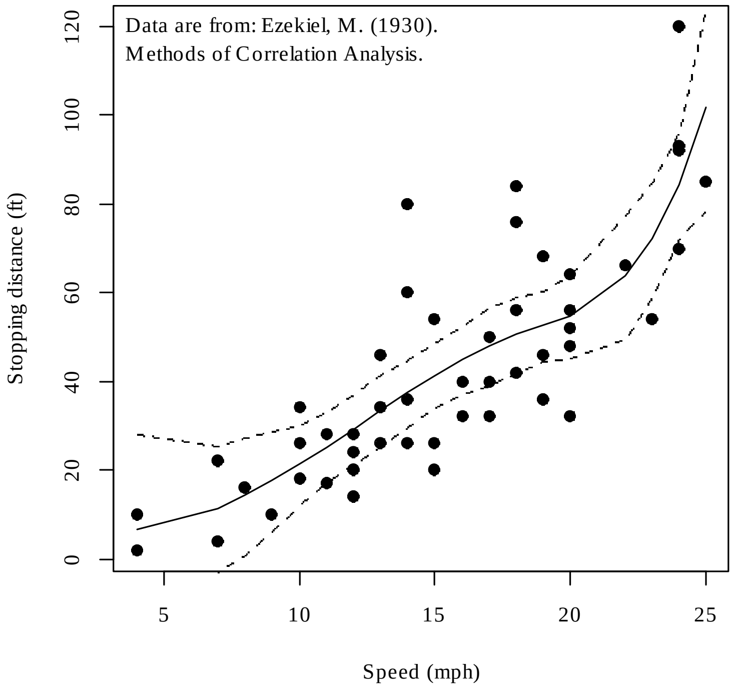
Data Analysis and Graphics Using R – An Introduction

J H Maindonald

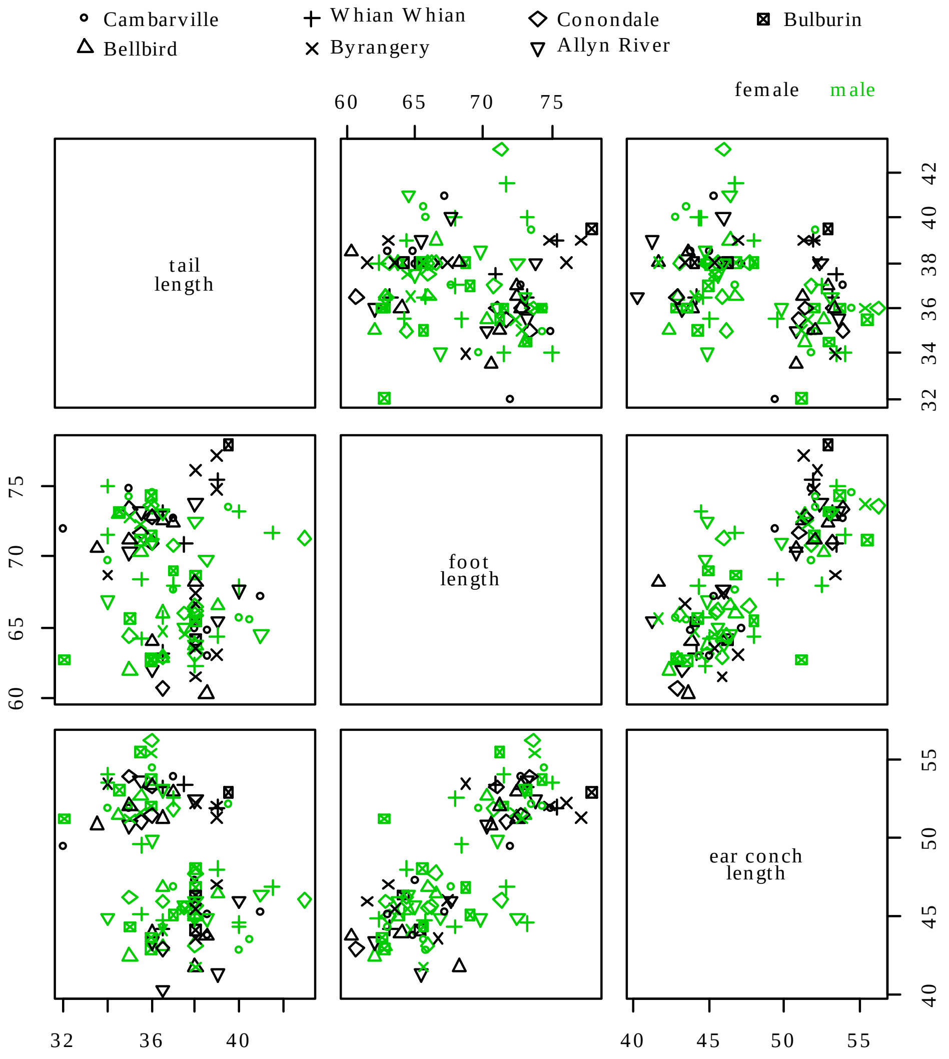
Statistical Consulting Unit of the Graduate School,

Australian National University.

3 March 2000



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.

Introduction

R implements a dialect of the S language which 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 and for Linux and other Unix systems. It is available through the Comprehensive R Archive Network (CRAN). Web addresses are given at the beginning of Chapter 1, and in Chapter 7.

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 you may want to consider:

1. R has extensive and powerful graphics abilities, which 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 you do in due course find that simple methods are not adequate, R has a huge range of more advanced abilities that you can call into use. Alternatively you can adapt the available abilities to give you what you need.
4. The R community is widely drawn, from application area specialists as well as statistical specialists. It is also a community that is sensitive to the potential for misuse of statistical techniques and suspicious of what might appear to be mindless use. You will find scepticism of the use of models which are not susceptible to some minimal form of data-based validation.
5. Because R is free, you have no right to expect attention, on the r-help list or elsewhere, to your particular problem. Be grateful for whatever help you get.

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 you the statistical expertise that you need 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, you will find traps which 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 statistical system you are using, 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 file (*r-notes.dis*) using the notes as commentary. You can either type the commands in at the console, or you can 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 1999. To download a copy, 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/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. The current version of R is designed for use with small to medium sized data sets. It uses a "fixed memory" model. Depending on available computer memory and on R workspace settings, the processing of a data set containing one hundred thousand observations and perhaps ten variables may press the limits of what R can reasonably handle.

The S language on which R is largely based is best known through its commercial S-PLUS implementation. Most of the independent libraries that were developed for S-PLUS have been adapted to run under R. These give access to up-to-date methodology from leading statistical researchers. Like S-PLUS, R has strong graphics abilities. R has no complete equivalent of S-PLUS trellis graphics, though coplot has a limited ability to produce scatterplot panels.

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. Do 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. Chapter 13 gives useful web addresses.

Jeff Wood (CMIS, CSIRO), Andreas Ruckstuhl (Technikum Winterthur Ingenieurschule, Switzerland) and John Braun (University of Winnipeg) 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 instructions on installation of R.

You then need to set up one or more R icons (or a folder containing one or more icons) on your screen.

1.1 Installation

At the time of writing, the latest Microsoft Windows version is RW-1.0.0 . Files are:

rwinst.exe (This controls the installation.)

rw1000b1.zip, rw1000b2.zip (Binaries for RW-1.0.0)

rw1000h.zip (Text help)

rw1000w.zip (html help)

rw1000ch.help [compiled html; activate with `options(chmhelp=T)`]

rw1000l.zip (latex help files - most users will not need these)

You should get the latest versions of the files. Make sure that all the files you want to install (certainly rw1000b1.zip and rw1000b2.zip, and probably rw1000h.zip) are in the same directory. Run rwinst.exe, and follow instructions. Finally, create a link (e. g. by right-clicking in blank space on the Windows screen, or in a folder where you want to include the link). Set the **target** to be

```
<path to binary>\rw1000\bin\rgui.exe
```

where <path to binary> is the path to the directory in which you placed the files used for installation[1].

You can also, optionally, set a **Start in** directory. This functions as a working directory, where any R objects that you create will by default be saved, providing that you request this before quitting. It pays to have a separate working (**Start in**) directory, and a separate associated icon, for each major project. For more details. see the README file that is included with the R distribution.

The Australian mirror site, from which you can download new releases of R as they appear, is:

```
http://mirror.aarnet.edu.au/CRAN/
```

For Windows 95 etc binaries, look in

```
http://mirror.aarnet.edu.au/CRAN/windows/windows-9x/
```

Look under the directory `base`. There is also a *contributed* directory, from which you can get libraries for R.

1.2 Getting started

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

In interactive use under Microsoft Windows there are two ways to input commands to R. You can use either or both of these forms of input at your discretion:

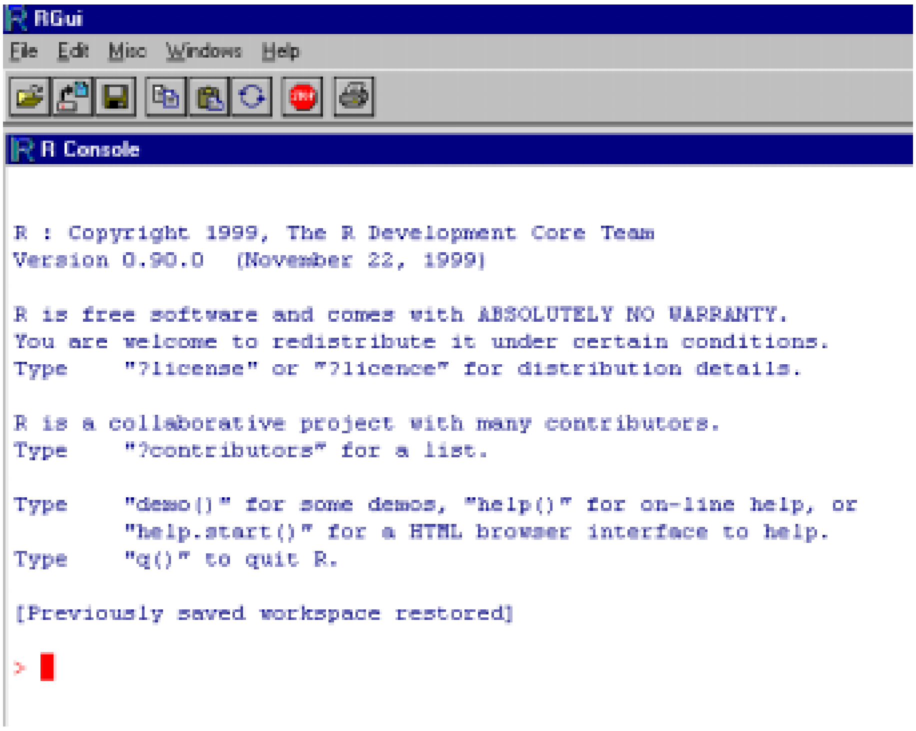
1. For the moment, we will type commands into the *command window,* at the command line prompt. Fig. 1 shows the command window as it appears when R has just been started.
2. The screen snapshot in Fig.2 shows a *display file* window. To get a display file window, go to the **File** menu. Then click 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.

[1] If you want a larger memory space than the default you may want a **target** akin to

```
<path to binary>\rw091\bin\rgui.exe --vsize 30M --nsize 1000k
```

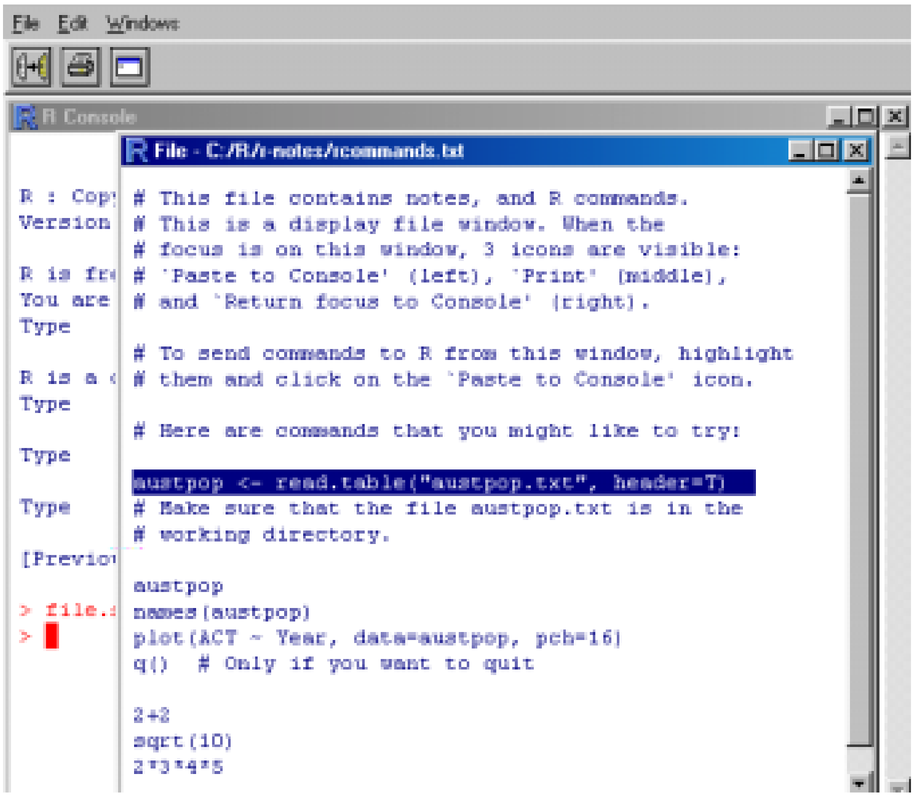
[The default is `--vsize 6M --nsize 250k` . The nsize (cons blocks) settings control the space for the building blocks of the language and the workspace. The vsize (heap) setting creates space for new objects as they are created. The setting `--vsize 6M` should be adequate for 5000 observations on 40 numeric variables.]

Any commands that are to be input to R are highlighted in the window. Clicking on the `Paste to console' icon, on the far left of the display file toolbar in Figs. 2 and 3, then sends these commands to R.



```
R : Copyright 1999, The R Development Core Team
Version 0.90.0  (November 22, 1999)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type    "?license" or "?licence" for distribution details.

R is a collaborative project with many contributors.
Type    "?contributors" for a list.

Type    "demo()" for some demos, "help()" for on-line help, or
        "help.start()" for a HTML browser interface to help.
Type    "q()" to quit R.

[Previously saved workspace restored]

>
```

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



```
# This file contains notes, and R commands.
# This is a display file window. When the
# focus is on this window, 3 icons are visible:
# `Paste to Console' (left), `Print' (middle),
# and `Return focus to Console' (right).

# To send commands to R from this window, highlight
# them and click on the `Paste to Console' icon.

# Here are commands that you might like to try:

austpop <- read.table("austpop.txt", header=T)
# Make sure that the file austpop.txt is in the
# working directory.

austpop
names(austpop)
plot(ACT ~ Year, data=austpop, pch=16)
q()  # Only if you want to quit

2+2
sqrt(10)
2*3*4*5
```

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

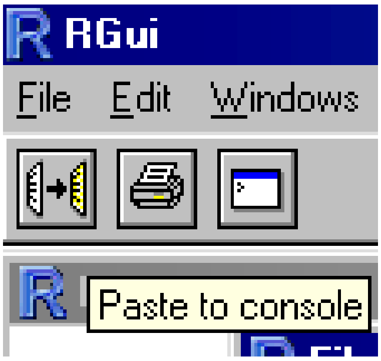


Fig. 3: The `paste to console', `print', and `return focus to console' icons.

Under Unix, the standard form of input is the command line interface. Under both Microsoft Windows and Unix, a further possibility is to run R from within the emacs editor[2].

1.2.1 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.

Just in case you want to quit from R at this point, you should know that the exit or quit command is

```
> q()
```

Alternatives are to click on the **File** menu and then on **Exit**, or to click on the ✕ in the top right hand corner of the R window.

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
```

[2] For this, you need to install on your PC both emacs and the emacs add-on call ESS. You can get ESS from http://franz.stat.wisc.edu/pub/ESS, or if ftp is easier from ftp:// franz.stat.wisc.edu/pub/ESS .

The following statement can be used to read in the data:

```
> austpop <- read.table(“a:/austpop.txt”, header=T)
```

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

Type in **austpop** at the command line prompt, and the object will be displayed, thus:

```
> austpop
  Year  NSW Vic.  Qld   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
3 1937 2693 1853  993  589  457  233   6  11  6836
4 1947 2985 2055 1106  646  502  257  11  17  7579
5 1957 3625 2656 1413  873  688  326  21  38  9640
6 1967 4295 3274 1700 1110  879  375  62 103 11799
7 1977 5002 3837 2130 1286 1204  415 104 214 14192
8 1987 5617 4210 2675 1393 1496  449 158 265 16264
9 1997 6274 4605 3401 1480 1798  474 187 310 18532
>
```

We will learn later that **austpop** is a special form of R object, known as a data frame. Data frames that consist entirely of numeric data have the same structure as numeric matrices.

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:

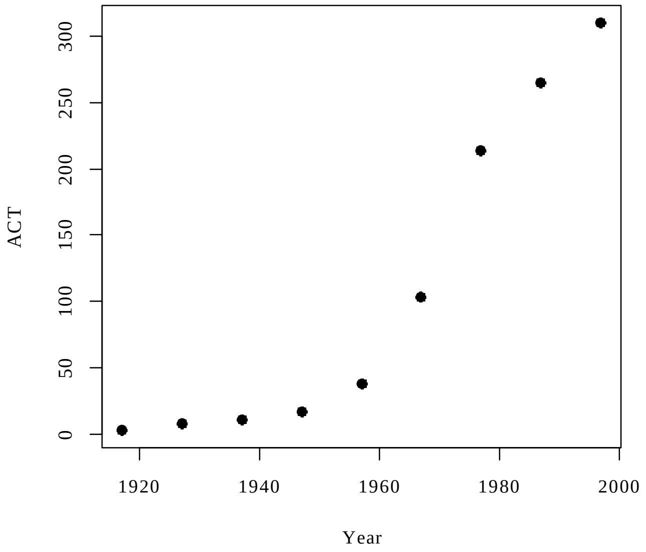


Fig. 4: ACT population, at various times between 1917 and 1997.

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

To quit from the R session type

```
> q()
```

You will be asked whether you want to save the workspace image. Unless you are quite sure that you do not want to save any of the objects that were newly created in your R session, click **Yes**.

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 which the band moved when released:

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

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

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

1.4 Further Notational Details

As noted earlier, the command line prompt is

```
>
```

R commands (expressions) are typed in following this prompt[3].

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.

When you type 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 which follows a `#` on the command line is taken as comment and ignored by R.

Note: Notice 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 in

```
> help()
```

[3] Multiple commands may appear on the one line, with the semicolon (`;`) as the separator.

In R for Windows, you can alternatively 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)
```

Often users need to experiment to discover precisely what a specific R function does. The documentation may be short on details of the specific formula that has been used. The documentation is however improving rapidly.

1.6 Exercise

1. In the data frame `elastic` from section 1.3.1, plot `distance` 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 entering the years separately you can type `1970:1979`]

ii. Plot `snow.cover` versus `time`.

iii Use the `hist()` command to plot a histogram of the snow cover values.

iv. Repeat ii and iii after taking logarithms of snow cover.

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 `orings` 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`, `erosion`, `blowby` 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

There is a special class of object called a *data frame*, used to store 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.

As a first example, consider the data frame `hills`, available from the Venables and Ripley MASS library. I have included these data with the data sets that accompany these notes. 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:

```
> data(hills)   # Gives access to the data frame hills
> 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
>
```

Thus we can immediately see that the range of distances (first column) is from 2 miles to 28 miles, and that the range of times (third column) is from 15.95 (minutes) to 204.6 minutes
We will discuss graphical summaries in the next section.

2.1.3 R has extensive abilities for graphical presentation

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, that was obtained by typing

```
> pairs(hills)
```

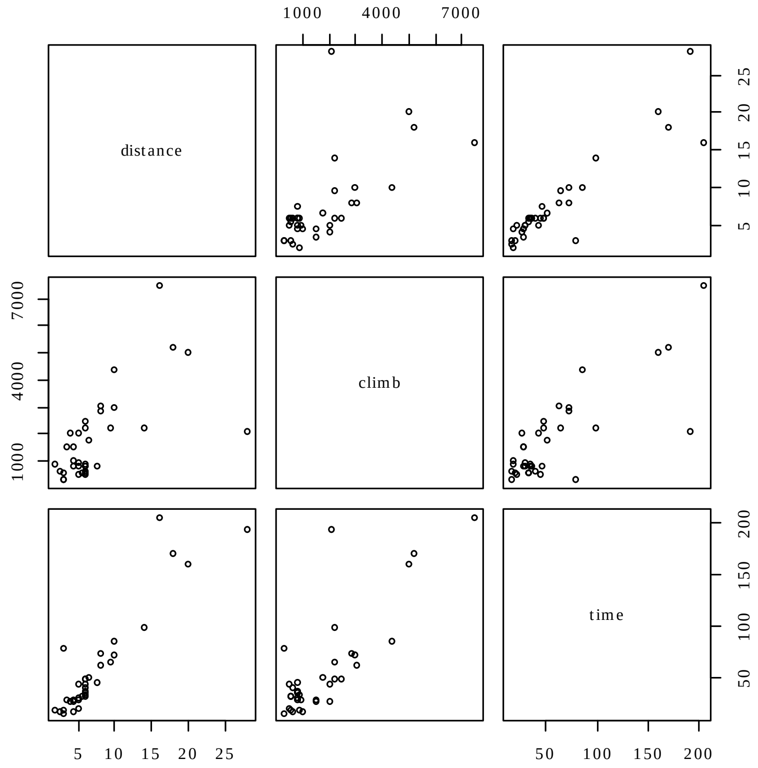


Fig. 5: Scatterplot matrix for the Scottish hill race data.

2.1.4 R will handle a variety of specific analyses

The examples that will be given are correlation and regression.

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). It is possible to write a function which will relabel in this way. Notice that the correlations between time and distance, and between time and climb, have reduced. Why do you think that has 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 `elastic` 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=elastic, pch=16)
> elastic.lm<-lm(distance~stretch,data=elastic)
> lm(distance~stretch,data=elastic)

Call:
lm(formula = distance ~ stretch, data = elastic)

Coefficients:
(Intercept)      stretch
    -63.571        4.554
```

You can get more complete information by typing in

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

Try it!

2.1.5 R is an Interactive Programming Language

Suppose we want to calculate the Fahrenheit temperatures which correspond to Celsius temperatures 25, 26, …, 30. Here is a good way to do this in R:

```
> 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 function language. There is a language core which uses standard forms of algebraic notation, allowing you to do the calculations described in Section 2.1.1. Beyond this, most computation is handled using functions. Even the action of quitting from an S session uses a function call. When you type

```
> q()
```

you are invoking the function `q` (for quit). In most expressions you can treat every object – vectors, arrays, lists and so on – as a whole. Use of operators and functions which operate on objects as a whole largely avoids the need for explicit loops. For an example, look back to section 2.1.5 above.

