

Using R for Data Analysis and Graphics

An Introduction

J H Maindonald

**Statistical Consulting Unit of the Graduate School,
Australian National University.**

©J. H. Maindonald 2001. A licence is granted for personal study and classroom use. Redistribution in any other form is prohibited.

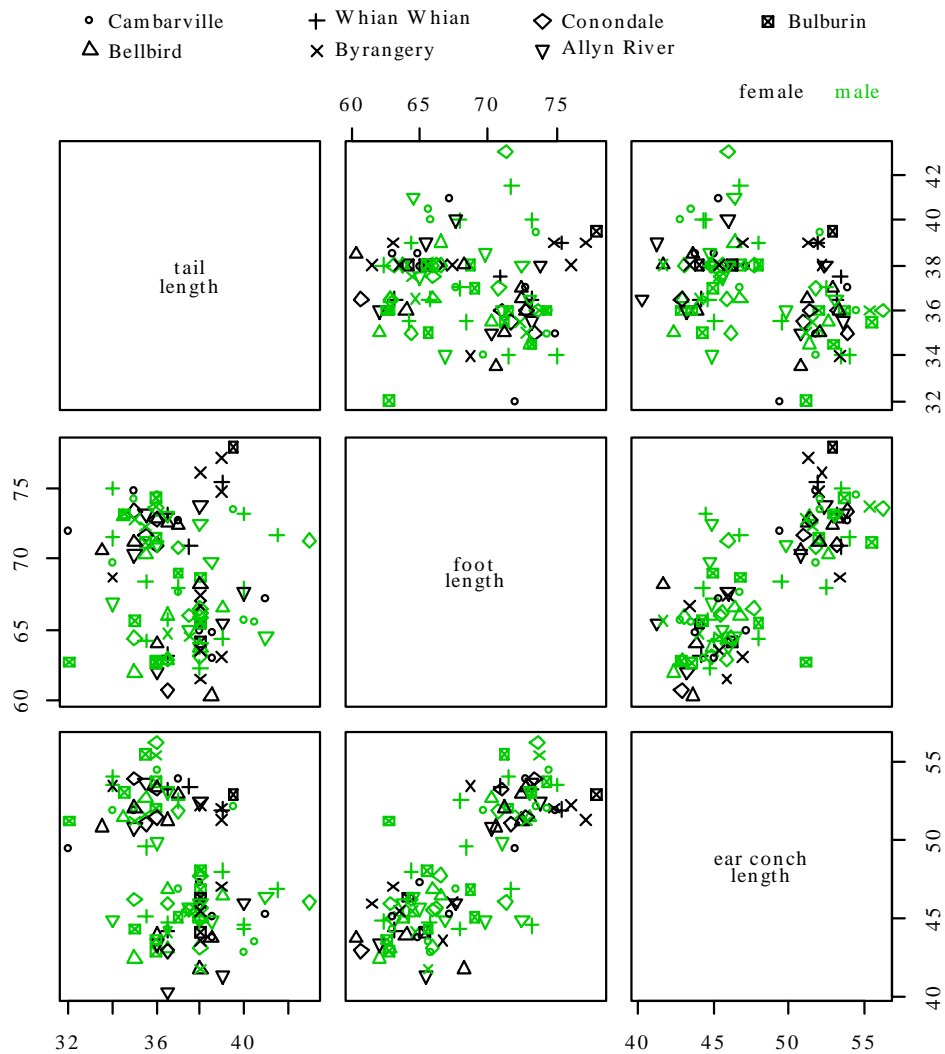
Alexis Diamond has customized this document for the 2004 Harvard Gov Dept Math Camp. All added text appears in a crimson Arial font.

This tutorial is designed to introduce the basics of R. If this is your first programming experience, it might take some time to get used to it. Hang in there; like any new language, fluency in R will require patience and practice. Before you know it, you'll be able to produce figures and analyze information in creative and innovative ways that serve your specific research goals.

Many thanks to J.H. Maindonald for granting permission to use his materials. His original document can be found at

<http://cran.r-project.org/doc/contrib/usingR.pdf>

Languages shape the way we think, and determine what we can think about (Benjamin Whorf.).



Lindenmayer, D. B., Viggers, K. L., Cunningham, R. B., and Donnelly, C. F. : Morphological variation among populations of the mountain brushtail possum, *trichosurus caninus* Ogibly (Phalangeridae:Marsupialia). *Australian Journal of Zoology* 43: 449-459, 1995.

possum *n.* **1** Any of many chiefly herbivorous, long-tailed, tree-dwelling, mainly Australian marsupials, some of which are gliding animals (e.g. *brush-tailed possum*, *flying possum*). **2** a mildly scornful term for a person. **3** an affectionate mode of address.

From the Australian Oxford Paperback Dictionary, 2nd ed, 1996.

Contents (Math Camp focuses on chapters 1-3 & 7, but the full T.O.C is retained below)

Introduction	1
1. Starting Up.....	3
1.1 <i>Getting started under Windows</i>	3
1.2 <i>Using the Console (or Command Line) Window</i>	5
1.3 <i>A Short R Session</i>	5
1.4 <i>Further Notational Details</i>	7
1.5 <i>On-line Help</i>	7
1.6 <i>Exercise</i>	8
2. An Overview of R	9
2.1 <i>The Uses of R</i>	9
2.2 <i>The Look and Feel of R</i>	11
2.3 <i>R Objects</i>	12
*2.4 <i>Looping</i>	12
2.5 <i>R Functions</i>	13
2.6 <i>Vectors</i>	14
2.7 <i>Data Frames</i>	16
2.8 <i>Common Useful Functions</i>	18
2.9 <i>Making Tables</i>	19
2.10 <i>The R Directory Structure</i>	19
2.11 <i>More Detailed Information</i>	20
2.11 <i>Exercises</i>	20
3. Plotting	21
3.1 <i>plot () and allied functions</i>	21
3.2 <i>Fine control – Parameter settings</i>	22
3.3 <i>Adding points, lines and text</i>	23
3.4 <i>Identification and Location on the Figure Region</i>	25
3.5 <i>Plots that show the distribution of data values</i>	26
3.6 <i>Other Useful Plotting Functions</i>	29
3.7 <i>Plotting Mathematical Symbols</i>	31
3.8 <i>Guidelines for Graphs</i>	31
3.9 <i>Exercises</i>	32
3.10 <i>References</i>	33
4. Lattice graphics, and coplot()	35
4.1 <i>Examples that Present Panels of Scatterplots – Using xypplot ()</i>	35
4.2 <i>Using coplot ()</i>	37
4.3 <i>Exercises</i>	37

5. Linear (Multiple Regression) Models and Analysis of Variance	39
5.1 <i>The Model Formula in Straight Line Regression</i>	39
5.2 <i>Regression Objects</i>	40
5.3 <i>Model Formulae, and the X Matrix</i>	41
5.4 <i>Multiple Linear Regression Models</i>	43
5.5 <i>Polynomial and Spline Regression</i>	45
5.6 <i>Using Factors in R Models</i>	48
5.7 <i>Multiple Lines – Different Regression Lines for Different Species</i>	51
5.8 <i>aov models (Analysis of Variance)</i>	52
5.9 <i>Exercises</i>	54
5.10 <i>References</i>	55
6. Multivariate and Tree-Based Methods	57
6.1 <i>Multivariate EDA, and Principal Components Analysis</i>	57
6.2 <i>Cluster Analysis</i>	58
6.3 <i>Discriminant Analysis</i>	58
6.4 <i>Decision Tree models (Tree-based models)</i>	60
6.5 <i>Exercises</i>	60
6.6 <i>References</i>	60
*7. R Data Structures	63
7.1 <i>Vectors</i>	63
7.2 <i>Missing Values</i>	63
7.3 <i>Data frames</i>	64
7.4 <i>Data Entry</i>	65
7.5 <i>Factors and Ordered Factors</i>	67
7.6 <i>Ordered Factors</i>	68
7.7 <i>Lists</i>	68
*7.8 <i>Matrices and Arrays</i>	69
7.9 <i>Different Types of Attachments</i>	70
7.10 <i>Exercises</i>	70
8. Useful Functions	73
8.1 <i>Confidence Intervals and Tests</i>	73
8.2 <i>Matching and Ordering</i>	73
8.3 <i>String Functions</i>	73
8.4 <i>Application of a Function to the Columns of an Array or Data Frame</i>	74
*8.5 <i>tapply()</i>	74
8.6 <i>Splitting Vectors and Data Frames Down into Lists – split()</i>	76
*8.7 <i>Merging Data Frames</i>	76
8.8 <i>Dates</i>	76

8.9 Exercises.....	77
9. Writing Functions and other Code.....	79
9.1 Syntax and Semantics	79
9.2 Issues for the Writing and Use of Functions.....	80
9.3 Functions as aids to Data Management	81
9.4 A Simulation Example	81
9.5 Exercises.....	82
*10. GLM, and General Non-linear Models.....	85
10.1 A Taxonomy of Extensions to the Linear Model.....	85
10.2 Logistic Regression.....	86
10.3 glm models (Generalized Linear Regression Modelling).....	90
10.4 Models that Include Smooth Spline Terms	90
10.5 Non-linear Models.....	90
10.6 Model Summaries	90
10.7 Further Elaborations.....	91
10.8 Exercises.....	91
10.9 References.....	91
*11. Multi-level Models, Time Series and Survival Analysis	93
11.1 Multi-Level Models, Including Repeated Measures Models.....	93
11.2 Time Series Models.....	97
11.3 Survival Analysis	98
11.4 Exercises.....	98
11.5 References.....	98
*12. Advanced Programming Topics	99
12.1. Methods	99
12.2 Extracting Arguments to Functions.....	99
12.3 Parsing and Evaluation of Expressions.....	100
12.4 Plotting a mathematical expression.....	101
12.4 Searching R functions for a specified token.....	102
13. R Resources.....	103
13.1 R Packages for Windows	103
13.2 Literature written by expert users.....	103
13.3 The R-help electronic mail discussion list	104
13.4 Competing Systems – XLISP-STAT.....	104
14. Appendix 1.....	105
14.1 Data Sets Referred to in these Notes	105
14.2 Answers to Selected Exercises	105

Introduction

R implements a dialect of the S language that was developed at AT&T Bell Laboratories by Rick Becker, John Chambers and Allan Wilks. Versions of R are available, at no cost, for 32-bit 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). Web addresses are given below.

The citation for John Chambers' 1998 Association for Computing Machinery Software award stated that S has "forever altered how people analyze, visualize and manipulate data." The R project enlarges on the ideas and insights that generated the S language.

Here are points relating to the use of R that potential users might consider:

1. R has extensive and powerful graphics abilities that are tightly linked with its analytic abilities.
2. Although there is no official support for R, its informal support network, accessible from the r-help mailing list, can be highly effective.
3. Simple calculations and analyses can be handled straightforwardly, albeit (in the current version) using a command line interface. Chapters 1 and 2 are intended to give the flavour of what is possible without getting deeply into the R language. If simple methods prove inadequate, there can be recourse to the huge range of more advanced abilities that R offers. Adaptation of available abilities allows even greater flexibility.
4. The R community is widely drawn, from application area specialists as well as statistical specialists. It is a community that is sensitive to the potential for misuse of statistical techniques and suspicious of what might appear to be mindless use. Expect scepticism of the use of models that are not susceptible to some minimal form of data-based validation.
5. Because R is free, users have no right to expect attention, on the r-help list or elsewhere, to queries. Be grateful for whatever help is given.

