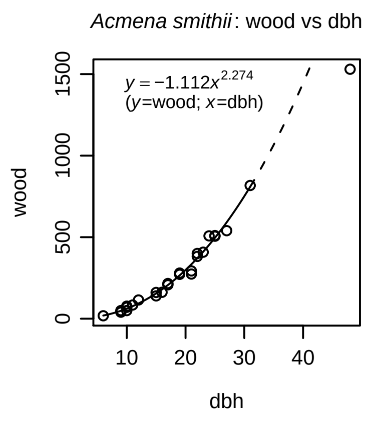


Figure 15: Plot of `wood` (wood biomass) vs `dbh` (diameter at breast height), for the species *Acmena smithii* (`rainforest`). The text describes the code used to obtain the annotation.

8.6 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 number 0. The frame number increases by 1 with each new function call. 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.

There are many other such functions, but these will do for now!

Here is a function that determines, from within the function, its name:

```
> test <- function(){
+  fname <- as.character(sys.call(sys.parent())[[1]])
+  fname
+ }
> test()
[1] "test"
> newtest <- test
> newtest()
[1] "newtest"
```

I like to call my figures, in the directory that relates to a paper that I am writing, `fig1()`, `fig2()`, etc. These functions will in turn call a function `gf()` that calls the graphics device, and specifies the file that will be used to store the graph. The definition of `gf()` is:

```
gf <-
    function(width=2.25, height=2.25, color=F, pointsize=8){
        funtxt <- sys.call(1)  # Sea
        fnam <- paste(funtxt, ".pdf", sep="")
        print(paste("Output will be to the file '",
              fnam, "'", sep=""))
        pdf(file=fnam, width=width, height=height, pointsize=
            pointsize)
    }
```

Now create a function that calls `gf()`:

```
graph1 <- function(){
    gf()            # Call with default parameters
    curve(sin, -pi, 2*pi)
    dev.off()
}
```

Output goes to the file **graph1.pdf**. For a function `graph2()` that calls `gf()`, the file name will be **graph2.pdf**. Similarly for `fig1()` or `fig2()` or any other function that calls `gf()`.

Bound and Unbound Variables

Variables that are passed as formal arguments to a function are bound variables. Local variables are those that are created within the body of the function. Unbound variables 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.

8.7 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.

8.8 Exercises

1. Modify the function `simulate.simple.distribution()` so that it allows, additonanally, the simulation of the lognormal, the loglogistic and the (Type I) extreme value distribution. You may call it `simulate.NotSoSimple.distribution()`.
2. Write a function that takes a vector of numerical values `x` and gives, in a 2 by layout
a. a density plot
b. a boxplot
c. a cumulative probability plot
d. a quantile-quantile plot, against some specified reference distribution.
3. Write a function that is designed to give a density plot for the output from `simulate.simple.distribution()`. On the density plot, it will overlay the density for the theoretical distribution. It will print the formula for the density, noting below the formula the actual parameter values that were used.
[Note: As it stands, this will be a tedious exercise. Hence, we settle for a proof of concept, i.e., set this up so that it works for the weibull, with placeholders for other distributions that can be filled later.]

9 Additional Notes on R

This section gathers up a number of further matters that can be important.

Key notions and language structures:

Entry of data	Where do I start when my attempt to input data generates an error? `scan()` is a more flexible alternative to `read.table()`
`sapply()` and friends	The functions `sapply()`, `lapply()` and `apply()` offer flexible mechanisms for carrying out operations in parallel across all columns of a data frame or, for (`apply()`, across all rows or columns of a matrix.
`Inf` and friends	The logarithm of zero returns `-Inf`. Take care!
Large datasets	A little knowhow can save a load of time.
Workspaces	Manage them carefully!

9.1 Entry of data using read.table() and scan()

read.table() – Errors when reading in data

Carefully check the option settings for the version of the input command that is in use. If some text strings have embedded single quotes, it may be necessary to set `quote = ""`. There may be text strings that have a # embedded; the default (which can of course be changed) is then to ignore the rest of the input line.