The structure of an R program looks very like the structure of the widely used general purpose language C and its successors C^{++} and Java[4].

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 working directory. An alternative to `ls()` is `objects()`. In both cases you can restrict the names to those with a particular pattern, e. g. starting with the letter `p'[5].

If you type the name of an object at the prompt, the contents of the object are printed out. Try typing in `q`, `mean`, etc.

Important: When you quit, R will ask whether you want to save the workspace image. This allows you to retain, for use in the next session in the same working directory, any objects that you have created in the current session. Careful housekeeping may be needed to ensure that you do not at the same time retain numerous objects that you will never use again. Before you type `q()`, use `rm()` to remove them. Saving the workspace image will them save everything else. The workspace image will be automatically loaded when you start another session in that directory.

*[6]2.4 Looping

In R there is often a better alternative to writing an explicit loop. Where possible, you should use one of the built-in functions to avoid explicit looping. A simple example of a `for` loop is[7]

```
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:

```
> # Fahrenheit to Celsius
> for (fahrenheit in 25:30)
+     print(c(fahrenheit, 9/5*fahrenheit + 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
>
```

[4] 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.

[5] 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.

[6] Asterisks (*) identify sections which are more technical and might be omitted at a first reading

[7] Other looping constructs are:

```
repeat <expression>    ## You'll need break somewhere inside
while (x>0) <expression>
```

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

2.4.1 More on looping

Here is a long-winded way to sum the three number 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.

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

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

Much of the art of using R effectively lies in avoiding unnecessary loops.

2.5 R Functions – Examples

2.5.1 An Approximate Miles to Kilometers Conversion

```
> miles.to.km <- function(miles)miles*8/5
```

The return value is the value of the final (and in this instance only) expression which appears in the function body[8].

Use the function thus

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

You can do the conversion for several distances, all at the one time. To convert a vector of the three distances 100, 200 and 300 miles to distances in kilometers, specify:

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

2.5.2 A Plotting function

In a tasting experiment twenty individuals were each given two milk samples. One sample was of milk that had one unit of an additive, and the other sample was of milk that had four units of an additive. Did the additive affect the perceived sweetness? The data frame `milk` (with columns `one` and `four`) has the result. Sweetness was measured on a scale that ran from 1 to 9, with 1 denoting "not nearly sweet enough", 5 denoting "just right" and 9 denoting "much too sweet".

We plot, for each participant, the `four` result against the `one` result, but insisting on the same range for the x and y axes.

```
xyrange <- range(milk)
plot(four ~ one, data = milk, xlim = xyrange, ylim = xyrange, pch = 16)
abline(0, 1)   # Line where `four' value = `one' value
```

In order to keep together the code used for the graph, we could store these in a function, thus:

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

```
fig6 <- function ()
{
    xyrange <- range(milk) # Range of all values in the data frame
    plot(four ~ one, data = milk, xlim = xyrange, ylim = xyrange,
        pch = 16)
    abline(0, 1)   # Line where `four' value = `one' value
}
```

The function's name is **fig6**. We enclose the function body in braces ({ }). The above function might be a first step to writing a function that takes any data frame, plots one named column against another named column, and shows the line y = x. Later, we will give this as an exercise.

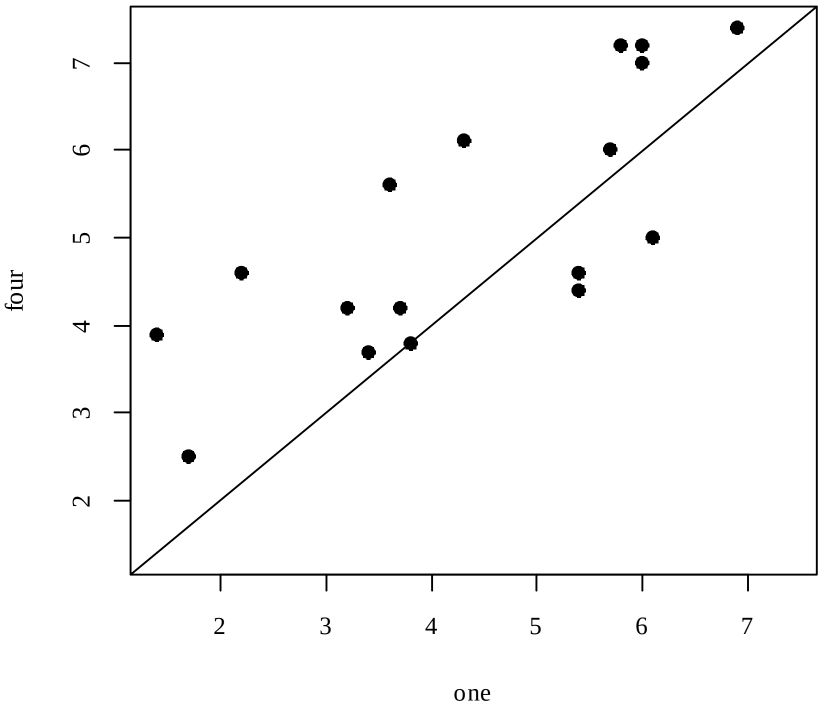


Fig. 6: Sweetness assessment for milk sample with four units of additive (y-axis) versus one unit (x-axis).

2.6 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 is **airquality**[9], giving measurements made on 111 successive days in New York. Because this is in the base library all you need do to bring it into the working directory is to type:

```
> data(airquality)      # Bring data set into working directory
```

Here is summary information on this data frame

```
> summary(airquality)
      Ozone            Solar.R           Wind             Temp
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00
```

[9] This holds the same kind of data as the S-PLUS data sets **environmental** and **air**.

```
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00
 NA's   : 37.00   NA's   :  7.0
      Month            Day
 Min.   :5.000   Min.   : 1.0
 1st Qu.:6.000   1st Qu.: 8.0
 Median :7.000   Median :16.0
 Mean   :6.993   Mean   :15.8
 3rd Qu.:8.000   3rd Qu.:23.0
 Max.   :9.000   Max.   :31.0
```

Type in `data()` to get a list of built-in data sets.

2.7 The R Directory Structure

R has a search list, which you can however change in the course of your session. This includes the list of libraries where R will look for the objects that are needed as your session proceeds. To get a full list of these directories, type in

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

So in addition to the global environment, there are objects which are in the working directory and the base package or library. If you add further libraries (also called packages), they will be added to this list. For example:

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

2.8 Exercises

1. For each of the following code sequences, predict the result. Then use R to 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`.

3. R Data Structures

3.1 Vectors

Vectors may have mode logical, numeric or character[10]. Examples of vectors are

```
> c(2,3,5,2,7,1)
[1] 2 3 5 2 7 1
> 3:10   # The numbers 3, 4, .., 10
[1]  3  4  5  6  7  8  9 10
> c(T,F,F,F,T,T,F)
[1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
> c("Canberra","Sydney","Newcastle","Darwin")
[1] "Canberra"  "Sydney"    "Newcastle" "Darwin"
```

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)` is an acronym for "concatenate", i. e. the meaning is: "Join these numbers together in to a vector"[11]. Observe how one can concatenate two vectors. 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
>
```

We will later meet lists. The concatenate function `c()` may also be used to join lists.

3.1.1 Subsets of Vectors

There are three ways to extract subsets of vectors.

1. Specify the numbers of the elements which 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
>
```

You can use negative numbers to omit elements:

[10] Below, we will meet the notion of "class", which is important for implementation of some of the 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, as one might expect, the value "factor".

[11] More generally, one can use this same construction to join lists.

```
> 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` which are greater than 10.

```
> x<-c(3,11,8,15,12)
> x>10
[1] FALSE  TRUE FALSE  TRUE  TRUE
> x[x>10]
[1] 11 15 12
```

3. Where elements are named, one can use a vector of names to extract the elements

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

3.1.2 Patterned Data

You can use 5:15 to generate the numbers 5, 6, …, 15. If you enter 15:5, this 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 you want four 2s, then four 3s, then four 5s, enter `rep(c(2,3,5), c(4,4,4))`. Another way to achieve the same effect is `rep(c(2,3,5), each=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`."

3.2 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)
> 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
>
```

<u>WARNING</u>: 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 which 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.

For example

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

When `x>2` yields a subscript on the left hand side that is `NA`, no replacement is made. The value on the right is irrelevant, except that there should be one value on the right corresponding to each element of `x>2` that is either `TRUE` or `NA`. Here is a further example of R's behaviour:

```
>  x <- c(1,6,2,NA,10)
> x
[1]  1  6  2 NA 10
> x>2
[1] FALSE  TRUE FALSE    NA  TRUE
> x[x>2] <- c(20,21,22)
> x
[1]  1 20  2 NA 22
>
```

One can use `!is.na(x)` to limit the selection, on both sides, to those elements of `x` that are not `NA`s. Specify

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

This does give the same result as in S-PLUS! For code that is to be used in both R and S-PLUS, you need to include the `!is.na(x) test`, as above. The S-PLUS result from `y[x>2] <- x[x>2]` is different from that above.

We will have more to say on missing values in the section on data frames which now follows.

3.3 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. If you want to be sure that a data frame of numeric data will behave like a matrix of numeric data, use `as.matrix()` to turn it into a matrix.

Lists are discussed below, in section 3.6.

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
Gorilla       207 406
Human          62 1320
"Rhesus monkey" 6.8 179
Chimp         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         Chimp   52.2     440
>
```

3.3.1 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 strings[12].

Problems may arise when small mistakes in the data cause R to interpret a column of supposedly numeric data as character strings. 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 3.3.2 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.

Where the file that you are reading in contains character as well as numeric data, whether by design or accident, the behaviour of `read.table()` may seem idiosyncratic. Until you are familiar with its idiosyncrasies, you may wish to use the parameter setting `as.is = TRUE`**.** [13]

3.3.2 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(".").`

[12] 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.

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

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

3.3.3 Separators when using `read.table()`

With data from spreadsheets[14], 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"`.

3.3.4 Component Parts of Data frames

Recall that the data frame `primates` has a column of species names, then `Bodywt` in column 2, then `Brainwt` in column 3. Any of the following will pick out column 3 of the data frame `primates` and store it in the vector `brain.wt`:

```
brain.wt<-primates$Brainwt
brain.wt<-primates[,3]
brain.wt<-primates[,”Brainwt”]
brain.wt<-primates[[3]]    # Take the object that is stored
                           # in the second list element.
```

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.

```
> 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 abbreviations have been used 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)`.

[14] 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 `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.]

3.3.5 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.

3.4 Factors

One justification for factors is that they provide an economical way of storing vectors of character strings in which many of the same character strings are stored a number of times. We start with this storage economy explanation of factors because it is a good context in which to explain the dual identity of factors[15]. Analysis of variance and regression models provide another more cogent reason for the use of factors, as will become apparent in chapter 6.

The data frame `islandcities` which 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 (`population`) 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.]
```

Rather than store `Australia’ three times, `Indonesia’ four times, and so on, it is more economical to different numerical codes for each of the different countries, then using a look-up table to associate code with country:

country	code
Australia	1
Cuba	2
Indonesia	3
Japan	4
Philippines	5
Taiwan	6
United Kingdom	7

If the column for country is stored as a factor, then the value is 1 when the country is Australia, 2 when the country is Cuba, and so on. The numbers 1, 2, . . ., 7 are factor “indices” or “codes”. The country names are the

[15] Factors are vectors which have mode numeric and class "factor". They have an attribute levels which specifies the level names.

factor levels. When you print out the contents of the country column, what you see are the names, not the codes. R does the translation invisibly. In fact the codes are invisible in most operations that you might want to do with factors. There are, though, annoying exceptions which can make the use of factors tricky.

The level names are stored as a factor attribute in the levels vector. We can get details of the level names by typing in `levels(islandcities$country)`, thus:

```
> levels(islandcities$country)
[1] "Australia"      "Cuba"           "Indonesia"      "Japan"
[5] "Philippines"    "Taiwan"         "United Kingdom"
>
```

When a factor is created, the default behaviour is that level names are in alphabetical order. This order of the level names is purely a convenience. Level names can be re-arranged to any order you like. [Later we will meet ordered factors, i. e. factors with ordered levels, where the order is not arbitrary.]

Note the dual identity of the factor `country`. It is at one and the same time a numeric vector and a vector of character strings. In truth it is neither of these, but rather a data structure that encompasses them both. The view which a factor presents depends on how you intend to use it.

Here is a simple example. The statement

```
> ff1<-factor(c("UC","UC", "ANU","ANU"))
> ff1
[1] UC  UC  ANU ANU
Levels:  ANU UC
```

stores the values (2, 2, 1, 1), with the levels vector equal to (`ANU, UC`). The levels vector has `ANU` first because the order of levels is by default alphanumeric order. You can place `“UC”` first by specifying

```
> ff2<-factor(c("UC","UC", "ANU","ANU"), levels=c("UC", "ANU"))
> ff2
[1] UC  UC  ANU ANU
Levels:  UC ANU
```

The “labels” parameter of factor allows you to change level names. The label text string that you specify for each level becomes the new level name. You need to exercise care that the label names are in the same order as the relevant level names vector.

```
> factor(c("UC","UC", "ANU","ANU"),
+  labels=c("Australian National University", "University of Canberra"))
[1] University of Canberra         University of Canberra
[3] Australian National University Australian National University
Levels:  Australian National University University of Canberra

> factor(c("UC","UC", "ANU","ANU"),levels=c("UC","ANU"),
+  labels=c("University of Canberra", "Australian National University"))
[1] University of Canberra         University of Canberra
[3] Australian National University Australian National University
Levels:  University of Canberra Australian National University
>
```

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

1. When you make a vector of character strings 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 you want it to remain character.

2. There are some contexts in which factors become numeric vectors. You can be sure of getting the vector of text strings by specifying e. g. `as.character(ff1)`.

3. To extract the numeric levels 1, 2, 3, …, specify `as.integer(ff1)`.

3.5 Ordered Factors

Actually, it is their levels which 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"
```

3.6 Lists

Lists allow you 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=elastic)
```

You will find 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"
>
```

Here are three different and equivalent ways to examine the first list element

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

Note that we can also ask for `elastic.lm[“coefficients”]` or `elastic.lm[1]`. These are subtly different. Either of these give us the list whose only element is the above vector. This is reflected 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
```

We defer discussion of list elements 3 to 9, interesting though they are. The tenth list element is

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

*3.7 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 which 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
     [,1] [,2]
[1,]    2   11
[2,]    3   12

> x34[2,]  # Extract the second row
[1]  2  5  8 11

> x34[-2,]  # Extract all rows except the second
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    3    6    9   12

> x34[-2,-3]  # Extract the matrix obtained by omitting row 2 & column 3
     [,1] [,2] [,3]
[1,]    1    4   10
[2,]    3    6   12
>
```

You can use the `dimnames()` function to assign and/or extract 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.

3.7.1 Arrays

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

Suppose you have 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
```

3.7.2 Conversion of Numeric Data frames into Matrices

Use `as.matrix()` for this purpose.

Suppose for example that you want to interchange the rows and columns of a data frame that contains only numbers. You can do this by using `t(as.matrix())` to convert it to a matrix and transpose it, then `data.frame()` to convert it back to a data frame.

The first three columns of the `moths`[16] data frame that accompanies these notes are numeric. So we can do this:

```
> transposed.moths <- data.frame(t(as.matrix(moths[,1:3])))
> transposed.moths
       X1 X2  X3 X4  X5 X6 X7 X8 X9 X10 X11 X12 X13 X14 X15 X16 X17 X18
meters 25 37 109 10 133 26  4  3  3  27  16   6  17   3   5 163  10   5
A       9  3   7  0   9  3  0  0  0  39   7  12   6   2   1   5   2   2
P       8 20   9  2   1 18  5  5  2   5  16   0   0   0   0   1   4   0
       X19 X20 X21 X22 X23 X24 X25 X26 X27 X28 X29 X30 X31 X32 X33 X34
meters  13  63   4   4  33 241  18   2 182  48  20   3  36 233  44  35
A       23  10   5   6   2   4   2   3   4   3   3   1   3   6   1   9
P        6  12   0   5   1   1   0   1   2   3   4   0   1   3   1   0
       X35 X36 X37 X38 X39 X40 X41
meters   8  55   6  90  44  21  36
A       10   2   0   6   0   0   9
P        0   0   2   2   4   4   1
```

3.8 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. You can use the `library()` command to add libraries. Alternatively, or in addition, you can place a list of R objects on the search list. A data frame is in fact a specialised list, with its columns as the objects. If you add a data frame to the search list, then you can refer to the columns by name, without the need to specify the data frame to which they belong. If there is any overlap of names, the order on the search list determines what name will be taken.

The documentation speaks of attaching databases.

3.8.1 Attaching Data Frames

Thus

```
> attach(primates)
```

then allows you to refer to `Brainwt` and `Bodywt`, where you would otherwise have to type `primates$Brainwt` and `primates$Bodywt`. This assumes that you do not have any other variables or columns of attached data frames that have either of these names.

[16] I am grateful to Sharyn Wragg, who was an honours student at ANU, for making these data available.

```
> Bodywt
 Potar monkey Gorilla Human Rhesus monkey Chimp
           10     207    62           6.8  52.2
> Brainwt
 Potar monkey Gorilla Human Rhesus monkey Chimp
          115     406  1320           179   440
```

To detach this data frame, type

```
> detach(“primates”)
```

i. e. quotes are now used.

Note how the use of quotes changes. You specify the name (without quotes) when you attach, and enclose the name between quotes when you detach.

3.9 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` which 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.
15. Remove all the data frames which you have brought into the working directory.

4. 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 which R offers, enter

```
        demo(graphics)
```

Press the Enter key to move to each new graph.

4.1 plot () and allied functions

The basic command is

```
        plot(y~x)
```

or

```
        plot(x,y)
```

where `x` and `y` must be the same length. This second form of command is the model that you need to follow for `points()`, `lines()`, `text()`, etc.

Try

```
        > plot((0:20)*pi/10, sin((0:20)*pi/10))
        > plot((1:50)*1.75, sin((1:50)*1.75))
```

Comment on the appearance which these graphs present. Would you have guessed, if you had not known the formula for plotting the data, that these pointsd lay on a sine curve?

Here are two further examples.

```
        > attach(elastic)           # R now knows where to find stretch & distance
        > plot(stretch, distance)  # Alternatively: plot(distance~stretch)
        > detach("elastic") # Not strictly necessary, but it is well to tidy up.

        > attach(austpop)   # These are the data we read in from austpop.txt
        plot(Year, ACT, type="l")  # Join the points
                                   # ("l" = "line")
        detach("austpop")
```

The `points()` function allows you to add points to a plot. The `lines()` function allows you to add lines to a plot[17]. The `text()` function allows you to place text anywhere on the plot. The `mtext()` function allows you to place text in the margins. The `axis()` function gives you fine control over axis ticks and labels.
You might also like to try

```
        > attach(austpop)
        > plot(spline(Year, ACT), type="l")  # Fit smooth curve thru the points
        > detach("austpop")
```

4.1.1 Newer plot methods

Above, I described the default plot method. There are other ways in which you can use `plot()`. The plot function is a generic function which has special methods for "plotting" various different classes of object. For example, you can plot a data frame. Plotting a data frame gives, for each numeric variable, a normal probability plot. Or you can plot the `lm` object that is created by the use of the `lm()` modelling function. This is designed to give helpful diagnostic and other information that will aid in the interpretation of regression results.

[17] Actually these functions are identical, differing 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.

Try

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

4.2 Fine control – Parameter settings

Much of the time, the default settings of parameters, such as character size, are adequate. If however you do need to change parameter settings, the `par()` function does this. For example,

```
        par(cex=1.25, mex=1.25)
```

increases the text and plot symbol size 25% above the default. I have added `mex=1.25` to ensure that there is 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 a good idea to store the existing settings, so that you can restore them later. For this, you can 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. You can then restore the original parameter settings later, with `par(oldpar)`. Inside a function it is a good idea to specify, e. g.

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

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

4.2.1 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. If you want a column by column layout, then use `mfcol`. In the example below we look at four different transformations of the primates data.

```
        par(mfrow=c(2,2))
        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))      # Restore to 1 figure per page
```

4.2.2 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. In R for Windows you can use win.`graph()` or `x11()` to set up the graphics page. It takes 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.