There is no substitute for experience and expert knowledge, even when the statistical analysis task may seem straightforward. Neither R nor any other statistical system will give the statistical expertise that is needed to use sophisticated abilities, or to know when naïve methods are not enough. Experience with the use of R is however, more than with most systems, likely to be an educational experience.

While R is as reliable as any statistical software that is available, and exposed to higher standards of scrutiny than most other systems, there are traps that call for special care. Many of the model fitting routines in R are leading edge. There may be a limited tradition of experience of the limitations and potential pitfalls of some of the newer abilities. Whatever the statistical system, and especially when there is some element of complication, check each step with care.

Hurrah for the R development team!

The Use of these Notes

The notes are designed so that users can run the examples in the script files (*ch1-2.R*, *ch3-4.R*, etc.) using the notes as commentary. Under Windows alternatives are can either to type the commands in at the console, or to open a display file window and feed the commands in one at a time from the display file window. Section 1.2 gives details of these alternative ways to input commands to R.

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 a copy, or to download a distribution set that includes this document, go to

<http://cran.r-project.org>

and look for the nearest CRAN (Comprehensive R Archive Network) site.

Australian users may wish to go directly to the site:

<http://mirror.aarnet.edu.au/pub/CRAN>

The R Project

The initial version of R was developed by Ross Ihaka and Robert Gentleman, both from the University of Auckland. Development of R is now overseen by a ‘core team’ of about a dozen people, widely drawn from different institutions worldwide. The development model is similar to that of the increasingly popular Linux operating system.

Like Linux, R is an “open source” system. Source-code is available for inspection or for adaptation to other systems. In principle, if it is unclear what a routine does, one can check the source code. Exposing code to the critical scrutiny of highly expert users has proved an extremely effective way to identify bugs and other inadequacies, and to elicit ideas for enhancement. Reported bugs are commonly fixed in the next minor-minor release, which will usually appear within a matter of weeks.

A point and click interface is at an early stage of development. Users should be aware that R is developing rapidly. Substantial new features appear every few months. As of version 1.2, R has a “dynamic memory” model. Depending on available computer memory, the processing of a data set containing one hundred thousand observations and perhaps twenty variables may press the limits of what R can reasonably handle.

Novice users will notice small but occasionally important differences between the S dialect that R implements and the commercial S-PLUS implementation of S. Those who write their own substantial functions and (more importantly) libraries will find large differences. Libraries that have been written for R offer abilities that are broadly comparable with, or in some instances go beyond, those in S-PLUS libraries. These give access to up-to-date methodology from leading statistical researchers. R has strong graphics abilities. The recently released beta version of the *lattice* graphics library gives many of the abilities that are in the S-PLUS trellis library.

R is attractive as a language environment for the development of new scientific computational tools. Computer-intensive components can, if computational efficiency demands, be handled by a call to a function that is written in the C language.

The R-help mailing list is a useful source of advice and help. Be sure to check the available documentation before posting this list. Archives are available that can be searched for questions that may have been previously answered. The final chapter gives useful web addresses.

Jeff Wood (CMIS, CSIRO), Andreas Ruckstuhl (Technikum Winterthur Ingenieurschule, Switzerland) and John Braun (University of Western Ontario) gave me exemplary help in getting the earlier S-PLUS version of this document somewhere near shipshape form. John Braun gave valuable help with proofreading, and provided several of the data sets and a number of the exercises. I take full responsibility for the errors that remain. I am grateful, also, to the various scientists named in the notes who have allowed me to use their data.

1. Starting Up

R must be installed on your system! If it is not, follow the installation instructions appropriate to the operating system. Installation is now especially straightforward for Windows users. Copy down the latest *SetupR.exe* from the relevant *base* directory on the nearest CRAN site, click on its icon to start installation, and follow instructions. Libraries that do not come with the base distribution must be downloaded and installed separately.

It pays to have a separate workspace directory for each major project. For more details, see the README file that is included with the R distribution. Users of Microsoft Windows may wish to create a separate icon for each such workspace. First create the directory that will be used for the new workspace. Then right click|copy¹ to copy an existing R icon, it, right click|paste to place a copy on the desktop, right click|rename on the copy to rename it², and then finally go to right click|properties to set the **Start in** directory to be the workspace directory that was set up earlier.

1.1 Getting started under Windows

Click on the R icon. Or if there is more than one icon, choose the icon that corresponds to the project that is in hand. For this demonstration I will click on my *r-notes* icon.

In interactive use under Microsoft Windows there are several ways to input commands to R. Figures 1 and 2 demonstrate two of the possibilities. Either or both of the following may be used at the user's discretion:

For the moment, we will type commands into the *command window*, at the command line prompt. Fig. 1 shows the command window as it appears when R has just been started, for version 0.90.0. At the time of writing, the latest version is 1.3.0.

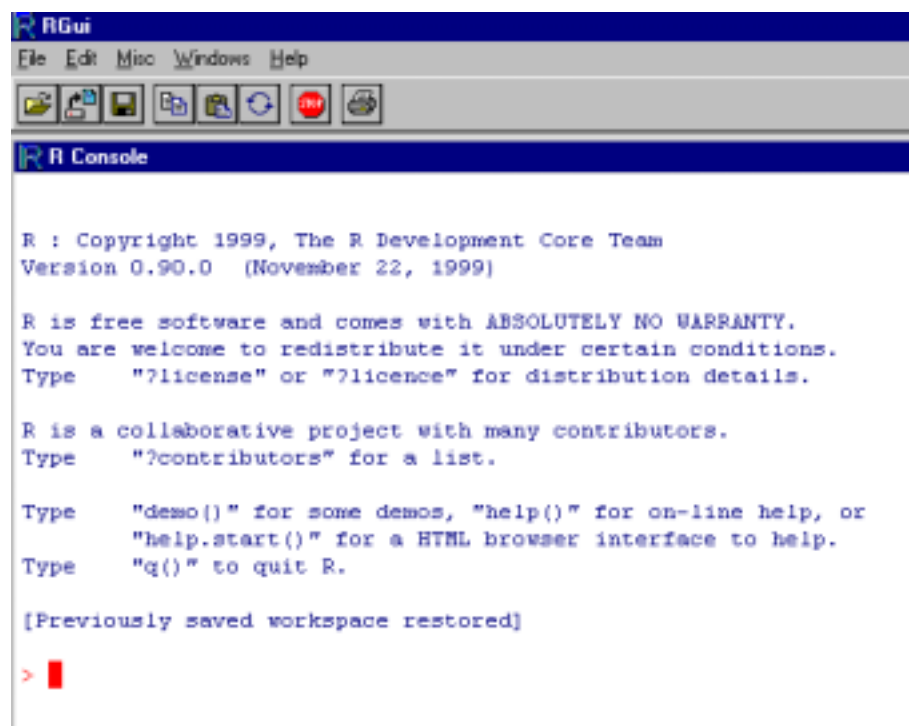


Fig. 1: The R console (command line) window.

¹ This is a shortcut for “right click, then left click on the copy menu item”.

² Enter the name of your choice into the name field. For ease of remembering, choose a name that closely matches the name of the workspace directory.

The screen snapshot in Fig.2 shows a *display file* window. This allows input to R of statements from a file that has been set up in advance. To get a display file window, go to the **File** menu. Then click on **Display File**. You will be asked for the name of a file whose contents are then displayed in the window. In Fig. 2 the file was `rcommands.txt`.

Highlight the commands that are intended for input to R. Click on the 'Paste to console' icon, on the far left of the display file toolbar in Figs. 2 and 3, to send these commands to R.

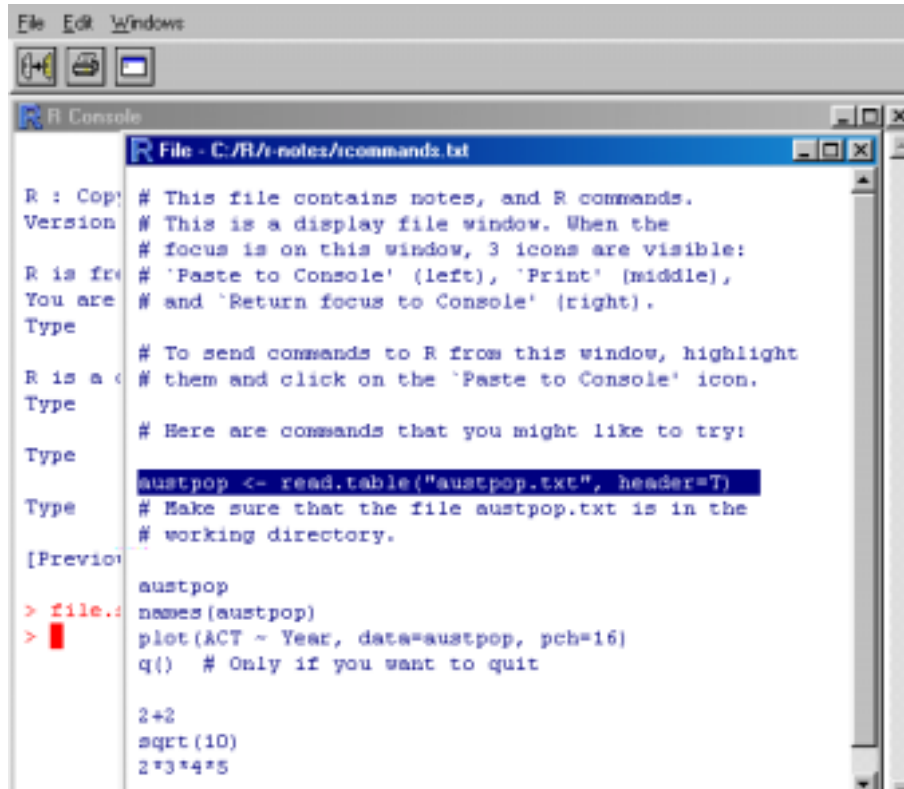


Fig. 2: The focus is on an R display file window, with the console window in the background.

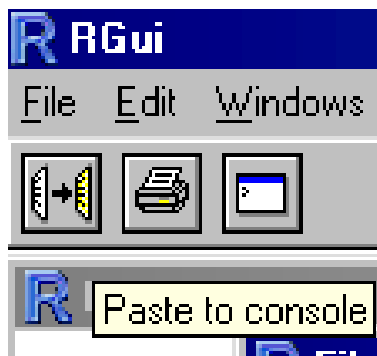


Fig. 3: The 'paste to console', 'print', and 'return focus to console' icons.

Under Unix, the standard form of input is the command line interface. Under both Microsoft Windows and Linux (or Unix), a further possibility is to run R from within the emacs editor³. This works much better under

³This requires both emacs and the emacs add-on call ESS. Both are free. look under Software|Other on the CRAN web page.

Linux/Unix than under Windows. Under Microsoft Windows, an attractive option is to use a utility that is designed for use with the shareware WinEdt editor⁴.

1.2 Using the Console (or Command Line) Window

Fig. 1 showed the console window when it was first opened.