The function `count.fields()` can be a useful way to determine how many fields the input function thinks it has found in each record. Alternatively, use `read.table()` with the parameter setting `fill=TRUE`, and carefully check the input data frame. Blank fields will be implicitly added, as needed, in order to ensure that all records have an equal number of fields.

read.table() – Additional points

Users can set parameters that give a great deal of detailed control over the way that `read.table()` works. These allow control over: the character(s) used to separate fields (specify `sep`), the missing value character(s) (specify `na.strings`), the quote character(s) (specify `quote`), the number of lines to skip at the beginning of the file (specify `skip`), etc. There are a number of aliases for `read.table()` that have different settings for these and other defaults. See `help(read.table)`. Note however that non-default option settings can, for large files, severely slow down data input.

Other points, additional to those made earlier, are:

- Columns that are taken as character are by default converted to factors. For now, think of such columns as vectors of character strings. The character strings are not stored directly; instead codes are stored that identify the character strings. In many contexts, the factors that result from conversion from vectors of character strings behave just like vectors of character strings. There are however some contexts where they do not, which can be a source of puzzlement and frustration for novices.
- When a vector of character strings is converted to a factor the different text strings, by default taken in alphanumeric order, become factor *levels*. Factor levels are stored in a table, in which the row number is the code for the level. The data frame holds the code, not the level. This can be useful in reducing memory requirements when some columns hold a small number of distinct and perhaps longish text strings, each of which is repeated a number of times.

The sma package has a function `read.spot()` that is designed for use with data that are output from the Spot image analysis package. This is a slight adaptation of `read.table()`.

*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 function `scan()` has the necessary flexibility. There may be other reasons for using `scan()`. With large files, data input is much faster than with `read.table()`.

The following code demonstrates the use of `scan()` to read in the file **molclock.txt**.

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

This could easily be put into a function that accepts as parameters the file name, the number of lines per record, and the what list.

Notice that that there were two calls to `scan()`, each time with the same file **molclock.txt**. The first (with `nlines=1` and `what=""`), recovered the column names, while the second [with `skip=1` and `what=c(list(""), rep(list(1),5)))`], recovered the entries for the several rows of data. The calls introduces the use of a the `what` parameter, which expects a list as argument. There is one list element for each column that is to be input. The ”” in the first list element indicates that the data is to be input as character. The remaining five list elements hold 1's, indicating numeric data.

Where there are a number of data files that have the same format, it makes sense to put the code into a function, probably with the what list as a parameter. Where records extend over several lines, it will be necessary to set `multi.line=TRUE`. The length of the `what` list gives the number of fields in each record.

Large files – Reading in part of a file

Difficulties with large files can often be eased by reading in part only of a file at any time. Both `read.table()` and `scan()` allow this. Give skip the value needed to get to the point where you will start reading. For `read.table()` set `nrows` to the number of lines to be read, while for `scan()` use `nlines` for this purpose.

9.2 The apply family of functions

The function `apply()` handles operations that are carried out across rows (dimension 1) or down columns (dimension 2) of matrices. The function `sapply()` is primarily for use with data frames. Thus, for work with microarrays, it is usually `apply()` that is required.

The `apply()` function

The function `apply()` may be used both with data frames and matrices.[8] 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.

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

Here is a more interesting example, using the multi-way table `UCBAdmissions` that is in the *base* package. We'd like to see how the admission rates, for males and females separately, compare across the departments. Here is how to do it:

```
> apply(UCBAdmissions, c(2,3), function(x)x[1]/sum(x))

Dept
Gender A B C D E F
Male 0.620606 0.6303571 0.3692308 0.3309353 0.2774869 0.05898123

Female 0.824074 0.6800000 0.3406408 0.3493333 0.2391858 0.07038123
```

[8]More generally, it can be used with arrays. Arrays are a generalization of matrices to allow an arbitrary number of dimensions.

Compare this with

```
> apply(UCBAdmissions, c(1,2), sum)
Gender
Admit Male Female
Admitted 1198 557
Rejected 1493 1278
```

Next, to get the overall admission rates, we demonstrate a double use of `apply()`.

```
> apply(apply(UCBAdmissions, c(1,2), sum), 2, function(x)x[1]/sum(x))

Male Female
0.4451877 0.3035422
```

The `sapply()` function

The function `sapply()` makes it possible to apply functions such as `mean()`, `range()`, etc., in parallel to all columns of a data frame. It takes as arguments the name of the data frame, and the function that is to be applied. Here are examples:

```
> sapply(molclock, range)
AvRate Myr Gpdh Sod Xdh
[1,] 12 55 1.5 13 12
[2,] 25 1100 40.0 46 32
```