4.3 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 Brainwt
         Potar monkey    10.0     115
              Gorilla   207.0     406
                Human    62.0    1320
```

```
Rhesus monkey     6.8     179
        Chimp    52.2     440
> attach(primates)  # Needed if primates is not already attached.
> plot(Bodywt, Brainwt, xlim=c(5, 250))
> # Specify xlim so that there is room for the labels
> text(x=Bodywt, y=Brainwt,
labels=row.names(primates), adj=0) # adj=0 implies left adjusted text
> detach("primates")
```

Fig. 7 shows the result.

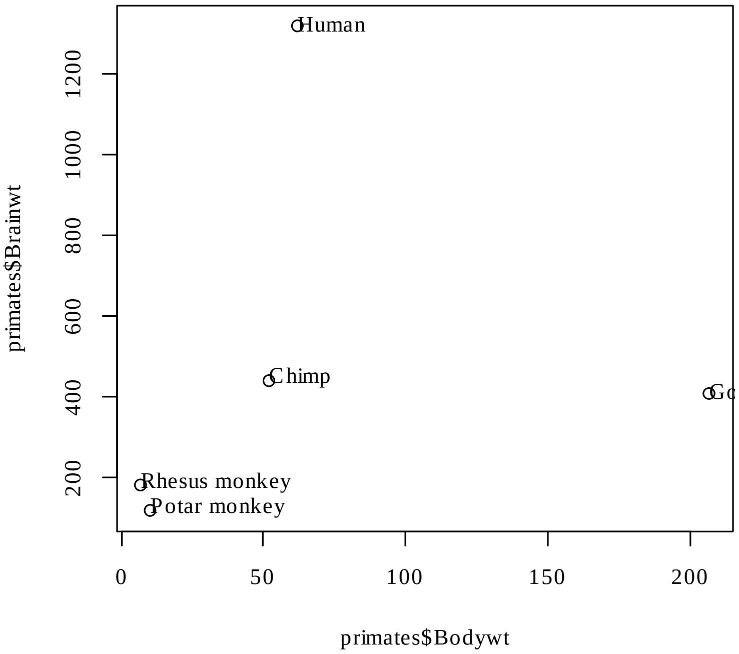


Fig. 7: Plot of the primate data, with labels on points.

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

In Fig. 8 we use the `xlab` (x-axis) and `ylab` (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.

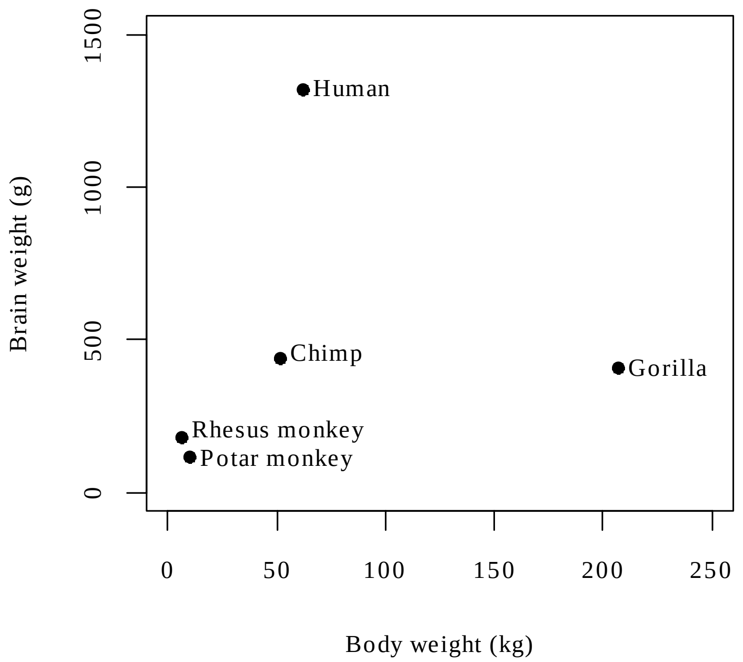


Fig. 8: Improved version of Fig. 7.

Here is the R code that we used in Fig. 8:

```
> plot(x=Bodywt, y=Brainwt, pch=16,
       xlab="Body weight (kg)", ylab="Brain weight (g)",
       xlim=c(5,250), ylim=c(0,1500))
> chw <- strwidth(" ")
> chh <- strheight(" ")
> text(x=Bodywt+chw, y=Brainwt+c(-.1,0,0,-.1,0)*chh,
       labels=primates$Species, adj=0)
```

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

```
> text(x=Bodywt- 0.75*chw, y=Brainwt+c(-.1,0,0,-.1,0)*chh,
       labels=primates$Species, adj=1)
```

4.3.1 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.

4.4 Other Useful Plotting Functions

4.4.1 Scatterplot smoothing

`scatter.smooth()` plots points, then adds a smooth curve through the points.

4.4.2 Normal probability plots

`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. It is a good idea to calibrate your eye to recognise plots which indicate non-normal variation by doing several normal probability plots for random samples of the relevant size from a normal distribution, as in Fig. 9. Here `y` is the difference between the columns four and one in the data frame `milk` that we encountered in section 2.4.2. For a one-sample t-test to be valid, this vector of differences should have an approximately Normal distribution.

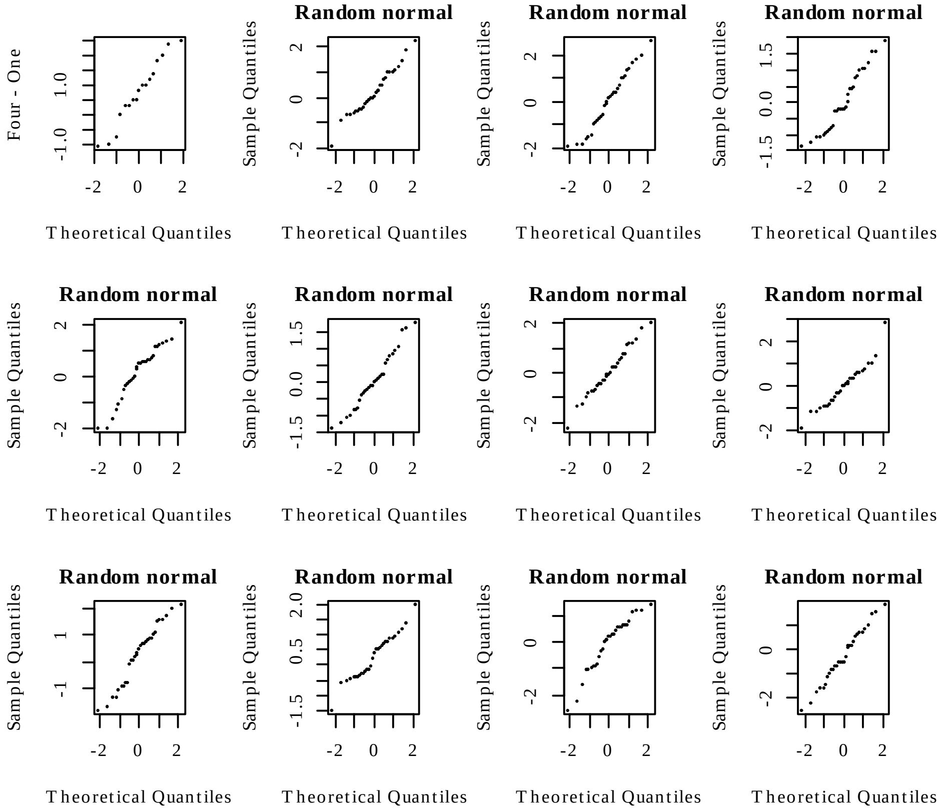


Fig. 9: Normal Probability Plots. The probability plot for the differences for the milk tasting data is the top left plot. Remaining plots are for random normal data.

The code which produced Fig. 9 is:

```
par(mfrow=c(3,4))                # A 3 by 4 layout of plots
y <- elastic$four-elastic$one
qqnorm(y, ylab="Four - One")
                             # Normal probability plot for differences
                             # in times, for the milk tasting data (20 values)
for(i in 1:11)qqnorm(rnorm(21))  # Plots for 11 normal random samples
                                # each of size 20.
par(mfrow=c(1,1))               # Change back to default layout
```

The plot for `y` (= `four` - `one` in the milk data frame) at the top left does seem just as linear as any of the plots of normal data. The idea is an important one. In order to judge whether data are normally distributed, one examines a number of randomly generated samples of the same size from a normal distribution. It is a way to train the eye.

By default, `rnorm()` generates random samples from a distribution with mean 0 and standard deviation 1.

4.4.3 Rug Plots

`rug(x)` adds, along the x-axis of the current plot, vertical bars showing the distribution of values of `x`. One can also add rugs along the y direction. Here is an example:

```
xyrange <- range(milk)
plot(four ~ one,
  data = milk,
  xlim = xyrange,
  ylim = xyrange, pch = 16)
rug(milk$one)
rug(milk$four, side = 2)
abline(0, 1)
```

Fig. 10 shows the result:

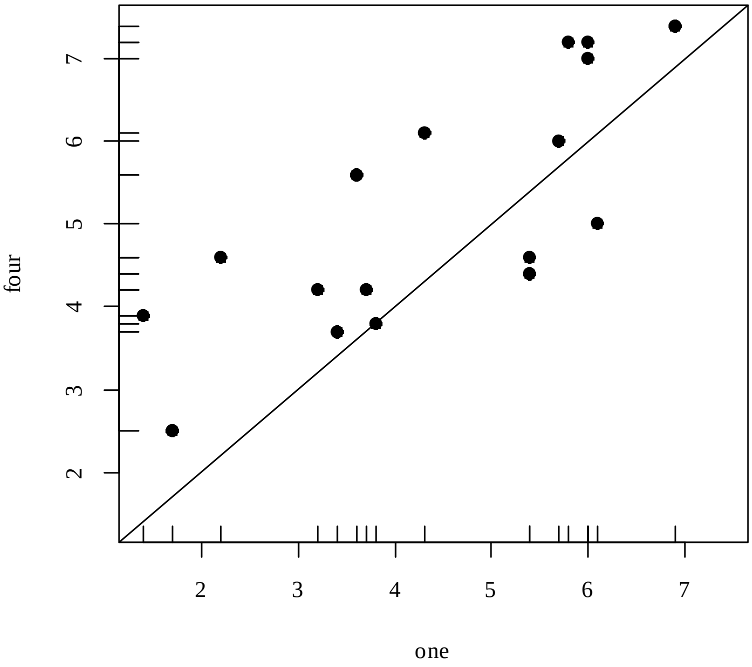


Fig. 10: Each of 20 panelists compared two milk samples for sweetness. The y-ordinate is the assessment for four units of additive, while the x-ordinate is the assessment for one unit of additive.

4.4.4 Scatterplot matrices

Section 2.1.3 demonstrated the use of the `pairs()` function. Note that this function allows only very limited control over graphics parameters. The parameter `cex` has no effect on the size of the plotting symbols. Settings of `mar` (inner margins) and `oma` (outer margins) are ignored, in the latter case with extensive warning messages.

4.4.5 Histograms and Density Plots

It is worth learning to understand and use density plots, as an alternative or adjunct to histograms. The appearance that histograms present can be strongly influenced by the choice of breakpoints between histogram cells. Try the following:

```
> data(islands)  # From the base library
> hist(islands)
> plot(density(islands))
> plot(density(sqrt(islands)))
> plot(density(log(islands)))
```

4.4.6 Dotplots

These can be a good alternative to barcharts. They have a much higher information to ink ratio! Try

```
dotplot(islands)  # Vector of named numeric values
```

Unfortunately there are many names, and there is substantial overlap. The following is better, but shrinks the sizes of the points so that they almost disappear:

```
dotplot(islands, cex=0.2)
```

4.5 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. Fig. 11 was produced with

```
plot(x, y, xlab="Radius", ylab=expression(Area == pi*r^2))
```

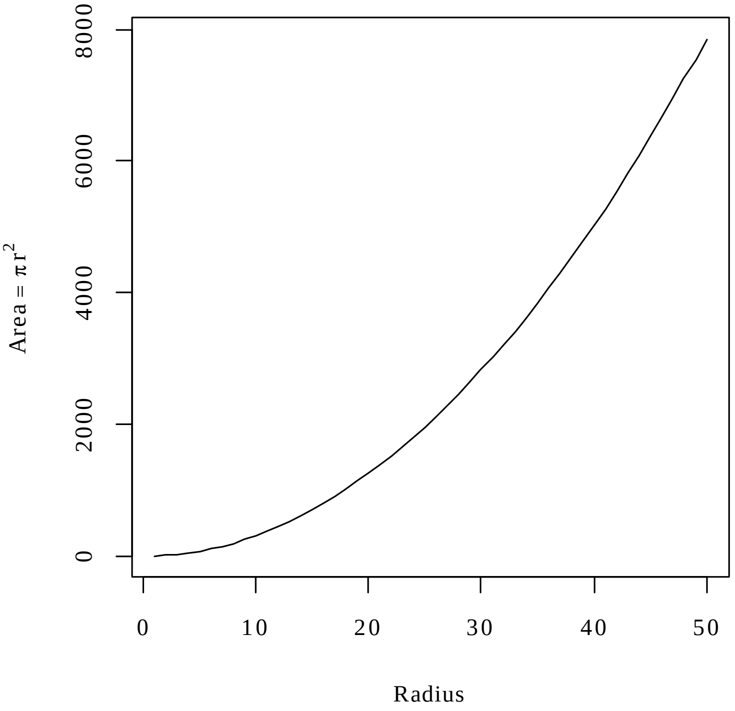


Fig. 11: The y-axis label is a mathematical expression.

The final plot from

```
demo(graphics)
```

shows some of the possibilities for plotting mathematical symbols.

4.6 Guidelines for Graphs

Design graphs to make their point tersely and clearly, with a minimum waste of ink. Label as necessary to identify important features. In scatterplots the graph should attract the eye's attention to the points that are plotted, and to important grouping in the data. Use solid points, large enough to stand out relative to other features, when there is little or no overlap.

When there is extensive overlap of plotting symbols, use open plotting symbols. Where points are dense, overlapping points will give a high ink density, which is exactly what one wants.

Use scatterplots in preference to bar or related graphs whenever the horizontal axis represents a quantitative effect.

Use graphs from which information can be read directly and easily in preference to those that rely on visual impression and perspective. Thus in scientific papers contour plots are much preferable to surface plots or two-dimensional bar graphs.

Draw graphs so that reduction and reproduction will not interfere with visual clarity.

Explain clearly how error bars should be interpreted — ± SE limits, ± 95% confidence interval, ± SD limits, or whatever. Explain what source of `error(s)' is represented. It is pointless to present information on a source of error that is of little or no interest, for example analytical error when the relevant source of `error' for comparison of treatments is between fruit.

Use colour or different plotting symbols to distinguish different groups. Take care to use colours that contrast.

The list of references at the end of this chapter has further comments on graphical and other presentation issues.

4.7 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(brain weight) versus log(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 1860-1986[18]. (Alternatively you can work with the vector `LakeHuron` from the ts library, which 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.

5. Check the distributions of head lengths (`hdlngth`) in the `possum`[19] data set that accompanies these notes. 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?

6. Try `x <- rnorm(10)`. Print out the numbers that you get. Look up the help for `rnorm`. Now generate a sample of size 10 from a normal distribution with mean 170 and standard deviation 4.

7. Use `mfrow()` to set up the layout for a 3 by 4 array of plots. In the top 4 rows, show normal probability plots (section 4.4.2) 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.

[18] Source: Great Lakes Water Levels, 1860-1986. U.S. Dept. of Commerce, National Oceanic and AtmosphericAdministration, National Ocean Survey.

[19] 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.

8. 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?

*9. If you find exercise 8 interesting, you might like to try it for some further 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.

4.8 References

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

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.

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

5. Panel plots

The `coplot()` function is the equivalent of the S-PLUS trellis graphics function `xyplot( )`, but with a limitation to two conditioning factors or variables only. It allows R users access to a very limited form of trellis graphics. Trellis plots allow the use of the layout on the page to reflect meaningful aspects of data structure.

5.1 Examples that Present Panels of Scatterplots – Using `coplot()`

The function that draws panels of scatterplots is `coplot()`. We will use data from an experiment on the effects of the tinting of car windows on visual performance[20]. Data are in the data frame `tinting`. 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, while `target` (2 levels) is a factor and `tint` (3 levels) is an ordered factor. The variable `sex` is coded 1 for males and 2 for females, while the factor `agegp` is coded 1 for young people (all in their early 20s) and 2 for older participants (all in the early 70s).

```
coplot(csoa~it|tint+target,data=tinting)
coplot(csoa~it|tint+target,col=palette()[tinting$sex],data=tinting)
  # Use different colours to identify sex
coplot(csoa~it|tint+target, data=tinting, panel=panel.smooth)
coplot(csoa~it|tint+target, pch=tinting$agegp, data=tinting,
panel=panel.smooth)
  # Different colours for different agegroups, and show smooth
  # There seems no easy way to do separate smooths by agegroup
```

All four of the above commands plot `csoa` against `it` for each combination of `tint` and `target`. The second command uses different colours for the different colours for males and females. The third command adds a smooth. The fourth command uses different symbols for males and females, and a smooth. The graph given by this fourth and final command is shown in Fig. 12.

[20] 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.

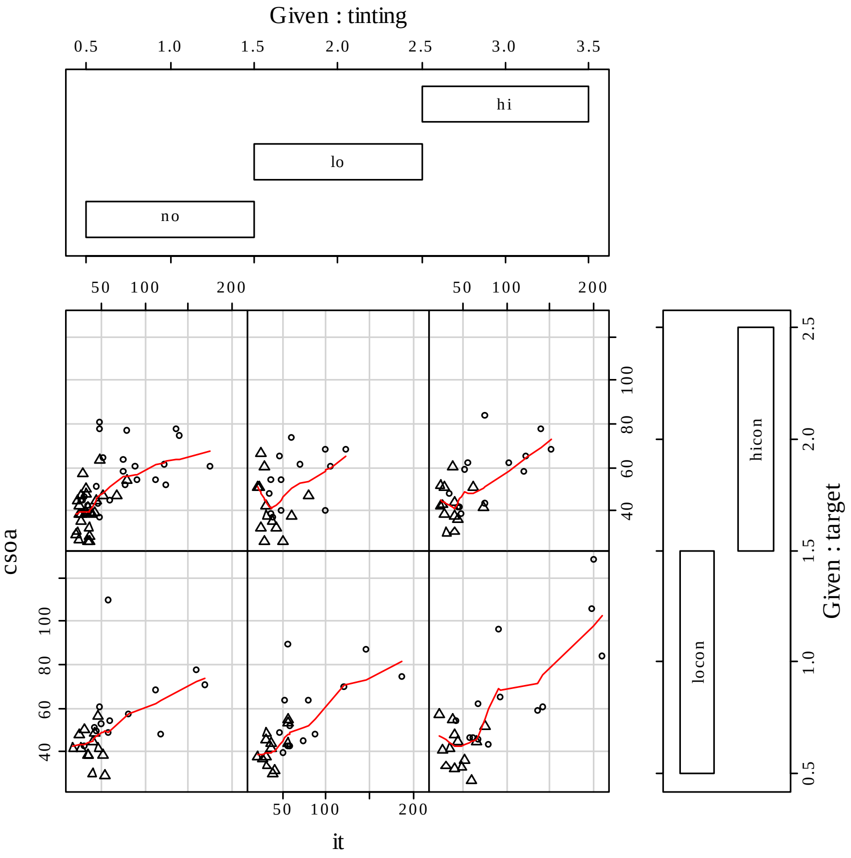


Fig. 12: The use of `coplot()`. For this graph, specify:
`coplot(csoa~it|tint+target, pch=tinting$agegp, data=tinting, panel=panel.smooth)`

Question: Why is it desirable to make `target` is an ordered factor, rather than simply a factor?

5.1.1 Using Ranges of Continuous Variables to Define Panels

Where conditioning is on a continuous variable, `coplot()` will break it down into ranges which in general overlap. The parameter `number` controls the number of ranges, and `overlap` controls the fraction of overlap. For example

```
coplot(time ~ distance | climb, data=hills, overlap=0.5, number=3)
```

By default `overlap` is 0.5, i. e. each successive pair of categories have around half their values in common.

5.2 The Panel Function

The panel function controls what appears in any panel. Users are able to supply their own.

5.3 Exercises

1. The following data gives milk volume (g/day) for smoking and nonsmoking mothers[21]:
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.

2. Repeat the plot as in exercise 1, but this time including a scatterplot smooth on each panel.

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

3. Taking the data frame `ships` that accompanies these notes, plot `incidents` against `service` for each level of `consyr` (construction period) and for each level of `period` (period of service).

4. Repeat the plot from exercise 4, but now plot log(`incidents`+1) against log(`service`), and use a different colour or plot symbol for each different `shiptype`.

5. For the possum data set, generate the following plots for each separate population (`Pop`) and for each sex (`sex`) separately:

a) histograms of `hdlngth` – use `histogram()`;

b) normal probability plots of `hdlngth` – use `qqmath()`;

c) density plots of `hdlngth` – use `densityplot()`.

The histogram function allows you to control the ratio of the y to x scales (`aspect`) and the number of intervals (`nint`). Investigate the effect of varying these. The `densityplot` function allows you to vary parameters `aspect` and `width`. The parameter `width` controls the width of the smoothing window. Investigate the effect of varying these paramters.

6. The following exercises relate to the data frame `possum` that accompanies these notes:

(a) Using the `coplot` function, explore the relation between `hdlngth` and `totlngth`, taking into account `sex` and `Pop`.

(b) Construct a contour plot of `chest` versus `belly` and `totlngth`.

(c) Construct box and whisker plots for `hdlngth`, using `site` as a factor.

(d) Construct normal probability plots for `hdlgth`, for each separate level of `sex` and `Pop`. Is there evidence that the distribution of `hdlgth` varies with the level of these other factors.

7. The frame `airquality` thas is in the base library 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.

6. Linear (Multiple Regression) Models and Analysis of Variance

6.1 The Model Formula in Straight Line Regression

We begin with the straight line regression example that appeared earlier, in section 2.1.4. First we will plot the data:

```
> plot(distance ~ stretch, data=elastic)
```

The code for the regression calculation is:

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

Here `distance ~ stretch` is a model formula. We will meet more general types of model formulae in the course of this chapter. Fig. 13 shows the plot:

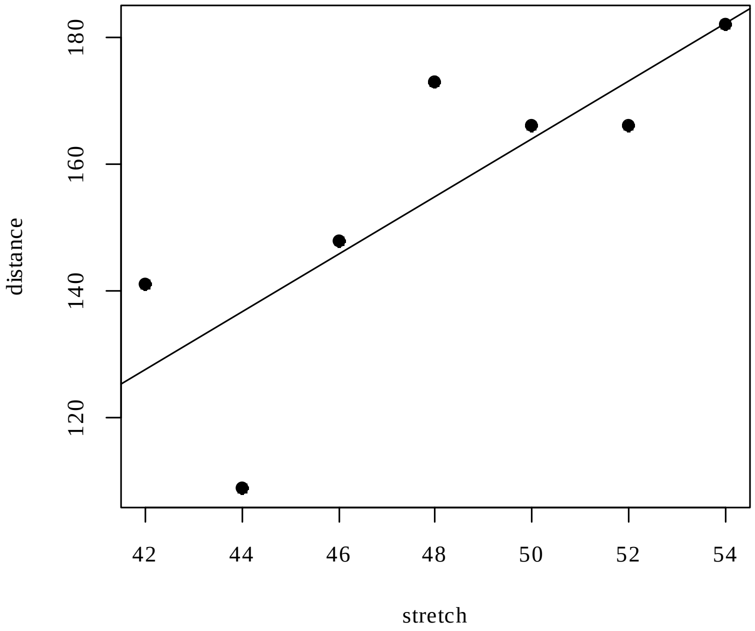


Fig. 13: Plot of distance (cm) versus stretch (mm) for the rubber band data, with fitted least squares line.

The output from the regression is an `lm` object, which we have called `elastic.lm`. Now examine a summary of the regression results. Notice that the documentation of the call gives details of the model formula.