The command line prompt, i.e. the `>`, is an invitation to start typing in your commands. For example, type in `2+2` and press the **Enter** key. Here is what I get on my screen:

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

Here 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()
```

Alternatives to the quit command are to click on the **File** menu and then **Exit**, or click on **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 all the objects that remain in the workspace – any that were there at the start of the session and any that have been added since.

1.3 A Short R Session

We will read into R a file that holds the population figures for Australian states and territories, and the total population, at various times since 1917. We will use information from this file to create a graph. Here is the information in the file:

Year	NSW	Vic.	Qld	SA	WA	Tas.	NT	ACT	Aust.
1917	1904	1409	683	440	306	193	5	3	4941
1927	2402	1727	873	565	392	211	4	8	6182
1937	2693	1853	993	589	457	233	6	11	6836
1947	2985	2055	1106	646	502	257	11	17	7579
1957	3625	2656	1413	873	688	326	21	38	9640
1967	4295	3274	1700	1110	879	375	62	103	11799
1977	5002	3837	2130	1286	1204	415	104	214	14192
1987	5617	4210	2675	1393	1496	449	158	265	16264
1997	6274	4605	3401	1480	1798	474	187	310	18532

To load all the necessary data sets into working memory (the R workspace), type the following:

```
> source("http://www.maths.anu.edu.au/~johnm/r/dsets/allldsets.R")
```

For your own future reference, syntax for loading a data set (eg., `data1.txt`) and saving it in working memory is

```
> data1 <- read.table("data1.txt", header = T)
```

The `<-` is a left diamond bracket (`<`) followed by a minus sign (`-`). It means “is assigned to”. Use of `header=T` causes R to use the first line to get header information for the columns. If column headings are not included in the file, the argument should be omitted.

Now type in `austpop` (one of the datasets you just loaded) at the command line prompt, displaying the object:

```
> austpop
  Year  NSW  Vi c.  Ql d  SA  WA  Tas.  NT  ACT  Aust.
1 1917 1904 1409  683  440  306  193   5   3  4941
2 1927 2402 1727  873  565  392  211   4   8  6182
. . .
```

⁴ The R-WinEdt utility, which is free, is a “plugin” for WinEdt. For links to the relevant web pages, for WinEdt and R-WinEdt, look under Software|Other on the CRAN web page.

We will now do a plot of the ACT population between 1917 and 1997. We will first of all remind ourselves of the column names:

```
> names(austpop)
[1] "Year" "NSW" "Vic." "Qld" "SA" "WA" "Tas." "NT"
[9] "ACT" "Aust."
```

A simple way to get the plot is:

```
> plot(ACT ~ Year, data=austpop, pch=16)
```

The option **pch=16** sets the plotting character to solid black dots. Fig. 4 shows the graph:

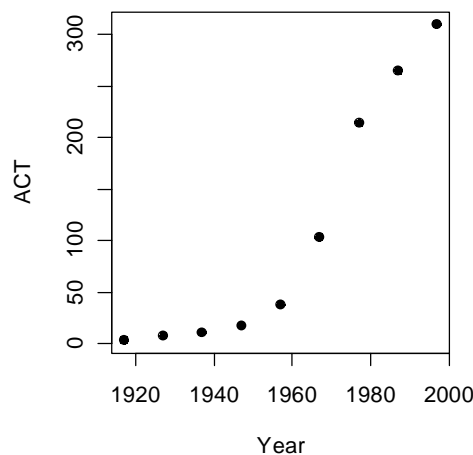


Figure 4: ACT population, at various times between 1917 and 1997.

This plot can be improved greatly. We can specify more informative axis labels, change size of text and plotting symbol, etc.

Try setting `pch` to different values, and appending `pch = 3, col = "red"` to the `plot` command above.

1.3.1 Entry of Data at the Command Line

A data frame is a rectangular array of columns of data. Here we will have two columns, and both columns will be numeric. The following data gives, for each amount by which an elastic band is stretched over the end of a ruler, the distance that the band moved when released:

Stretch (mm)	Distance (cm)
46	148
54	182
48	173
50	166
44	109
42	141
52	166

One can use `data.frame()` to input these (or other) data directly at the command line. We will give the data frame the name `elasticband`:

```
elasticband <- data.frame(stretch=c(46,54,48,50,44,42,52),
distance=c(148,182,173,166,109,141,166))
```

The `c()` command above means "concatenate"; i.e.: "Join these numbers together into a vector."

1.3.2 Options for use of `read.table()`

The function `read.table()` takes, optionally various parameters additional to the file name that holds the data. Specify `header=TRUE` if there is an initial row of header names. The default is `header=FALSE`. In addition users can specify the separator character or characters. Command alternatives to the default use of a

space are `sep=","` and `sep="\t"`. This last choice makes tabs separators. Similarly, users can control over the choice of missing value character or characters, which by default is **NA**. If the missing value character is a period (“.”), specify `na.strings="."`.

R has several variants of `read.table()` that differ only in having different default parameter settings. Note in particular `read.csv()`, which has settings that are suitable for comma delimited (csv) files that have been generated from Excel spreadsheets.

If `read.table()` detects that lines in the input file have different numbers of fields, data input will fail, with an error message that draws attention to the discrepancy. It is then often useful to use the function `count.fields()` to report the number of fields that were identified on each separate line of the file.

1.3.3 Options for plot() and allied functions

The function `plot()` and related functions accept parameters that control the plotting symbol, and the size and colour of the plotting symbol. Details will be given in section 3.3.

1.4 Further Notational Details

As noted earlier, the command line prompt is

```
>
```

R commands (expressions) are typed in following this prompt⁵.

There is also a continuation prompt, used when, following a carriage return, the command is still not complete. By default, the continuation prompt is

```
+
```

In these notes, we often continue commands over more than one line, but omit the + that will appear on the commands window if the command is typed in as we show it.

For the names of R objects or commands, case is significant. Thus **Austpop** is different from **austpop**. For file names however, the Microsoft Windows conventions apply, and case does not distinguish file names. On Unix systems letters that have a different case are treated as different.

Anything that follows a # on the command line is taken as comment and ignored by R.

Note: Recall that, in order to quit from the R session we had to type `q()`. This is because `q` is a function. Typing `q` on its own, without the parentheses, displays the text of the function on the screen. Try it!

1.5 On-line Help

To get a help window (under R for Windows) with a list of help topics, type:

```
> help()
```

In R for Windows, an alternative is to click on the help menu item, and then use key words to do a search. To get help on a specific R function, e.g. `plot()`, type in

```
> help(plot)
```

The two search functions `help.search()` and `apropos()` can be a huge help in finding what one wants. Examples of their use are:

```
> help.search("matrix")
```

This lists all functions whose help pages have a title or alias in which the text string “matrix” appears.

```
> apropos(matrix)
```

This lists all function names that include the text “matrix”.

Experimentation often helps clarify the precise action of an R function.

See Alexis Diamond's [handout on using R-help for guidance using the help pages](#).

⁵ Multiple commands may appear on the one line, with the semicolon (;) as the separator.

1.6 Exercise

1. In the data frame **el asti cband** from section 1.3.1, plot **di stance** against **stretch**.
2. The following ten observations, taken during the years 1970-79, are on October snow cover for Eurasia. (Snow cover is in millions of square kilometers):

```
year snow.cover
1970 6.5
1971 12.0
1972 14.9
1973 10.0
1974 10.7
1975 7.9
1976 21.9
1977 12.5
1978 14.5
1979 9.2
```

- i. Enter the data into R. [Section 1.3.1 showed one way to do this. To save keystrokes, enter the successive years as **1970: 1979**]
 - ii. Plot **snow.cover** versus **year**.
 - iii Use the **hi st()** command to plot a histogram of the snow cover values.
 - iv. Repeat ii and iii taking **natural** logarithms of snow cover. (Hint: entering **log(5)** gives the natural log of 5)
3. Input the following data, on damage that had occurred in space shuttle launches prior to the disastrous launch of Jan 28 1986. These are the data, for 6 launches out of 24, that were included in the pre-launch charts that were used in deciding whether to proceed with the launch. (Data for the 23 launches where information is available is in the data set **ori ngs** that accompanies these notes.)

Temperature (F)	Erosion incidents	Blowby incidents	Total incidents
53	3	2	5
57	1	0	1
63	1	0	1
70	1	0	1
70	1	0	1
75	0	2	1

Enter these data into a data frame, with (for example) column names **temperature**, **erosi on**, **bl owby** and **total** . (Refer back to Section 1.3.1). Plot total incidents against temperature.

2. An Overview of R

2.1 The Uses of R

2.1.1 R may be used as a calculator.

R evaluates and prints out the result of any expression that one types in at the command line in the console window. Expressions are typed following the prompt (>) on the screen. The result, if any, appears on subsequent lines

```
> 2+2
[1] 4
> sqrt(10)
[1] 3.162278
> 2*3*4*5
[1] 120
> 1000*(1+0.075)^5 - 1000 # Interest on $1000, compounded annually
[1] 435.6293
>                                     # at 7.5% p.a. for five years
> pi # R knows about pi
[1] 3.141593
> 2*pi*6378 #Circumference of Earth at Equator, in km; radius is 6378 km
[1] 40074.16
> sin(c(30,60,90)*pi/180) # Convert angles to radians, then take sin()
[1] 0.5000000 0.8660254 1.0000000
```

2.1.2 R will provide numerical or graphical summaries of data

A special class of object, called a *data frame*, stores rectangular arrays in which the columns may be vectors of numbers or factors or text strings. Data frames are central to the way that all the more recent R routines process data. For now, think of data frames as matrices, where the rows are observations and the columns are variables.

The data frame below was loaded into the workspace in Ch. 1 using the `source` command.

As a first example, consider the data frame `hills` that accompanies these notes⁶. This has three columns (variables), with the names `distance`, `climb`, and `time`. Typing in `summary(hills)` gives summary information on these variables. There is one column for each variable, thus:

```
> summary(hills)
  distance      climb      time
Min. : 2.000    Min. : 300    Min. : 15.95
1st Qu.: 4.500  1st Qu.: 725    1st Qu.: 28.00
Median: 6.000    Median: 1000   Median: 39.75
Mean: 7.529      Mean: 1815     Mean: 57.88
3rd Qu.: 8.000  3rd Qu.: 2200   3rd Qu.: 68.62
Max. : 28.000   Max. : 7500    Max. : 204.60
```

We may for example require information on ranges of variables. Thus the range of distances (first column) is from 2 miles to 28 miles, while the range of times (third column) is from 15.95 (minutes) to 204.6 minutes.

We will discuss graphical summaries in the next section.

⁶ There is also a version in the Venables and Ripley MASS library.

2.1.3 R has extensive graphical abilities

The main R graphics function is `plot()`. In addition to `plot()` there are functions for adding points and lines to existing graphs, for placing text at specified positions, for specifying tick marks and tick labels, for labelling axes, and so on.

There are various other alternative helpful forms of graphical summary. A helpful graphical summary for the `hills` data frame is the scatterplot matrix, shown in Fig. 5. For this, type:

```
> pairs(hills)
```

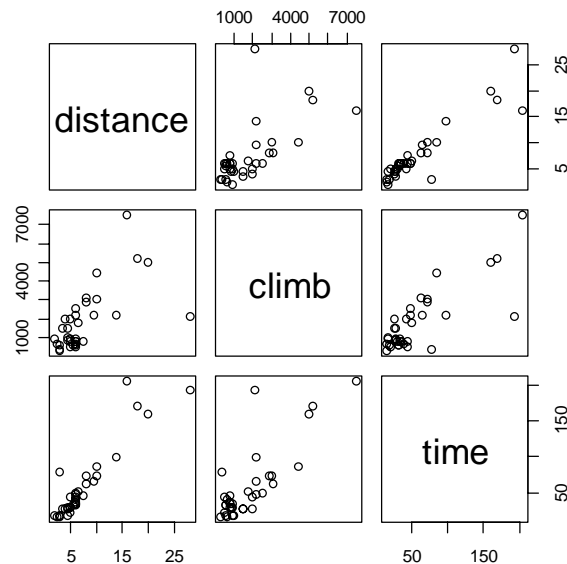


Figure 5: Scatterplot matrix for the Scottish hill race data

2.1.4 R will handle a variety of specific analyses

The examples that will be given are correlation and regression.

If you are not familiar with these topics, don't worry about it. All of this will be taught in GOV 1000.

Correlation:

We calculate the correlation matrix for the `hills` data:

```
> options(digits=3)
> cor(hills)
      distance climb time
distance  1.000 0.652 0.920
climb     0.652 1.000 0.805
time      0.920 0.805 1.000
```

Suppose we wish to calculate logarithms, and then calculate correlations. We can do all this in one step, thus:

```
> cor(log(hills))
      distance climb time
distance  1.00 0.700 0.890
climb     0.70 1.000 0.724
time      0.89 0.724 1.000
```

Unfortunately R was not clever enough to relabel distance as `log(distance)`, climb as `log(climb)`, and time as `log(time)`. Notice that the correlations between time and distance, and between time and climb, have reduced. Why has this happened?

Straight Line Regression:

Here is a straight line regression calculation. One specifies an `lm` (= linear model) expression, which R evaluates. The data are stored in the data frame `elasticband` that accompanies these notes. The variable names are the names of columns in that data frame. The command asks for the regression of distance travelled by the elastic band (distance) on the amount by which it is stretched (stretch).

