Using R for Data Analysis and Graphics

John Maindonald

2024-01-26

Table of contents

Preface

This document was designed, when it first appeared in 2000, to cover elementary aspects of the R language syntax and semantics, and to demonstrate by example a limited selection of R graphics and data analysis abilities. It was the starting point for the Cambridge University Press text "Data Analysis and Graphics Using R" (J. Maindonald and Braun 2010).[1]

This 2024 fourth revision should now replace, for who find it still useful, earlier versions. Attention has been primarily on corrections and clarifications, with very limited attention to new features.

Primarily, the focus is on using examples that readers can follow and work through. It demonstrates the use of R for a range of data manipulation, graphical presentation, and statistical analysis tasks. Examples in Chapters 3 and later come with what is often limited explanatory comment.

Those who want to explore the Hadley Wickham *tidyverse* R package collection, using its functions for data manipulation and graphics in place of the limited range of base R functions that get attention in this present document, can find very extensive resources online.[2]

The history of R

R implements a dialect of the S language that was developed at AT&T Bell Laboratories by Rick Becker, John Chambers and Allan Wilks. 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.

The initial version of R was developed by Ross Ihaka and Robert Gentleman, at that time at the University of Auckland. Development of R is now overseen by a core team, widely drawn from different institutions worldwide.

Source code is available for users to adapt and/or improve, or take across to other systems. Exposing code to the critical scrutiny of expert users has proved an effective way to identify bugs and other inadequacies, and to elicit ideas for enhancement. New minor releases appear four or five times a year.

[1] A text that is a derivative of this latter text, titled "A Practical Guide to Data Analysis Using R" (J. Maindonald, Braun, and Andrews 2024, forthcoming), is due for publication towards the middle of 2024.

[2] Notably, check https://www.tidyverse.org/

Obtaining R

Versions of R are available, at no cost, for Microsoft Windows, for Linux, for Unix, and for Macintosh OS X. It is available through the Comprehensive R Archive Network (CRAN). Standard R installations come with *base* and *recommended* packages. As the name suggests the *base* packages provide a base on which other R packages are built. Most users will want to supplement *base* and *recommended* packages with packages that target their own specific requirements.

A language and an environment

The R language environment is designed to facilitate the development of new scientific computational tools. The packages give access to up-to-date methodology from leading statistical and other researchers.

R is a functional language. There is a language core that uses standard forms of algebraic notation, allowing the calculations such as `2+3`, or `3^11`. Beyond this, most computation is handled using functions. The action of quitting from an R session uses the function call `q()`, either directly or invoked by a click on a menu item that is provided for this purpose.

It is often possible and desirable to operate on objects — vectors, arrays, lists and so on – as a whole. This largely avoids the need for explicit loops, leading to clearer code. Section 1.8 has an example.

With the very large address spaces now possible, and as a result of continuing improvements in the efficiency of R's coding and memory management, R's routines can readily process data sets that by historical standards seem large — e.g., on a Unix machine with 2GB of memory, a regression with 500,000 cases and 100 variables is feasible. With very large datasets, the main issue is often manipulation of data, and systems that are specifically designed for such manipulation may be preferable.

Data structure is, typically, an even more important issue for large data sets than for small data sets. Consider that repeated smaller analyses with subsets of the total data may give insight that is not available from a single global analysis.

The use of these notes

The notes are designed so that users can run the examples in the script files (*ch1.R*, *ch2.R*, etc.) using the notes as commentary.

The url https://www.r-bloggers.com/2015/12/how-to-learn-r-2/ has links to a range of different tutorials for learning R. The RStudio dropdown Tutorial menu offers a wide range of tutorial content. Readers of these notes may find it helpful to have available for reference the document: *"An Introduction to R"*, written by W N Venables, D M Smith and the R

Development Core Team, and available from https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf. Books that provide a more extended commentary include J. Maindonald and Braun (2010) .

Points to note are:

> Users who want a point and click interface should investigate the R Commander (*Rcmdr* package) interface.
>
> While R is as reliable as any statistical software that is available, and exposed to higher standards of scrutiny than most other systems, there are traps that call for special care. Some of the model fitting routines are leading edge, with a limited tradition of experience of the limitations and pitfalls. Whatever the statistical system, and especially when there is some element of complication, check each step with care.
>
> The R community is widely drawn, from application area specialists as well as statistical specialists. It is a community that is sensitive to the potential for misuse of statistical techniques and suspicious of what might seem mindless use. Expect scepticism of the use of models that are not susceptible to some minimal form of data-based validation.
>
> The skills needed for the computing are not on their own enough. Neither R nor any other statistical system will give the statistical expertise needed to use sophisticated abilities, or to know when naïve methods are inadequate. A butcher's meat-cleaving skills are unlikely to be adequate for effective animal surgery. Experimentation with the use of R is however, more than with most systems, likely to be an educational experience.

Hurrah for the R development team!

Web Pages and Email Lists

For a variety of official and contributed documentation, for copies of various versions of R, and for other information, go to http://cran.r-project.org and find the nearest CRAN (Comprehensive R Archive Network) mirror site.

There is no official support for R. The r-help email list gives access to an informal support network that can be highly effective. Details of the R-help list, and of other lists that serve the R community, are available from the web site for the R project at https://www.R-project.org/. A search on Stack Overflow can often yield information that is very helpful.

Binary installations of R are available from CRAN sites, for Windows, for MacOS, and for four different flavors of Linux. These come with all the base and recommended packages. Other packages must be installed.

Installation instructions appropriate to the operating system can be found on CRAN sites. Copy down the relevant setup file, click on its icon to start installation, and follow instructions.

For running the examples in the first chapter, ensure that the *DAAG* package is installed. The following command line instruction can be used:

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

1 An overview of the R system

1.1 Learn by typing at the command line

This document mostly assumes that users will type commands into the *command window*, at the command line prompt. The command line prompt, i.e. the `>`, is an invitation to start typing in commands. For example, type `2+2` and press the **Enter** key. Here is what appears on the 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.

For later reference, note that the exit or quit command is

```
q()
```

An alternative to the use of `q()` is to click on the **File** menu and then on **Exit**. There will be a message asking whether to save the workspace image. Clicking **Yes** (the safe option) will save all the objects that remain in the workspace — any that remain from the start of the session and any that have been added since. RStudio users can click on one of the options that are available under the **Session** menu header.

The *workspace* is the name given to a *database* (as it is called) that holds objects (datasets and functions) that have been created or copied in by the user, and have not been subsequently deleted. Depending on session settings, these may add to objects that were saved from a previous session and re-loaded at startup.

Some notational details

As noted earlier, the command line prompt is

```
>
```

R commands (expressions) are typed following this prompt.

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

```
+
```

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

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

Anything that follows `#` on the command line is taken as comment.

Note: Recall that, in order to quit from the R session we could 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!

Base, recommended, and other packages

```
## Base packages
names(which(installed.packages()[ ,"Priority"] == "base", ))
 [1] "base"      "compiler"  "datasets"  "graphics"  "grDevices" "grid"
 [7] "methods"   "parallel"  "splines"   "stats"     "stats4"    "tcltk"
[13] "tools"     "utils"
## Recommended packages
names(which(available.packages(repos =
  c(CRAN = "https://cran.r-project.org"))[ ,"Priority"] == "recommended", ))
 [1] "boot"       "class"      "cluster"    "codetools"  "foreign"
 [6] "KernSmooth" "lattice"    "MASS"       "Matrix"     "mgcv"
[11] "nlme"       "nnet"       "rpart"      "spatial"    "survival"
```

Packages that do not come with the initial distribution must be downloaded and installed separately. A number of packages are by default attached at startup. Names of packages (additional to *base*) that are by default attached at startup can be checked thus:

```
getOption('defaultPackages')
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"
```

To see which packages have been attached at any point in a session, type:

```
search()
```

These may, if earlier session was saved upon quitting and restored on startup for the current session, include packages that were attached in the earlier session.

It pays to have a separate working directory for each major project. RStudio makes it straightforward, both to set up a new project in a new directory, and to move between projects.