```
> options(digits=4)
> summary(elastic.lm)

Call:
lm(formula = distance ~ stretch, data = band)

Residuals:
     1       2      3       4       5      6      7
  2.107  -0.321  18.000   1.893 -27.786  13.321  -7.214

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -63.57      74.33   -0.86    0.431
stretch         4.55       1.54    2.95    0.032
```

```
Residual standard error: 16.3 on 5 degrees of freedom
Multiple R-Squared: 0.635,      Adjusted R-squared: 0.562
F-statistic: 8.71 on 1 and 5 degrees of freedom,       p-value: 0.0319
```

6.2 Regression Objects

An `lm` object is a list of named elements. Above, we created the object `elastic.lm` . Here are the names of its elements:

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

Various functions are available for extracting information that you might want from the list. This is better than manipulating the list directly. Examples are:

```
> coef(elastic.lm)
(Intercept)     stretch
    -63.571       4.554
> resid(elastic.lm)
       1        2        3        4        5        6        7
  2.1071  -0.3214  18.0000   1.8929 -27.7857  13.3214  -7.2143
>
```

The function most often used to inspect regression output is `summary()`. It extracts the information that users are most likely to want. For example, in section 6.1, we had

```
summary(elastic.lm)
```

There is a plot method for `lm` objects. It gives diagnostic information. Fig. 13 shows the result of typing in:

```
par(mfrow = c(2, 2))
plot(elastic.lm)
```

By default the first, second and fourth plot use the row names to identify the three most extreme residuals. [If explicit row names are not given when the data frame is created, then the row numbers will be used.]

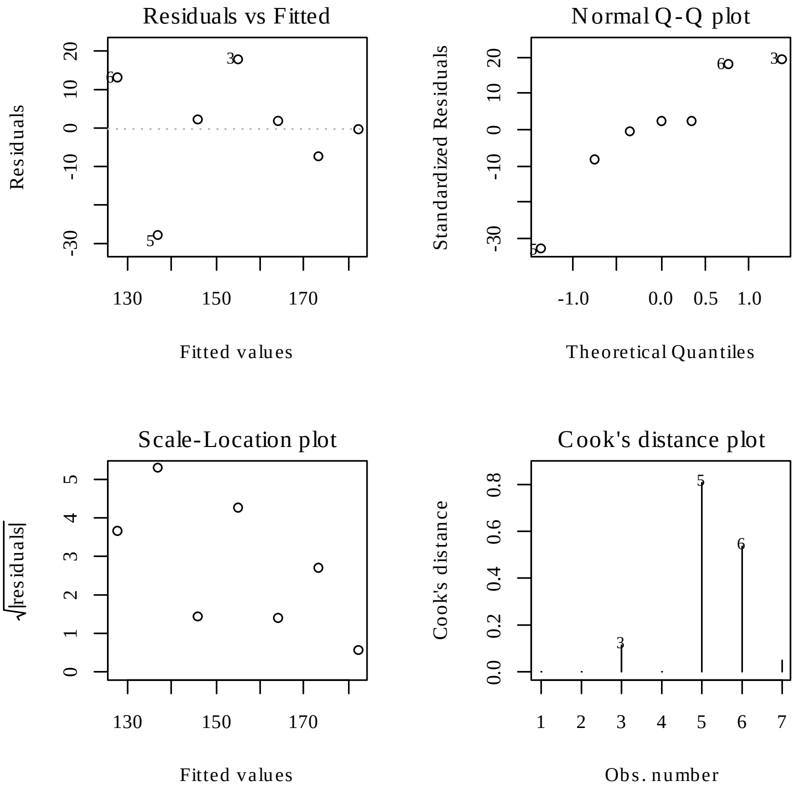


Fig. 14: Diagnostic plot of lm object, obtained by `plot(elastic.lm)`.

6.3 Model Formulae, and the X Matrix

The model formula for the elastic band example was `distance ~ stretch` . The model formula is a recipe for setting up the calculations. All the calculations described in this chapter require the use of an model matrix or X matrix, and a vector y of values of the dependent variable. For some of the examples we discuss later, it helps to know what the X matrix looks like. Details for the elastic band example follow.

The straight line model is

$y = a + b\,x$ + residual

which we write as

$y = 1 \times a + x \times b$ + residual

The X matrix, with the y-vector alongside, is:

X		y
	Stretch (mm)	Distance (cm)
1	46	148
1	54	182
1	48	173
1	50	166
1	44	109
1	42	141
1	52	166

The function `model.matrix()` prints out the model matrix. Thus:

```
> model.matrix(distance ~ stretch, data=elastic)
(Intercept) stretch
1           1      46
2           1      54
3           1      48
4           1      50
5           1      44
6           1      42
7           1      52
attr(,"assign")
[1] 0 1
>
```

Another possibility, with `elastic.lm` as in section 6.1, is:

```
> model.matrix(elastic.lm)
```

For each row, one takes some multiple of the value in the first column of the model matrix, another multiple of the value in the second column, and adds them, to give *fitted* values. Another name is *predicted* values. The aim is to reproduce, as closely as possible, the values in the y-column. The *residuals* are the differences between the values in the y-column and the fitted values. Least squares regression, which is the form of regression that we describe in this course, chooses *a* and *b* so that the sum of squares of the residuals is as small as possible.

The following are the fitted values and residuals that we get with the estimates of *a* (= -63.6) and *b* (= 4.55) that a least squares regression program chooses for us:

X		□	*y*	□
Stretch (mm)		*(Fitted)*	*(Observed)*	*(Residual)*
× -63.6	× 4.55	1 × -63.6 + 4.55 × Stretch	Distance (mm)	Observed - Fitted
1	46	-63.6 + 4.55 × 46 = 145.7	148	148-145.7 = 2.3
1	54	-63.6 + 4.55 × 54 = 182.1	182	182-182.1 = -0.1
1	48	-63.6 + 4.55 × 48 = 154.8	173	173-154.8 = 18.2
1	50	-63.6 + 4.55 × 50 = 163.9	166	166-163.9 = 2.1
1	44	-63.6 + 4.55 × 44 = 136.6	109	109-136.6 = -27.6
1	42	-63.6 + 4.55 × 42 = 127.5	141	141-127.5 = 13.5
1	52	-63.6 + 4.55 × 52 = 173.0	166	166-173.0 = -7.0

Note that we use □ [pronounced y-hat] as the symbol for predicted values.

We might alternatively fit the simpler (no intercept) model. For this we have

$$y = x \times b + e$$

where *e* is random variation with mean 0. The X matrix then consists of a single column, the x's.

6.3.1 Model Formulae in General

Here is what model formulae look like:

`y~x+z` : lm, glm,, etc.

`y~x+fac+fac:x` : lm, glm, aov, etc. (If `fac` is a factor and `x` is a variable, `fac:x` allows a different slope for each different level of `fac`.)

Model formulae are widely used to set up most of the model calculations in R.

The R parser[22] makes no distinction between model formulae and the sorts of formulae that are used for specifying coplots. The difference may matter once one tries to do something with the formula. By way of reminder, here is a graph formula for coplots.

```
y~x | fac1+fac2
```

This gives a plot of `y` against `x` for each different combination of levels of `fac1` (across the page) and `fac2` (up the page).

*6.3.2 Manipulating Model Formulae

Model formulae can be assigned, e. g.

```
formyxz <- formula(y~x+z)
```

or

```
formyxz <- formula("y~x+z")
```

[22] The parser is a part of the R implementation code. It takes R statements and turns them into code which can be more directly executed by the computer.

The argument to formula() can, as just demonstrated, be a text string. This makes it straightforward to paste the argument together from components that are stored in text strings.

For example

```
> names(elastic)
[1] "stretch"  "distance"
> nam_names(elastic)
> formds<-paste(nam[1],"~",nam[2])
> lm(formds,data=elastic)

Call:
lm(formula = formds, data = elastic)

Coefficients:
(Intercept)     distance
    26.3780       0.1395
```

6.4 Multiple Linear Regression Models

6.4.1 The data frame Rubber

The data set `Rubber` from the *MASS* library is from the accelerated testing of tyre rubber[23]. The variables are `loss` (the abrasion loss in gm/hr), `hard` (hardness in `Shore' units), and `tens` (tensile strength in kg/sq m).

We examine the scatterplot matrix (Fig. 15)

[23] The original source is O.L. Davies (1947) Statistical Methods in Research and Production. Oliver and Boyd, Table 6.1 p. 119.

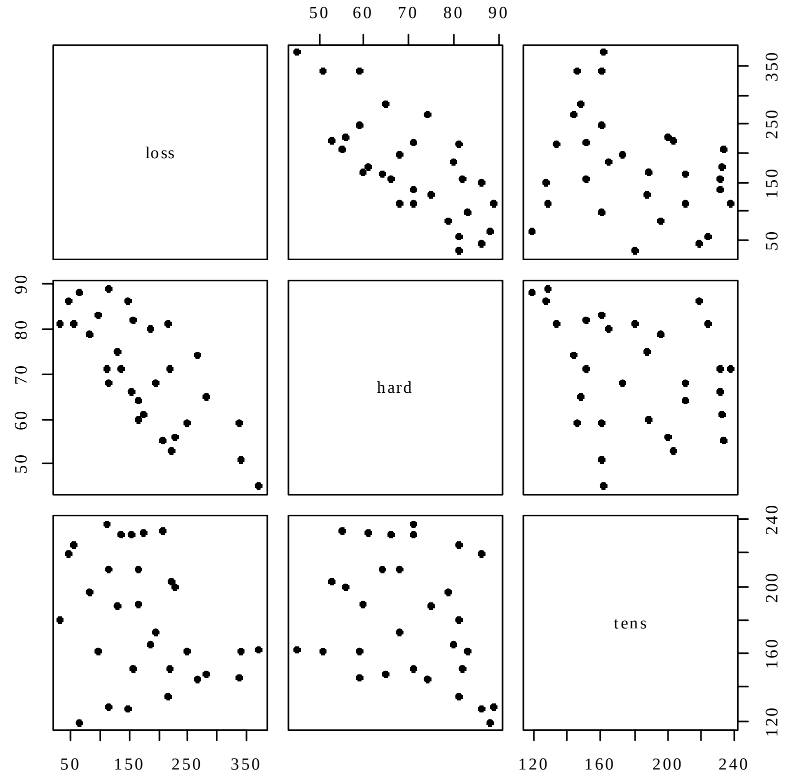


Fig. 15: Scatterplot matrix for the Rubber data frame.

There is a negative correlation between loss and hardness. We proceed to regress loss on `hard` and `tens`.

```
Rubber.lm <- lm(loss~hard+tens, data=Rubber)
> options(digits=3)
> summary(Rubber.lm)

Call:
lm(formula = loss ~ hard + tens, data = Rubber)

Residuals:
   Min     1Q Median     3Q    Max
-79.38 -14.61   3.82  19.75  65.98

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  885.161     61.752   14.33  3.8e-14
hard          -6.571      0.583  -11.27  1.0e-11
tens          -1.374      0.194   -7.07  1.3e-07

Residual standard error: 36.5 on 27 degrees of freedom
Multiple R-Squared: 0.84,      Adjusted R-squared: 0.828
F-statistic:   71 on 2 and 27 degrees of freedom,       p-value: 1.77e-011
```

The examination of diagnostic plots is left as an exercise.

6.4.2 Weights of Books

The books to which the data in the data set `oddbooks` (accompanying these notes) refer were chosen to cover a wide range of weight to height ratios. Here are the data:

```
> oddbooks
   thick height width weight
1     14   30.5  23.0   1075
2     15   29.1  20.5    940
3     18   27.5  18.5    625
4     23   23.2  15.2    400
5     24   21.6  14.0    550
6     25   23.5  15.5    600
7     28   19.7  12.6    450
8     28   19.8  12.6    450
9     29   17.3  10.5    300
10    30   22.8  15.4    690
11    36   17.8  11.0    400
12    44   13.5   9.2    250
```

Notice that as thickness increases, weight reduces.

```
> logbooks <- log(oddbooks) # We might expect weight to be
>                           # proportional to thick * height * width
> logbooks.lm1<-lm(weight~thick,data=logbooks)
> summary(logbooks.lm1)$coef
            Estimate Std. Error t value Pr(>|t|)
(Intercept)     9.69      0.708    13.7 8.35e-08
thick          -1.07      0.219    -4.9 6.26e-04

> logbooks.lm2<-lm(weight~thick+height,data=logbooks)
> summary(logbooks.lm2)$coef
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -1.263      3.552  -0.356   0.7303
thick          0.313      0.472   0.662   0.5243
height         2.114      0.678   3.117   0.0124

> logbooks.lm3<-lm(weight~thick+height+width,data=logbooks)
> summary(logbooks.lm3)$coef
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -0.719      3.216  -0.224    0.829
thick          0.465      0.434   1.070    0.316
height         0.154      1.273   0.121    0.907
width          1.877      1.070   1.755    0.117
```

So is `weight` proportional to `thick * height * width`?

The correlations between `thick`, `height` and `width` are so strong that if one tries to use more than one of them as a explanatory variables, the coefficients are ill-determined. They contain very similar information, as is evident from the scatterplot matrix. The regressions on `height` and `width` give plausible results, while the coefficient of the regression on `thick` is entirely an artefact of the way that the books were selected.

The design of the data collection really is important for the interpretation of coefficients from a regression equation. The design for the collection of these data was about as bad as it gets!

6.5 Polynomial and Spline Regression

We show how calculations that have the same structure as multiple linear regression may be used to model a curvilinear response. We build up curves from linear combinations of transformed values. A warning is that the use of polynomial curves of high degree are in general unsatisfactory. Spline curves, which are constructed by joining together low order polynomial curves (typically cubics) in such a way that the slope changes smoothly, are in general preferable.

6.5.1 Polynomial Terms in Linear Models

The data frame seedrates[24] that accompanies these notes gives, for each of a number of different seeding rates, the number of barley grain per head.

```
> plot(grains ~ rate, data=seedrates)   # Plot the data
```

Fig. 16 shows the data, with fitted quadratic curve:

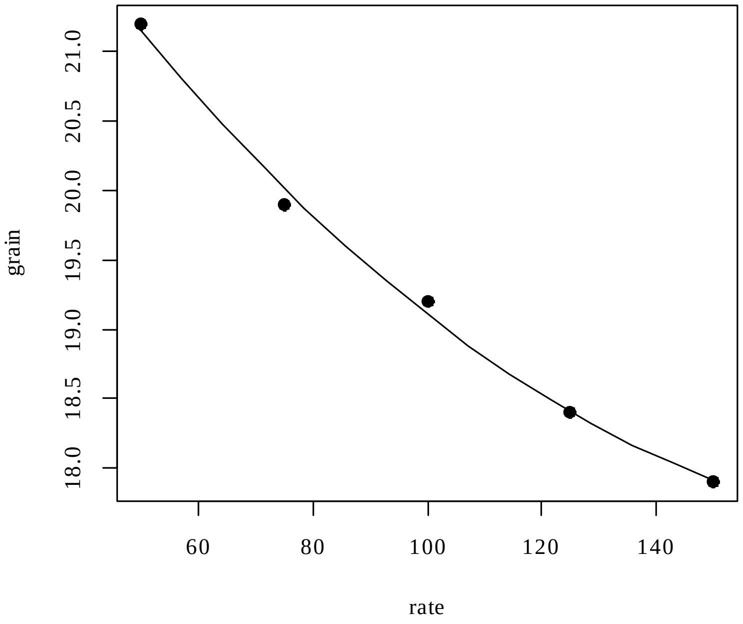


Fig. 16: Plot of number of grain per head versus seeding rate, for the barley seeding rate data, with fitted quadratic curve.

We will need an X-matrix with a column of ones, a column of values of rate, and a column of values of rate^2. We can achieve this by putting both rate and I(rate^2) into the model formula.

```
> seedrates.lm2<-lm(grain~rate+I(rate^2),data=seedrates)
> summary(seedrates.lm2)

Call:
lm(formula = grain ~ rate + I(rate^2), data = seedrates)

Residuals:
      1        2        3        4        5
 0.04571 -0.12286  0.09429 -0.00286 -0.01429
```

[24] Data are from McLeod, C. C. (1982) Effect of rates of seeding on barley grown for grain. New Zealand Journal of Agriculture 10: 133-136. Summary details are in Maindonald, J. H. (1992).

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 24.060000   0.455694   52.80  0.00036
rate        -0.066686   0.009911   -6.73  0.02138
I(rate^2)    0.000171   0.000049    3.50  0.07294

Residual standard error: 0.115 on 2 degrees of freedom
Multiple R-Squared: 0.996,      Adjusted R-squared: 0.992
F-statistic:  256 on 2 and 2 degrees of freedom,        p-value: 0.0039
> hat <- predict(seedrates.lm2)
> lines(spline(seedrates$rate, hat))
> # Placing the spline fit through the fitted points allows a smooth curve.
> # For this to work the values of seedrates$rate must be ordered.
>
```

Again, check the form of the model matrix. Type in:

```
> model.matrix(grain~rate+I(rate^2),data=seedrates)
  (Intercept) rate I(rate^2)
1           1   50      2500
2           1   75      5625
3           1  100     10000
4           1  125     15625
5           1  150     22500
attr(,"assign")
[1] 0 1 2
```

This example demonstrates a way to extend linear models to handle specific types of non-linear relationships. We can use any transformation we wish to form columns of the model matrix. We could, if we wished, add an x^3 column.

Once the model matrix has been formed, we are limited to taking linear combinations of columns.

6.5.2 What order of polynomial?

A polynomial of degree 2, i. e. a quadratic curve, looked about right for the above data. How does one check?

One way is to fit polynomials, e. g. of each of degrees 1 and 2, and compare them thus:

```
> seedrates.lm1<-lm(grain~rate,data=seedrates)
> seedrates.lm2<-lm(grain~rate+I(rate^2),data=seedrates)
> anova(seedrates.lm2,seedrates.lm1)
Analysis of Variance Table

Model 1: grain ~ rate + I(rate^2)
Model 2: grain ~ rate
  Res.Df Res.Sum Sq Df    Sum Sq F value  Pr(>F)
1      2   0.026286
2      3   0.187000 -1 -0.160714  12.228 0.07294
```

The F-value is large, but on this evidence there are too few degrees of freedom to make a totally convincing case for preferring a quadratic to a line. However the paper from which these data come gives an independent estimate of the error mean square (0.17 on 35 d.f.) based on 8 replicate results that were averaged to give each value for number of grains per head. If we compare the change in the sum of squares (0.1607, on 1 df) with a mean square of 0.17^2 (35 df), the F-value is now 5.4 on 1 and 35 degrees of freedom, and we have p=0.024 . The increase in the number of degrees of freedom more than compensates for the reduction in the F-statistic.

```
> # However we have an independent estimate of the error mean
> # square. The estimate is 0.17^2, on 35 df.
> 1-pf(0.16/0.17^2, 1, 35)
[1] 0.0244
```

Finally note that R^2 was 0.972 for the straight line model. This may seem good, but given the accuracy of these data it was not good enough! The statistic is an inadequate guide to whether a model is adequate. Even for any one context, R^2 will in general increase as the range of the values of the dependent variable increases. (R^2 is larger when there is more variation to be explained.) A predictive model is adequate when the standard errors of predicted values are acceptably small, not when R^2 achieves some magic threshold.

6.5.3 Spline Terms in Linear Models

By now, readers of this document will be used to the idea that it is possible to use linear models to fit terms that may be highly nonlinear functions of one or more variables. The fitting of polynomial functions was a simple example of this. Spline functions variables extend this idea further. The splines that I demonstrate are constructed by joining together cubic curves, in such a way the joins are smooth. The places where the cubics join are known as `knots'. It turns out that, once the knots are fixed, and depending on the class of spline curves that are used, spline functions of a variable can be constructed as a linear combination of basis functions, where each basis function is a transformation of the variable.

The data frame `cars` is in the base library.

```
> data(cars)
> plot(dist~speed,data=cars)
> library(splines)
> cars.lm<-lm(dist~bs(speed),data=cars)  # By default, there are no knots
> hat<-predict(cars.lm)
> lines(cars$speed,hat,lty=3)  # NB assumes values of speed are sorted
> cars5.lm<-lm(dist~bs(speed,5),data=cars) # try for a closer fit (1 knot)
> ci5<-predict(cars.lm5,interval="confidence",se.fit=T)
> names(ci5)
[1] "fit"           "se.fit"        "df"            "residual.scale"
> lines(cars$speed,ci5$fit[,"fit"])
> lines(cars$speed,ci5$fit[,"lwr"],lty=2)
> lines(cars$speed,ci5$fit[,"upr"],lty=2)
>
```

6.6 Using Factors in R Models

Factors are essential, when there are categorical or "factor" variables, for specifying R models. Consider data from an experiment that compared houses with and without cavity insulation[25]. While one would not usually handle these calculations using an `lm` model, it makes a simple example to illustrate the choice of a baseline level, and a set of contrasts. Different choices, although they fit equivalent models, give output in which some of the numbers are different and must be interpreted differently.

We begin by entering the data from the command line:

```
> insulation <- factor(c(rep("without", 8), rep("with", 7)))
      # 8 without, then 7 with
```

[25] Data are from Hand, D. J.; Daly, F.; Lunn, A. D.; Ostrowski, E., eds. (1994). A Handbook of Small Data Sets. Chapman and Hall.

```
    # `with’ precedes `without’ in alphanumeric order, & is the baseline
> kWh <- c(10225, 10689, 14683, 6584, 8541, 12086, 12467,
         12669, 9708, 6700, 4307, 10315, 8017, 8162, 8022)
```

To formulate this as a regression model, we take kWh as the dependent variable, and the factor insulation as the explanatory variable.

```
> insulation <- factor(c(rep("without", 8), rep("with", 7)))
> # 8 without, then 7 with
> kWh <- c(10225, 10689, 14683, 6584, 8541, 12086, 12467,
+ 12669, 9708, 6700, 4307, 10315, 8017, 8162, 8022)
> insulation.lm <- lm(kWh ~ insulation)
> summary(insulation.lm, corr=F)

Call:
lm(formula = kWh ~ insulation)

Residuals:
   Min     1Q Median     3Q    Max
 -4409   -979    132   1575   3690

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)     7890        874    9.03  5.8e-07
insulation      3103       1196    2.59    0.022

Residual standard error: 2310 on 13 degrees of freedom
Multiple R-Squared: 0.341,      Adjusted R-squared: 0.29
F-statistic: 6.73 on 1 and 13 degrees of freedom,       p-value: 0.0223
```

The p-value is 0.022, which may be taken to indicate ($p < 0.05$) that we can distinguish between the two types of houses. But what does the “intercept” of 7890 mean, and what does the value for “insulation” of 3103 mean? To interpret this, we need to know that the factor levels are, by default, taken in alphabetical order, and that the initial level is taken as the baseline. So `with` comes before `without`, and `with` is the baseline. Hence:

Average for Insulated Houses = 7980

To get the estimate for uninsulated houses take 7980 + 3103 = 10993.

The standard error of the difference is 1196.

6.6.1 The Model Matrix

It often helps to think in terms of the model matrix or X matrix. Here are the X and the y that are used for the calculations. Note that the first eight data values were all `without`s:

Contrast	kWh

× 7980	× 3103	Add to get	Compare with	Residual
1	1	7980+3103=10993	10225	10225-10993
1	1	7980+3103=10993	10689	10689-10993
1	1	7980+3103=10993	14683	etc.
1	1	7980+3103=10993	6584	
1	1	7980+3103=10993	8541	
1	1	7980+3103=10993	12086	
1	1	7980+3103=10993	12467	
1	1	7980+3103=10993	12669	
1	0	7980+0	9708	9708-7980
1	0	7980+0	6700	6700-7980
1	0	7980+0	4307	etc.
1	0	7980+0	10315	
1	0	7980+0	8017	
1	0	7980+0	8162	
1	0	7980+0	8022	

Type in

```
model.matrix(kWh~insulation)
```

and check that you get the above model matrix.