```
> plot(distance ~ stretch, data=elasticband, pch=16)
> elastic.lm <- lm(distance~stretch, data=elasticband)
> lm(distance ~stretch, data=elasticband)
```

Call:

```
lm(formula = distance ~ stretch, data = elasticband)
```

Coefficients:

(Intercept)	stretch
-63.571	4.554

More complete information is available by typing

```
> summary(lm(distance~stretch, data=elasticband))
```

Try it! Also, to draw the regression line (a best-fit line), before closing the graphics window try typing

```
> abline(elastic.lm, col = "blue", lwd = 2)
```

The `lwd` parameter sets the line width. Type `?abline` for more info.

2.1.5 R is an Interactive Programming Language

We calculate the Fahrenheit temperatures that correspond to Celsius temperatures 25, 26, ..., 30:

```
> celsius <- 25:30
> fahrenheit <- 9/5*celsius+32
> conversion <- data.frame(Celsius=celsius, Fahrenheit=fahrenheit)
> print(conversion)
  Celsius Fahrenheit
1      25        77.0
2      26        78.8
3      27        80.6
4      28        82.4
5      29        84.2
6      30        86.0
```

We could also have used a loop. In general it is preferable to avoid loops whenever, as here, there is a good alternative. Loops may involve severe computational overheads.

2.2 The Look and Feel of R

R is a functional language. There is a language core that uses standard forms of algebraic notation, allowing the calculations described in Section 2.1.1. Beyond this, most computation is handled using functions. Even the action of quitting from an R session uses, as noted earlier, the function call `q()`.

It is often possible and desirable to operate on objects – vectors, arrays, lists and so on – as a whole. This largely avoids the need for explicit loops, leading to clearer code. Section 2.1.5 above gave an example.

The structure of an R program looks very like the structure of the widely used general purpose language C and its successors C++ and Java⁷.

⁷ Note however that R has no header files, most declarations are implicit, there are no pointers, and vectors of text strings can be defined and manipulated directly. The implementation of R relies heavily on list processing ideas from the LISP language. Lists are a key part of R syntax.

2.3 R Objects

All R entities, including functions and data structures, exist as objects. They can all be operated on as data. Type in `ls()` to see the names of all objects in your workspace. An alternative to `ls()` is `objects()`. In both cases there is provision to specify a particular pattern, e.g. starting with the letter 'p'⁸.

Typing the name of an object causes the printing of its contents. Try typing `q`, `mean`, etc.

Important: On quitting, R offers the option of saving the workspace image. This allows the retention, for use in the next session in the same workspace, any objects that were created in the current session. Careful housekeeping may be needed to distinguish between objects that are to be kept and objects that will not be used again. Before typing `q()` to quit, use `rm()` to remove objects that are no longer required. Saving the workspace image will then save everything remains. The workspace image will be automatically loaded upon starting another session in that directory.

*⁹2.4 Looping

In R there is often a better alternative to writing an explicit loop. Where possible, use one of the built-in functions to avoid explicit looping. A simple example of a `for` loop is¹⁰

```
for (i in 1:10) print(i)
```

Here is another example of a `for` loop, to do in a complicated way what we did very simply in section 2.1.5:

```
> # Celsius to Fahrenheit
> for (celsius in 25:30)
+   print(c(celsius, 9/5*celsius + 32))
[1] 25 77
[1] 26.0 78.8
[1] 27.0 80.6
[1] 28.0 82.4
[1] 29.0 84.2
[1] 30 86
```

2.4.1 More on looping

Here is a long-winded way to sum the three numbers 31, 51 and 91:

```
> answer <- 0
> for (j in c(31, 51, 91)){answer <- j + answer}
> answer
[1] 173
```

The calculation iteratively builds up the object `answer`, using the successive values of `j` listed in the vector (31,51,91). i.e. Initially, `j` =31, and `answer` is assigned the value 31 + 0 = 31. Then `j` =51, and `answer` is assigned the value 51 + 31 = 82. Finally, `j` =91, and `answer` is assigned the value 91 + 81 = 173. Then the procedure ends, and the contents of `answer` can be examined by typing in `answer` and pressing the **Enter** key.

⁸ Type in `help(ls)` and `help(grep)` to get details. The pattern matching conventions are those used for `grep()`, which is modelled on the Unix `grep` command.

⁹ Asterisks (*) identify sections that are more technical and might be omitted at a first reading

¹⁰ Other looping constructs are:

```
repeat <expression> ## break must appear somewhere inside the loop
while (x>0) <expression>
```

Here `<expression>` is an R statement, or a sequence of statements that are enclosed within braces

There is a much easier (and better) way to do this calculation:

```
> sum(c(31, 51, 91))
[1] 173
```

Skilled R users have limited recourse to loops. There are often, as in the example above, better alternatives.

2.5 R Functions

We give two simple examples of R functions.

2.5.1 An Approximate Miles to Kilometers Conversion

```
mi | es. to. km <- functi on(mi | es)mi | es*8/5
```

The return value is the value of the final (and in this instance only) expression that appears in the function body¹¹. Use the function thus

```
> mi | es. to. km(175) # Approximate distance to Sydney, in miles
[1] 280
```

The function will do the conversion for several distances all at once. To convert a vector of the three distances 100, 200 and 300 miles to distances in kilometers, specify:

```
> mi | es. to. km(c(100, 200, 300))
[1] 160 320 480
```

2.5.2 A Plotting function

The data set **fl ori da** has the votes in the 2000 election for the various US Presidential candidates, county by county in the state of Florida. The following plots the vote for Buchanan against the vote for Bush.

```
attach(fl ori da)
pl ot(BUSH, BUCHANAN, xl ab="Bush", yl ab="Buchanan")
detach(fl ori da) # In S-PLUS, speci fy detach(" fl ori da")
```

Here is a function that makes it possible to plot the figures for any pair of candidates.

```
pl ot. fl ori da <- functi on(xvar="BUSH", yvar="BUCHANAN"){
  x <- fl ori da[, xvar]
  y<- fl ori da[, yvar]
  pl ot(x, y, xl ab=xvar, yl ab=yvar)
  mtext(si de=3, l i ne=1.75,
    "Votes in Florida, by county, in \nthe 2000 US Presi denti al electi on")
}
```

The line `x <- florida[,xvar]` extracts all data in the "xvar" column in the florida data set, and saves this (a vector) as "x". "BUSH" is the default value of "xvar", but this can be customized (see the example below).

Note that the function body is enclosed in braces (`{ }`).

As well as `pl ot. fl ori da()`, this allows, e.g.

```
pl ot. fl ori da(yvar="NADER") # yvar="NADER" over-ri des the default
pl ot. fl ori da(xvar="GORE", yvar="NADER")
```

Fig. 6 shows the graph produced by `pl ot. fl ori da()`, i.e. parameter settings are left at their defaults.

¹¹ Alternatively a return value may be given using an explicit `return()` statement. This is however an uncommon construction

Votes in Florida, by county, in the 2000 US Presidential election

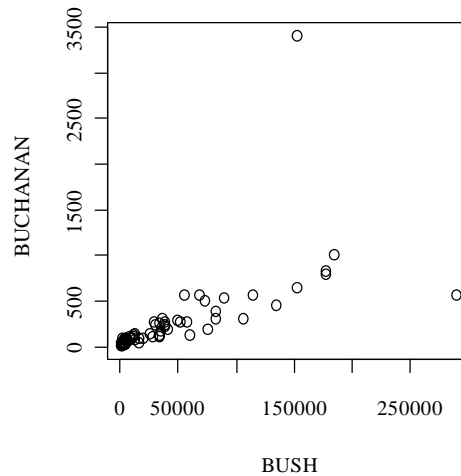


Figure 6: Election night count of votes received, by county, in the US 2000 Presidential election.

2.6 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("Canberra", "Sydney", "Newcastle", "Darwin")
```

Vectors may have mode logical, numeric or character¹². 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 missing value symbol, which is **NA**, can be included as an element of a vector.

2.6.1 Joining (concatenating) vectors

The **c** in **c(2, 3, 5, 7, 1)** above is an acronym for “concatenate”, i.e. the meaning is: “Join these numbers together into a vector. Existing vectors may be included among the elements that are to be concatenated. In the following we form vectors **x** and **y**, which we then concatenate to form a vector **z**:

```
> x <- c(2, 3, 5, 2, 7, 1)
> x
[1] 2 3 5 2 7 1
> y <- c(10, 15, 12)
> y
[1] 10 15 12

> z <- c(x, y)
> z
[1] 2 3 5 2 7 1 10 15 12
```

¹² Below, we will meet the notion of “class”, which is important for some of the more sophisticated language features of S-PLUS. The logical, numeric and character vectors just given have class **NULL**, i.e. they have no class. There are special types of numeric vector which do have a class attribute. Factors (see section 2.6.3) are an most important example.

The concatenate function `c()` may also be used to join lists.

2.6.2 Subsets of Vectors

There are two common ways to extract subsets of vectors¹³.

1. Specify the numbers 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
```

One can use negative numbers 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 `T`. Thus suppose we want to extract values of `x` that are greater than 10.

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

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

2.6.3 The Use of NA in Vector Subscripts

Note that any arithmetic operation or relation that involves `NA` generates an `NA`. Set

```
y <- c(1, NA, 3, 0, NA)
```

Be warned that `y[y==NA] <- 0` leaves `y` unchanged. The reason is that all elements of `y==NA` evaluate to `NA`. This does not select an element of `y`, and there is no assignment.

To replace all `NA`s by 0, use

```
y[is.na(y)] <- 0
```

2.6.4 Factors

A factor is a special type of vector, stored internally as a numeric vector with values 1, 2, 3, k . The value k is the number of levels. An attributes table gives the 'level' for each integer value¹⁴. Factors provide a compact way to store character strings. They are crucial in the representation of categorical effects in model and graphics formulae. The class attribute of a factor has, not surprisingly, the value "factor".

Consider a survey that has data on 691 females and 692 males. If the first 691 are females and the next 692 males, we can create a vector of strings that that holds the values thus:

¹³ A third more subtle method is available when vectors have named elements. One can then use a vector of names to extract the elements, thus:

```
> c(Andreas=178, John=185, Jeff=183)[c("John", "Jeff")]
John Jeff
185 183
```

¹⁴ The `attributes()` function makes it possible to inspect attributes. For example

```
attributes(factor(1:3))
```

The function `levels()` gives a better way to inspect factor levels.