One can specify `na.rm=T` as a third argument to the function `sapply`. This argument is then automatically passed to the function that is given in the second argument position. The following shows the syntax:

```
> sapply(molclock, range, na.rm=T)
AvRate Myr Gpdh Sod Xdh
[1,] 12 55 1.5 13 12
[2,] 25 1100 40.0 46 32
```

The function `lapply()` has the same syntax as `sapply()`, but returns a list that has one element corresponding to each element of the argument. It is the appropriate function to use when the function that is the second argument either
(i) does not return a vector, or
(ii) returns vectors that are of different lengths for different elements of the list or data frame that is the first parameter.
(For a data frame, the different columns of the data frame count as different list elements.)

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

Warning: It is possible to use `sapply()` or `lapply()` with a matrix. However the result is different to that from use of these functions with a data frame that has the same dimensions. The functions `sapply()` and `lapply()` treat the matrix as a vector that has as many elements as there are matrix elements, applying the function that is specified as the second argument to each element in turn. This can be a trap for functions that return one or more of their output structures as matrices rather than data frames, and is particularly disastrous if the matrix is large. For example, the scores that are returned by the principal components function `prcomp()` are stored in a matrix. To find the ranges of each of the columns either use `apply()`, which is designed for use with matrices, or else first convert the matrix to a data frame.

More efficient alternatives to apply() and sapply()

The `apply()` family of functions may take an unreasonably long time when data sets are large, e.g., depending on available memory, 50,000 or 100,000 rows. If there is a ready alternative that uses matrix multiplication or `sweep()`, this is likely to be much faster. The speed of execution of these functions may improve greatly at some future time, with the implementation of major components of the calculations in compiled C code.

9.3 Logarithms, with some zero or negative numbers

This can be a particular issue for microarrray data. Numbers that are zero or negative require careful attention when data are to be transformed to a logarithmic scale, to avoid unnecessary loss of information and to avoid numerical problems that may arise if logarithms are taken regardless of zero or negative arguments. Zeros are an issue for the `mouse.data` data in the *sma* package. There are, adding over all six slides, 74 spots where `R`, `Rb`, `G` and `Gb` are all zero. For all other spots `R>Rb` and `G>Gb`.

Calculation of `log(0)` yields `–Inf`, which R treats, for some purposes at least, as a number. The attempt to calculate, e.g., `log(-1)` generates an `NaN`, which is for most purposes treated as an `NA`. The quantities `–Inf` and `Inf` are not missing values, and are not treated as such. Some calculations that cope with `NA`s by omitting them before proceeding will however fail if they encounter –Inf or Inf. Perhaps the easiest way to deal with them is to turn all such quantities into `NA`s, which can then be detected and omitted using `na.omit()`. There is some loss of information.

The *sma* package has a function `log.na()` that is a replacement for `log()`. This returns `NA` whenever the function argument is zero or negative. Points where there are `NA`s are then omitted in plots and in subsequent calculations.

Rather than omitting negative or zero numbers prior to taking logarithms, we might prefer to retain the information that these are the smallest of the expression values. For zeros, this can be achieved by replacing zero values by a suitable small number, smaller for background values than for the signal.

In replacing the zeros by small numbers, how might we proceed? In order to get an idea of suitable small numbers for data in the `mouse.data` object we find, for each of `R`, `Rb`, `G` and `Gb`, the smallest non-zero value. These are:

```
> attach(mouse.data)
> min(R[R>0])
[1] 929.2
> min(Rb[Rb>0])
[1] 688
> min(G[G>0])
[1] 231.63
> min(Gb[Gb>0])
[1] 100.04
> detach(mouse.data)
```

It might be reasonable to replace zeros of `R` with 475 (around half of 929.2), zeros of `Rb` with 350, zeros of `G` with 115, and zeros of `Gb` with 50.

Note that different functions handle `NA`s in different ways. The `plot()` function is accepting of `NA`s, infinities and `NaN`s, handling such points by omitting them. If however we wish to use `lowess()` to put a smooth curve through the plot, we need first to remove `NA`s.

9.4 Computations with Large Datasets

Most of R's modeling functions (regression, smoothing, discriminant analysis, etc.) are designed to work with data frames. If the data set is so large that these calculations are slow, consider whether it makes sense to run the function separately on different subsets of the data, or on a random subset of the data, or on some suitably summarized version of the data. Where a random subset is used, it can be advantageous to repeat the analysis for several different random subsets.