*6.6.2 Other Choices of Contrasts

There are other ways to set up the X matrix. In technical jargon, there are other contrasts that one can choose. One obvious alternative is to make `without` the first factor level, so that it becomes the baseline. You can do this in the following way:

```
insulation <- factor(insulation, labels=c("without", "with"))
  # Make `without’ the baseline
```

Another possibility is to use what are called the “sum” contrasts. With the “sum” contrasts the baseline is the mean over all factor levels. The effect for the first level is omitted; the user has to calculate it as minus the sum of the remaining effects. Here is the output you get if you use the `sum’ contrasts[26]:

```
> options(contrasts = c("contr.sum", "contr.poly"), digits = 2)
   #  Try the `sum’ contrasts
> insulation <- factor(insulation, levels=c("without", "with"))
   # Make `without' the baseline
> insulation.lm <- lm(kWh ~ insulation)
> summary(insulation.lm, corr=F)

Call:
```

[26] The second string element, i. e. `"contr.poly"`, is the default setting for factors with ordered levels. [One uses the function ordered() to create ordered factors.]

```
lm(formula = kWh ~ insulation)

Residuals:
   Min     1Q Median     3Q    Max
 -4409   -979    132   1575   3690

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)     9442        598   15.78  7.4e-10
insulation      1551        598    2.59    0.022

Residual standard error: 2310 on 13 degrees of freedom
Multiple R-Squared: 0.341,      Adjusted R-squared: 0.29
F-statistic: 6.73 on 1 and 13 degrees of freedom,   p-value: 0.0223
```

Here is the interpretation:

average of (mean for "without", "mean for with") = 9442

To get the estimate for uninsulated houses (the first level), take 9442 + 1551 = 10993

The `effects' sum to one. So the effect for the second level (`with') is -1551. Thus

to get the estimate for insulated houses (the first level), take 9442 - 1551 = 7980.

The sum contrasts are sometimes called "analysis of variance" contrasts.

You can set the choice of contrasts for each factor separately, with a statement such as:

```
> insulation <- C(insulation, contr=treatment)
```

Also available are the Helmert contrasts. These are not at all intuitive, even though S-PLUS uses them as the default. Novices should avoid them[27].

6.7 Multiple Lines – Different Regression Lines for Different Species

The terms which appear on the right of the model formula may be variables or factors, or interactions between variables and factors, or interactions between factors. Here we take advantage of this to fit different lines to different subsets of the data.

In the example which follows, we had weights for a porpoise species (*Stellena styx*) and for a dolphin species (*Delphinus delphis*). We take x_1 to be a variable which has the value 0 for *Delphinus delphis*, and 1 for *Stellena styx*. We take x_2 to be body weight. Then possibilities we may want to consider are:

A: A single line: $y = a + b\, x_2$

B: Two parallel lines: $y = a_1 + a_2\, x_1 + b\, x_2$
[For the first group (*Stellena styx;* $x_1 = 0$) the constant term is a_1, while for the second group (*Delphinus delphis;* $x_1 = 1$) the constant term is $a_1 + a_2$.]

C: Two separate lines: $y = a_1 + a_2\, x_1 + b_1\, x_2 + b_2\, x_1\, x_2$
[For the first group (*Delphinus delphis;* $x_1 = 0$) the constant term is a_1 and the slope is b_1. For the second group (*Stellena styx;* $x_1 = 1$) the constant term is $a_1 + a_2$, and the slope is $b_1 + b_2$.]

[27] The interpretation of the helmert contrasts is simple enough when there are just two levels. With >2 levels, the helmert contrasts give parameter estimates which in general do not make a lot of sense, basically because the baseline keeps changing, to the average for all previous factor levels. You do better to use either the treatment contrasts, or the sum contrasts. With the sum contrasts the baseline is the overall mean.

S-PLUS makes helmert contrasts the default, perhaps for reasons of computational efficiency. This was an unfortunate choice.

We show results from fitting the first two of these models, i. e. A and B:

```
> plot(logheart ~ logweight, data=dolphins)  # Plot the data
> options(digits=4)
> cet.lm1 <- lm(logheart ~ logweight, data = dolphins)
> summary(cet.lm1, corr=F)

Call:
lm(formula = logheart ~ logweight, data = dolphins)

Residuals:
     Min       1Q   Median       3Q      Max
-0.15874 -0.08249  0.00274  0.04981  0.21858

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    1.325      0.522    2.54    0.024
logweight      1.133      0.133    8.52  6.5e-07

Residual standard error: 0.111 on 14 degrees of freedom
Multiple R-Squared: 0.838,      Adjusted R-squared: 0.827
F-statistic: 72.6 on 1 and 14 degrees of freedom,   p-value: 6.51e-007
>
```

For model B (parallel lines) we have

```
> cet.lm2 <- lm(logheart ~ species + logweight, data=dolphins)
```

Check what the model matrix looks like:

```
> model.matrix(cet.lm2)
   (Intercept) factor(species) logweight
1            1               1     3.555
2            1               1     3.738
3            1               1     4.263
4            1               1     4.174
5            1               1     4.143
6            1               1     4.159
7            1               1     3.807
8            1               0     3.989
9            1               0     4.078
10           1               0     3.912
11           1               0     3.738
12           1               0     4.007
13           1               0     3.611
14           1               0     3.850
15           1               0     3.689
16           1               0     3.951
attr(,"assign")
[1] 0 1 2
attr(,"contrasts")
[1] "contr.treatment"
```

Now look at an output summary:

```
> summary(cet.lm2)

Call:
lm(formula = logheart ~ factor(species) + logweight, data = dolphins)

Residuals:
    Min      1Q  Median      3Q     Max
-0.1163 -0.0649 -0.0114  0.0606  0.1282

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)       1.6052     0.4139    3.88   0.0019
factor(species)   0.1435     0.0448    3.21   0.0069
logweight         1.0458     0.1067    9.80  2.3e-07

Residual standard error: 0.0859 on 13 degrees of freedom
Multiple R-Squared: 0.91,       Adjusted R-squared: 0.896
F-statistic: 65.5 on 2 and 13 degrees of freedom,   p-value: 1.62e-007
>
> plot(cet.lm2)  # Plot diagnostic information for the model just fitted.
```

For model C, the statement is:

```
> cet.lm3 <- lm(logheart ~ factor(species) + logweight +
    factor(species):logweight, data=dolphins)
```

Check what the model matrix looks like:

```
> model.matrix(cet.lm3)
   (Intercept) factor(species) logweight factor(species).logweight
1            1               1     3.555                     3.555
2            1               1     3.738                     3.738
3            1               1     4.263                     4.263
4            1               1     4.174                     4.174
5            1               1     4.143                     4.143
6            1               1     4.159                     4.159
7            1               1     3.807                     3.807
8            1               0     3.989                     0.000
9            1               0     4.078                     0.000
10           1               0     3.912                     0.000
11           1               0     3.738                     0.000
12           1               0     4.007                     0.000
13           1               0     3.611                     0.000
14           1               0     3.850                     0.000
15           1               0     3.689                     0.000
16           1               0     3.951                     0.000
attr(,"assign")
[1] 0 1 2 3
attr(,"contrasts")$"factor(species)"
[1] "contr.treatment"
```

Now see why one should not waste time on model C.

```
> anova(cet.lm1,cet.lm2,cet.lm3)
Analysis of Variance Table
```

```
Model 1: logheart ~ logweight
Model 2: logheart ~ factor(species) + logweight
Model 3: logheart ~ factor(species) + logweight + factor(species):logweight
  Res.Df Res.Sum Sq Df Sum Sq F value Pr(>F)
1     14     0.1717
2     13     0.0959  1 0.0758   10.28 0.0069
3     12     0.0949  1 0.0010    0.12 0.7346
>
```

6.8 aov models (Analysis of Variance)

The class of models which can be directly fitted as `aov` models is quite limited. In essence, `aov` provides, for data where all combinations of factor levels have the same number of observations, another view of an `lm` model. One can however specify the error term that is to be used in testing for treatment effects. See section 6.8.2 below.

By default, R uses the treatment contrasts for factors, i. e. the first level is taken as the baseline or reference level. A useful function is `relevel()`. The parameter `ref` can be used to set the level that you want as the reference level.

6.8.1 Plant Growth Example

Here is a simple randomised block design:

```
> data(PlantGrowth)              # From the MASS library
> attach(PlantGrowth)
> boxplot(split(weight,group))    # Looks OK
> data()
> PlantGrowth.aov<-aov(weight~group)
> summary(PlantGrowth.aov)
            Df  Sum Sq Mean Sq F value  Pr(>F)
group        2  3.7663  1.8832  4.8461 0.01591
Residuals   27 10.4921  0.3886
> summary.lm(PlantGrowth.aov)

Call:
aov(formula = weight ~ group)

Residuals:
    Min      1Q  Median      3Q     Max
-1.0710 -0.4180 -0.0060  0.2627  1.3690

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.0320     0.1971  25.527   <2e-16
grouptrt1    -0.3710     0.2788  -1.331   0.1944
grouptrt2     0.4940     0.2788   1.772   0.0877

Residual standard error: 0.6234 on 27 degrees of freedom
Multiple R-Squared: 0.2641,     Adjusted R-squared: 0.2096
F-statistic: 4.846 on 2 and 27 degrees of freedom,      p-value: 0.01591
```

```
> help(cabbages)
> data(cabbages)                  # From the MASS library
> names(cabbages)
[1] "Cult"   "Date"   "HeadWt" "VitC"
> coplot(HeadWt~VitC|Cult+Date,data=cabbages)
```

Examination of the plot suggests that cultivars differ greatly in the variability in head weight. Variation in the vitamin C levels seems relatively consistent between cultivars.

```
> VitC.aov<-aov(VitC~Cult+Date,data=cabbages)
> summary(VitC.aov)
            Df  Sum Sq Mean Sq F value    Pr(>F)
Cult         1 2496.15 2496.15 53.0411 1.179e-09
Date         2  909.30  454.65  9.6609 0.0002486
Residuals   56 2635.40   47.06
>
```

*6.8.2 Shading of Kiwifruit Vines

These data (yields in kilograms) are in the data frame `kiwishade` which accompanies these notes. They are from an experiment[28] where there were four treatments - no shading, shading from August to December, shading from December to February, and shading from February to May. Each treatment appeared once in each of the three blocks. The northernmost plots were grouped in one block because they were similarly affected by shading from the sun. For the remaining two blocks shelter effects, in one case from the east and in the other case from the west, were thought more important. Results are given for each of the four vines in each plot. In experimental design parlance, the four vines within a plot constitute subplots.

The `block:shade` mean square (sum of squares divided by degrees of freedom) provides the error term. (If this is not specified, one still gets a correct analysis of variance breakdown. But the F-statistics and p-values will be wrong.)

```
> kiwishade$shade <- relevel(kiwishade$shade, ref="none")
> ## Make sure that the level “none” (no shade) is used as reference
> kiwishade.aov<-aov(yield~block+shade+Error(block:shade),data=kiwishade)
> summary(kiwishade.aov)

Error: block:shade
          Df  Sum Sq Mean Sq F value   Pr(>F)
block      2  172.35   86.17  4.1176 0.074879
shade      3 1394.51  464.84 22.2112 0.001194
Residuals  6  125.57   20.93

Error: Within
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals 36 438.58   12.18
> coef(kiwishade.aov)
(Intercept) :
(Intercept)
```

[28] Data relate to the paper: Snelgar, W.P., Manson. P.J., Martin, P.J. 1992. Influence of time of shading on flowering and yield of kiwifruit vines. Journal of Horticultural Science 67: 481-487.
Further details, including a diagram showing the layout of plots and vines and details of shelter, are in Maindonald (1992). The two papers have different shorthands (e.g. Sept-Nov versus Aug-Dec) for describing the time periods for which the shading was applied.

```
    96.5327

block:shade :
  blocknorth    blockwest shadeAug2Dec shadeDec2Feb shadeFeb2May
    0.993125    -3.430000     3.030833   -10.281667    -7.428333

Within :
numeric(0)
```

6.9 Exercises

1. Here are two sets of data that were obtained the same apparatus, including the same rubber band, as the data frame elastic. For the data set `elastic1`, the values are:
`stretch` (mm): 46, 54, 48, 50, 44, 42, 52
`distance` (cm): 183, 217, 189, 208, 178, 150, 249.

For the data set `elastic2`, the values are:
`stretch` (mm): 25, 45, 35, 40, 55, 50 30, 50, 60
`distance` (cm): 71, 196, 127, 187, 249, 217, 114, 228, 291.

Using a different symbol and/or a different colour, plot the data from the two data frames `elastic1` and `elastic2` on the same graph. Do the two sets of results appear consistent.

2. For each of the data sets `elastic1` and `elastic2`, determine the regression of stretch on distance. In each case determine (i) fitted values and standard errors of fitted values and (ii) the R^2 statistic. Compare the two sets of results. What is the key difference between the two sets of data?

3. Use the method of section 6.7 to determine, formally, whether one needs different regression lines for the two data frames `elastic1` and `elastic2`.

4. Using the data frame `cars` (in the base library), plot `distance` (i. e. stopping distance) versus `speed`. Fit a line to this relationship, and plot the line. Then try fitting and plotting a quadratic curve. Does the quadratic curve give a useful improvement to the fit? If you have studied the dynamics of particles, can you find a theory that would tell you how stopping distance might change with speed?

5. Using the data frame `hills` (in library *MASS*), regress `time` on `distance` and `climb`. What can you learn from the diagnostic plots which you get when you plot the `lm` object? Try also regressing `log(time)` on `log(distance)` and `log(climb)`. Which of these regression equations would you prefer?

6. Using the data frame `beams` (in the data sets accompanying these notes), carry out a regression of `strength` on `SpecificGravity` and `Moisture`. Carefully examine the regression diagnostic plot, obtained by supplying the name of the `lm` object as the first parameter to `plot()`. What does this indicate?

7. Type

```
> hosp<-rep(c(”RNC”,”Hunter”,”Mater”),2)
> hosp
> fhosp<-factor(hosp)
> levels(fhosp)
```

Now repeat the steps involved in forming the factor `fhosp`, this time keeping the factor levels in the order `RNC`, `Hunter`, `Mater`.

Use `contrasts(fhosp)` to form and print out the matrix of contrasts. Do this using helmert contrasts, treatment contrasts, and sum contrasts. Using an outcome variable

```
y <- c(2,5,8,10,3,9)
```

fit the model `lm(y~fhosp),` repeating the fit for each of the three different choices of contrasts. Comment on what you get.

For which choice(s) of contrasts do the parameter estimates change when you re-order the factor levels?

8. In section 6.7 check the form of the model matrix (i) for fitting two parallel lines and (ii) for fitting two arbitrary lines when one uses the *sum* contrasts. Repeat the exercise for the *helmert* contrasts.

9. In the data set `cement` (*MASS* library), examine the dependence of *y* (amount of heat produced) on *x1*, *x2*, *x3* and *x4* (which are proportions of four constituents). Begin by examining the scatterplot matrix. As the explanatory variables are proportions, do they require transformation, perhaps by taking log(*x*/(100-*x*))? What alternative strategies one might use to find an effective prediction equation?

10. In the data set `pressure` (*base* library), examine the dependence of pressure on temperature. [Transformation of temperature makes sense only if one first converts to degrees Kelvin. Consider transformation of pressure. A logarithmic transformation is too extreme; the direction of the curvature changes. What family of transformations might one try?

*11. Repeat the analysis of the `kiwishade` data (section 6.8.2), but replacing `Error(block:shade)` with `block:shade`. Comment on the output that you get from `summary()`. To what extent is it potentially misleading? Also do the analysis where the `block:shade` term is omitted altogether. Comment on that analysis.

6.10 References

Atkinson, A. C. 1986. Comment: Aspects of diagnostic regression analysis. Statistical Science 1, 397–402.

Atkinson, A. C. 1988. Transformations Unmasked. Technometrics 30: 311-318.

Cook, R. D. and Weisberg, S. 1999. Applied Regression including Computing and Graphics. Wiley.

Harrell, F. E., Lee, K. L., and Mark, D. B. 1996. Tutorial in Biostatistics. Multivariable Prognostic Models: Issues in Developing Models, Evaluating Assumptions and Adequacy, and Measuring and Reducing Errors. Statistics in Medicine 15: 361-387.

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

Venables, W. N. and Ripley, B. D., 2nd edn 1997. Modern Applied Statistics with S-Plus. Springer, New York.

Weisberg, S., 2nd edn, 1985. Applied Linear Regression. Wiley.

Williams, G. P. 1983. Improper use of regression equations in the earth sciences. Geology 11: 195-197

7. Multivariate and Tree-Based Methods

7.1 Multivariate EDA, and Principal Components Analysis

Principal components analysis is often a useful exploratory tool for multivariate data. The idea is to replace the variables with which we start by a small number of "principal components" which together may explain most of the variation in the data. The first principal component is the component (linear combination of the variables with which we started) that explains the greatest part of the variation. The second principal component is the component that, among linear combinations of the variables that are uncorrelated with the first principal component, explains the greatest part of the variation, and so on.

The data set `possum` that accompanies these notes has nine morphometric measurements on each of 102 mountain brushtail possums, trapped at seven sites from southern Victoria to central Queensland[29]. It is good practice to begin by examining relevant scatterplot matrices. This may draw attention to gross errors in the data. A plot in which the sites and/or the sexes are identified will draw attention to any very strong structure in the data. For example one site may be quite different from the others, for some or all of the variables.

Here are some of the scatterplot matrix possibilities:

```
pairs(possum[,6:14], col=palette()[as.integer(possum$sex)])
pairs(possum[,6:14], col=palette()[as.integer(possum$site)])
here<-!is.na(possum$footlgth)    # We need to exclude missing values
print(sum(!here))          # Check how many values are missing
```

We now look at particular views of the data that we get from a principal components analysis:

```
library(mva)               # Load x-variate analysis library
possum.prc <- princomp(possum[here,6:14])  # Principal components
# Print scores on second pc versus scores on first pc,
# by populations and sex, identified by site
coplot(possum.prc$scores[,2] ~
  possum.prc$scores[,1]|possum$Pop[here]+possum$sex[here],
  col=palette()[as.integer(possum$site)])
```

Fig. 17, which uses different plot symbols for different sites, was produced using:

```
coplot(possum.prc$scores[,2] ~
  possum.prc$scores[,1]|possum$Pop[here]+possum$sex[here],
  pch=as.integer(possum$site))
```

[29] Data relate to the paper: Lindenmayer, D. B., Viggers, K. L., Cunningham, R. B., and Donnelly, C. F. 1995. Morphological variation among columns of the mountain brushtail possum, *Trichosurus caninus* Ogilby (Phalangeridae: Marsupiala). Australian Journal of Zoology 43: 449-458.

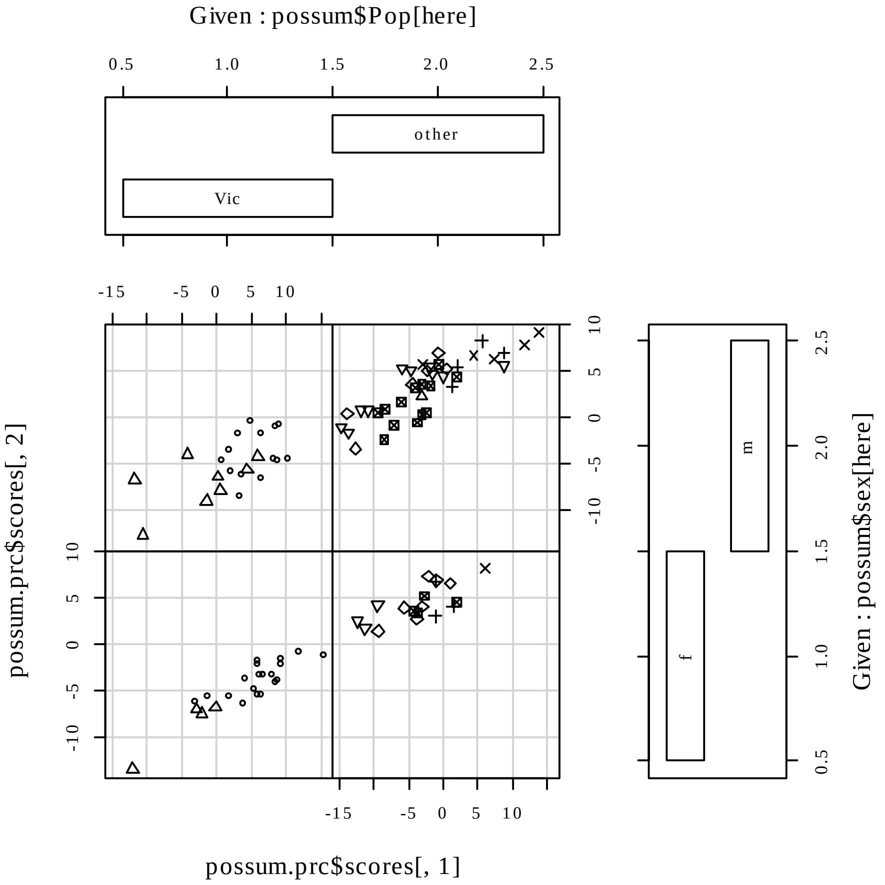


Fig. 17: Second principal component versus first principal component, by population and by sex, for the possum data.

7.2 Cluster Analysis

In the language of Ripley (1996)[30], cluster analysis is a form of unsupervised classification. It is "unsupervised" because the clusters are not known in advance. There are two types of algorithms – algorithms based on *hierachical agglomeration*, and algorithms based on *iterative relocation*.

In *hierarchical agglomeration* each observation starts as a separate group. Groups that are "close" to one another are then successively merged. The output yields a hierarchical clustering tree which shows the relationships between observations and between the clusters into which they are successively merged. A judgement is then needed on the point at which further merging is unwarranted.

In *iterative relocation*, the algorithm starts with an initial classification, which it then tries to improve. How does one get the initial classification? Typically, by a prior use of a hierarchical agglomeration algorithm.

The *mva* library has the cluster analysis routines. The function dist() calculates distances. The function hclust() does hierarchical agglomerative clustering, with a choice of methods available. The function kmeans() (k-means clustering) implements iterative relocation.