```
gender <- c(rep("female", 691), rep("male", 692))
```

(The usage is that `rep("female", 691)` creates 691 copies of the character string "female", and similarly for the creation of 692 copies of "male".)

We can change the vector to a factor, by entering:

```
gender <- factor(gender)
```

Internally the factor `gender` is stored as 691 1's, followed by 692 2's. It has stored with it a table that looks like this:

1	female
2	male

Once stored as a factor, the space required for storage is reduced.

Whenever the context seems to demand a character string, the 1 is translated into "female" and the 2 into "male". The values "female" and "male" are the *levels* of the factor. By default, the levels are in alphanumeric order, so that "female" precedes "male". Hence:

```
> levels(gender) # Assumes gender is a factor, created as above
[1] "female" "male"
```

The order of the levels in a factor determines the order in which the levels appear in graphs that use this information, and in tables. To cause "male" to come before "female", use

```
gender <- relevel(gender, ref="male")
```

An alternative is

```
gender <- factor(gender, levels=c("male", "female"))
```

This last syntax is available both when the factor is first created, or later when one wishes to change the order of levels in an existing factor. Incorrect spelling of the level names will generate an error message. Try

```
gender <- factor(c(rep("female", 691), rep("male", 692)))
table(gender)
gender <- factor(gender, levels=c("male", "female"))
table(gender)
gender <- factor(gender, levels=c("Male", "female"))
# Erroneous - "male" rows now hold missing values
table(gender)
rm(gender) # Remove gender
```

2.7 Data Frames

Data frames are fundamental to the use of the R modelling and graphics functions. A data frame is a generalisation of a matrix, in which different columns may have different modes. All elements of any column must however have the same mode, i.e. all numeric or all factor, or all character.

Among the data sets that are supplied to accompany these notes is one called `Cars93.summary`, created from information in the `Cars93` data set in the Venables and Ripley *mass* library. **It was sourced into the workspace in Ch 1.**

```
> Cars93.summary
      Mi n. passengers Max. passengers No. of. cars abbrev
Compact           4             6           16         C
Large             6             6           11         L
Midsize           4             6           22         M
Small            4             5           21         Sm
Sporty           2             4           14         Sp
Van              7             8            9         V
```

The data frame has row labels (access with `row.names(Cars93.summary)`) Compact, Large, . . . The column names (access with `names(Cars93.summary)`) are `Min. passengers` (i.e. the minimum number

of passengers for cars in this category), **Max. passengers**, **No. of cars.**, and **abbrev**. The first three columns have mode numeric, and the fourth has mode character. Columns can be vectors of any mode. The column **abbrev** could equally well be stored as a factor.

Any of the following¹⁵ will pick out the fourth column of the data frame **Cars93.summary**, then storing it in the vector **type**.

```
type <- Cars93.summary$abbrev
type <- Cars93.summary[, 4]
type <- Cars93.summary[, "abbrev"]
type <- Cars93.summary[[4]] # Take the object that is stored
# In the fourth list element.
```

2.7.1 Data frames as lists

A data frame is a list¹⁶ of column vectors, all of equal length. Just as with any other list, subscripting extracts a list. Thus **Cars93.summary[4]** is a data frame with a single column, which is the fourth column vector of **Cars93.summary**. As noted above, use **Cars93.summary[[4]]** or **Cars93.summary[, 4]** to extract the column vector.

The use of matrix-like subscripting, e.g. **Cars93.summary[, 4]** or **Cars93.summary[1, 4]**, takes advantage of the rectangular structure of data frames.

2.7.2 Inclusion of character string vectors in data frames

When data are read in using **read.table()**, or when the **data.frame()** function is used to create data frames, vectors of character strings are by default turned into factors. Often this is convenient. If not, the parameter setting **as.is=T** will prevent this behaviour, both with **read.table()** and with **data.frame()**.

2.7.3 Built-in data sets

We will often use data sets that accompany one of the R libraries, usually stored as data frames. One such data frame, in the *base* library, is **trees**, which gives girth, height and volume for 31 Black Cherry Trees. This data frame was loaded into the workspace in Chapter 1, using the **source** command.

Here is summary information on this data frame

```
> summary(trees)
      Girth      Height      Volume
Min.   : 8.30  Min.   : 63  Min.   : 10.20
1st Qu.: 11.05 1st Qu.: 72  1st Qu.: 19.40
Median : 12.90 Median : 76  Median : 24.20
Mean   : 13.25 Mean   : 76  Mean   : 30.17
3rd Qu.: 15.25 3rd Qu.: 80  3rd Qu.: 37.30
Max.   : 20.60 Max.   : 87  Max.   : 77.00
```

Type **data()** to get a list of built-in data sets in the libraries that have been loaded¹⁷.

¹⁵ Also legal is **Cars93.summary[2]**. This gives a data frame with the single column **Type**.

¹⁶ In general forms of list, elements that are of arbitrary type. They may be any mixture of scalars, vectors, functions, etc.

¹⁷ The list include all libraries that are in the current environment.

2.8 Common Useful Functions

```
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
sort()     # Sort elements into order, but omitting NAs
order()    # x[order(x)] orders elements of x, with NAs last
```

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

By default, `sort()` omits any NAs. The function `order()` places NAs last. Hence:

```
> x <- c(1, 20, 2, NA, 22)
> order(x)
[1] 1 3 2 5 4
> x[order(x)]
[1] 1 2 20 22 NA
> sort(x)
[1] 1 2 20 22
```

2.8.1 Applying a function to all columns of a data frame

The function `sapply()` does this. It takes as arguments the name of the data frame, and the function that is to be applied. Here are examples, using the supplied data set `rainforest`¹⁸.

```
> sapply(rainforest, is.factor)
  dbh  wood  bark  root rootsk branch species
FALSE FALSE FALSE FALSE  FALSE  FALSE  TRUE
> sapply(rainforest[, -7], range) # The final column (7) is a factor
  dbh wood bark root rootsk branch
[1,] 4  NA  NA  NA  NA  NA
[2,] 56 NA  NA  NA  NA  NA
```

The `rainforest[, -7]` component deletes the 7th column of the data frame, leaving all others unchanged.

In its entirety, the command line says: "Form a subset of the rainforest data frame that includes all but the 7th column of the rainforest data set, and then output the range of each remaining column vector."

Notice how any vector with an NA automatically outputs a range of {NA, NA}. Any mathematical operation performed on NA is equal to NA. Try typing `sum(c(2, NA, 0.5))` and compare to `sum(c(2, 5))`.

The functions `mean` and `range`, and several of the other functions noted above, have parameters `na.rm`. For example

```
> range(rainforest$branch, na.rm=T) # Omit NAs, then determine the range
[1] 4 120
```

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 specified in the second argument position. For example:

¹⁸ Source: Ash, J. and Southern, W. 1982: Forest biomass at Butler's Creek, Edith & Joy London Foundation, New South Wales, Unpublished manuscript. See also Ash, J. and Helman, C. 1990: Floristics and vegetation biomass of a forest catchment, Kioloa, south coastal N.S.W. *Cunninghamia*, 2(2): 167-182.

```
> sapply(rainforest[, -7], range, na.rm=T)
      dbh wood bark root rootsk branch
[1, ]  4    3    8    2    0.3    4
[2, ] 56 1530 105 135 24.0 120
```

Chapter 8 has further details on the use of `sapply()`. There is an example that shows how to use it to count the number of missing values in each column of data.

2.9 Making Tables

`table()` makes a table of counts. Specify one vector of values (often a factor) for each table margin that is required. Here are some examples

```
> table(rainforest$species) # rainforest is a supplied data set

Acacia mabellae      C. fraseri  Acmena smithii    B. myrtifolia
              16              12              26              11

> table(Barley$Year, Barley$Site)
      C D GR M UF W
1931 5 5 5 5 5 5
1932 5 5 5 5 5 5
```

WARNING: NAs are by default ignored. 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)`

```
> x_c(1, 5, NA, 8)
> x <- factor(x)
> x
[1] 1 5 NA 8
Levels: 1 5 8
> factor(x, exclude=NULL)
[1] 1 5 NA 8
Levels: 1 5 8 NA
```

2.9.1 Numbers of NAs in subgroups of the data

The following gives information on the number of NAs in subgroups of the data:

```
> table(rainforest$species, !is.na(rainforest$branch))

              FALSE TRUE
Acacia mabellae      6  10
C. fraseri           0  12
Acmena smithii     15  11
B. myrtifolia       1  10
```

Thus for *Acacia mabellae* there are 6 NAs for the variable `branch` (i.e. number of branches over 2cm in diameter), out of a total of 16 data values.

2.10 The R Directory Structure

R has a search list where it looks for objects. This can be changed in the course of a session. To get a full list of these directories, called *databases*, type:

```
> search() # This is for version 1.2.3 for Windows
```

```
[1] ". Global Env" "Autoloads" "package: base"
```

At this point, just after startup, the search list consists of the workspace ("**G**lobal Env"), a slightly mysterious database with the name Autoloads, and the *base* package or library. Addition of further libraries (also called packages) extends this list. For example:

```
> library(ts) # Time series library, included with the distribution
> search()
[1] ". Global Env" "package: ts" "Autoloads" "package: base"
```

2.11 More Detailed Information

This chapter has given the minimum detail that seems necessary for getting started. Look in chapters 7 and 8 for a more detailed coverage of the topics in this chapter. It may pay, at this point, to glance through chapters 7 and 8 to see what is there. Remember also to use the R help.

Topics from chapter 7, additional to those covered above, that may be important for relatively elementary uses of R include:

- The entry of patterned data (7.1.3)
- The handling of missing values in subscripts when vectors are assigned (7.2)
- Unexpected consequences (e.g. conversion of columns of numeric data into factors) from errors in data (7.4.1).

2.11 Exercises

1. For each of the following code sequences, predict the result. Then do the computation:

a) `answer <- 0`
`for (j in 3:5){ answer <- j+answer }`

b) `answer <- 10`
`for (j in 3:5){ answer <- j+answer }`

c) `answer <- 10`
`for (j in 3:5){ answer <- j*answer }`

2. Look up the help for the function `prod()`, and use `prod()` to do the calculation in 1(c) above. Alternatively, how would you expect `prod()` to work? Try it!

3. Add up all the numbers from 1 to 100 in two different ways: using `for` and using `sum`. Now apply the function to the sequence 1:100. What is its action?

4. Multiply all the numbers from 1 to 50 in two different ways: using `for` and using `prod`.

5. The volume of a sphere of radius r is given by $4\pi r^3/3$. For spheres having radii 3, 4, 5, ..., 20 find the corresponding volumes and print the results out in a table. Use the technique of section 2.1.5 to construct a data frame with columns `radius` and `volume`.

6. Use `sapply()` to apply the function `is.factor` to each column of the supplied data frame `tinting`. For each of the columns that are identified as factors, determine the levels. Which columns are ordered factors? [Use `is.ordered()`].

3. Plotting

The functions `plot()`, `points()`, `lines()`, `text()`, `mtext()`, `axis()`, `identify()` etc. form a suite that plots points, lines and text. To see some of the possibilities that R offers, enter

```
demo(graphics)
```

Press the Enter key to move to each new graph.

3.1 plot () and allied functions

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

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

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))
```

Comment on the appearance that these graphs present. Is it obvious that these points lie on a sine curve? How can one make it obvious? (Place the cursor over the lower border of the graph sheet, until it becomes a double-sided arrow. Drag the border in towards the top border, making the graph sheet short and wide.)