Some ways to use R

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)                 # Type ?sqrt to see help for `sqrt`
[1] 3.162278
2*3*4*5
[1] 120
1000*(1+0.045)^5 - 1000  # Interest on $1000, compounded annually
[1] 246.1819
                         # at 4.5% 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
deg <- c(30,60,90)
  # Save the numeric vector `c(30,60,90)` with the name `deg`
sin(deg*pi/180)          # Convert angles to radians, then take sin()
[1] 0.5000000 0.8660254 1.0000000
sin(c(30,60,90)*pi/180)  # The result is the same
[1] 0.5000000 0.8660254 1.0000000
```

Observe that `c()` is a function that joins vectors together

R makes it easy to create plots and other forms of data summary

As a relatively simple example, where all columns are numeric, consider the data frame `austpop` that holds population figures (in thousands) for Australian states and territories, and total population, at various times since 1917. Assuming that the *DAAG* package has been installed, this can be accessed as `DAAG::austpop`. The data are:

```
austpop <- DAAG::austpop
head(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
```

Figure 1.1 uses the function `plot()` to show a plot of the Australian Capital Territory (ACT) population between 1917 and 1997.

Figure 1.1 uses the function `plot()` to show a plot of the Australian Capital Territory (ACT) population between 1917 and 1997.

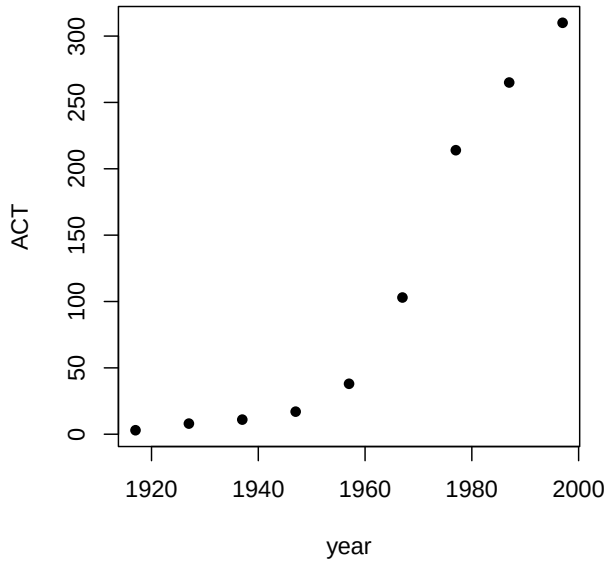

Figure 1.1: Australian Capital Territory (ACT) population between 1917 and 1997

Code is:

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

The option `pch=16` sets the plotting character to a solid black dot. 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.

R provides a huge range of abilities for working with data

Between the base, recommended, and huge range of contributed packages, R offers wide-ranging abilities for data manipulation, for data summary, for graphical display, for fitting models, for simulation, and for a variety of other computations.

Data frames are a standard way to store data. A dataframe is a list of columns, all of the same length. Columns can be numeric, or character, or logical (values are `TRUE` or `FALSE`), or factor, or dates. As a first example, consider the data frame `hills`. This has three columns (variables), with the names distance, climb, and time. Typing `summary(hills)` gives summary information on these variables. There is one column for each variable, thus:

```
hills <- DAAG::hills  # Copy the dataframe `hills`, from the
                      # DAAG package, into the  workspace.
summary(hills)
      dist            climb          time
 Min.   : 2.000   Min.   : 300   Min.   :0.2658
 1st Qu.: 4.500   1st Qu.: 725   1st Qu.:0.4667
 Median : 6.000   Median :1000   Median :0.6625
 Mean   : 7.526   Mean   :1815   Mean   :0.9646
 3rd Qu.: 8.000   3rd Qu.:2200   3rd Qu.:1.1438
 Max.   :28.000   Max.   :7500   Max.   :3.4103
```

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

A helpful graphical summary for the hills data frame is the scatterplot matrix, shown in Figure 1.2.

```
pairs(DAAG::hills)
```

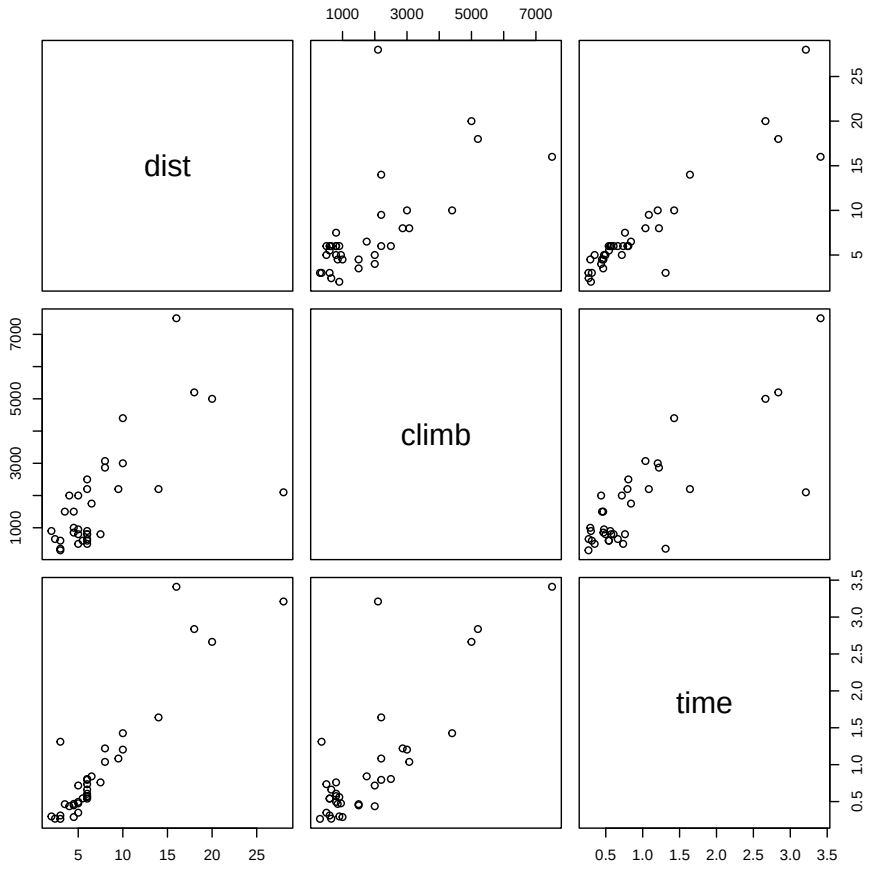

Figure 1.2: Scatterplot matrix for the Scottish hill race data

To reproduce the plot, type

```
pairs(DAAG::hills)
```

Correlation calculations are a form of data summary. The correlation matrix for the `hills` data is:

```
options(digits=3)
cor(DAAG::hills)
       dist climb  time
dist  1.000 0.652 0.920
climb 0.652 1.000 0.805
time  0.920 0.805 1.000
```

There is a case for taking logarithms of data values, and then calculating correlations. This can all be done in one step, thus:

```
cor(log(DAAG::hills))
      dist climb  time
dist  1.00 0.700 0.890
climb 0.70 1.000 0.724
time  0.89 0.724 1.000
```

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

In the straight line regression calculations now demonstrated, the variable names are the names of columns in the data frame `DAAG::elasticband`. The formula that is supplied to the `lm()` (linear model) command asks for the regression of distance traveled by the elastic band (`distance`) on the amount by which it is stretched (`stretch`).

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

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

Coefficients:
(Intercept)      stretch
     -63.57         4.55
```

More complete information is available by typing

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

Figure 1.3 plots the data and adds the regression line:

```
## Code
plot(distance ~ stretch,data=elasticband, pch=16)
abline(elastic.lm)  # Add regression line to graph
```

R is an interactive programming language

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

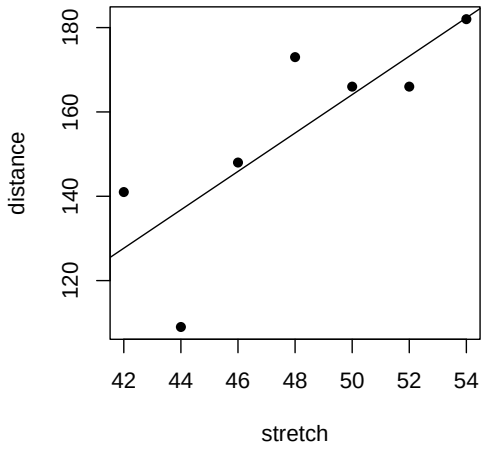

Figure 1.3: Elastic band `distance` versus `stretch`, with regression line added

```
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
```

Extensive help is available from the command line

To get a help on an R object, use `help()` or `?`, thus

Thus, to get help on the R function `plot()`, type:

```
?plot
```

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

```
help.search("matrix")
  ## This lists all functions whose help pages have a title or alias in
  ## which the text string "matrix" appears.
```

```
apropos("matrix")
  ## This lists all function names that include the text "matrix"
```

The function `help.start()` opens a browser window that gives access to the full range of documentation for syntax, packages and functions.