7.3 Discriminant Analysis

We start with data which are classified into several groups, and want a rule which will allow us to predict the group to which a new data value will belong. In the language of Ripley (1996), our interest is in supervised classification. For example, we may wish to predict, based on prognostic measurements and outcome information for previous patients, which future patients will remain free of disease symptoms for twelve months

[30] References are at the end of the chapter.

or more. Here are calculations for the `possum` data frame, using the `lda()` function from the Venables & Ripley *MASS* library:

```
> library(mass)                 # Only if not already attached.
> here<- !is.na(possum$footlgth)
> possum.lda <- lda(site~hdlngth+skullw+totlngth+
+ taill+footlgth+earconch+eye+chest+belly,data=possum,
+ subset=here)
> options(digits=4)
> possum.lda$svd   # Examine the singular values
[1] 15.7578  3.9372  3.1860  1.5078  1.1420  0.7772
>
> plot(possum.lda, dimen=3)
> # Scatterplot matrix for scores on 1st 3 canonical variates, as in Fig. 18
```

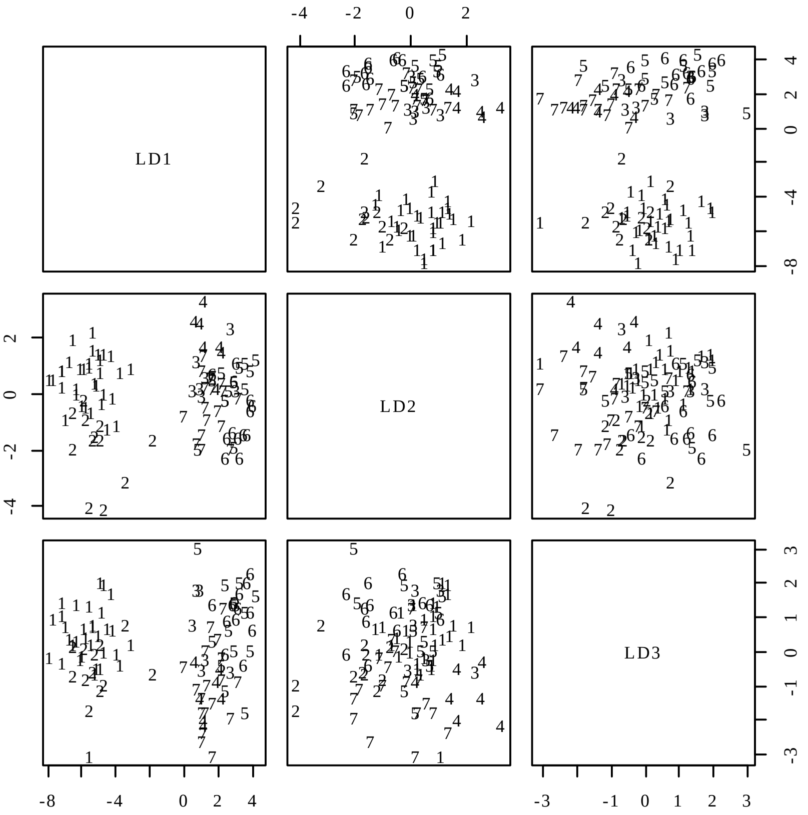


Fig. 18: Scatterplot matrix for the first 3 canonical variates.

The singular values are the ratio of between to within group sums of squares, for the canonical variates in turn. Clearly canonical variates after the third will have little if any discriminatory power. One can use `predict.lda()` to get (among other information) scores on the first few canonical variates.

Where there are two groups, logistic regression is often effective. Perhaps the best source of code for handling more general supervised classification problems is Hastie and Tibshirani's `mda` (mixture discriminant analysis) library. There is a brief overview of this library in the Venables and Ripley `Complements', referred to in section 13.2 .

7.4 Decision Tree models (Tree-based models)

We include tree-based classification here because it is a multivariate supervised classification, or discrimination, method. A tree-based regression approach is available for use for regression problems. Tree-based methods seem more suited to binary regression and classification than to regression with an ordinal or continuous dependent variable.

Tree-based models, also known as "Classification and Regression Trees" (CART), may be suitable for regression and classification problems when there are extensive data. One advantage of such methods is that they automatically handle non-linearity and interactions. Output includes a "decision tree" which is immediately useful for prediction.

```
library(rpart)
data(fgl)    # Forensic glass fragment data; from MASS library
glass.tree <- rpart(type ~ RI+Na+Mg+Al+Si+K+Ca+Ba+Fe, data=fgl)
plot(glass.tree);  text(glass.tree)
summary(glass.tree)
```

To use these models effectively, you also need to know about approaches to pruning trees, and about cross-validation. Methods for reduction of tree complexity that are based on significance tests at each individual node (i. e. branching point) typically choose trees that over-predict.

The Atkinson and Therneau RPART (recursive partitioning) library is closer to CART than is the S-PLUS tree library. It integrates cross-validation with the algorithm for forming trees.

7.5 Exercises

1. Using the data set `painters` (*MASS* library), apply principal components analysis to the scores for `Composition`, `Drawing`, `Colour`, and `Expression`. Examine the loadings on the first three principal components. Plot a scatterplot matrix of the first three principal components, using different colours or symbols to identify the different schools.

2. The data set `Cars93` is in the *MASS* library. Using the columns of continuous or ordinal data, determine scores on the first and second principal components. Investigate the comparison between (i) USA and non-USA cars, and (ii) the six different types (`Type`) of car. Now create a new data set in which binary factors become columns of 0/1 data, and include these in the principal components analysis.

3. Repeat the calculations of exercises 1 and 2, but this time using the function `lda()` from the *MASS* library to derive canonical discriminant scores, as in section 7.3.

4. The MASS library has the `Aids2` data set, containing de-identified data on the survival status of patients diagnosed with AIDS before July 1 1991. Use tree-based classification (`rpart())` to identify major influences on survival.

5. Investigate discrimination between plagiotropic and orthotropic species in the data set `leafshape`[31].

7.6 References

Chambers, J. M. and Hastie, T. J. 1992. Statistical Models in S. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove CA.

Everitt, B. S. and Dunn, G. 1992. Applied Multivariate Data Analysis. Arnold, London.

Friedman, J., Hastie, T. and Tibshirani, R. (1998). Additive logistic regression: A statistical view of boosting. Available from the internet.

Magidson, Jay 1996. SPSS for Windows CHAID Release 6. SPSS Inc., Chicago.

Ripley, B. D. 1996. Pattern Recognition and Neural Networks. Cambridge University Press, Cambridge UK.

Therneau, T. M. and Atkinson, E. J. 1997. An Introduction to Recursive Partitioning Using the RPART Routines. This is one of two documents included in:
`http://www.stats.ox.ac.uk/pub/SWin/rpartdoc.zip`

[31] Data relate to the paper: King. D.A. and Maindonald, J.H. 1999. Tree architecture in relation to leaf dimensions and tree stature in temperate and tropical rain forests. *Journal of Ecology* 87: 1012-1024.

Venables, W. N. and Ripley, B. D., 2nd edn 1997. Modern Applied Statistics with S-Plus. Springer, New York.

8. Useful Functions

8.1 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
> diff()     # Replace a vector by the 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
```

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
```

8.2 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(islandcities$country)     # islandcities accompanies these notes.
 Australia Cuba Indonesia Japan Philippines Taiwan United Kingdom
         3    1         4     6           2      1              2
>
> table(Barley$Year,Barley$Site)  # Barley accompanies these notes
C D GR M UF W
  1931 5 5  5 5  5 5
  1932 5 5  5 5  5 5
>
```

<u>WARNING:</u> `NA`s are ignored unless you specify otherwise. The action needed to get `NA`s 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 you need to generate a new factor in which `“NA”` is included 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
```

8.3 Matching and Ordering

```
> match(<vec1>, <vec2>)  ## For each element of <vec1>, returns the
                         ## position of the first occurrence in <vec2>
> order(<vector>)        ## Returns the vector of subscripts giving
                         ## the order in which elements must be taken
                         ## so that <vector> will be sorted.
> rank(<vector>)         ## Returns the ranks of the successive elements.
```

Numeric vectors will be sorted in numerical order. Character vectors will be sorted in alphanumeric order.

The function `match()` can be used in all sorts of clever ways to pick out subsets of data. For example:

```
> x <- rep(1:5,rep(3,5))
> x
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> two4 <- match(x,c(2,4), nomatch=0)
> two4
 [1] 0 0 0 1 1 1 0 0 0 2 2 2 0 0 0
> # We can use this to pick out the 2s and the 4s
> as.logical(two4)
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[13] FALSE FALSE FALSE
> x[as.logical(two4)]
[1] 2 2 2 4 4 4
```

8.4 String Functions

```
substring(<vector of text strings>, <first position>, <last position>)
nchar(<vector of text strings>)
              ## Returns vector of number of characters in each element.
```

*8.4.1 Operations with Vectors of Text Strings – A Further Example

The following stores, in `nblank,` the position of the first occurrence of a blank space in the column `Make` in the dataset `Cars93` from the *MASS* library.

```
nblank <- sapply(Cars93$Make, function(x){n <- nchar(x);
      a <- substring(x, 1:n, 1:n); m <- match(" ", a,nomatch=1); m})
```

To extract the first part of the name, up to the first space, specify

```
car.brandnames <- substring(Cars93$Make, 1, nblank-1)
> car.brandnames[1:5]
[1] "Acura" "Acura" "Audi"  "Audi"  "BMW"
```

8.5 Application of a Function to the Columns of an Array or Data Frame

```
apply(<array>, <dimension>, <function>)
lapply(<list>, <function>)
               ## N. B. A dataframe is a list.  Output is a list.
sapply(<list>, <function>)
               ## As lapply(), but simplify (e. g. to a vector
               ## or matrix), if possible.
```

The function sapply() can be useful for getting information about the columns of a data frame. We illustrate with the data frame `moths` which accompanies these notes:

```
> sapply(moths,is.factor)        # Determine which columns are factors
 meters       A       P habitat
  FALSE   FALSE   FALSE    TRUE
> # How many levels does each factor have?
> sapply(moths, function(x)if(!is.factor(x))return(0) else
length(levels(x)))
 meters       A       P habitat
      0       0       0       8
```

The function `sapply()` often works most conveniently if we can ensure that the function we use returns just one element for each column. We can reduce the 8 levels for `habitat` to one character string, with the levels separated by spaces `(“ “)`, by specifying `paste(, collapse=” “)`

```
> sapply(moths, function(x)if(!is.factor(x))return("") else
paste(levels(x),collapse=" "))
                                                           meters
                                                               ""
                                                                A
                                                               ""
                                                                P
                                                               ""
                                                          habitat
"Bank Disturbed Lowerside NEsoak NWsoak SEsoak SWsoak Upperside"
```

8.6 tapply()

The arguments are a variable, a list of factors, and a function which operates on a vector to return a single value. For each combination of factor levels, the function is applied to corresponding values of the variable. The output is an array with as many dimensions as there are factors. Where there are no data values for a particular combination of factor levels, NA is returned.

Often one wishes to get back, not an array, but a data frame with one row for each combination of factor levels. For example, we may have a data frame with two factors and a numeric variable, and want to create a new data frame with all possible combinations of the factors, and the cell means as the response. Here is an example of how to do it.

First, use `tapply()` to produce an array of cell means. The function `dimnames()`, applied to this array, returns a list whose first element holds the row names (i. e. for the level names for the first factor), and whose second element holds the column names. [Further dimensions are possible.] We pass this list (row names, column names) to `expand.grid()`, which returns a data frame with all possible combinations of the factor levels. Finally, stretch the array of means out into a vector, and append this to the data frame. Here is an example using the data set `cabbages` from the MASS library.

```
> data(cabbages)
> names(cabbages)
[1] "Cult"   "Date"   "HeadWt" "VitC"
> sapply(cabbages, levels)
```

```
$Cult
[1] "c39" "c52"

$Date
[1] "d16" "d20" "d21"

$HeadWt
NULL

$VitC
NULL

> attach(cabbages)
> cabbages.tab <- tapply(HeadWt, list(Cult, Date), mean)
> cabbages.tab             # Two varieties by three planting dates
     d16  d20  d21
c39 3.18 2.80 2.74
c52 2.26 3.11 1.47
> cabbages.nam <- dimnames(cabbages.tab)
> cabbages.nam      # There are 2 dimensions, therefore 2 list elements
[[1]]
[1] "c39" "c52"

[[2]]
[1] "d16" "d20" "d21"
>      ## We now stretch the array of means out into a vector, and create
>   ## a new column of cabbages.df, named Means, that holds the means.
> cabbages.df <- expand.grid(Cult=factor(cabbages.nam[[1]]),
+                          Date=factor(cabbages.nam[[2]]))
> cabbages.df$Means <- as.vector(cabbages.tab)
> cabbages.df
  Cult Date Means
1  c39  d16  3.18
2  c52  d16  2.26
3  c39  d20  2.80
4  c52  d20  3.11
5  c39  d21  2.74
6  c52  d21  1.47
```

In a case where there are no data for some combinations of factor levels, one might want to omit the corresponding rows.

8.7 Breaking Vectors and Data Frames Down into Lists – split()

As an example,

```
split(cabbages$HeadWt, cabbages$Date)
```

returns a list with three elements, the first named “d16” and containing values of **HeadWt** where **Date** has the level **d16**, and similarly for the remaining lists with names “d20” and “d21”. You need to use **split()** in this way in order to do side by side boxplots. The function **boxplot()** takes as its first element a list in which the first list element is the vector of values for the first boxplot, the second list element is the vector of values for the second boxplot, and so on.

You can use split to split up a data frame into a list of data frames. For example

```
split(cabbages[,-1], cabbages$Date)  # Split remaining columns
                                        # by levels of Date
```

*8.8 Merging Data Frames

The data frame `Cars93` (*MASS* library) holds extensive information on data from 93 cars on sale in the USA in 1993. One of the variables, stored as a factor, is `Type`. I have created a data frame `type.df` which holds two character abbreviations of each of the car types, suitable for use in plotting.

```
> type.df
     Type abbrev
1 Compact     Co
2   Large      L
3 Midsize      M
4   Small     Sm
5  Sporty     Sp
6     Van      V
```

As yet R does not have a `merge()` function, for merging data frames. Here however our demands are simple, and we can proceed thus:

```
> subs <- match(Cars93$Type, type.df$Type, nomatch=0)
> Cars93$abbrev <- type.df$abbrev[subs]
```

This creates a data frame which has the abbreviations in the additional column with name `“abbrev”`.

If there had been rows with missing values of `Type`, these would have been omitted from the new data frame. One can avoid this by making sure that Type has NA as one of its levels, in both data frames. In `type.df`, one would need to specify a value of `abbrev` corresponding to NA, perhaps “nk”.

8.9 Dates

There are two libraries for working with dates — the *date* library and the *chron* library.

We demonstrate the use of the *date* library. The function as.`date()` will convert a character string into a dates object. By default, dates are stored using January 1 1960 as origin. This is important when you use `as.integer` to convert a date into an integer value.

```
> as.date("1/1/60", order="dmy")
[1] 1Jan60
> as.date("1/12/60","dmy")
[1] 1Dec60
> as.date("1/12/60","dmy")-as.date("1/1/60","dmy")
[1] 335
> as.date("31/12/60","dmy")
[1] 31Dec60
> as.date("31/12/60","dmy")-as.date("1/1/60","dmy")
[1] 365
> as.integer(as.date("1/1/60","dmy"))
[1] 0

> as.integer(as.date("1/1/2000","dmy"))
[1] 14610
> as.integer(as.date("29/2/2000","dmy"))
[1] 14669
> as.integer(as.date("1/3/2000","dmy"))
```

```
[1] 14670
```

A wide variety of different formats are possible. Among the legal formats are 8-31-2000 (or 31-8-2000 if you specify `order="dmy"`), 8/31/2000 (cf 31/8/2000), or August 31 2000.

Observe that one can subtract two dates and get the time between them in days. There are several functions (including `date.ddmmmyy()`) for printing out dates in various different formats.

8.10 Exercises

1. For the data frame `Cars93`, get the information provided by `summary()` for each level of `Type`. (Use `split()`.)
2. Determine the number of cars, in the data frame `Cars93`, for each `Origin` and `Type`.
3. In the data frame `claims`: (a) determine the number of rows of information for each age category (`age`) and car type (`type`); (b) determine the total number of claims for each age category and car type; (c) determine, for each age category and car type, the number of rows for which data are missing; (d) determine, for each age category and car type, the total cost of claims.
4. Determine the number of days, according to R, between the following dates:

   a) January 1 in the year 1700, and January 1 in the year 1800

   b) January 1 in the year 1998, and January 1 in the year 2000

9. Writing Functions and other Code

We have already met several functions. Here is a function to convert Fahrenheit to Celsius:

```
> fahrenheit2celsius <- function(fahrenheit=32:40)(fahrenheit-32)*5/9
> # Now invoke the function
> fahrenheit2celsius(c(40,50,60))
[1]  4.444444 10.000000 15.555556
```

The function returns the value `(fahrenheit-32)*5/9`. More generally, a function returns the value of the last statement of the function. Unless the result from the function is assigned to a name, the result is printed.

Here is a function that prints out the mean and standard deviation of a set of numbers:

```
> mean.and.sd <- function(x=1:10){
+ av <- mean(x)
+ sd <- sqrt(var(x))
+ c(mean=av, SD=sd)
+ }
>
> # Now invoke the function
> mean.and.sd()
    mean       SD
5.500000 3.027650

> mean.and.sd(hills$climb)
    mean       SD
1815.314 1619.151
```

9.1 Syntax and Semantics

A function is created using an assignment. On the right hand side, the parameters appear within round brackets. You can, if you wish, give a default. In the example above the default was x = 1:10, so that users can run the function without specifying a parameter, just to see what it does.

Following the closing ")" the function body appears. Except where the function body consists of just one statement, this is enclosed between curly braces ({ }). The return value usually appears on the final line of the function body. In the example above, this was the vector consisting of the two named elements mean and sd.

9.2 A Function that gives Data Frame Details

First we will define a function which accepts a vector `x` as its only argument. It will allow us to determine whether x is a factor, and if a factor, how many levels it has. The built-in function `is.factor()` will return T if `x` is a factor, and otherwise F. The following function `faclev()` uses `is.factor()` to test whether `x` is a factor. It prints out 0 if `x` is not a factor, and otherwise the number of levels of `x`.

```
> faclev <- function(x)if(!is.factor(x))return(0) else
                                             length(levels(x))
```

Earlier, we encountered the function `sapply() which` can be used to repeat a calculation on all columns of a data frame. [More generally, the first argument of `sapply()` may be a list.] To apply `faclev()` to all columns of the data frame `moths` we can specify

```
> sapply(moths, faclev)
```

We can alternatively give the definition of `faclev` directly as the second argument of `sapply`, thus

```
> sapply(moths, function(x)if(!is.factor(x))return(0)
                                                  else length(levels(x))
```

Finally, we may want to do similar calculations on a number of different data frames. So we create a function `check.df()` which encapsulates the calculations. Here is the definition of `check.df()`.

```
check.df <- function(df=moths)
          sapply(df, function(x)if(!is.factor(x))return(0) else
                                                length(levels(x)))
```

9.3 Naming and Record-Keeping Issues

As far as possible, make code self-documenting. Use meaningful names for R objects. Ensure that the names used reflect the hierarchies of files, data structures and code.

R allows the use of names for elements of vectors and lists, and for rows and columns of arrays and dataframes. Consider the use of names rather than numbers when you pull out individual elements, columns etc. Thus `dead.tot[,"dead"]` is more meaningful and safer than `dead.tot[,2]`.

Structure computations so that it is easy to retrace them. For this reason substantial chunks of code should be incorporated into functions sooner rather than later.

9.4 Data Management

Where data, labelling etc must be pulled together from a number of sources, and especially where you may want to retrace your steps some months later, take the same care over structuring data as over structuring code. Thus if there is a factorial structure to the data files, choose file names that reflect it. You can then generate the file names automatically, using `paste()` to glue the separate portions of the name together.

Lists are a useful mechanism for grouping together all data and labelling information that one may wish to bring together in a single set of computations. Use as the name of the list a unique and meaningful identification code. Consider whether you should include objects as list items, or whether identification by name is preferable. Bear in mind, also, the use of `switch()`, with the identification code used to determine what `switch()` should pick out, to pull out specific information and data that is required for a particular run.

Concentrate in one function the task of pulling together data and labelling information, perhaps with some subsequent manipulation, from a number of separate files. This structures the code, and makes the function a source of documentation for the data.

Use user-defined data frame attributes to document your data. For example, given the data frame `elastic` containing the amount of stretch and resulting distance of movement of a rubber band, you might specify

```
attributes(rubber)$title <-
    "Extent of stretch of band, and Resulting Distance"
```

9.5 Issues for the Writing and Use of Functions

There can be many functions. Choose the names for your own functions carefully, so that they are meaningful.

Choose meaningful names for arguments, even if this means that they are longer than you would like. Remember that they can be abbreviated in actual use.

Settings that you may need to change in later use of the function should appear as default settings for parameters. Use lists, where this seems appropriate, to group together parameters that belong together conceptually.

Where appropriate, provide a demonstration mode for functions. Such a mode will print out summary information on the data and/or on the results of manipulations prior to analysis, with appropriate labelling. The code needed to implement this feature has the side-effect of showing by example what the function does, and may be useful for debugging.

Break your functions up into a small number of sub-functions or "primitives". Re-use existing functions wherever possible. Write any new "primitives" so that they can be re-used. This helps ensure that functions contain well-tested and well-understood components. Watch s-news (section 1.9) for useful functions for routine tasks.

If at all possible, give parameters sensible defaults. Often a good strategy is to use as defaults parameters that will serve for a demonstration run of the function.

NULL is a useful default where the parameter mostly is not required, but where the parameter if it appears may be any one of several types of data structure. The test `if(!is.null())` then determines whether one needs to investigate that parameter further.

Structure code to avoid multiple entry of information.

9.6 Graphs

Use graphs freely to shed light both on computations and on data. One of R's big pluses is its tight integration of computation and graphics.

9.7 A Simulation Example

We would like to know how well such a student might do by random guessing, on a multiple choice test consisting of 100 questions each with five alternatives. We can get an idea by using simulation. Each question corresponds to an independent Bernoulli trial with probability of success equal to 0.2. We can simulate the correctness of the student for each question by generating an independent uniform random number. If this number is less than .2, we say that the student guessed correctly; otherwise, we say that the student guessed incorrectly.

This will work, because the probability that a uniform random variable is less than .2 is exactly .2, while the probability that a uniform random variable exceeds .2 is exactly .8, which is the same as the probability that the student guesses incorrectly. Thus, the uniform random number generator is simulating the student. R can do this as follows:

```
guesses<-runif(100)
correct.answers = 1*(guesses < .2)
```

The multiplication by 1 causes `(guesses<.2)`, which is calculated as `TRUE` or `FALSE`, to be coerced to 1 (`TRUE`) or 0 (`FALSE`). The vector `correct.answers` thus contains the results of the student's guesses. A 1 is recorded each time the student correctly guesses the answer, while a 0 is recorded each time the student is wrong.

One can thus write an R function which simulates a student guessing at a True-False test consisting of some arbitrary number of questions. We leave this as an exercise.

9.7.1 Poisson Random Numbers

You can think of the Poisson distribution as the distribution of the total for occurrences of rare events. For example, the occurrence of an accident at an intersection on any one day should be a rare event. The total number of accidents over the course of a year may well follow a distribution which is close to Poisson. [However the total number of people injured is unlikely to follow a Poisson distribution. Why?] We can generate Poisson random numbers using `rpois()`. It is similar to the `rbinom` function, but there is only one parameter – the mean. Suppose for example traffic accidents occur at an intersection with a Poisson distribution that has a mean rate of 3.7 per year. To simulate the annual number of accidents for a 10-year period, we can specify `rpois(10,3.7)`.

We pursue the Poisson distribution in an exercise below.

9.8 Exercises

1. Use the `round` function together with `runif()` to generate 100 random integers between 0 and 99. Now look up the help for `sample()`, and use it for the same purpose.

2. Write a general function to carry out the calculations of section 8.6. More specifically, the function will take as its arguments a list of response variables, a list of factors, a data frame, and a function. It will return a data frame in which each value for each combination of factor levels is summarised in a single statistic, for example the mean or the median.