Here are two further examples.

```
attach(elasticband) # R now knows where to find distance & stretch
plot(distance ~ stretch)
plot(ACT ~ Year, data=austpop, type="l")
plot(ACT ~ Year, data=austpop, type="b")
```

The `points()` function adds points to a plot. The `lines()` function adds lines to a plot¹⁹. 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

```
attach(austpop)
plot(spline(Year, ACT), type="l") # Fit smooth curve through points
detach(austpop) # In S-PLUS, specify detach("austpop")
```

3.1.1 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, plotting a data frame gives, for each numeric variable, a normal probability plot. Plotting the `lm` object that is created by the use of the `lm()` modelling function gives diagnostic and other information that is intended to help in the interpretation of regression results.

Try

```
plot(hills) # Has the same effect as pairs(hills)
```

¹⁹ 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.

3.2 Fine control – Parameter settings

The default settings of parameters, such as character size, are often adequate. When it is necessary to change parameter settings for a subsequent plot, the `par()` function does this. For example,

```
par(cex=1.25, mex=1.25) # character (cex) & margin (mex) expansion
```

increases the text and plot symbol size 25% above the default. The addition of `mex=1.25` makes room in the margin to accommodate the increased text size.

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

```
oldpar <- par(cex=1.25, mex=1.25)
```

This stores the existing settings in `oldpar`, then changes parameters (here `cex` and `mex`) as requested. To restore the original parameter settings at some later time, enter `par(oldpar)`. Here is an example:

```
attach(elsa)
oldpar <- par(cex=1.5, mex=1.5)
plot(distance ~ stretch)
par(oldpar) # Restores the earlier settings
detach(elsa)
```

Inside a function specify, e.g.

```
oldpar <- par(cex=1.25, mex=1.25)
on.exit(par(oldpar))
```

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

3.2.1 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. In the example below we present four different transformations of the primates data, in a two by two layout:

```
par(mfrow=c(2,2), pch=16)
data(Animals) # Needed if Animals (MASS library) is not already loaded
attach(Animals)
plot(body, brain)
plot(sqrt(body), sqrt(brain))
plot((body)^0.1, (brain)^0.1)
plot(log(body), log(brain))
detach(Animals)
par(mfrow=c(1,1), pch=1) # Restore to 1 figure per page
```

3.2.2 The shape of the graph sheet

Often it is desirable to exercise control over the 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.

3.3 Adding points, lines and text

Here is a simple example that shows how to use the function `text()` to add text labels to the points on a plot.

```
> primates
      Bodywt Brai nwt
Potar monkey  10.0   115
  Gorilla    207.0   406
   Human     62.0  1320
Rhesus monkey   6.8   179
   Chim p    52.2   440
```

Observe that the row names store labels for each row²⁰.

```
> attach(primates) # Needed if primates is not already attached.
> plot(Bodywt, Brai nwt, xlim=c(5, 250))
> # Specify xlim so that there is room for the labels
> text(x=Bodywt, y=Brai nwt, labels=row.names(primates), adj=0)
  # adj=0 implies left adjusted text
> detach(primates)
```

Fig. 7 shows the result.

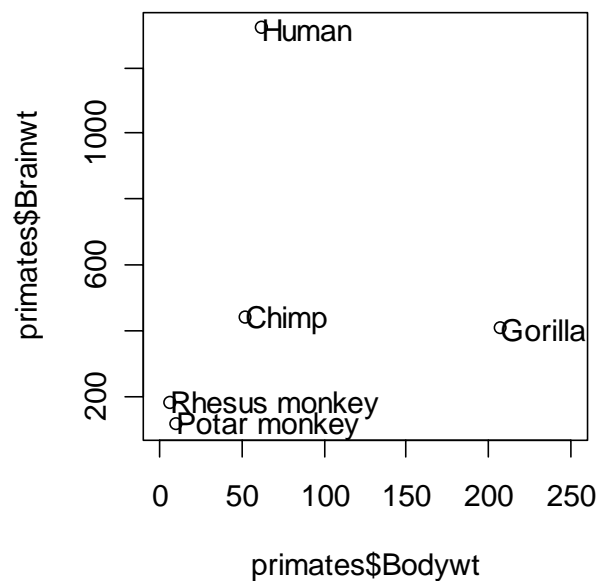


Figure 7: Plot of the primate data, with labels on points

Fig. 7 would be adequate for identifying points, but is not a presentation quality graph. We now show how to improve it.

²⁰ Row names can be created in several different ways. They can be assigned directly, e.g.

```
row.names(primates) <- c("Potar monkey", "Gorilla", "Human", "Rhesus monkey", "Chim p")
```

When using `read.table()` to input data, the parameter `row.names` is available to specify, by number or name, a column that holds the row names.

In Fig. 8 we use the **xl ab** (x-axis) and **yl ab** (y-axis) parameters to specify meaningful axis titles. We move the labelling to one side of the points by including appropriate horizontal and vertical offsets. We use **chw <- par()\$cxy[1]** to get a 1-character space horizontal offset, and **chh <- par()\$cxy[2]** to get a 1-character height vertical offset. I've used **pch=16** to make the plot character a heavy black dot. This helps make the points stand out against the labelling.

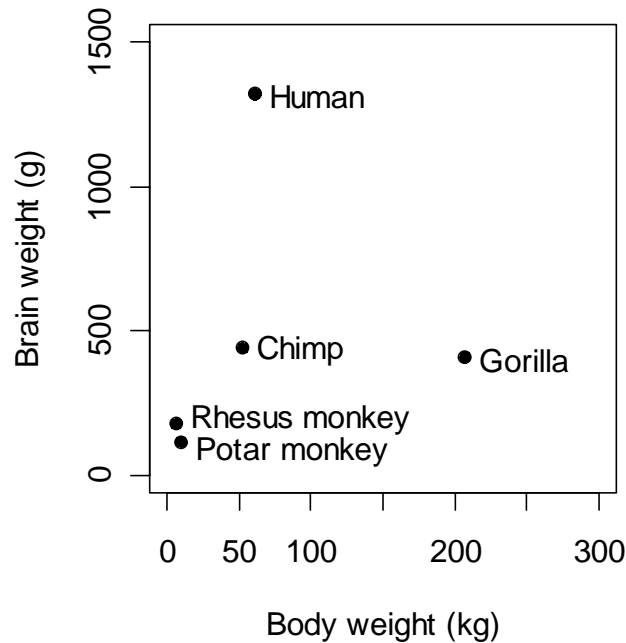


Figure 8: Improved version of Fig. 7.

Here is the R code for Fig. 8:

```
attach(primates)
plot(x=Bodywt, y=Brainwt, pch=16,
      xlab="Body weight (kg)", ylab="Brain weight (g)",
      xlim=c(5, 240), ylim=c(0, 1500))
chw <- par()$cxy[1]
chh <- par()$cxy[2]
text(x=Bodywt+chw, y=Brainwt,
      labels=row.names(primates), adj=0)
```

To place the text to the left of the points, specify

```
text(x=Bodywt- 0.75*chw, y=Brainwt,
      labels=row.names(primates), adj=1)
```

3.3.1 Size, colour and choice of plotting symbol

For **plot()** and **points()** the parameter **cex** (“character expansion”) controls the size, while **col** (“colour”) controls the colour of the plotting symbol. The parameter **pch** controls the choice of plotting symbol.

The parameters **cex** and **col** may be used in a similar way with **text()**. Try

```
plot(1, 1, xlim=c(1, 7.5), ylim=c(0,5), type="n") # Do not plot points
points(1:7, rep(4.5, 7), cex=1:7, col=1:7, pch=0:6)
text(1:7, rep(3.5, 7), labels=paste(0:6), cex=1:7, col=1:7)
```

The following, added to the plot that results from the above three statements, demonstrates other choices of pch.

```

points(1:7, rep(2, 7), pch=(0:6)+7)          # Plot symbols 7 to 13
text((1:7)+0.25, rep(2, 7), paste((0:6)+7)) # Label with symbol number
points(1:7, rep(1, 7), pch=(0:6)+14)        # Plot symbols 14 to 20
text((1:7)+0.25, rep(1, 7), paste((0:6)+14)) # Labels with symbol number

```

Here is the plot:

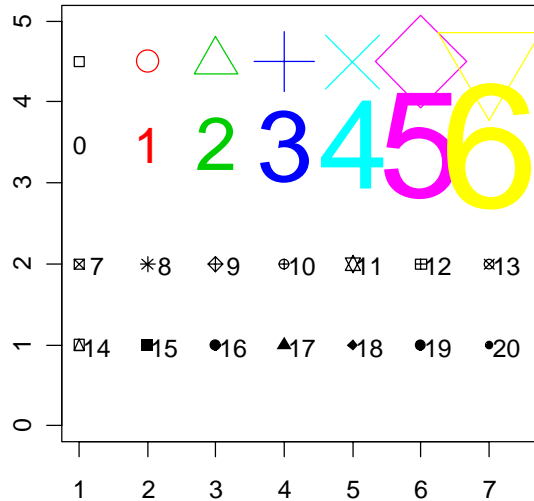


Figure 9: Different plot symbols, colours and sizes

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)
  rai nchars <- c("R", "O", "Y", "G", "B", "I", "V")
  text(1:7, rep(1.5, 7), rai nchars, 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()
```

3.3.2 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.

3.4 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.

- `identify()` labels points. One positions the cursor near the point that is to be identified, and clicks the left mouse button.

- `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`.

3.4.1 identify()

This function requires specification of a vector `x`, a vector `y`, and a vector of text strings that are available for use as labels. The data set `florida` has the votes for the various Presidential candidates, county by county in the state of Florida. We plot the vote for Buchanan against the vote for Bush, then invoking `identify()` so that we can label selected points on the plot.