Functions that are specifically written for use with large data sets may prefer data that are stored in matrix form. Microarray applications, where it would not make sense to run calculations on a subset of spots ("genes"), are an example. The matrix or matrices may be list element(s) in a more complex data structure.

Note the following points:

Use matrices, if possible, in preference to data frames: Computations with large matrices are typically much faster than the equivalent computations with large data frames.

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(500000),ncol=50)
> dim(xy)
[1] 10000 50
> system.time(xy+1)
[1] 0.05 0.00 0.08 0.00 0.00
> ## Times are: user cpu, system cpu, elapsed, subproc1, subproc2
> xy.df <- data.frame(xy)
> system.time(xy.df+1)
[1] 3.24 0.00 8.23 0.00 0.00
```

The two non-zero numbers are of interest to us. The first is processor time and the second is elapsed time. The above was on a 4Mhz Macintosh G4 laptop with 768MB of random access memory.

Use the most efficient coding: Matrix arithmetic can be much faster than the equivalent computations that use `apply()`. Here are timings for some alternative ways to find the sums of rows of a matrix:

```
> xy <- matrix(rnorm(200000), nrow=2000)
> system.time(apply(xy,1,sum))
[1] 0.25 0.00 0.60 0.00 0.00
> system.time(xy %*% rep(1,100))
[1] 0.01 0.00 0.06 0.00 0.00
> system.time(rowSums(xy))
[1] 0.01 0.00 0.01 0.00 0.00
>
> xy2 <- matrix(rnorm(200000), nrow=100)

> system.time(apply(xy2,1,sum))
[1] 0.10 0.00 0.25 0.00 0.00
> system.time(xy2%*%rep(1,2000))
[1] 0.01 0.00 0.02 0.00 0.00
> system.time(rowSums(xy2))
[1] 0.01 0.00 0.02 0.00 0.00
```

The first time is processor time, in seconds. The third is elapsed time.

Unnecessary formation of a diagonal matrix should however be avoided. The following are equivalent:

```
> xy <- matrix(rnorm(200000), nrow=2000)
> dd <- sample(2000)
> system.time(diag(dd)%*%xy)
[1] 3.26 0.00 8.44 0.00 0.00
> system.time(sweep(xy, 1, dd, "*"))
[1] 0.42 0.00 1.07 0.00 0.00
> system.time(xy*dd)
[1] 0.02 0.00 0.10 0.00 0.00
```

Suppose however that we want to multiply each column of `xy` by a constant. Is it helpful to transpose `xy`?

```
> dd100 <- sample(100)
> system.time(xy%*%diag(dd100))
[1] 0.12 0.00 0.45 0.00 0.00
> system.time(sweep(xy, 2, dd100, "*"))
[1] 0.12 0.00 0.19 0.00 0.00
> system.time(t(xy)*dd100)
[1] 0.06 0.00 0.29 0.00 0.00
```

Observe that processor time decreases, but elapsed time increases, between methods 2 and 3.

Skinny Matrices can be Advantageous Processing can be more efficient for skinny matrices (many rows and few columns). Internally, successive elements in a column are stored in sequence. The requirement for fresh memory access is reduced when elements are accessed one after another down a column.

The singular value decomposition gives the same information whether applied to the matrix $\mathbf{X}$ or to its transpose. Calculation of $\mathbf{X} = \mathbf{U}\ \mathbf{D}\ \mathbf{V}^T$ is equivalent to
$\mathbf{X}^T = \mathbf{V}\ \mathrm{D}\ \mathbf{U}^T$,
where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices and $\mathbf{D}$ is diagonal.