3. Rewrite the function used for Fig. 8 (section 2.5.2) so that, given the name of a data frame and of any two of its columns, it will plot the second named column against the first named column, showing also the line *y*=*x*.

4. Write a function that prints, with their row and column labels, only those elements of a correlation matrix for which abs(correlation) >= 0.9.

5. Write your own wrapper function for one-way analysis of variance which provides a side by side boxplot of the distribution of values by groups. If no response variable is specified, the function will generate random normal data (no difference between groups) and provide the analysis of variance and boxplot information for that.

6. Write a function which adds a text string containing documentation information as an attribute to a dataframe.

7. Write a function that computes a moving average of order 2 of the values in a given vector. Apply the above function to the data (in the data set `huron` that accompanies these notes) for the levels of Lake Huron. Repeat for a moving average of order 3.

8. Find a way of computing the moving averages in exercise 3 that does not involve the use of a for loop.

9. Create a function to compute the average, variance and standard deviation of 1000 randomly generated uniform random numbers, on [0,1]. (Compare your results with the theoretical results: the expected value of a uniform random variable on [0,1] is 0.5, and the variance of such a random variable is 0.0833.)

10. Write a function which generates 100 independent observations on a uniformly distributed random variable on the interval [3.7, 5.8]. Find the mean, variance and standard deviation of such a uniform random variable. Now modify the function so that you can specify an arbitrary interval.

11. Look up the help for the `sample()` function. Use it to generate 50 random integers between 0 and 99, sampled without replacement. (This means that we do not allow any number to be sampled a second time.) Now, generate 50 random integers between 0 and 9, with replacement.

12. Write an R function which simulates a student guessing at a True-False test consisting of 40 questions. Find the mean and variance of the student's answers. Compare with the theoretical values of .5 and .25.

13. Write an R function which simulates a student guessing at a multiple choice test consisting of 40 questions, where there is chance of 1 in 5 of getting the right answer to each question. Find the mean and variance of the student's answers. Compare with the theoretical values of .2 and .16.

14. Write an R function which simulates the number of working light bulbs out of 500, where each bulb has a probability .99 of working. Using simulation, estimate the expected value and variance of the random variable X, which is 1 if the light bulb works and 0 if the light bulb does not work. What are the theoretical values?

15. Write a function that does an arbitrary number `n` of repeated simulations of the number of accidents in a year, plotting the result in a suitable way. Assume that the number of accidents in a year follows a Poisson distribution. Run the function assuming an average rate of 2.8 accidents per year.

16. Write a function which simulates the repeated calculation of the coefficient of variation (= the ratio of the mean to the standard deviation), for independent random samples from a normal distribution.

17. Write a function which, for any sample, calculates the median of the absolute values of the deviations from the sample median.

*18. Generate random samples from normal, exponential, t (2 d. f.), and t (1 d. f.), thus:

a) `xn<-rnorm(100)`

b) `xe<-rexp(100)`

c) `xt2<-rt(100, df=2)`

d) `xt2<-rt(100, df=1`)

Apply the function from exercise 17 to each sample. Compare with the standard deviation in each case.

*19. The vector `x` consists of the frequencies

```
    5, 3, 1, 4, 6
```

The first element is the number of occurrences of level 1, the second is the number of occurrences of level 2, and so on. Write a function which takes any such vector x as its input, and outputs the vector of factor levels, here `1 1 1 1 1 2 2 2 3 . . .`

[You’ll need the information that is provided by cumsum(x). Form a vector in which 1’s appear whenever the factor level is incremented, and is otherwise zero. . . .]

*20. Write a function which calculates the minimum of a quadratic, and the value of the function at the minimum.

*21. A “between times” correlation matrix, has been calculated from data on heights of trees at times 1, 2, 3, 4, . . . Write a function that calculates the average of the correlations for any given lag.

*22. Given data on trees at times 1, 2, 3, 4, . . ., write a function that calculates the matrix of “average” relative growth rates over the several intervals. Apply your function to the data frame `rats` that accompanies these notes.

[The relative growth rate may be defined as $\frac{1}{w}\frac{dw}{dt} = \frac{d \log w}{dt}$. Hence its is reasonable to calculate the average over the interval from t_1 to t_2 as $\frac{\log w_2 - \log w_1}{t_2 - t_1}$.]

10. GLM, and General Non-linear Models

GLM models are Generalized Linear Models. They extend the multiple regression model. The GAM (Generalized Additive Model) model is a further extension.

10.1 A Taxonomy of Extensions to the Linear Model

R allows a variety of extensions to the multiple linear regression model. In this chapter we describe the alternative functional forms.

The basic model formulation[32] is:

Observed value = Model Prediction + Statistical Error

Often it is assumed that the statistical error values (values of ε in the discussion below) are independently and identically distributed as Normal. Generalized Linear Models, and the other extensions we describe, allow a variety of non-normal distributions. In the discussion of this section, our focus is on the form of the model prediction, and we leave until later sections the discussion of different possibilities for the "error" distribution.

Multiple regression model

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon$$

Use `lm()` to fit multiple regression models. The various other models we describe are, in essence, generalizations of this model.

Generalized Linear Model (e. g. logit model)

$$y = g(a + b_1 x_1) + \varepsilon$$

Here $g(.)$ is selected from one of a small number of options.

For logit models, $y = \pi + \varepsilon$, where

$$\log(\frac{\pi}{1-\pi}) = a + b_1 x_1$$

Here π is an expected proportion, and

$$\log(\frac{\pi}{1-\pi}) = \text{logit}(\pi) \text{ is log(odds).}$$

We can turn this model around, and write

$$y = g(a + b_1 x_1) + \varepsilon = \frac{\exp(a + b_1 x_1)}{1 + \exp(a + b_1 x_1)} + \varepsilon$$

Here $g(.)$ undoes the logit transformation.

We can add more explanatory variables: $a + b_1 x_1 + \ldots + \mathrm{b_p x_{p.}}$

Use `glm()` to fit generalized linear models.

[32] This may be generalized in various ways. Models which have this form may be nested within other models which have this basic form. Thus there may be `predictions' and `errors' at different levels within the total model.

Additive Model

$$y = \phi_1(x_1) + \phi_2(x_2) + ... + \phi_p(x_p) + \varepsilon$$

Additive models are a generalization of `lm` models. In 1 dimension

$$y = \phi_1(x_1) + \varepsilon$$

Some of $z_1 = \phi_1(x_1), z_2 = \phi_2(x_2),..., z_p = \phi_p(x_p)$ may be smoothing functions, while others may be the usual linear model terms. The constant term gets absorbed into one or more of the ϕ s.

Generalized Additive Model

$$y = g(\phi_1(x_1) + \phi_2(x_2) + ... + \phi_p(x_p)) + \varepsilon$$

Generalized Additive Models are a generalisation of Generalized Linear Models. For example, *g*(.) may be the function that undoes the logit transformation, as in a logistic regression model.

Some of $z_1 = \phi_1(x_1), z_2 = \phi_2(x_2),..., z_p = \phi_p(x_p)$ may be smoothing functions, while others may be the usual linear model terms.

We can transform to get the model

$$y = g(z_1 + z_2 + ...z_p) + \varepsilon$$

Notice that even if *p* = 1, we may still want to retain both ϕ_1(.) and *g*(.), i. e.

$$y = g(\phi_1(x_1)) + \varepsilon$$

The reason is that *g*(.) is a specific function, such as the inverse of the logit function. The function $\phi_1(.)$ does any further necessary smoothing, in case *g*(.) is not quite the right transformation. One wants *g*(.) to do as much of possible of the task of transformation, with $\phi_1(.)$ giving the transformation any necessary additional flourishes.

At the time of writing, R has no specific provision for generalized additive models. The fitting of spline (`bs()` or `ns()`) terms in a linear model or a generalized linear model will often do what is needed.

10.2 Logistic Regression

We will use a logistic regression model as a starting point for discussing Generalized Linear Models.

With proportions that range from less than 0.1 to 0.99, it is not reasonable to expect that the expected proportion will be a linear function of *x*. Some such transformation (`link' function) as the logit is required. A good way to think about logit models is that they work on a log(odds) scale. If *p* is a probability (e. g. that horse A will win the race), then the corresponding odds are *p*/(1-*p*), and

log(odds) = log($\frac{p}{1-p}$) = log(*p*) -log(1-*p*)

The linear model predicts, not *p*, but log($\frac{p}{1-p}$). Fig. 19 shows the logit transformation

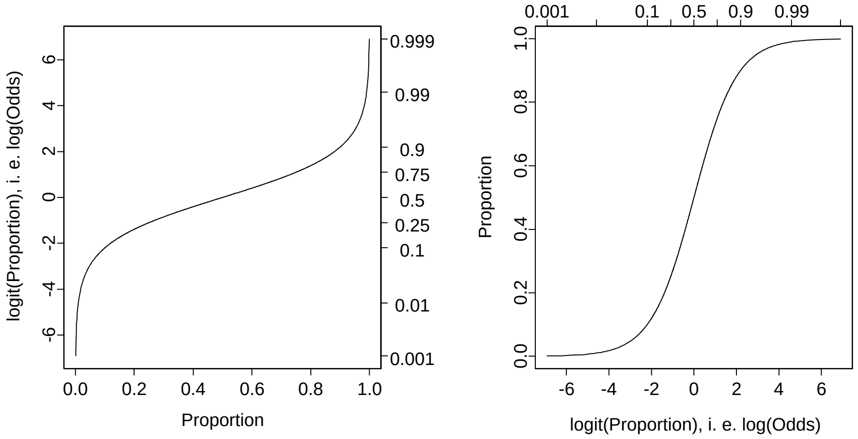


Fig. 19: The logit or log(odds) transformation. The left panel shows a plot of log(odds) versus proportion, while the right panel shows a plot of proportion versus log(odds). Notice how the range is stretched out at both ends.

The logit or log(odds) function turns expected proportions into values that may range from -∞ to +∞. It is not satisfactory to use a linear model to predict proportions. The values from the linear model may well lie outside the range from 0 to 1. It is however in order to use a linear model to predict logit(proportion). The logit function is an example of a link function.

There are various other link functions that we can use with proportions. One of the commonest is the complementary log-log function.

10.2.1 Anaesthetic Depth Example

Thirty patients were given an anaesthetic agent which was maintained at a pre-determined [alveolar] concentration for 15 minutes before making an incision[33]. It was then noted whether the patient moved, i. e. jerked or twisted. The interest is in estimating how the probability of jerking or twisting varies with increasing concentration of the anaesthetic agent.

The response is best taken as nomove, for reasons that will emerge later. There is a small number of concentrations; so we begin by tabulating proportion that have the nomove outcome against concentration.

	Alveolar Concentration					
Nomove	0.8	1	1.2	1.4	1.6	2.5
0	6	4	2	2	0	0
1	1	1	4	4	4	2
Total	7	5	6	6	4	2

Table 1: Patients moving (0) and not moving (1), for each of six different alveolar concentrations.

Fig. 20 then displays a plot of these proportions.

[33] I am grateful to John Erickson (Anesthesia and Critical Care, University of Chicago) and to Alan Welsh (Centre for Mathematics & its Applications, Australian National University) for allowing me use of these data.

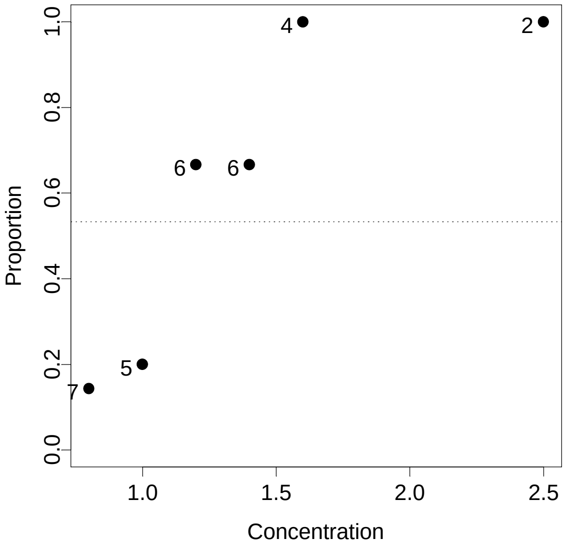


Fig. 20: Plot, versus concentration, of proportion of patients not moving. The dotted horizontal line is the estimate of the proportion of moves one would expect if the concentration had no effect.

We fit two models, the logit model and the complementary log-log model. We can fit the models either directly to the 0/1 data, or to the proportions in Table 1. To understand the output, you need to know about "deviances". A deviance has a role very similar to a sum of squares in regression. Thus we have:

Regression	Logistic regression
degrees of freedom	degrees of freedom
sum of squares	deviance
mean sum of squares	mean deviance
(divide by d.f.)	(divide by d.f.)
We prefer models with a small mean residual sum of squares.	We prefer models with a small mean deviance.

If individuals respond independently, with the same probability, then we have Bernoulli trials. Justification for assuming the same probability will arise from the way in which individuals are sampled. While individuals will certainly be different in their response the notion is that, each time a new individual is taken, they are drawn at random from some larger population. Here is the R code:

```
> anaes.logit <- glm(nomove ~ conc, family = binomial(link = logit),
+        data = anaesthetic)
```

The output summary is:

```
> summary(anaes.logit)

Call: glm(formula = nomove ~ conc, family = binomial(link = logit),
         data = anaesthetic)
Deviance Residuals:
   Min     1Q Median    3Q  Max
 -1.77 -0.744 0.0341 0.687 2.07
```

```
Coefficients:
            Value Std. Error t value
(Intercept) -6.47       2.42   -2.68
       conc  5.57       2.04    2.72

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 41.5 on 29 degrees of freedom
Residual Deviance: 27.8 on 28 degrees of freedom
Number of Fisher Scoring Iterations: 5

Correlation of Coefficients:
     (Intercept)
conc -0.981
```

Fig. 21 is a graphical summary of the results:

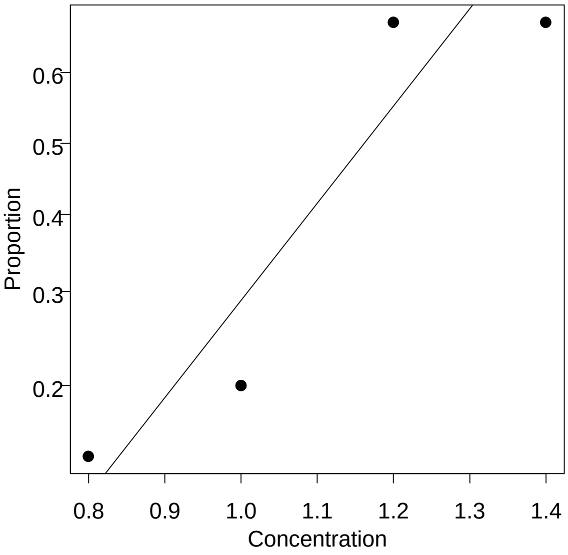


Fig. 21: Plot, versus concentration, of logit(proportion) of patients not moving. The line is the estimate of the proportion of moves, based on the fitted logit model.

With such a small sample size it is impossible to do much that is useful to check the adequacy of the model.

You can also try `plot(anaes.logit)` and `plot.gam(anaes.logit)`.

10.3 glm models (Generalized Linear Regression Modelling)

In the above we had

```
anaes.logit <- glm(nomove ~ conc, family = binomial(link = logit),
                   data=anaesthetic)
```

The `family` parameter specifies the distribution for the dependent variable. There is an optional argument which allows us to specify the link function. Below we give further examples.

10.3.2 Data in the form of counts

Data that are in the form of counts can often be analysed quite effectively assuming the `poisson` family. The link that is commonly used here is `log`. The `log` link transforms from positive numbers to numbers in the range -∞ to +∞ which a linear model may predict.

10.3.3 The gaussian family

If no family is specified, then the family is taken to be `gaussian`. The default link is then the `identity,` as for an `lm` model. This way of formulating an `lm` type model does however have the advantage that one is not restricted to the identity link.

```
Data(airquality)
air.glm<-glm(Ozone^(1/3) ~ Solar.R + Wind + Temp, data = airquality)
    # Assumes gaussian family, i.e. normal errors model
summary(air.glm)
```

10.4 Models that Include Smooth Spline Terms

These make it possible to fit spline and other smooth transformations of explanatory variables. One can request a `smooth' b-spline or n-spline transformation of a column of the X matrix. In place of `x` one specifies `bs(x)`or `ns(x)`. One can control the smoothness of the curve, but often the default works quite well. You need to install the *splines* library. R does not at present have a facility for plots that show the contribution of each term to the model.

10.4.1 Dewpoint Data

The data set `dewpoint`[34] has columns `mintemp`, `maxtemp` and `dewpoint`. The dewpoint values are averages, for each combination of mintemp and maxtemp, of monthly data from a number of different times and locations. We fit the model:

`dewpoint` = mean of `dewpoint` + smooth(`mintemp`) + smooth(`maxtemp`)

Taking out the mean is a computational convenience. Also it provides a more helpful form of output. Here are details of the calculations:

```
> dewpoint.lm <- lm(dewpoint ~ bs(mintemp) + bs(maxtemp),
                    data = dewpoint)
> options(digits=3)
> summary(dewpoint.lm)
```

10.5 Non-linear Models

You can use `nls()` (non-linear least squares) to obtain a least squares fit to a non-linear function.

10.6 Model Summaries

Type in

```
?methods(summary)
```

to get a list of the summary methods that are available. You may want to mix and match, e.g. `summary.lm()` on an aov or glm object. The output may not be what you might expect. So be careful!

[34] I am grateful to Dr Edward Linacre, Visiting Fellow, Geography Department, Australian National University, for making these data available.

10.7 Further Elaborations

Generalised Linear Models were developed in the 1970s. They unified a huge range of diverse methodology. They have now become a stock-in-trade of statistical analysts. Their practical implementation built on the powerful computational abilities which, by the 1970s, had been developed for handling linear model calculations.

Practical data analysis demands further elaborations. An important elaboration is to the incorporation of more than one term in the error structure. The R `nlme` library implements such extensions, both for linear models and for a wide class of nonlinear models.

Each such new development builds on the theoretical and computational tools that have arisen from earlier developments. Exciting new analysis tools will continue to appear for a long time yet. This is fortunate. Most professional users of R will regularly encounter data where the methodology that the data ideally demands is not yet available.

10.8 Exercises

1. Fit a Poisson regression model to the data in the data frame `moths` that Accompanies these notes. Allow different intercepts for different habitats. Use log(meters) as a covariate.

10.9 References

Dobson, A. J. 1983. An Introduction to Statistical Modelling. Chapman and Hall, London.

Hastie, T. J. and Tibshirani, R. J. 1990. Generalized Additive Models. Chapman and Hall, London.

McCullagh, P. and Nelder, J. A., 2nd edn., 1989. Generalized Linear Models. Chapman and Hall.

Venables, W. N. and Ripley, B. D., 2nd edn 1997. Modern Applied Statistics with S-Plus. Springer, New York.

11. Multi-level Models, Time Series and Survival Analysis

Repeated measures models are a special case of multi-level models.

*11.1 Multi-Level Models, Including Repeated Measures Models

Models have both a fixed effects structure and an error structure. For example, in an inter-laboratory comparison there may be variation between laboratories, between observers within laboratories, and between multiple determinations made by the same observer on different samples. If we treat laboratories and observers as random, the only fixed effect is the mean.

The functions `lme()` and `nlme()`, from the Pinheiro and Bates library, handle models in which a repeated measures error structure is superimposed on a linear (`lme`) or non-linear (`nlme`) model. Version 3 of lme, which is currently in β-test, is broadly comparable in its abilities to Proc Mixed which is available in the widely used SAS statistical package. The function `lme` has associated with it highly useful abilities for diagnostic checking and for various insightful plots.

There is a strong link between a wide class of repeated measures models and time series models. In the time series context there is usually just one realisation of the series, which may however be observed at a large number of time points. In the repeated measures context there may be a large number of realisations of a series which is typically quite short.

11.1.1 The Kiwifruit Shading Data, Again

Refer back to section 6.8.2 for details of these data. The fixed effects are `block` and treatment (`shade`). The random effects are `block` (though making block a random effect is optional), `plot` within `block`, and units within each block/plot combination. Here is the analysis:

```
> library(nlme)
Loading required package: nls
> kiwishade$plot<-paste(kiwishade$block,kiwishade$shade,sep=".")
> kiwishade.lme<-lme(yield~shade,random=~1|block/plot,
data=kiwishade)
> summary(kiwishade.lme)
Linear mixed-effects model fit by REML
 Data: kiwishade
       AIC      BIC    logLik
  265.9663 278.4556 -125.9831

Random effects:
 Formula: ~1 | block
        (Intercept)
StdDev:    2.019373

 Formula: ~1 | plot %in% block
        (Intercept) Residual
StdDev:    1.478639 3.490378

Fixed effects: yield ~ shade
                 Value Std.Error DF  t-value p-value
(Intercept)  100.20250  1.761621 36 56.88086  <.0001
shadeAug2Dec   3.03083  1.867629  6  1.62282  0.1558
shadeDec2Feb -10.28167  1.867629  6 -5.50520  0.0015
```

```
shadeFeb2May  -7.42833  1.867629  6 -3.97741  0.0073
 Correlation:
             (Intr) shdA2D shdD2F
shadeAug2Dec -0.53
shadeDec2Feb -0.53   0.50
shadeFeb2May -0.53   0.50   0.50

Standardized Within-Group Residuals:
       Min         Q1        Med         Q3        Max
-2.4153887 -0.5981415 -0.0689948  0.7804597  1.5890938

Number of Observations: 48
Number of Groups:
          block plot %in% block
              3              12
> anova(kiwishade.lme)
            numDF denDF  F-value p-value
(Intercept)     1    36 5190.552  <.0001
shade           3     6   22.211  0.0012
```

This was a balanced design, which is why in section 6.8.2 we could use `aov()` for an analysis. We can get an output summary that is helpful for showing how the error mean squares match up with standard deviation information given above thus:

```
> intervals(kiwishade.lme)
Approximate 95% confidence intervals

 Fixed effects:
                 lower       est.      upper
(Intercept)   96.62977 100.202500 103.775232
shadeAug2Dec  -1.53909   3.030833   7.600757
shadeDec2Feb -14.85159 -10.281667  -5.711743
shadeFeb2May -11.99826  -7.428333  -2.858410

 Random Effects:
  Level: block
                    lower     est.   upper
sd((Intercept)) 0.5473014 2.019373 7.45086
  Level: plot
                    lower     est.    upper
sd((Intercept)) 0.3702555 1.478639 5.905037

 Within-group standard error:
   lower     est.    upper
2.770678 3.490378 4.397024
>
```

We are interested in the three estimates. By squaring the standard deviations and converting them to variances we get the information in the following table:

	Variance component	Notes
block	2.019^2 = 4.076	Three blocks
plot	1.479^2= 2.186	4 plots per block
residual (within group)	3.490^2=12.180	4 vines (subplots) per plot

The above allows us to put together the information for an analysis of variance table. We have:

	Variance component	Mean square for anova table	d.f.
block	4.076	12.180 + 4 × 2.186 + 16 × 4.076 = 86.14	2 (3-1)
plot	2.186	12.180 + 4 × 2.186 = 20.92	6 (3-1) ×(2-1)
residual (within group)	12.180	12.18	3×4×(4-1)

Now find see where these same pieces of information appeared in the analysis of variance table of section 6.8.2:

```
> summary(kiwishade.aov)

Error: block:shade
          Df  Sum Sq Mean Sq F value   Pr(>F)
block      2  172.35   86.17  4.1176 0.074879
shade      3 1394.51  464.84 22.2112 0.001194
Residuals  6  125.57   20.93

Error: Within
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals 36 438.58   12.18
```

11.1.2 The Tinting of Car Windows

In section 5.1 we encountered data from an experiment that aimed to model the effects of the tinting of car windows on visual performance[35]. The authors are mainly interested in effects on side window vision, and hence in visual recognition tasks that would be performed when looking through side windows.