```
attach(florida)
plot(BUSH, BUCHANAN, xlab="Bush", ylab="Buchanan")
identify(BUSH, BUCHANAN, County)
detach(florida)
```

Click to the left or right, and slightly above or below a point, depending on the preferred positioning of the label. When labelling is terminated (click with the right mouse button), the row numbers of the observations that have been labelled are printed on the screen, in order.

3.4.2 locator()

Left click at the locations whose coordinates are required

```
attach(florida) # if not already attached
plot(BUSH, BUCHANAN, xlab="Bush", ylab="Buchanan")
locator()
detach(florida)
```

The function can be used to mark new points (specify `type="p"`) or lines (specify `type="l"`) or both points and lines (specify `type="b"`).

This tutorial does not include some of Maindonald's material on graphing techniques. Interested students should feel encouraged to look up his original document and read more on their own.

3.5 Exercises

1. Plot the graph of brain weight (**brain**) versus body weight (**body**) for the data set **Animals** from the *MASS* library. Label the axes appropriately.

[To access this data frame, specify `library(mass); data(Animals)`]

2. Repeat the plot 1, but this time plotting $\log(\text{brain weight})$ versus $\log(\text{body weight})$. Use the row labels to label the points with the three largest body weight values. Label the axes in untransformed units.

3. Repeat the plots 1 and 2, but this time place the plots side by side on the one page.

4. The data set **huron** that accompanies these notes has mean July average water surface elevations, in feet, IGLD (1955) for Harbor Beach, Michigan, on Lake Huron, Station 5014, for 1860-1986²². (Alternatively you can work with the vector **LakeHuron** from the *ts* library, that has mean heights for 1875-1772 only.)

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 $i-1$.

²² Source: Great Lakes Water Levels, 1860-1986. U.S. Dept. of Commerce, National Oceanic and Atmospheric Administration, National Ocean Survey.

²³ Data relate to the paper: Lindenmayer, D. B., Viggers, K. L., Cunningham, R. B., and Donnelly, C. F. 1995. Morphological variation among populations of the mountain brush tail possum, *Trichosurus caninus* Ogilby (Phalangeridae: Marsupialia). Australian Journal of Zoology 43: 449-458.

*7. R Data Structures

7.1 Vectors

Recall that vectors may have mode logical, numeric or character³⁶.

7.1.1 Subsets of Vectors

Recall (section 2.6.2) two common ways to extract subsets of vectors:

1. Specify the numbers of the elements that are to be extracted. One can use negative numbers to omit elements.
2. Specify a vector of logical values. The elements that are extracted are those for which the logical value is T. Thus suppose we want to extract values of **x** that are greater than 10.

The following demonstrates a third possibility, for vectors that have named elements:

```
> c(Andreas=178, John=185, Jeff=183)[c("John", "Jeff")]
John Jeff
 185  183
```

A vector of names has been used to extract the elements.

7.1.2 Patterned Data

Use 5:15 to generate the numbers 5, 6, ..., 15. Entering 15:5 will generate the sequence in the reverse order.

To repeat the sequence (2, 3, 5) four times over, enter `rep(c(2, 3, 5), 4)` thus:

```
> rep(c(2, 3, 5), 4)
[1] 2 3 5 2 3 5 2 3 5 2 3 5
>
```

If instead one wants four 2s, then four 3s, then four 5s, enter `rep(c(2, 3, 5), c(4, 4, 4))`.

```
> rep(c(2, 3, 5), c(4, 4, 4)) # An alternative is rep(c(2, 3, 5), each=4)
[1] 2 2 2 2 3 3 3 3 5 5 5 5
```

Note further that, in place of `c(4, 4, 4)` we could write `rep(4, 3)`. So a further possibility is that in place of `rep(c(2, 3, 5), c(4, 4, 4))` we could enter `rep(c(2, 3, 5), rep(4, 3))`.

In addition to the above, note that the function `rep()` has an argument `length.out`, meaning “keep on repeating the sequence until the length is `length.out`.”

7.2 Missing Values

In R, the missing value symbol is **NA**. Any arithmetic operation or relation that involves **NA** generates an **NA**. This applies also to the relations `<`, `<=`, `>`, `>=`, `==`, `!=`. The first four compare magnitudes, `==` tests for equality, and `!=` tests for inequality. Unless you think carefully about the implications for working with expressions that include **NA**s, you may not get the results that you expect. Specifically, note that `x==NA` generates **NA**.

Be sure to use `is.na(x)` to test which values of **x** are **NA**. As `x==NA` gives a vector of **NA**s, you get no information at all about **x**. For example

```
> x <- c(1, 6, 2, NA)
```

³⁶ Below, we will meet the notion of “class”, which is important for some of the more sophisticated language features of R. The logical, numeric and character vectors just given have class `NULL`, i.e. they have no class. There are special types of numeric vector which do have a class attribute. Factors are the most important example. Although often used as a compact way to store character strings, factors are, technically, numeric vectors. The class attribute of a factor has, not surprisingly, the value “factor”.

```

> !is.na(x) # TRUE for when NA appears, and otherwise FALSE
[1] FALSE FALSE FALSE TRUE
> x==NA     # All elements are set to NA
[1] NA NA NA NA
> NA==NA
[1] NA

```

WARNING: This is chiefly for those who may move between R and S-PLUS. In important respects, R's behaviour with missing values is more intuitive than that of S-PLUS. Thus in R

```
y[x>2] <- x[x>2]
```

gives the result that the naïve user might expect, i.e. replace elements of **y** with corresponding elements of **x** wherever **x>2**. Wherever **x>2** gives the result **NA**, no action is taken. In R, any **NA** in **x>2** yields a value of **NA** for **y[x>2]** on the left of the equation, and a value of **NA** for **x[x>2]** on the right of the equation.

In S-PLUS, the result on the right is the same, i.e. an **NA**. However, on the left, elements that have a subscript **NA** drop out. The vector on the left to which values will be assigned has, as a result, fewer elements than the vector on the right.

Thus the following has the effect in R that the naïve user might expect, but not in S-PLUS:

```

x <- c(1, 6, 2, NA, 10)
y <- c(1, 4, 2, 3, 0)
y[x>2] <- x[x>2]
y

```

In S-PLUS it is essential to specify, in the example just considered:

```
y[!is.na(x)&x>2] <- x[!is.na(x)&x>2]
```

Here is a further example of R's behaviour:

```

> x <- c(1, 6, 2, NA, 10)
> x>2
[1] FALSE TRUE FALSE NA TRUE
> x[x>3] <- c(21, 22) # This does not give what the naïve user might expect
Warning message:
number of items to replace is not a multiple of replacement length
> x
[1] 1 21 2 NA 21

```

The safe way, in both S-PLUS and R, is to use **!is.na(x)** to limit the selection, on one or both sides as necessary, to those elements of **x** that are not **NA**s. We will have more to say on missing values in the section on data frames that now follows.

7.3 Data frames

The concept of a data frame is fundamental to the use of most of the R modelling and graphics functions. A data frame is a generalisation of a matrix, in which different columns may have different modes. All elements of any column must however have the same mode, i.e. all numeric or all factor, or all character.

Data frames where all columns hold numeric data have some, but not all, of the properties of matrices. There are important differences that arise because data frames are implemented as lists. To turn a data frame of numeric data into a matrix of numeric data, use **as.matrix()**.

Lists are discussed below, in section 7.6.

7.3.1 Extraction of Component Parts of Data frames

Consider the data frame `Barley`. A version is available with the data sets that are supplied to complement these notes. The data set `immer` that is bundled with the Venables and Ripley *MASS* library has the same data, but arranged differently. This data frame was loaded into the workspace in Chapter 1, using the `source` command.

```
> names(Barley)
[1] "Site" "Variety" "Year" "Yield"
> levels(Barley$Site)
[1] "C" "D" "GR" "M" "UF" "W"
> levels(Barley$Variety)
[1] "Manchuria" "Peatland" "Svansota" "Trebi" "Velvet"
```

Notice that the data frame has abbreviations for site names, while variety names are given in full.

We will extract the data for 1932, at the `D` site.

```
> Duluth1932 <- Barley[Barley$Year=="1932" & Barley$Site=="D",
+ c("Variety", "Yield")]
> Duluth1932
  Variety Yield
56 Manchuria 67.7
57 Svansota 66.7
58 Velvet 67.4
59 Trebi 91.8
60 Peatland 94.1
```

The first column holds the row labels, which in this case are the numbers of the rows that have been extracted. In place of `c("Variety", "Yield")` we could have written, more simply, `c(2, 4)`.

7.3.2 Data Sets that Accompany R Libraries

Type in `data()` to get a list of data sets (mostly data frames) associated with all libraries that are in the current search path. To get information on the data sets that are included in the base library, specify

```
data(package="base") # Here you must specify `package`, not `library`.
```

and similarly for any other library.

In order to bring any of these data frames into the working directory, specifically request it. (Ensure though that the relevant library is attached.) Thus to bring in the data set `airquality` from the base library, type in

```
data(airquality)
```

The default Windows distribution includes the libraries `BASE`, `EDA`, `STEPFUN` (empirical distributions), and `TS` (time series). Other libraries must be explicitly installed. For remaining sections of these notes, it will be useful to have the `MASS` library installed. The current Windows version is bundled in the file `VR61-6.zip`, which you can download from the directory of contributed packages at any of the CRAN sites.

The base library is automatically attached at the beginning of the session. To attach any other installed library, use the `library()` (or, equivalently `package()`) command.

7.4 Data Entry

The function `read.table()` offers a ready means to read a rectangular array into an R data frame. Suppose that the file `primates.dat` contains:

```
"Potar monkey" 10 115
Gori lla        207 406
Human          62 1320
"Rhesus monkey" 6.8 179
Chi mp         52.2 440
```

Then

```
primates <- read.table("a:/primates.txt")
```

will create the data frame `primates`, from a file on the `a:` drive. The text strings in the first column will become the first column in the data frame.

Suppose that `primates` is a data frame with three columns – species name, body weight, and brain weight. You can give the columns names by typing in:

```
names(primates) <- c("Species", "Bodywt", "Brainwt")
```

Here then are the contents of the data frame.

```
> primates
  Species Bodywt Brainwt
1 Potar monkey  10.0    115
2 Gorilla    207.0    406
3 Human      62.0   1320
4 Rhesus monkey  6.8    179
5 Chimpanzee 52.2    440
```

Specify `header=TRUE` if there is an initial row of header information. If the number of headers is one less than the number of columns of data, then the first column will be used, providing entries are unique, for row labels.

7.4.1 Idiosyncrasies

The function `read.table()` is straightforward for reading in rectangular arrays of data that are entirely numeric. When, as in the above example, one of the columns contains text strings, the column is by default stored as a factor with as many different levels as there are unique text strings³⁷.

Problems may arise when small mistakes in the data cause R to interpret a column of supposedly numeric data as character strings, which are automatically turned into factors. For example there may be an `O` (oh) somewhere where there should be a `0` (zero), or an `el` (`l`) where there should be a `one` (`1`). If you use any missing value symbols other than the default (`NA`), you need to make this explicit see section 7.3.2 below. Otherwise any appearance of such symbols as `*`, `period(.)` and `blank` (in a case where the separator is something other than a space) will cause to whole column to be treated as character data.

Users who find this default behaviour of `read.table()` confusing may wish to use the parameter setting `as.is = TRUE`.³⁸ If the column is later required for use as a factor in a model or graphics formula, it may be necessary to make it into a factor at that time. Some functions do this conversion automatically.

7.4.2 Missing values when using `read.table()`

The function `read.table()` expects missing values to be coded as `NA`, unless you set `na.strings` to recognise other characters as missing value indicators. If you have a text file that has been output from SAS, you will probably want to set `na.strings=c(" ")`.

There may be multiple missing value indicators, e.g. `na.strings=c("NA", ". ", " **", "")`. The `" "` will ensure that empty cells are entered as `NA`s.

7.4.3 Separators when using `read.table()`

With data from spreadsheets³⁹, it is sometimes necessary to use tab (`"\t"`) or comma as the separator. The default separator is white space. To set tab as the separator, specify `sep="\t"`.

³⁷ Storage of columns of character strings as factors is efficient when a small number of distinct strings are each repeated a large number of times.

³⁸ Specifying `as.is = T` prevents columns of (intended or unintended) character strings from being converted into factors.

³⁹ One way to get mixed text and numeric data across from Excel is to save the worksheet in a `.csv` text file with comma as the separator. If for example file name is `myfile.csv` and is on drive `a:`, use

7.5 Factors and Ordered Factors

We discussed factors in section 2.6.4. They provide an economical way to store vectors of character strings in which there are many multiple occurrences of the same strings. More crucially, they have a central role in the incorporation of qualitative effects into model and graphics formulae.

Factors have a dual identity. They are stored as integer vectors, with each of the values interpreted according to the information that is in the table of levels⁴⁰.

The data frame `isl andci ti es` that accompanies these notes holds the populations of the 19 island nation cities with a 1995 urban centre population of 1.4 million or more. The row names are the city names, the first column (**country**) has the name of the country, and the second column (**popul ati on**) has the urban centre population, in millions. Here is a table that gives the number of times each country occurs