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

1.2 Vectors

Examples of vectors are

```
c(2,3,5,2,7,1)
3:10   # The numbers 3, 4, .., 10
c(TRUE,FALSE,FALSE,FALSE,TRUE,TRUE,FALSE)
c("Canberra","Sydney","Newcastle","Darwin")
```

Vectors may have mode logical, numeric or character . The first two vectors above are numeric, the third is logical (i.e. a vector with elements of mode logical), and the fourth is a string vector (i.e. a vector with elements of mode character). The missing value symbol, which is NA, can be included as an element of a vector.

Joining (concatenating) vectors

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

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

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

Subsets of Vectors

There are two common ways to extract subsets of vectors.
1. Specify the numbers of the elements that are to be extracted, e.g.

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

One can use negative numbers to omit elements:

```
x <- c(3,11,8,15,12)
x[-c(2,3)]
[1]  3 15 12
```

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

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

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

Vectors can have named elements, in which case elements can be extracted by name. For example:

```
height <- c(Andreas=178, John=185, Jeff=183)
height[c("John","Jeff")]
John Jeff
 185  183
```

Patterned Data

Use `5:15` to generate the numbers 5, 6, . . ., 15. Entering 15:5 will generate the sequence in the reverse order. To repeat the sequence (2, 3, 5) four times over, enter `rep(c(2,3,5), 4)` thus:

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

If instead one wants four 2s, then four 3s, then four 5s, enter

```
rep(c(2,3,5),c(4,4,4))    # 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)`.

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`."

1.3 Lists, with dataframes as an important special case

Lists collect together, under a single name, what can be an arbitrary set of R objects. These might be vectors of several different modes and lengths, scalars, dates, matrices or more general arrays, or functions, etc.

Dataframes are lists!

A data frame is a list of variables, all of equal length. Variables can be vectors (integer, or numeric, or character, or logical, or complex). Among other possibilities, they can also be factors, or date objects. For the moment, attention will be limited to dataframes where the columns are integer, or numeric, or character, or logical.

Just as with any other list, subscripting extracts a list. Thus `Cars93.summary[4]` is a data frame with a single column, which is the fourth column vector of Cars93.summary. Use `Cars93.summary[[4]]` or `Cars93.summary[,4]` to extract the column vector.

R packages include a wide variety of datasets, mostly in the form of *dataframe*s. Data frames have a central role in the way that R is set up to process data, fit models, and display graphs.

Operations with dataframes

Among the datasets in the *DAAG* package is `Cars93.summary`, created from information in the `Cars93` data set in the Venables and Ripley *MASS* package. Here it is:

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

Notice that the final column has the *mode* character. Different columns can have different modes – including numeric, character, and logical (values `TRUE` and `FALSE`).

The data frame has row labels (access with `row.names(Cars93.summary)`) `Compact`, `Large`, . . . The column names (access with `names(Cars93.summary)`) are `Min.passengers` (i.e. the minimum number of passengers for cars in this category), `Max.passengers`, `No.of.cars.`, and `abbrev`. The first three columns have mode numeric, and the fourth has mode character. Columns can be vectors of any mode. The column `abbrev` could equally well be stored as a factor – more on that in due course.

Accessing parts of data frames

A data frame that has only the first four rows and omits the third column can be extracted thus:

```
cars <- Cars93.summary[1:4, c(1,2,4)]
cars <- Cars93.summary[1:4, -3]   # Alternative --- specify what to omit
```

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

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

*Merging Data Frames – a simple example {.unnumbered}

The data frame `MASS::Cars93` holds extensive information on data from 93 cars on sale in the USA in 1993. The data frame `DAAG::Cars93.summary` has as row names the distinct values of the factor `Type`. The final column, with the name `abbrev`, holds two character abbreviations of each of the car type names, suitable for use in plotting.

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

We proceed thus to add a column that has the abbreviations to the data frame.

```
Cars93 <- MASS::Cars93
new.Cars93 <- merge(x=Cars93, y=Cars93.summary[,4,drop=F],
            by.x="Type", by.y="row.names")
```

Notice that the row names of `Cars93.summary` are treated as a column of the data frame, with name `row.names`. The effect is to create a data frame that 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. This can be avoided by ensuring that Type has `NA` as one of its levels, in both data frames.

Lists more generally

Output from fitting a model is a list

As an illustration consider the list object that R creates as output from an `lm()` linear model fit.

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

The object `elastic.lm` is a list that brings together several different kinds of objects. The names are:

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

The first list element is:

```
elastic.lm$coefficients
(Intercept)     stretch
     -63.57        4.55
```

Alternative ways to extract this first list element are:

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

We can alternatively ask for the sublist whose only element is the vector `elastic.lm$coefficients`. For this, specify

```
elastic.lm["coefficients"]
$coefficients
(Intercept)     stretch
     -63.57        4.55
# elastic.lm[1] gives the same result
```

Notice that the information is in this case preceded by $coefficients, meaning "list element with name coefficients". The result is a list, now with just the first element of `elastic.lm`.

The second list element is a vector of length 7:

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

The tenth list element documents the function call:

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

1.4 Factors, dates, NAs, and more

Factors and ordered factors

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

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

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

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

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

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

Internally the factor gender is stored as 691 1's, followed by 692 2's. It has stored with it the vector:

```
levels(gender)
[1] "female" "male"
```

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

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

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

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

An alternative is

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

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

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

The following adds site names to the `possum` dataframe:

```
possum <- DAAG::possum
possumsites <- DAAG::possumsites
possum$sitenam <- rownames(possumsites)[DAAG::possum$site]
with(possum, table(sitenam))
sitenam
Allyn River    Bellbird    Bulburin   Byrangery Cambarville   Conondale
          7          13          18          13          33          13
Whian Whian
          7
sitefac <- factor(possum$sitenam)
table(sitefac)
sitefac
Allyn River    Bellbird    Bulburin   Byrangery Cambarville   Conondale
          7          13          18          13          33          13
Whian Whian
          7
```

It is the integer values that are stored. Along with the vector of integer values are stored a list of *attributes*, which holds the level names and the *class* vector:

```
mode(sitefac)
[1] "numeric"
  # The attributes list is for an object of this class
attributes(sitefac)
```

```
$levels
[1] "Allyn River" "Bellbird"    "Bulburin"    "Byrangery"   "Cambarville"
[6] "Conondale"   "Whian Whian"

$class
[1] "factor"
  # This controls the interpretation of `sitefac` a factor
```

Printing the contents of the column with the name `sitefac` gives the names, not the integer values. As in most operations with factors, R does the translation invisibly. There are though annoying exceptions that can make the use of factors tricky. To get back the site names as a character vector and tabulate the result, specify

```
table(as.character(possum$sitefac))
< table of extent 0 >
```

To get the integer values and tabulate the result, specify

```
table(unclass(possum$sitefac))
< table of extent 0 >
```

We might prefer the names to appear in order of latitude, from North to South. We can change the order of the level names to reflect this desired order:

```
ordnam <- rownames(possumsites)[order(possumsites$Latitude)]
possum$sitefac <- factor(possum$sitenam, levels=ordnam)
with(possum, table(sitefac))
sitefac
   Bellbird Cambarville Allyn River Whian Whian   Byrangery   Conondale
         13          33           7           7          13          13
   Bulburin
         18
```

Factors have the potential to cause surprises. Points to note are:
* When a vector of character strings becomes a column of a data frame, R by default turns it into a factor. Enclose the vector of character strings in the wrapper function `I()` if it is to remain character. With tibbles, this is not an issue. * There are some contexts in which factors become numeric vectors. To be sure of getting the vector of text strings, specify e.g. `as.character(possum$sitefac)`.

Ordered Factors

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

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

Ordered factors `inherit` the attributes of factors, and have a further ordering attribute. Asking for the class of an object returns details both of the class of the object, and of any classes from which it inherits.
Thus:

```
class(ordf.stress)
[1] "ordered" "factor"
attributes(ordf.stress)
$levels
[1] "low"    "medium" "high"

$class
[1] "ordered" "factor"
```

Inclusion of character string vectors in data frames

When data are input using `read.table()`, or when the `data.frame()` function is used to create data frames, vectors of character strings are by default turned into factors. The parameter setting `stringsAsFactors=TRUE`, available both with `read.table()` and with `data.frame()`, will if needed ensure that character strings are input without such conversion. For `read.table()`, an alternative is `as.is=TRUE`. When input uses functions in the *readr* package, character strings are left unchanged.

Dates

See `?Dates`, `?as.Date` and `?format.Date` for information on functions in base R for working with dates. Use `as.Date()` to convert text strings into dates. The default is that the year comes first, then the month, and then the day of the month, thus:

```
# Electricity Billing Dates
dd <- as.Date(c("2003/08/24","2003/11/23","2004/02/22","2004/05/23"))
diff(dd)
Time differences in days
[1] 91 91 91
```

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

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

The default format is `%Y-%m-%d`. The function `as.Date()` takea `vector of character strings that has an appropriate format, and converts it into a dates object. By default, dates are stored using January 1 1970 as origin. This becomes apparent when`as.integer()‘ is used to convert a date into an integer value. Examples are:

```
as.Date("1/1/1960", format="%d/%m/%Y")
[1] "1960-01-01"
as.Date("1:12:1960",format="%d:%m:%Y")
[1] "1960-12-01"
as.Date("1960-12-1")-as.Date("1960-1-1")
Time difference of 335 days
as.Date("31/12/1960","%d/%m/%Y")
[1] "1960-12-31"
as.integer(as.Date("1/1/1970","%d/%m/%Y"))
[1] 0
as.integer(as.Date("1/1/2000","%d/%m/%Y"))
[1] 10957
```

The function `format()` allows control of the formatting of dates when they are printed. `See ?format.Date`.

```
dec1 <- as.Date("2004-12-1")
format(dec1, format="%b %d %Y")
[1] "Dec 01 2004"
format(dec1, format="%a %b %d %Y")
[1] "Wed Dec 01 2004"
```

As with factors, the underlying storage mode is a numeric vector.

```
mode(dd)
[1] "numeric"
julian(dd)
[1] 12288 12379 12470 12561
attr(,"origin")
[1] "1970-01-01"
  # Makes it clear that date is days since '1970-01-01'
julian(as.Date('1970-01-01'))
[1] 0
attr(,"origin")
[1] "1970-01-01"
```

`NA`s (missing Values), `NaN` (not a number) and `Inf`

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. Users who do not carefully consider implications for expressions that include `NA`s may be puzzled by the results. 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, this gives no information 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             # Result is NA, for all elements of `x`
[1] NA NA NA NA
NA == NA
[1] NA
```

The Use of NA in variable subscripts

Any arithmetic operation or relation that involves NA generates an NA. Set

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

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

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

The following, where the subscript vector on both sides has one or more missing values, generates an error message:

```
x <- c(1,6,2,NA)
y <- 11:15
y[x>2] <- x[x>2]
Error in y[x > 2] <- x[x > 2]: NAs are not allowed in subscripted assignments
```

Use !is.na(x) to limit the selection, on both sides, to those elements of x that are not NAs.

Inf and NaN

The following are allowed:

```
c(-1/0, 1/0, 0/0, 1/Inf)
[1] -Inf  Inf  NaN    0
```

It is up to the user to ensure that allowing such calculations to proceed leads to results that make sense.

1.5 Matrices and arrays

All elements of a matrix have the same mode, i.e. all numeric, or all character. Thus a matrix is a more restricted structure than a data frame. One reason for numeric matrices is that they allow a variety of mathematical operations that are not available for data frames. Matrices are likely to be important for those users who wish to implement new regression and multivariate methods. The matrix construct generalizes to array, which may have more than two dimensions. Matrices are stored columnwise, in a single vector. 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 one long vector `x`. In the example just given, we get back the elements 1, 2, . . . , 6. Matrices have the attribute "dimension". Thus

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

Thus, a matrix is a vector (numeric or character or logical) whose dimension attribute has length 2.
Now set

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

Examples of the extraction of columns or rows or submatrices are:

```
x34 <- matrix(1:12, nrow=3)
x34[2:3, c(1,4)]  # Extract rows 2 & 3 & columns 1 & 4
x34[2,]           # Extract the second row
x34[-2,]          # Extract all rows except the second
x34[-2,-3]        # Omit row 2 & column 3
```

The `dimnames()` function assigns and/or extracts matrix row and column names. The is 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 generalizes in the obvious way for use with arrays, which we now discuss.

Arrays

The generalization from a matrix (2 dimensions) to allow more than 2 dimensions gives an array. A matrix is a 2-dimensional array. Consider a numeric vector of length 24. So that we can easily keep track of the elements, we will make them 1, 2, .., 24. Thus

```
x <- 1:24
dim(x) <- c(2,12)
# `x` is then a 2 x 12 matrix.
x
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    1    3    5    7    9   11   13   15   17    19    21    23
[2,]    2    4    6    8   10   12   14   16   18    20    22    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
```

Conversion of numeric dataframes into matrices

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

1.6 Data entry and editing

Entry of Data at the Command Line

The following data gives, for each amount by which an elastic band is stretched over the end of a ruler, the distance that the band moved when released:

```
stretch   46  54  48   50  44  42  52
distance 148 182 173  166 109 141 166
```

The function `data.frame()` can be used to input these (or other) data directly at the command line. We will give the data frame the name `elasticband`:

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

Entry and/or editing of data in an editor window

To edit the data frame elasticband in a spreadsheet-like format, type

```
elasticband <- edit(elasticband)
```

Options for `read.table()`

Under RStudio, input of data from files is very conveniently handled by selecting *File* | *ImportDataset* from the menu. Functions in the *tidyverse* collection greatly extend the range of data structure types that can be readily input into R. For this purpose, it is convenient to work from the RStudio menu, where the default is to read data into the data frame variant that has the name *tibble*.

The base R function `read.table()` takes, optionally various parameters additional to the file name that holds the data. Specify `header=TRUE` if there is an initial row of header names. The default is `header=FALSE`. In addition users can specify the separator character or characters. Command alternatives to the default use of a space are `sep=","` and `sep="\\t"`. This last choice makes tabs separators. Similarly, users can control over the choice of missing value character or characters, which by default is `NA`. If the missing value character is a period (“.”), specify `na.strings="."`.

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

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

1.7 R objects, the workspace, and attached packages

To attach the *MASS* package, type:

```
library("MASS")

Attaching package: 'MASS'
The following object is masked _by_ '.GlobalEnv':

    hills
```

It will then be attached at position 2 on the 'search list', which is the list of *databases* (as they are termed) that R searches for datasets or functions whose source has not been specifically identified. Position 1 is reserved for *workspace* objects that have been created or copied in by the user.

Functions and datasets are specific types of R *objects*. Notice that the first name on the search list is `.GlobalEnv`, which refers to the workspace. The following code 'loads' the dataframe `cabbages`, from the *MASS* package, into the workspace:

```
cabbages <- MASS::cabbages
```

If the *MASS* package has earlier been attached, there will then be two places on the search list where it can be found. The name `cabbages`, appearing on its own, would be taken to refer to the version in the workspace, not to that in the attached package *MASS*.