```
> system.time(svd(xy))         # xy is 2000 x 100
[1] 1.28 0.00 3.35 0.00 0.00
> xyt <- t(xy)
> system.time(svd(xyt))        # xy2 is 100 x 2000
[1] 1.90 0.00 4.81 0.00 0.00
```

Data Base Connections

The relevant tables can be huge, so that bringing them across into R may involve huge delay. Ripley and Fei Chen (2003) discuss ways to make such manipulations tractable.

9.5 Workspace management

Even with careful housekeeping, the workspace may soon become cluttered, and data and other objects that are in use must be identified from among the clutter. Large data objects, such as are common in expression array work, take up what may for some tasks be a scarce memory resource.

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.

It is straightforward to move, within an R session, from one working directory to another. Use `save.image()` to save the contents of the current workspace in the current working directory, `rm(list=ls())` to clear the workspace, `setwd("newdir")` to set the new working directory to, e.g., `newdir`, and `load(".Rdata")` to load the new workspace. These operations can all be carried out from the menu, where available. The effect is to change `.Globalenv`.

Use `getwd()` to check the name and path of the current working directory. Note also the utility function `dir()` (get the names of files, by default in the current working directory).

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.

Under Windows, and under the Mac OSX/Carbon version of R, there are File menu items **Save Workspace...**, **Change Dir...** and **Load Workspace...**. These can be used to save the current workspace, then moving to a new workspace in a new 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. A good precaution, before removing any objects, is to make an archive of the workspace. For this, type:

```
save.image(file="archive.Rdata")
```

In place of `archive`, it might be better to use, e.g., the date when the file was created, e.g.

```
save.image(file="a31-1-03.Rdata")
```

Objects can then be removed without too much worry. If some objects that were removed turn out to be needed, they can be recovered from the "archive".

Movement of files between computers

Files that are saved in the default binary save file format can usually be moved between different computer systems. There may be unusual platforms where the XDR binary interchange format used by R is unavailable; in this case use `save()` with the parameter setting `ascii = TRUE`.

*Further possibilities – saving objects in text form

Data frames and vectors can be dumped to disk with a command such as