Data are in the data frame `tinting`. 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, while `tint` (3 levels) and `target` (2 levels) are ordered factors. The variable `sex` is coded 1 for males and 2 for females, while the variable `agegp` is coded 1 for young people (all in their early 20s) and 2 for older participants (all in the early 70s).

We have two levels of variation – within individuals (who were each tested on each combination of `tint` and `target`), and between individuals. So we need to specify `id` (identifying the individual) as a random effect. Plots such as we examined in section 5.1 make it clear that, to get variances that are approximately homogeneous, we need to work with log(`csoa`) and log(`it`). Here we examine the analysis for log(`it`). We start with a model that is likely to be more complex than we need (it has all possible interactions):

```
itstar.lme<-lme(log(it)~tint*target*agegp*sex,
```

[35] 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.

```
    random=~1|id, data=tinting,method="ML")
```

A reasonable guess is that first order interactions may be all we need, i.e.

```
it2.lme<-lme(log(it)~(tint+target+agegp+sex)^2,
  random=~1|id, data=tinting,method="ML")
```

Finally, there is the very simple model, allowing only for main effects:

```
it1.lme<-lme(log(it)~(tint+target+agegp+sex),
  random=~1|id, data=tinting,method="ML")
```

Note that we have fitted all these models by maximum likelihood. This is so that we can do the equivalent of an analysis of variance comparison. Here is what we get:

```
> anova(itstar.lme,it2.lme,it1.lme)
           Model df       AIC      BIC    logLik   Test  L.Ratio p-value
itstar.lme     1 26  8.146187 91.45036 21.926906
it2.lme        2 17 -3.742883 50.72523 18.871441 1 vs 2  6.11093  0.7288
it1.lme        3  8  1.138171 26.77022  7.430915 2 vs 3 22.88105  0.0065
```

The model that limits attention to first order interactions is adequate. We will need to examine the first order interactions individually. For this we re-fit the model used for `it2.lme`, but now with `method="REML"`.

```
it2.reml<-update(it2.lme,method="REML")
```

We now examine the estimated effects:

```
> options(digits=3)
> summary(it2.reml)$tTable
                      Value Std.Error  DF t-value  p-value
(Intercept)         6.05231    0.7589 145   7.975 4.17e-13
tint.L              0.22658    0.0890 145   2.547 1.19e-02
tint.Q              0.17126    0.0933 145   1.836 6.84e-02
targethicon        -0.24012    0.1010 145  -2.378 1.87e-02
agegp              -1.13449    0.5167  22  -2.196 3.90e-02
sex                -0.74542    0.5167  22  -1.443 1.63e-01
tint.L.targethicon -0.09193    0.0461 145  -1.996 4.78e-02
tint.Q.targethicon -0.00722    0.0482 145  -0.150 8.81e-01
tint.L.agegp       -0.13075    0.0492 145  -2.658 8.74e-03
tint.Q.agegp       -0.06972    0.0520 145  -1.341 1.82e-01
tint.L.sex          0.09794    0.0492 145   1.991 4.83e-02
tint.Q.sex         -0.00542    0.0520 145  -0.104 9.17e-01
targethicon.agegp   0.13887    0.0584 145   2.376 1.88e-02
targethicon.sex    -0.07785    0.0584 145  -1.332 1.85e-01
agegp.sex           0.33164    0.3261  22   1.017 3.20e-01
```

Further explanation may appear in the next revision of these notes.

11.1.3 The Michelson Speed of Light Data

Here is an example, using the Michelson speed of light data from the Venables and Ripley *MASS* library. The model allows the determination to vary linearly with `Run` (from 1 to 20), with the slope varying randomly between the five experiments of 20 runs each. We assume an autoregressive dependence structure of order 2. We allow the variance to change from one experiment to another. Maximum likelihood tests suggest that one needs at least this complexity in the variance and dependence structure to represent the data accurately. A model that has neither fixed nor random `Run` effects seems all that is justified statistically. To test this, one needs to fit models with and without these effects, setting `method=”ML”` in each case, and compare the likelihoods. (I leave this as an exercise!) For purposes of doing this test, a first order autoregressive model would probably be adequate. A model which ignores the sequential dependence entirely does give misleading results.

```
> michelson$Run <- as.numeric(michelson$Run) # Ensure Run is a variable
> mich.lme20 <- lme(fixed = Speed ~ Run, data = michelson,
        random =  ~ Run| Expt, correlation = corARMA(form =  ~ 1 | Expt,
      p = 2, q = 0), weights = varIdent(form =  ~ 1 | Expt))

       > summary(mich.lme20)
Linear mixed-effects model fit by maximum likelihood
 Data: michelson
   AIC  BIC logLik
  1117 1148 -546.4

Random effects:
 Formula:  ~ Run | Expt
 Structure: General positive-definite
             StdDev   Corr
(Intercept)  47.031 (Inter
        Run   3.628 -1
   Residual 121.930

Correlation Structure: ARMA(2,0)
 Parameter estimate(s):
   Phi1    Phi2
 0.6321 -0.3106
Variance function:
 Structure: Different standard deviations per stratum
 Formula: ~ 1 | Expt
 Parameter estimates:
 1      2      3      4      5
 1 0.2993 0.6276 0.5678 0.4381
Fixed effects: Speed ~ Run
             Value Std.Error z-value p-value
(Intercept)  860.9      27.2    31.6     0.0
        Run   -1.6       2.1    -0.7     0.5
 Correlation:
    (Intr)
Run -0.962

Standardized Within-Group Residuals:
    Min      Q1    Med     Q3   Max
 -2.905 -0.6207 0.1222 0.7373 1.955

Number of Observations: 100
Number of Groups: 5
> # Now plot population residuals versus BLUP fitted values
> plot(mich.lme20, fitted(.) ~ Run | Expt,
                between = list(x = 0.25, y = 0.25), type = "b")
> # NB R invokes plot.lme()
> # Plot BLUP fitted effects versus Run, to help explain previous plot
> plot(mich.lme20, resid(., type = "p") ~
+ fitted(.) | Expt, between = list(x = 0.25, y = 0.25))
```

11.2 Time Series Models

The R *ts* (time series) package has a number of functions for manipulating and plotting time series, and for calculating the autocorrelation function.

There are (at least) two types of method – time domain methods and frequency domain methods. In the time domain models may be conventional "short memory" models where the autocorrelation function decays quite rapidly to zero, or the relatively recently developed "long memory" time series models where the autocorrelation function decays very slowly as observations move apart in time. A characteristic of "long memory" models is that there is variation at all temporal scales. Thus in a study of wind speeds it may be possible to characterise windy days, windy weeks, windy months, windy years, windy decades, and perhaps even windy centuries. R does not yet have functions for fitting the more recently developed long memory models.

The function `stl()` decomposes a times series into a trend and seasonal components, etc. The functions `ar()` (for "autoregressive" models) and associated functions, and `arima0()` ("autoregressive integrated moving average models") fit standard types of time domain short memory models.

The function `spectrum()` and related functions is designed for frequency domain or "spectral" analysis.

11.3 Survival Analysis

For example times at which subjects were either lost to the study or died ("failed") may be recorded for individuals in each of several treatment groups. Engineering or business failures can be modelled using this same methodology. The R *survival5* library has state of the art abilities for survival analysis.

11.5 Exercises

1. Use the function `acf()` to plot the autocorrelation function of lake levels in successive years in the data set `huron`. Do the plots both with `type="correlation"` and with `type="partial"`.

11.4 References

Chambers, J. M. and Hastie, T. J. 1992. Statistical Models in S. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove CA.

Diggle, Liang & Zeger (1996). Analysis of Longitudinal Data. Clarendon Press, Oxford.

Everitt, B. S. and Dunn, G. 1992. Applied Multivariate Data Analysis. Arnold, London.

Hand, D. J. & Crowder, M. J. (1996). Practical longitudinal data analysis. Chapman and Hall, London.

Little, R. C., Milliken, G. A., Stroup, W. W. and Wolfinger, R. D. (1996). SAS Systems for Mixed Models. SAS Institute Inc., Cary, New Carolina.

Pinheiro, J. C. and Bates, D. M. (1998). Mixed effects methods and classes for S and S-PLUS. Unpublished manuscript.

Venables, W. N. and Ripley, B. D., 2nd edn 1997. Modern Applied Statistics with S-Plus. Springer, New York.

12. Advanced Programming Topics

12.1. Methods

R is an object-oriented language. Objects may have a "class". For functions such as `print()`, `summary()`, etc., the class of the object determines what action will be taken. Thus in response to `print(x)`, R determines the class attribute of `x`, if one exists. If for example the class attribute is "factor", then the function which finally handles the printing is `print.factor()`. The function `print.default()` is used to print objects which have not been assigned a class.

More generally, the class attribute of an object may be a vector of strings. If there are "ancestor" classes – parent, grandparent, . . ., these are specified in order in subsequent elements of the class vector. For example, ordered factors have the class "ordered", which inherits from the class "factor". Thus:

```
> fac<-ordered(1:3)
> class(fac)
[1] "ordered" "factor"
>
```

Here `fac` has the class "ordered", which inherits from the parent class "factor".

The function `print.ordered()`, which is the function that is called when you invoke `print()` with an ordered factor, makes use of the fact that "ordered" inherits from "factor".

```
> print.ordered
function (x, quote = FALSE)
{
    if (length(x) <= 0)
        cat("ordered(0)\n")
    else print(levels(x)[x], quote = quote)
    cat("Levels: ", paste(levels(x), collapse = " < "), "\n")
    invisible(x)
}
```

Note that it is a convenience for `print.ordered()` to call `print.factor()`. The function `print.glm()` does not call `print.lm()`, even though glm objects inherit from lm objects.

12.2 Extracting Arguments to Functions

How, inside a function, can one extract the value assigned to a parameter when the function was called? Below there is a function `extract.arg()`. When it is called as `extract.arg(a=xx)`, we want it to return `"xx"`. When it is called as `extract.arg(a=xy)`, we want it to return `"xy"`. Here is how it is done.

```
extract.arg <-
function (a)
{
      s <- substitute(a)
      as.character(s)
}


> extract.arg(a=xy)
[1] "xy"
```

If the argument is a function, we may want to get at the arguments to the function. Here is how one can do it

```
deparse.args <-
```

```
function (a)
{
      s <- substitute (a)
      if(mode(s) == "call"){
            # the first element of a 'call' is the function called
            # so we don't deparse that, just the arguments.
           print(paste(“The function is:  “, s[1],”()”, collapse=””))
            lapply (s[-1], function (x)
            paste (deparse(x), collapse = "\n"))
           }
      else stop ("argument is not a function call")
}
```

For example:

```
> deparse.args(list(x+y, foo(bar)))
[1] "The function is:   list ()"
[[1]]
[1] "x + y"

[[2]]
[1] "foo(bar)"
```

12.3 Parsing and Evaluation of Expressions

When you type in an expression such as `mean(x+y)` or `cbind(x,y)` for R to evaluate, there are two steps:

1. The text string which you type in is parsed and turned into an expression, i. e. the syntax is checked and it is turned into code which the R engine can more immediately evaluate.
2. The expression is evaluated.

If you type in

```
expression(mean(x+y))
```

the output is the unevaluated expression `expression(mean(x+y))`. By setting

```
my.exp <- expression(mean(x+y))
```

you can store this unevaluated expression in `my.exp` . Actually what is actually stored in `my.exp` is a little different from what is printed out. R gives you as much information as it judges is (most of the time) helpful for you to know.

Note that `expression(mean(x+y))` is different from `expression(“mean(x+y)”),` as is obvious when the expression is evaluated. A text string is a text string is a text string, unless you explicitly change it into an expression or part of an expression.

Let’s see how this works in practice

```
> x <- 101:110
> y <- 21:30
> my.exp <- expression(mean(x+y))
> my.txt <- expression("mean(x+y)")
> eval(my.exp)
[1] 131
> eval(my.txt)
[1] "mean(x+y)"
>
```

What if we already have “mean(x+y)” stored in a text string, and we want to turn it into an expression? The answer is to use the function parse(), but you must tell it that you are supplying text rather than the name of a file. Thus

```
> parse(text="mean(x+y)")
expression(mean(x + y))
```

Let’s store the expression in my.exp2, and then evaluate it

```
> my.exp2 <- parse(text="mean(x+y)")
> eval(my.exp2)
[1] 131
>
```

Here is a function that creates a new data frame from an arbitrary set of columns of an existing data frame. Once in the function, we attach the data frame so that we can leave off the name of the data frame, and use only the column names

```
make.new.df <- function(old.df = austpop, colnames = c("NSW", "ACT"))
{
      attach(old.df)
      on.exit(detach("old.df"))
      argtxt <- paste(colnames, collapse = ",")
      exprtxt <- paste("data.frame(", argtxt, ")", sep = "")
      expr <- parse(text = exprtxt)
      df <- eval(expr)
      names(df) <- colnames
      df
}
```

To verify that the function does what it should, type in

```
> make.new.df()
   NSW ACT
1 1904   3
2 2402   8
3 2693  11
4 2985  17
5 3625  38
6 4295 103
7 5002 214
8 5617 265
9 6274 310
```

The function do.call() may be convenient if you want to keep the function name and the argument list in separate text strings. When do.call is used it is only necessary to use parse() in generating the argument list.

For example

```
make.new.df <-
function(old.df = austpop, colnames = c("NSW", "ACT"))
{
      attach(old.df)
      on.exit(detach("old.df"))
      argtxt <- paste(colnames, collapse = ",")
      listexpr <- parse(text=paste("list(", argtxt, ")", sep = ""))
      df <- do.call(“data.frame”, eval(listexpr))
```

```
        names(df) <- colnames
        df
}
```

12.4 Searching R functions for a specified token.

A token is a syntactic entity; for example function names are tokens. For example, we search all functions in the working directory. The purpose of using `unlist()` in the code below is to change `myfunc` from a list into a simple vector of characters.

```
mygrep <-
function(str)
{
##  Assign the names of all objects in current R
##  working directory to the string vector tempobj
##
  tempobj <- ls(envir=sys.frame(-1))
  objstring <- character(0)
  for(i in tempobj) {
          myfunc <- get(i)
    if(is.function(myfunc))
         if(length(grep(str,
             deparse(myfunc))))
     objstring <- c(objstring, i)
}
return(objstring)
}
```

13. R Resources

13.1 R Packages for Windows

To get information on R packages (libraries), go to:

`http://cran.r-project.org`

An Australian link is:

`http://mirror.aarnet.edu.au/CRAN/`

For Windows 95 etc binaries, look in

`http://mirror.aarnet.edu.au/CRAN/windows/windows-9x/`

Look in the directory **contrib** for libraries.

New libraries are being added all the time. So it pays to check the CRAN site from time to time. Also, watch for announcements on r-help and r-announce.

13.2 Literature written by expert users

Much literature that has been written for S-PLUS is highly relevant for R.

Burns, P. J. 1998. S Poetry.
This 439 page document is available from

`http://www.seanet.com/~pburns/Spoetry/.`

The style is leisurely. However this assumes some prior knowledge of computing language terms. It may be a good book to work through once you have some initial knowledge of R.

Chambers, J. M. 1998. Programming with Data. A Guide to the S Language. Springer-Verlag, New York.

This is a book for specialists. It describes a new version of the S language, which is the basis for version 5 of R. Version 5 of S-PLUS is so far available for Unix only.

Chambers, J. M. and Hastie, T. J. 1992. Statistical Models in S. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove CA.

This is the basic reference on R and S-PLUS model formulae and models.

Everitt, B. S. 1994. A Handbook of Statistical Analyses using S-PLUS. Chapman and Hall, London.

The choice of analysis methods may seem idiosyncratic. It has little on the more recently developed methods which are S-PLUS's strength.

Harrell, F. 1997. An Introduction to S-PLUS and the Hmisc and Design Libraries.
The latest version of this manual is available from

`http://hesweb1.med.virginia.edu/biostat/s/index.html`

Chapters 1-4 and 9-10 are a good introduction to S-PLUS, likely to be particularly helpful to anyone who comes to R or S-PLUS from SAS. The examples in this manual are largely medical.

Krause, A. and Olsen, M. 1997. The Basics of S and S-PLUS. Springer 1997.

This is an introductory book, at about the same level as Spector.

R Development Core Team 1999. An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics. [Available from the CRAN sites noted in section 13.1.]
This is derived from an original set of notes, written by Bill Venables and Dave Smith for the S and S-PLUS environments.

Spector, P. 1994. An Introduction to S and S-PLUS. Duxbury Press.

This is a readable and compact beginner's guide to the S-PLUS language. Copies are available from the ANU Co-op bookshop.

Venables, W. N. and Ripley, B. D., 2nd edn 1997. Modern Applied Statistics with S-PLUS. Springer, New York.

This has become a text book for the use of S-PLUS for applied statistical analysis. It assumes a fair level of statistical sophistication. Explanation is careful, but often terse. Together with the ‘Complements’ it gives brief introductions to extensive libraries of functions that have been written or adapted by Ripley, Venables, and a number of other statisticians. Supplementary material (`Complements’) is available from

`http://www.stats.ox.ac.uk/pub/MASS2/.`

The supplementary material is extensive, and is continually supplemented. The present version of the statistical `Complements’ has extensive information on new libraries that have come from third party sources. There is helpful information, additional to what is in the book, that is specific to the S-PLUS 4.0 and S-PLUS 4.5 releases for Microsoft Windows.

R Development Core Team 1999. An Introduction to R.
This document is available from the CRAN sites noted in section 13.1.

13.3 The R-help electronic mail discussion list

To subscribe (or unsubscribe) to this list send a message with `subscribe` (or `unsubscribe`) in the body of the message (not in the subject!) to

`r-help-request@stat.math.ethz.ch`

Information about the list can be obtained by sending an email with info as its contents to

`r-help-request@stat.math.ethz.ch`

To send a message to everyone on the r-help mailing list, send email to

`r-help@stat.math.ethz.ch`

Details on two further lists – r-announce and r-devel – are available from

`http://cran.r-project.org`

There is an archive of past discussion which you can search by going to the web page

`http://www.ens.gu.edu.au/robertk/R/`

13.4 Competing Systems – XLISP-STAT

XLISP-STAT is a lisp-based system that, like S-PLUS and R, allows a seamless extensibility. It is available from

`http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html`

See also the code designed to accompany Cook and Weisberg’s book “Applied Regression Including Computing and Graphics” (Wiley 1999), available from

`http://www.stat.umn.edu/arc`

14. Appendix 1

14.1 Data Sets Referred to in these Notes

Data sets accompanying these notes

Barley austpop beams dewpoint dolphins

elastic huron islandcities kiwishade leafshape

milk moths oddbooks orings possum

primates seedrates tinting

Data Set from Library ts

LakeHuron

Data Sets from Library BASE

airquality attitude cars islands

Data Sets from Library MASS

Aids2 Animals Cars93 PlantGrowth Rubber

cement cpus fgl michelson mtcars

painters pressure ships

14.2 Answers to Selected Exercises

Section 1.6

1. plot(distance~stretch,data=elastic)

2. (ii), (iii), (iv)

```
plot(snow.cover ~ year, data = snow)
hist(snow$snow.cover)
hist(log(snow$snow.cover))
```

Section 2.8

1. The value of answer is (a) 12, (b) 22, (c) 600.

2. `prod(c(10,3:5))`

3(i) `bigsum <- 0;  for (i in 1:100) {bigsum <- bigsum+i }; bigsum`

3(ii) `sum(1:100)`

4(i) `bigprod <- 1;  for (i in 1:50) {bigprod <- bigprod*i };  bigprod`

4(ii) `prod(1:50)`

5.
```
radius <- 3:20;  volume <- 4*pi*radius^3/3
sphere.data <- data.frame(radius=radius, volume=volume)
```

Section 3.9

1. `x <- seq(101,112)` or `x <- 101:112`

2. `rep(c(4,6,3),4)`

3. `c(rep(4,8),rep(6,7),rep(3,9))` or `rep(c(4,6,3),c(8,7,9))`

4. `rep(seq(1,9),seq(1,9))` or `rep(1:9, 1:9)`

5. Use `summary(airquality)` to get this information.

6(a) `2 7 7  5 12 12 4`

6(b) `2 9 8 6 17 15 7`

7.
```
airquality[airquality$Ozone == max(airquality$Ozone),]
airquality$Wind[airquality$Ozone > quantile(airquality$Ozone, .75)]
```

8.
```
mean(snow$snow.cover[seq(2,10,2)])
mean(snow$snow.cover[seq(1,9,2)])
```

9.
```
sapply(claims, is.factor)
levels(Cars93$Manufacturer), etc.
```

To check which are ordered factors, type in

```
sapply(claims, is.ordered)
```

10. `summary(airquality); summary(attitude); summary(cpus)`

Comment on ranges of values, whether distributions seem skew, etc.

11. `mtcars6<-mtcars[mtcars$cyl==6,]`

12. `Cars93[Cars93$Type==”Small”|Cars93$Type==”Sporty”,]`

13. `mat34 <- matrix(rep(c(4,6,3),4), nrow=3, ncol=4)`

14. `mat64 <- matrix(c(rep(4,8),rep(6,7),rep(3,9)), nrow=6, ncol=4)`

Section 4.7

1.
```
plot(Animals$body, Animals$brain, pch=1,
    xlab="Body weight (kg)",ylab="Brain weight (g)")
```

2.
```
plot(log(Animals$body),log(Animals$brain),pch=1,
     xlab="Body weight (kg)", ylab="Brain weight (g)", axes=F)
brainaxis <- 10^seq(-1,4)
bodyaxis <-10^seq(-2,4)
axis(1, at=log(bodyaxis), lab=bodyaxis)
axis(2, at=log(brainaxis), lab=brainaxis)
box()
identify(log(Animals$body), log(Animals$brain), labels=row.names(Animals))
```

(See problem 4.)

3. `par(mfrow = c(1,2)), etc.`

Additional solutions will be included in later versions of this document.