The function `with()`

The function `with()` attaches the data frame (or, it can be a list) that is given as its first argument, within a specially created *environment* for the duration of the calculation(s) that are specified by its second argument. The *environment* here is neither the workspace nor an attached database. See `?environment` for details of what, in general, constitutes an environment.

For example:

```
av <- with(trees, mean(Height))
  # The assignment places the result in the worksapce.
```

The environment is then the first place searched, looking for a column with the name `Height`.

```
with(list(x=1:3, y=5:8), mean(y))
[1] 6.5
```

Saving the workspace

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

Typing the name of an object causes the printing of its contents. Try typing q, mean, etc. In a long session, it makes sense to save the contents of the working directory from time to time. It is also possible to save individual objects, or collections of objects into a named image file. Some possibilities are:

```
save.image()           # Save contents of workspace, into the file .RData
save.image(file="archive.RData")      # Save into the file archive.RData
save(celsius, fahrenheit, file="tempscales.RData")
```

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

The function `save()` can be used to save a specific set of R objects into a named image file.

Functions and datasets – one at a time, or per database?

The dataset `possum` from the *DAAG* can be accessed as `DAAG::possum`, provided *DAAG* is installed on the computer. This has the advantage of leaving no room for ambiguity over the source of the dataset.

Datasets can alternatively be made available, using the functions `attach()` or `load()`, on a per database basis. For this purpose, a database is an R package, or an *image* file that holds some or all of the datasets that have been saved from an R session. Or it can be a list or data frame, allowing the list elements or data frame columns to be referred to directly, without reference to the list or data frame. The following first saves the dataframes `DAAG::possum` and `DAAG::cuckoos` to the image file `misc.RData` in the working directory (enter `getwd()` to check where that is), and then attaches `misc.RData`, making both these datasets available from a database that is placed at position 2 on the search list:

```
possum <- DAAG::possum; cuckoos <- DAAG::cuckoos
save(possum, cuckoos, file="misc.Rdata")
rm(possum, cuckoos)  # Remove from the workspace
# Use `attach()` to make these available again.
attach("misc.RData")
```

Be aware that if an object of the same name happens to be present in the workspace, that will be taken instead. The function `load()` can alternatively be used to load the objects saved in `misc.RData` into the workspace, which might be a safer way to proceed.

Individual dataframes can also be attached or loaded.

```
possum <- DAAG::possum   # Load `possum` into the workspace
attach(DAAG::possum)     # Add DAAG::possum to the search list
  # Its columns can then be directly referenced by name
```

Databases that are attached in the course of a session are by default added at position 2 on the search list. Set the argument `pos` to a value greater than 2 in order to add at a later position.

Image files, from the working directory or (with the path specified) from another directory, can be attached, thus making objects in the file available on request. For example

```
attach("tempscales.RData")
ls(pos=2)   # Check the contents of the file that has been attached
```

The parameter `pos` gives the position on the search list.

Datasets in R packages

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

```
data(package="datasets")
```

and similarly for any other package. In most packages, data from an attached package are automatically available. Use of e.g., `data(airquality)` to attach the data set `airquality` (*datasets* package) is unnecessary. The out-of-the-box Windows and other binary distributions include a number of commonly required packages, including *datasets*. Other packages must be explicitly installed.

The base package, and several other packages (including *datasets*), are automatically attached at the beginning of the session. To attach any other installed package, use the `library()` command.

1.8 Functions in R

We give two simple examples of R functions.

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 that appears in the function body . Use the function thus

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

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

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

A function that returns 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)
}
```

Notice that a default argument is supplied. Now invoke the function:

```
mean.and.sd()       # Uses default argument
mean   SD
5.50 3.03
mean.and.sd(hills$climb)
mean   SD
1815 1619
```

Looping – the `for()` function

A simple example of a for loop is

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

Here is another example:

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

A better way to formulate the calculation is:

```
celsius <- 25:30
print(9/5*celsius+32)
[1] 77.0 78.8 80.6 82.4 84.2 86.0
```

Skilled R users have limited recourse to loops. There are often, as in this and earlier examples, better alternatives.

Function syntax and semantics

A function is created using an assignment. On the right hand side, the parameters appear within round brackets. A default can, optionally, be provided. 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`.

An example of a user function

The data set `mtcars` in the *datasets* package has data on 63 cars, as given in the 1974 Motor Trend US magazine.

```
with(mtcars, plot(hp, mpg))
## Alternatively:
with(mtcars, plot(mpg ~ hp))   # mpg ~ hp is a graphics formula
```

Here is a function that makes it possible to plot the figures for any pair of columns of `mtcars`:

```
plot.mtcars <- function(form=mpg~hp, data=mtcars){
    plot(form, data=data)
    vars <- all.vars(form)   # Extract the variable names
    mtext(side=3, line=1.5,
     paste("Plot of", vars[1], "versus", vars[2]))
}
```

Observe that the function body is enclosed in braces ({ }). Figure 1.4 shows the graph produced by `plot.mtcars()`. Parameter settings were left at their defaults.

```
plot.mtcars()
```

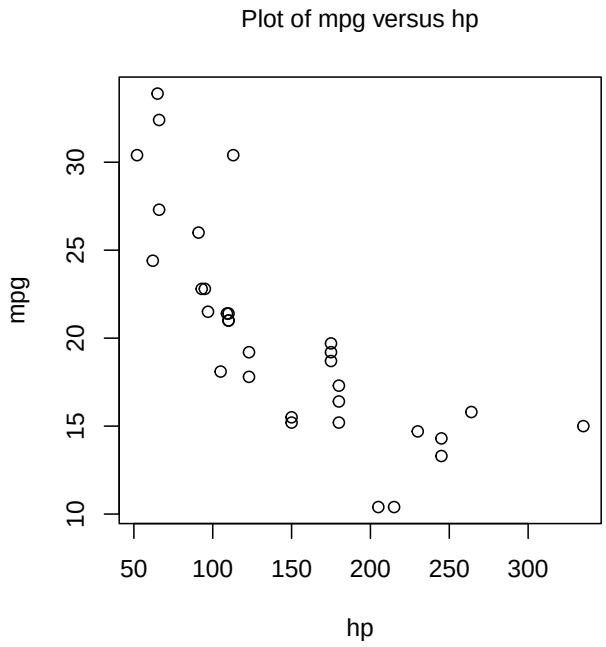

Figure 1.4: Plot of miles per gallon versus horsepower, for cars in the `mtcars` data frame

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, allow the argument `na.rm=T`; i.e. remove `NA`s, then proceed with the calculation. By default, `sort()` omits any `NA`s. The function `order()` places `NA`s 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
```

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.
```

The functions `sapply()`, `lapply()`, `apply()`, and `tapply()`

The functions are called as follows:

```
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.
apply(<array>, <dimension>, <function>)
```

Both `sapply()` ('s'='simplify') and `lapply()` ('l'='list') can be used with lists or with vectors, as well as with dataframes.
The function `lapply()` works in the same way as `sapply()`, but generates a list, and does not attempt to bring the list elements together into a common structure.

to the rows or columns of a matrix, can also be used with matrices. It can also be used with arrays of more than two dimensions.

The functions `sapply()` and `lapply()` take as arguments data frame, and the function that is to be applied. The following applies the function `is.factor()` to all columns of the supplied data frame rainforest.

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

One can specify na.rm=TRUE as a third argument to `sapply()`.
This argument is then automatically passed to the function that is specified in the second argument position. For example:

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

The following code returns that number of missing values in each column of the data frame `airquality`.