```
dump("molclock", file="molclock.R")
source("molclock.R")     # Use to retrieve molclock
```

The same form of `source()` command can be used to input R script files, i.e., files that contain R code. It can in principle be used with any R objects. However there are checks on dependencies that can cause strife. Problems can usually be resolved by editing the R source file to remove lines that lead to problems.

9.6 Summary

`scan()` can be a useful alternative to `read.table()` when large data sets are input, or when row spreads over more than one line of the file.

`apply(), sapply()` and `reshape()` can be useful for manipulations with data frames and matrices

Specific action may be needed, when some numbers are zero or negative, in taking logarithms

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

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.

9.7 References

Ripley, B.D. and Fei Chen, R. 2003. Data mining by scaling up open source software. Invited paper, Proceedings of the 2003 session of the International Statistical Institute.

Venables, W.N. and Ripley, B.D. 2000. S Programming. Springer-Verlag, New York.

10 Packages

10.1 Base Packages

These are *base*, *methods*, *stats*, *graphics*, *grDevices*, *utils* and *datasets*. These are all, in a custom installation, attached by default at startup

> The *methods* package has formally defined methods and classes for R objects, plus other programming tools, as described in the "Green Book"
> ("Programming with Data" (1998), by John M. Chambers, Springer.)
>
> The *graphics* package has R's regular graphics functions.
>
> The *stats* package has basic abilities for handling density estimation, tests, linear models, multivariate analysis, smoothing, time series, maximum likelihood maximization and profiling, nonlinear least squares, etc.
>
> The *utils* package has a variety of utility functions, including `Sweave()` and functions for editing data.
>
> The *grDevices* package has device drivers.
>
> The *datasets* package has datasets.

10.2 Recommended packages

"Recommended" packages that are included along with the base packages in all binary distributions are:

> *boot*: Functions and datasets for bootstrapping from the book "Bootstrap Methods and Their Applications" by A. C. Davison and D. V. Hinkley, 1997, Cambridge University Press.
>
> *class*: Functions for classification (k-nearest neighbor and LVQ). Contained in the VR bundle.
>
> *cluster*: Functions for cluster analysis.
>
> *foreign*: Functions for reading and writing data stored by statistical software like Minitab, SAS, SPSS, Stata, etc.
>
> *grid:* This provides a framework, alternative to that of the `graphics` package, for graphics. The *lattice* package uses this framework.
>
> *tcltk:* Interface and language bindings to Tcl/Tk GUI elements.
>
> *nlme:* Linear and non-linear maximum likelihood estimation.
>
> *KernSmooth*: Functions for kernel smoothing (and density estimation) corresponding to the book "Kernel Smoothing" by M. P. Wand and M. C. Jones, 1995.
>
> *lattice*: Lattice graphics, an implementation of Trellis Graphics.
>
> *MASS*: Functions and datasets from the main package of Venables and Ripley, "Modern Applied Statistics with S". Contained in theVR bundle
>
> *mgcv*: Routines for GAMs and other genralized ridge regression problems with multiple smoothing parameter selection by GCV or UBRE
>
> *nlme*: Fit and compare Gaussian linear and nonlinear mixed-effects and repeated measures models. Note also *lme4*, which is a beta version of package, based on S4 classes and methods, that includes and extends `lme()` and related functions from the *nlme* package.

nnet: Software for single hidden layer perceptrons ("feed-forward neural networks"), and for multinomial log-linear models. Contained in the VR bundle

rpart: Recursive PARTitioning and regression trees.

spatial: Functions for kriging and point pattern analysis from "Modern Applied Statistics with S" by W. Venables and B. Ripley. Contained in the VR bundle

survival: Functions for survival analysis, proportional hazards regression and related analyses.

10.3 Contributed Packages

Note that new packages appear every few weeks. The following are in any case a small selection from what is available.

Bayesian methods

gbayes:

GLMMGibbs: Generalized Linear Mixed Models

lmm: Linear mixed models, using Bayesian methods

mcmc: Markov Chain Monte Carlo.

ordinal: Models and utilities for categorical data.

Categorical data

cat: Analysis of categorical data where some data are missing.

repeated: Models for non-normal repeated measurements, including categorical measurements

vcd: Functions and data sets based on the book "Visualizing Categorical Data" by Michael Friendly.

GUIs

Rcmdr: A basic-statistics graphical user interface to R

Time series

In addition to abilities in *stats*, note

pear: for modeling periodic time series

tseries: for modeling nonlinear time series

fracdiff: for modeling long-memory time series

strucchange: for estimation of change points in time series

Note also the time series abilities in *nlme*.

Design of Experiments

AlgDesign: Algorithmic experimental designs. "Calculates exact and approximate theory experimental designs for D,A, and I criteria. Very large designs may be created. Experimental designs may be blocked or blocked designs created from a candidate list, using several criteria. The blocking can be done when whole and within plot factors interact."

conf.design: Construction of factorial designs

crossdes: Design and Randomization in Crossover Studies

The following packages are designed for applications in population genetics:

ldDesign: Design of experiments for detection of linkage disequilibrium

powerpkg: Power analyses for the affected sib pair and the TDT design

qtlDesign: Design of QTL experiments

The brewing and use of colors

dichromat: Simulation of the effect of two different types of red-green color blindness