```
Australia Cuba Indonesia Japan Philippines Taiwan United Kingdom
      3     1         4     6             2     1             2
[There are 19 cities in all.]
```

Printing the contents of the column with the name **country** gives the names, not the codes. As in most operations with factors, R does the translation invisibly. There are though annoying exceptions that can make the use of factors tricky. To be sure of getting the country names, specify

```
as.character(isl andci ti es$country)
```

To get the codes, specify

```
as.integer(isl andci ti es$country)
```

By default, R sorts the level names in alphabetical order. If we form a table that has the number of times that each country appears, this is the order that is used:

```
> table(isl andci ti es$country)
Australia Cuba Indonesi a Japan Phi l i p p i n e s Tai wan Uni ted Ki ngdom
      3     1         4     6             2     1             2
```

This order of the level names is purely a convenience. We might prefer countries to appear in order of latitude, from North to South. We can change the order of the level names to reflect this desired order:

```
> lev <- levels(isl andci ti es$country)
> lev[c(7, 4, 6, 2, 5, 3, 1)]
[1] "Uni ted Ki ngdom" "Japan"           "Tai wan"         "Cuba"
[5] "Phi l i p p i n e s" "I ndonesi a"     "Austral i a"
> country <- factor(isl andci ti es$country, levels=lev[c(7, 4, 6, 2, 5, 3, 1)])
> table(country)
Uni ted Ki ngdom Japan Tai wan Cuba Phi l i p p i n e s I ndonesi a Austral i a
      2     6     1     1             2         4     3
```

In ordered factors, i.e. factors with ordered levels, there are inequalities that relate factor levels.

Factors have the potential to cause a few surprises, so be careful! Here are two points to note:

1. When a vector of character strings becomes a column of a data frame, R by default turns it into a factor. Enclose the vector of character strings in the wrapper function `I ()` if it is to remain character.
2. There are some contexts in which factors become numeric vectors. To be sure of getting the vector of text strings, specify e.g. `as.character(country)`.
3. To extract the numeric levels 1, 2, 3, ..., specify `as.numeri c(country)`.

`read.table("a:/myfile.csv", sep=",")` to read the data into R. This copes with any spaces which may appear in text strings. [But watch that none of the cell entries include commas.]

⁴⁰ Factors are vectors which have mode numeric and class "factor". They have an attribute `levels` that holds the level names.

7.6 Ordered Factors

Actually, it is their levels that are ordered. To create an ordered factor, or to turn a factor into an ordered factor, use the function `ordered()`. The levels of an ordered factor are assumed to specify positions on an ordinal scale. Try

```
> stress.level <- rep(c("low", "medium", "high"), 2)
> ordf.stress <- ordered(stress.level, levels=c("low", "medium", "high"))
> ordf.stress
[1] low    medium high    low    medium high
Levels: low < medium < high
> ordf.stress < "medium"
[1] TRUE FALSE FALSE TRUE FALSE FALSE
> ordf.stress >= "medium"
[1] FALSE TRUE TRUE FALSE TRUE TRUE
```

Later we will meet the notion of inheritance. Ordered factors inherit the attributes of factors, and have a further ordering attribute. When you ask for the class of an object, you get details both of the class of the object, and of any classes from which it inherits. Thus:

```
> class(ordf.stress)
[1] "ordered" "factor"
```

7.7 Lists

Lists make it possible to collect an arbitrary set of R objects together under a single name. You might for example collect together vectors of several different modes and lengths, scalars, matrices or more general arrays, functions, etc. Lists can be, and often are, a rag-tag of different objects. We will use for illustration the list object that R creates as output from an `lm` calculation.

For example, suppose that we create a linear model (`lm`) object `elastic.lm` (c. f. sections 1.1.4 and 2.1.4) by specifying

```
elastic.lm <- lm(distance~stretch, data=elasticband)
```

It is readily verified that `elastic.lm` consists of a variety of different kinds of objects, stored as a list. You can get the names of these objects by typing in

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

The first list element is:

```
> elastic.lm$coefficients
(Intercept)    stretch
 -63.571429    4.553571
```

Alternative ways to extract this first list element are:

```
elastic.lm[["coefficients"]]
elastic.lm[[1]]
```

We can alternatively ask for the sublist whose only element is the vector `elastic.lm$coefficients`. For this, specify `elastic.lm["coefficients"]` or `elastic.lm[1]`. There is a subtle difference in the result that is printed out. The information is preceded by `$coefficients`, meaning “list element with name `coefficients`”.

```
> elastic.lm[1]
$coefficients
(Intercept)    stretch
 -63.571429    4.553571
```

The second list element is a vector of length 7

```
> options(digits=3)
> elastic.lm$residuals
      1      2      3      4      5      6      7
2.107 -0.321 18.000  1.893 -27.786 13.321 -7.214
```

The tenth list element documents the function call:

```
> elastic.lm$call
lm(formula = distance ~ stretch, data = elasticband)
> mode(elastic.lm$call)
[1] "call"
```

*7.8 Matrices and Arrays

In these notes the use of matrices and arrays will be quite limited. For almost everything we do here, data frames have more general relevance, and achieve what we require. Matrices are likely to be important for those users who wish to implement new regression and multivariate methods.

All the elements of a matrix have the same mode, i.e. all numeric, or all character. Thus a matrix is a more restricted structure than a data frame. One reason for numeric matrices is that they allow a variety of mathematical operations that are not available for data frames. Another reason is that **matrix** generalises to **array**, which may have more than two dimensions.

Note that matrices are stored columnwise. Thus consider

```
> xx <- matrix(1:6, ncol=3) # Equivalently, enter matrix(1:6, nrow=2)
> xx
      [, 1] [, 2] [, 3]
[1, ]    1    3    5
[2, ]    2    4    6
```

If **xx** is any matrix, the assignment

```
x <- as.vector(xx)
```

places columns of **xx**, in order, into the vector **x**. In the example above, we get back the elements 1, 2, . . . , 6.

Names may be assigned to the rows and columns of a matrix. We give details below.

Matrices have the attribute “dimension”. Thus

```
> dim(xx)
[1] 2 3
```

In fact a matrix *is* a vector (numeric or character) whose dimension attribute has length 2.

Now set

```
> x34 <- matrix(1:12, ncol=4)
> x34
      [, 1] [, 2] [, 3] [, 4]
[1, ]    1    4    7   10
[2, ]    2    5    8   11
[3, ]    3    6    9   12
```

Here are examples of the extraction of columns or rows or submatrices

```
x34[2:3, c(1,4)] # Extract rows 2 & 3 & columns 1 & 4
x34[2, ] # Extract the second row
x34[-2, ] # Extract all rows except the second
x34[-2, -3] # Extract the matrix obtained by omitting row 2 & column 3
```

The **dimnames()** function assigns and/or extracts matrix row and column names. The **dimnames()** function gives a list, in which the first list element is the vector of row names, and the second list element is the vector of column names. This generalises in the obvious way for use with arrays, which we now discuss.

7.8.1 Arrays

The generalisation from a matrix (2 dimensions) to allow > 2 dimensions gives an array. A matrix is a 2-dimensional array.

Consider a numeric vector of length 24. So that we can easily keep track of the elements, we will make them 1, 2, ..., 24. Thus

```
x <- 1:24
```

Then

```
dim(x) <- c(4, 6)
```

turns this into a 4 x 6 matrix.

```
> x
     [, 1] [, 2] [, 3] [, 4] [, 5] [, 6]
[1, ]    1    5    9   13   17   21
[2, ]    2    6   10   14   18   22
[3, ]    3    7   11   15   19   23
[4, ]    4    8   12   16   20   24
```

Now try

```
> dim(x) <-c(3, 4, 2)
```

```
> x
```

```
, , 1
     [, 1] [, 2] [, 3] [, 4]
[1, ]    1    4    7   10
[2, ]    2    5    8   11
[3, ]    3    6    9   12
```

```
, , 2
     [, 1] [, 2] [, 3] [, 4]
[1, ]   13   16   19   22
[2, ]   14   17   20   23
[3, ]   15   18   21   24
```

7.8.2 Conversion of Numeric Data frames into Matrices

There are various manipulations that are available for matrices, but not for data frames. Use `as.matrix()` to handle any conversion that may be necessary.

7.9 Different Types of Attachments

When R starts up, it has a list of directories where it looks, in order, for objects. You can inspect the current list by typing in `search()`. The working directory comes first on the search list.

You can extend the search list in two ways. The `library()` command adds libraries. Alternatively, or in addition, the `attach()` command places a data frame on the search list. A data frame is in fact a specialised list, with its columns as the objects. Recall the syntax

```
> attach(primates)      # NB: No quotes
> detach(primates)     # NB: S-PLUS requires detach("primates")
```

7.10 Exercises

1. Generate the numbers 101, 102, ..., 112, and store the result in the vector `x`.
2. Generate four repeats of the sequence of numbers (4, 6, 3).

3. Generate the sequence consisting of eight 4s, then seven 6s, and finally nine 3s.
4. Create a vector consisting of one 1, then two 2's, three 3's, etc., and ending with nine 9's.
5. Determine, for each of the columns of the data frame **airquality** (base library), the median, mean, upper and lower quartiles, and range.
[Specify **data(airquality)** to bring the data frame **airquality** into the working directory.]
6. For each of the following calculations, decide what you would expect, and then check to see if you were right!
 - a)


```
answer <- c(2, 7, 1, 5, 12, 3, 4)
for (j in 2:length(answer)){ answer[j] <- max(answer[j], answer[j-1])}
```
 - b)


```
answer <- c(2, 7, 1, 5, 12, 3, 4)
for (j in 2:length(answer)){ answer[j] <- sum(answer[j], answer[j-1])}
```
7. In the built-in data frame **airquality** (a) extract the row or rows for which **Ozone** has its maximum value; and (b) extract the vector of values of **Wind** for values of **Ozone** that are above the upper quartile.
8. Refer to the Eurasian snow data that is given in Exercise 1.6 . Find the mean of the snow cover (a) for the odd-numbered years and (b) for the even-numbered years.
9. Determine which columns of the data frame **Cars93** (MASS library) are factors. For each of these factor columns, print out the levels vector. Which of these are ordered factors?
10. Use **summary()** to get information about data in the data frames **airquality**, **attitude** (both in the base library), and **cpus** (MASS library). Write brief notes, for each of these data sets, on what you have been able to learn.
11. From the data frame **mtcars** (MASS library) extract a data frame **mtcars6** that holds only the information for cars with 6 cylinders.
12. From the data frame **Cars93** (MASS library) extract a data frame which holds only information for small and sporty cars.
13. Store the numbers obtained in exercise 2, in order, in the columns of a 3 x 4 matrix.
14. Store the numbers obtained in exercise 3, in order, in the columns of a 6 by 4 matrix. Extract the matrix consisting of rows 3 to 6 and columns 3 and 4, of this matrix.