```
sapply(airquality, function(x)sum(is.na(x)))
  Ozone Solar.R    Wind    Temp   Month     Day
     37       7       0       0       0       0
```

The function `apply()` can be used on data frames as well as matrices and arrays. An example that calculates means for each column is:

```
apply(airquality,2,mean)   # All elements must be numeric!
  Ozone Solar.R    Wind    Temp   Month     Day
     NA      NA    9.96   77.88    6.99   15.80
apply(airquality,2,mean,na.rm=TRUE)
  Ozone Solar.R    Wind    Temp   Month     Day
  42.13  185.93    9.96   77.88    6.99   15.80
```

The use of `apply(airquality,1,mean)` will give means for each row.
These are not, for these data, useful information!

*An example that uses `sapply()` with a vector of text strings {.unnumbered}

We will work with the column `Make` in the dataset `MASS::Cars93`. To find the position at which the first space appears in the information on make of car, we might do the following:

```
Cars93 <- MASS::Cars93
car.brandnames <- sapply(strsplit(as.character(Cars93$Make), " ", fixed=TRUE),
                         function(x)x[1])
car.brandnames[1:5]
[1] "Acura" "Acura" "Audi"  "Audi"  "BMW"
```

Using `aggregate()` and `tapply()`

The arguments are in each case a variable, a list of factors, and a function that 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 function `aggregate()` returns a data frame. For example:

```
cabbages <- MASS::cabbages
str(cabbages)
'data.frame':   60 obs. of  4 variables:
 $ Cult  : Factor w/ 2 levels "c39","c52": 1 1 1 1 1 1 1 1 1 1 ...
 $ Date  : Factor w/ 3 levels "d16","d20","d21": 1 1 1 1 1 1 1 1 1 1 ...
 $ HeadWt: num  2.5 2.2 3.1 4.3 2.5 4.3 3.8 4.3 1.7 3.1 ...
 $ VitC  : int  51 55 45 42 53 50 50 52 56 49 ...
with(cabbages, aggregate(HeadWt, by=list(Cult=Cult, Date=Date), FUN=mean))
  Cult Date    x
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
```

The syntax for `tapply()` is similar, except that the name of the second argument is `INDEX` rather than `by`. 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.

Functions for Confidence Intervals and Tests

Two of the simpler functions are `t-test` (allows both a one-sample and a two-sample test) and `chisq.test()` for testing for no association between rows and columns in a two way table.

(This assumes counts enter independently into the cells of the table. The test is invalid if there is clustering in the data.)

Use the help pages to get more complete information.

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 operator `%in%` can be used 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 <- x %in% c(2,4)
two4
 [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[13] FALSE FALSE FALSE
# Now pick out the 2s and the 4s
x[two4]
[1] 2 2 2 4 4 4
```

Compare Working Directory Data Sets with a Reference Set

At the beginning of a new session, we might store the names of the objects in the working directory in the vector dsetnames, thus:

```
dsetnames <- objects()
```

Now suppose that we have a function additions(), defined thus:

```
additions <- function(objnames = dsetnames)
{
        newnames <- objects(pos=1)
        existing <- as.logical(match(newnames, objnames, nomatch = 0))
        newnames[!existing]
}
```

At some later point in the session, we can enter

```
additions(dsetnames)
```

to get the names of objects that have been added since the start of the session.

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 15 questions each with five alternatives. If the choice is truly random, with the same probability for all 5 possibilities, we can simulate the correctness of the student for each question by using `sample(1:5, size=1)`, taking `1` as 'correct' and any other number as 'wrong'. For 15 questions we can do `sample(1:5, replace=T, size=15)`, and repeat this perhaps 1000 times to get an idea of the distribution of number of right answers. Or we can calculate the distribution as `dbinom(x, size=15, prob=0.2)`, with `x` running from 0 to 15.

```
resp1000 <- matrix(sample(1:5, replace=T, size=15000),nrow=1000)
  # Each row counts as one set of responses
correct <- apply(resp1000, 1, function(x)sum(x==1))
## Look at distribution of number of correct answers.
table(correct)/1000
correct
    0     1     2     3     4     5     6     7     8     9
0.032 0.123 0.235 0.238 0.212 0.099 0.043 0.015 0.002 0.001
## Now use `dbinom()` to obtain expected numbers
setNames(round(dbinom(0:15, size=15, prob=0.2), 3), 0:15)[1:11]
    0     1     2     3     4     5     6     7     8     9    10
0.035 0.132 0.231 0.250 0.188 0.103 0.043 0.014 0.003 0.001 0.000
  # For x>0, the probability is no more than 0.0005
round(pbinom(8, size=15, prob=0.2, lower.tail=F),4)  ## 9 or more
[1] 8e-04
```

Poisson Random Numbers

One can think of the Poisson distribution as the distribution of the total for occurrences of rare events. For example, 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 that is close to Poisson. However, the total number of people injured is unlikely to follow a Poisson distribution. Why?

The function using `rpois()` generates Poisson random numbers. Suppose for example that 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.

Functions that assist with data management

Where data, labeling 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 labeling 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 labeling 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.

Issues for the Writing and Use of Functions

There can be many functions. Choose their names and argument names carefully, so that they are meaningful, even if this means that they are longer than one would like.
There are mechanisms by which names and argument names can be abbreviated in actual use. 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]. Settings that 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 labeling. 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 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 the r-help electronic mail list (section 13.3) for useful functions for routine tasks.

Wherever 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 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.

Structure code to avoid multiple entry of information.

1.9 Making Tables

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

```
library(lattice)    # The data frame `barley` is included in lattice
table(barley$year, barley$site)

      Grand Rapids Duluth University Farm Morris Crookston Waseca
  1932           10     10              10     10        10     10
  1931           10     10              10     10        10     10
```

WARNING: `NA`s are by default ignored. 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 it is necessary to generate a new factor that includes `NA` as a level. Specify `x <- factor(x,exclude=NULL)`.

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

Numbers of NAs in subgroups of the data

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

```
rainforest <- DAAG::rainforest
table(rainforest$species, !is.na(rainforest$branch))

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

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

1.10 Pipes – A "do this, then do that" syntax

The following calculates the difference between the largest and the smallest tree height, in the `trees` dataframe (*datasets* package):

```
# Use the relation: y/x = exp(log(y/x) = exp(log(y)-log(x))
diff(range(trees$Height))
[1] 24
```

Using a pipe, this can be written:

```
trees$Height |>     # Pipe vector of heights to function `range()`
  range() |>        # Pipe (min, max) vector to function `diff()`
  diff()
[1] 24
```

The data object (here, in both cases, a vector) is taken as the first argument of the function to which it is piped. Notice the ability to add commentary, at each step, that describes the operation that is to be performed.

To assign result, use either of the following:

```
d <- trees$Height |> range() |> diff()
trees$Height |> range() |> diff() -> d
  # Uses right assignment operator
```

The assignment is the last operation performed. It is then in the spirit of the use of pipes to use the right assignment operator (`->`) to position the assignment last.

The following creates a new data frame where the columns hold the logarithms of the values of `Girth`, `Height`, and `Volume`, and changes the column names accordingly:

```
logtrees <- setNames(log(trees), c("logGirth","logHeight","logVolume"))
```

There are three steps here – taking the logarithms, and changing the names. What is happening may be clearer if the pipe syntax is used to set them out in order:

```
trees |> log() |>
         setNames(c("logGirth","logHeight","logVolume")) -> logtrees
```

Notice the specifying of the new names as a second argument, following the argument that is piped through from the previous line, to the function `setnames()`.

Pipe to several functions, or to a later argument than the first

The trick is to create an anonymous function for which the argument concerned is the only argument:

```
LakeHuron |>
    (function(x)c(mean = mean(x), SD = sd(x), n=length(x)))() |>
  print(digits=3)
  mean     SD      n
579.00   1.32  98.00
```

Wrapping round brackets around the function definition causes it to be treated as a function name.

Note that `function(x)` can be abbreviated to `\(x)`.

1.11 Methods

R is an object-oriented language. Objects may have a *class*. Functions that have generic implementations include `print()`, `summary()`, and `plot()` the class of the object supplied as argument 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 that have not been assigned a class.

More generally, the class attribute of an object may be a character vector. 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"
```

1.12 Some Further Programming Niceties

Extracting Arguments to Functions

When an argument appears inside a function, it is taken to be another name for the object to which the argument refers. Other possibilities are that one might want the name itself, or the character vector that has the name. Thus, consider:

```
fun1 <- function(x)x
fun2 <- function(x)substitute(x)
fun3 <- function(x)deparse(substitute(x))
```

Now see what happens, in the three cases, when the argument is the linear model object created thus:

```
cars.lm <- lm(dist ~ speed, data=cars)
fun1(cars.lm)

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
```

```
(Intercept)        speed
     -17.58         3.93
fun2(cars.lm)
cars.lm
fun3(cars.lm)
[1] "cars.lm"
```

Parsing and Evaluation of Expressions

When R encounters an expression such as `mean(x+y)` or `cbind(x,y)`, there are two steps:
* The text string is parsed and turned into an expression, i.e. the syntax is checked and it is turned into code that the R computing engine can more immediately evaluate. * The expression is evaluated.
Upon typing in

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

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

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

stores this unevaluated expression in `my.exp`. The actual contents of `my.exp` are a little different from what is printed out. R gives as much information as it thinks helpful.