RColorBrewer: Discrete color combinations that are effective for coloring maps and images.

Fun and recreational packages

fortunes: R fortunes. Attach the package and type `fortunes()`

magic: Create and investigate magic squares.

Other packages

There is a huge variety of other packages. Here is a small selection:

classPP: Projection Pursuit for supervised classification.

clim.pact: Climate analysis and downscaling for monthly and daily data

clines: Calculates Contour Lines

nlmeODE: Combine the **nlme** and *odesolve* packages for mixed-effects modelling using differential equations.

nlrq: Nonlinear quantile regression.

corpora: Statistical analysis of corpus frequency data.

TeachingDemos: Demonstrations for teaching and learning.

10.4 Multivariate analysis – what does R offer?

The most important of the multivariate libraries that are part of the R distribution are *stats*, *cluster* and *mass*. Below, I list functions that may be of interest from these libraries. For more complete details, type in commands of the form `help(package=cluster)`, and then look at help pages for individual functions that are of interest. See also Venables and Ripley (2002).

cluster

The available clustering functions are:
agnes: Agglomerative Nesting
clara: Clustering Large Applications
daisy: Dissimilarity Matrix Calculation
diana: Divisive Analysis
fanny: Fuzzy Analysis
mona: Monothetic Analysis
pam: Partitioning Around Medoids

Multivariate abilities in package *stats*

as.hclust: Convert Objects to Class hclust
biplot: Biplot of Multivariate Data
biplot.princomp: Biplot for Principal Components
cancor: Canonical Correlations
cmdscale: Classical (Metric) Multidimensional Scaling
cutree: Cut a tree into groups of data
dist: Distance Matrix Computation
hclust: Hierarchical Clustering
identify.hclust: Identify Clusters in a Dendrogram
kmeans: K-Means Clustering
prcomp: Principal Components Analysis
princomp: Principal Components Analysis
rect.hclust: Draw Rectangles Around Hierarchical Clusters

MASS

isoMDS: Kruskal's Non-metric Multidimensional Scaling
kde2d: Two-dimensional Kernel Density Estimation
lda: Linear Discriminant Analysis
ldahist: Histograms or Density Plots of Multiple Groups
lqs: Linear Quantile Smoothing, i.e., resistant regression and covariance estimation, which aims to identify and use the "good" points in the data.
pairs.lda: Produce pairwise scatterplots from an 'lda' Fit
parcoord: Parallel Coordinates plot
plot.lda: Plot method for class 'lda'
predict.lda: Classify multivariate observations by Linear Discrimination
predict.qda: Classify using Quadratic Discriminant Analysis
qda: Quadratic Discriminant Analysis
sammon: Sammon's non-linear mapping.

11 References and Bibliography

11.1 Books and Papers on R

Dalgaard, P. 2002. Introductory Statistics with R. Springer-Verlag, New York.
[An excellent R-based introductory statistics text]

Fox, J. 2002. An R and S-PLUS Companion to Applied Regression. Sage Books.
(web page `http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/index.html`)
[This is particularly aimed at classical types of regression calculations.]

Kuhnert, P. and Venables, W. 2005. An Introduction to R: Software for Statistical Modelling & Computing. CSIRO Australia. Available from
`http://cran.r-project.org/doc/contrib/Kuhnert+Venables-R_Course_Notes.zip`

Ihaka, R. & Gentleman, R. 1996. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics 5: 299-314.

Maindonald, J. H. & Braun, J. B. 2003. Data Analysis & Graphics Using R. An Example-Based Approach. Cambridge University Press, Cambridge, UK.
(web page `http://www.maths.anu.edu.au/~johnm/r-book.html`
[This is aimed at researchers who have had some previous exposure to statistics, and at applied statisticians. A new edition is due in October 2006.]

Venables, W.N. and Ripley, B.D. 2000. S Programming. Springer-Verlag, New York.
[This treats both R and S-PLUS.]

Venables, W.N. and Ripley, B.D., 4^{th} edn 2002. Modern Applied Statistics with S. Springer.
[This demands a relatively high level of sophistication. This treats both R and S-PLUS.]

11.2 Graphics

Cleveland, W. S. 1985. The Elements of Graphing Data. Wadsworth, Monterey, California.

Maindonald J H 1992. Statistical design, analysis and presentation issues. New Zealand Journal of Agricultural Research 35: 121-141.

Murrell, P. 2005. R Graphics. Chapman & Hall/CRC.
`http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html.`
[This is a detailed exposition of the R graphics systems, with examples of their use.]

Tufte, E. R. 1983. The Visual Display of Quantitative Information. Graphics Press, Cheshire, Connecticut, U.S.A.

Tufte, E. R. 1990. Envisioning Information. Graphics Press, Cheshire, Connecticut, U.S.A.

Tufte, E. R. 1997. Visual Explanations. Graphics Press, Cheshire, Connecticut, U.S.A.

Wainer, H. 1997. Visual Revelations. Springer-Verlag, New York

Literature on trellis (lattice) graphics

Bell Lab's Trellis Page:
`http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/`

Becker, R.A., Cleveland, W.S. and Shyu, M. The Visual Design and Control of Trellis Display. Journal of Computational and Graphical Statistics.

Cleveland, W. S. 1993. Visualizing Data. Hobart Press, Summit, New Jersey.

11.3 Influences on the Development of R – Lisp-Stat

de Leeuw, J. 2005. On Abandoning XLISP-STAT. *Journal of Statistical Software* 13, issue 7.

Narasimhan, B. 2005. Lisp-Stat to Java to R. *Journal of Statistical Software* 13, Issue 4.

Tierney, L. 2005. Some Notes on the Past and Future of Lisp-Stat. *Journal of Statistical Software* 13, Issue 9.

Papers that appear in the Journal of Statistical Software can be downloaded from `http://www.jstatsoft.org/`

Index of Functions

Functions not otherwise identified are from one of the base packages that, in most “out-of-the-box” installations, are loaded at startup: *methods*, *stats*, *graphics*, *grDevices*, *utils*, *datasets*.