Note that `expression(mean(x+y))` is different from `expression("mean(x+y)")`, as is obvious when the expression is evaluated.
Here is an example:

```
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)"
```

The function `parse()`, used with the argument `text`, takes code that is stored in a text string and turns it into an expression.

```
parse(text="mean(x+y)")
expression(mean(x + y))
expression(mean(x + y))
expression(mean(x + y))
# We store the expression in my.exp2, and then evaluate it
my.exp2 <- parse(text="mean(x+y)")
eval(my.exp2, list(x=1:5, y=13:17))
[1] 18
with(list(x=1:5, y=13:17), eval(my.exp2))
[1] 18
```

1.13 Next steps

It may pay, at this point, to glance through chapters 6 and 7, which have a more detailed coverage of the topics in this chapter. Remember also to use R's help pages and functions. Topics from chapter 6, additional to those covered above, that may be important for relatively elementary uses of R include:

- The entry of patterned data (6.1.3)
- The handling of missing values in subscripts when vectors are assigned (7.2)
- Unexpected consequences (e.g. conversion of columns of numeric data into factors) from errors in data (6.4.1).

1.14 Exercises

1. In the data frame `elasticband` from section 1.3.1, plot `distance` against `stretch`.
2. The following creates a data frame that has October snow cover (millions of square kilometers)) in Eurasia for the years 1970-79:

```
snowCover <- data.frame(
  year = 1970:1979,
  snow.cover = c(6.5, 12, 14.9, 10, 10.7, 7.9, 21.9, 12.5, 14.5, 9.2))
```

i. Enter the data into R.
ii. Plot `snow.cover` versus `year`.

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`, from the *DAAG* package.)

Temperature	Erosion	Blowby	Total
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 dataframe, with (for example) column names `temperature`, `erosion`, `blowby` and `total`. (Refer back to Section 1.6). Plot total incidents against temperature.

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

```
## a)
answer <- 0
for (j in 3:5){ answer <- j+answer }
## b)
answer<- 10
for (j in 3:5){ answer <- j+answer }
## c)
answer <- 10
for (j in 3:5){ answer <- j*answer }
```

5. 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!

6. 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?

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

8. The volume of a sphere of radius r is $4r^3/3$. For spheres having radii 3, 4, 5, ..., 20 find the corresponding volumes and print the results out in a table. Construct a data frame with columns `radius` and `volume`.

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

10. Generate and save the following sets of numbers:

(a) Generate the numbers 101, 102, …, 112, and store the result in the vector `x`.

(b) Generate four repeats of the sequence of numbers (4, 6, 3).

(c) Generate the sequence consisting of eight 4s, then seven 6s, and finally nine 3s. Store the numbers obtained , in order, in the columns of a 6 by 4 matrix.

(d) Create a vector consisting of one 1, then two 2's, three 3's, etc., and ending with nine 9's.

11. For each of the following calculations, what you would expect? 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])}
```

12. In the data frame `airquality` (*datasets* package):

(a) Determine, for each of the columns, the median, mean, upper and lower quartiles, and range;

(b) Extract the row or rows for which Ozone has its maximum value;

(c) extract the vector of values of `Wind` for values of `Ozone` that are above the upper quartile.

13. 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.

14. Determine which columns of the data frame `MASS::Cars93` are factors. For each of these factor columns, print out the levels vector. Which of these are ordered factors?

15. Use `summary()` to get information about data in the data frame `attitude` (in the `datasets` package), and `MASS::cpus`. Comment, for each of these data sets, on what this reveals.

16. From the data frame `MASS::mtcars` extract a data frame `mtcars6` that holds only the information for cars with 6 cylinders.

17. From the data frame `MASS::Cars93`, extract a data frame which holds information for small and sporty cars.

18. Use the function `sample()` function to generate 100 random integers between 0 and 19. Now look up the help for `runif()`, and use it for the same purpose. [Count a value between 0 and 0.05 as 0, where the range is from 0 to 1.]

19. Write a function that will take as its arguments a list of response variables, a list of factors, a data frame, and a function such as mean or median. It will return a data frame in which each value for each combination of factor levels is summarized in a single statistic, for example the mean or the median.

20. 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

21. Seventeen people rated the sweetness of each of two samples of a milk product on a continuous scale from 1 to 7, one sample with four units of additive and the other with one unit of additive. The data frame `DAAG::milk`, with columns `four` and `one`, shows the ratings.
    Here is a function that plots, for each patient, the four result against the one result, with the same range for the `x` and `y` axes.

```
plot.one <- function(){
  xyrange <- range(milk)   # Calculates the range of all values in the data frame
  par(pin=c(6.75, 6.75))      # Set plotting area = 6.75 in. by 6.75 in.
  plot(four, one, data=milk, xlim=xyrange, ylim=xyrange, pch=16)
  abline(0,1)                  # Line where four = one
}
```

Rewrite this function 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`.

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

23. Write q wrapper function for one-way analysis of variance that 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 show the boxplot.

24. Use the function `zoo::rollmean` to compute the moving average of order 2 of the data for levels of Lake Erie, included in the dataset `DAAG::greatLakes`. Repeat for order 3.

25. 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.

26. Write a function that 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.

27. 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.) Repeat several times. Now, generate 50 random integers between 0 and 9, with replacement. Determine the proportion of integers that occur more than once.

28. Write an R function that 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.

29. Write an R function that simulates a student guessing at a multiple choice test consisting of 40 questions, where there is a 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.

30. Write an R function that 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?

31. 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.

32. Write a function that 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.

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

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

(a) `xn <- rnorm(100)` (b) `xe <- rexp(100)`

(b) `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.

35. *The vector x consists of the frequencies
5, 3, 1, 4, 6
The first element is the number of occurrences of level 1 of a factor, the second is the number of occurrences of level 2, and so on. Write a function that 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. . . .]

36. *Write a function that calculates the minimum of a quadratic expression, and the value of the function at the minimum.

37. *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.

38. *Given data on trees at times 1, 2, 3, 4, . . ., write a function that calculates the matrix of "average" relative growth rates over the for each interval.

1.15 References and reading

Braun and Murdoch (2021) . A first course in statistical programming with R.

Dalgaard (2008) . Introductory Statistics with R.
[This introductory text has a biostatistical emphasis.]

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.
[Appendix A gives a brief overview of the R language and system.]

Matloff (2011) . The Art of R Programming.

Murrell (2009) . Introduction to data technologies.

Muenchen (2011) . R for SAS and SPSS Users.

Venables and Ripley (2002) . Modern Applied Statistics with S.
[This assumes a fair level of statistical sophistication. Explanation is careful, but often terse.]

Wickham (2016) . R for Data Science.

References to packages

DAAG: J. H. Maindonald and Braun (2022)

lattice: Sarkar (2023)

MASS: Ripley (2023)

2 Base R Graphics

Most of the attention will be on base graphics. Base graphics functions in the style of `plot()` and allied functions are designed to directly create graphs that appear in printed output or on the screen.

This contrasts with the approach in *lattice* and in *ggplot2* graphics, where graphics functions create graphics *objects*, i.e., instructions for creating a plot, rather than a plot on the screen or page. Plotting (or printing) is then a process by which those instructions are used to create a plot on the graph or screen. In this context, `plot()` (or `print()`) are used as generic functions, whose action is determined by the *class* of object.

2.1 The base graphics `plot()` scatterplot function

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

```
demo(graphics)
```

Press the Enter key to move to each new graph.

Plot methods for different classes of object

The plot function is a generic function that has special methods for "plotting" various different classes of object. For example, plotting a data frame gives, for each numeric variable, a normal probability plot. Plotting the lm object that is created by the use of the `lm()` linear model function gives diagnostic and other information that is designed to help in the interpretation of regression results. Try

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

2.2 `plot()` and allied functions – some further details

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

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

It is usual for `x` and `y` to be numeric vectors of the same length. (Also possible, but rarely used, is for `x` to be a factor.) Try

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

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

Two further examples are:

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

The `points()` function adds points to a plot. The `lines()` function adds lines to a plot . The `text()` function adds text at specified locations. The `mtext()` function places text in one of the margins. The `axis()` function gives fine control over the adding of axis ticks and labels, usually on a graph where one or both axes have been omitted (`xaxt='n'` or `yaxt='n'` or `axes=false`) from the initial plot.

Here is a further possibility

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

Size, colour, and choice of plotting symbol

For `plot()` and `points()` the parameter `cex` ("character expansion") controls the size, while `col` ("colour") controls the colour of the plotting symbol. The parameter `pch` controls the choice of plotting symbol. The parameters `cex` and `col` may be used in a similar way with `text()`. Try

```
oldpar <- par(mar=rep(0.15,4))
plot(1, 1, xlim=c(1, 7.35), ylim=c(1.75,5), type="n", axes=F, xlab="",
        ylab="")   # Do not plot points or axes
box()              # Draw a box around the plot area
points(1:7, rep(4.65, 7), cex=2:8, col=1:7, pch=0:6, lwd=4)
text(1:7,rep(3.5, 7), labels=paste(0:6), cex=2:8, col=1:7)
```

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

```
points(1:7,rep(2.5,7), pch=(0:6)+7, cex=3)  # Plot symbols 7 to 13
text((1:7), rep(2.5,7), paste((0:6)+7), pos=4, cex=1)  # Add symbol number
points(1:7,rep(2,7), pch=(0:6)+14, cex=3)       # Plot symbols 14 to 20
text((1:7), rep(2,7), paste((0:6)+14), pos=4, cex=1)   # Add symbol number
par(oldpar)
```

Note the use of pos=4 to position the text to the right of the point (1=below, 2=left, 3=top, 4=right). Figure 2.1 shows the plots:

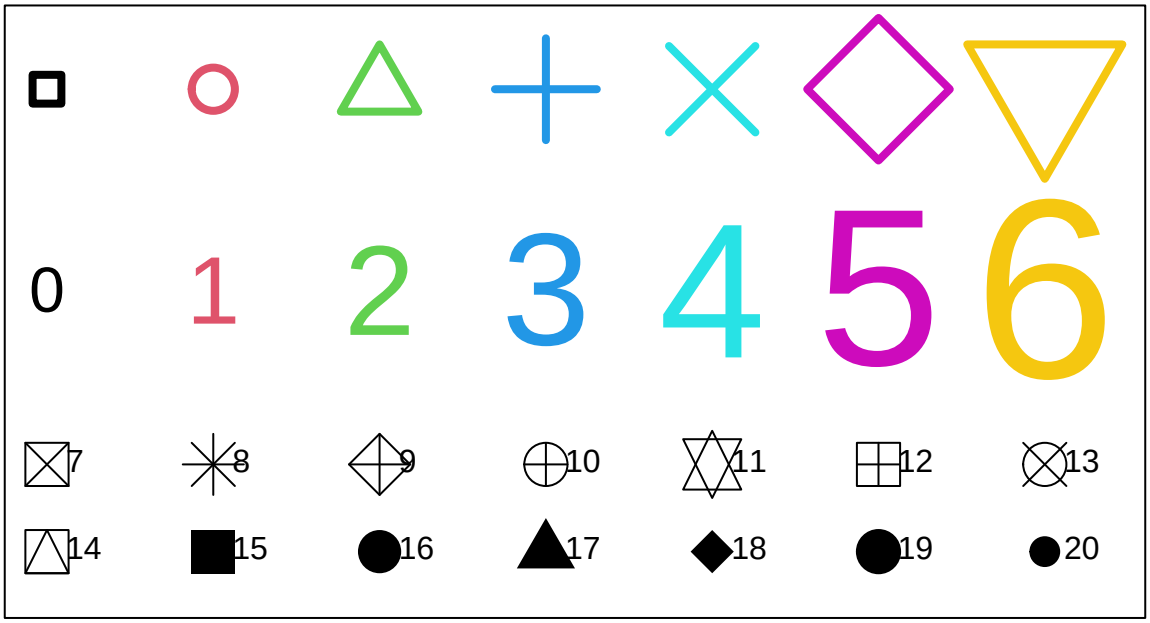

Figure 2.1: Different plot symbols, colours and sizes

A variety of color palettes are available. Figure 2.2 shows some of the possibilities:

```
view.colors <- function(xlim=c(0.55,10.9), ylim=c(0,4)){
  oldpar <- par(mar=rep(0.15,4))
    plot(1, 1, xlim=xlim, ylim=ylim, type="n", axes=F,
         xlab="",ylab="", xaxs="i", yaxs="i")
    text(1:8, rep(3.5,8), paste(1:8), col=palette()[1:8], cex=4)
    text(9, 3.5, "Default palette", adj=0)
    rainchars <- c("R","O","Y","G","B","I","V")
    text(1:7, rep(2.5,7), rainchars, col=rainbow(7), cex=4)
    text(9, 2.5, "rainbow(7)", adj=0)
  ## topo.colors()
    rect((1:8)-0.5, rep(1.05,8), (1:8)+0.5, rep(1.95,8), col=topo.colors(8))
    text(1:8, rep(1.5,8), paste(1:8), col="gray", cex=2)
    text(9, 1.5, "topo.colors(8)", adj=0)
    ## heat.colors()
    rect((1:8)-0.5, rep(0.05,8), (1:8)+0.5, rep(0.95,8), col=heat.colors(8))
    text(1:8, rep(0.5,8), paste(1:8), col="gray", cex=2)
    text(9, 0.5, "heat.colors(8)", adj=0)
    par(oldpar)
}
```

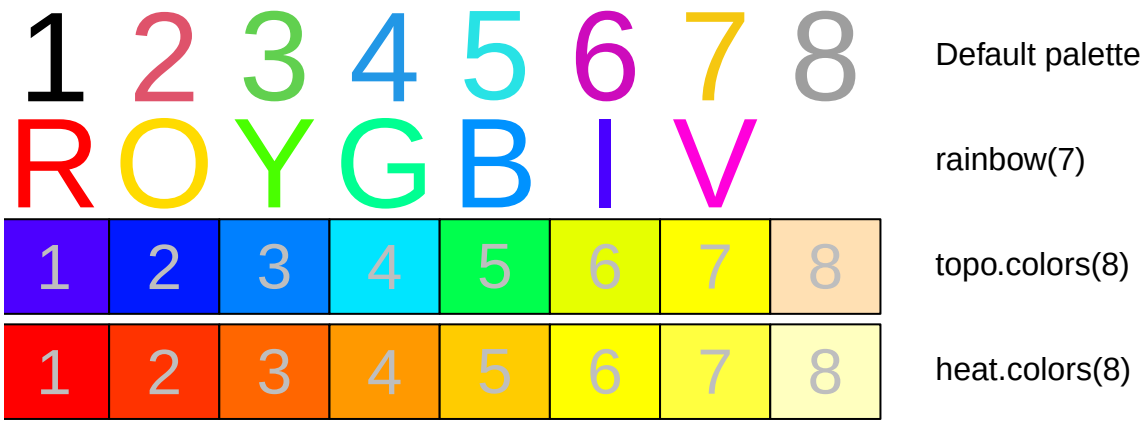

Figure 2.2: Some available colour palettes

To run the function, enter

Fine control – Parameter settings

The default settings of parameters, such as character size, are often adequate. In order to change parameter settings for a subsequent plot, use the `par()` function. For example

```
par(cex=1.25)   # character expansion
This increases the text and plot symbol size 25% above the default.
```

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

```
oldpar <- par(cex=1.25, mex=1.25)  # mex=1.25 expands the margin by 25%
```

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

```
oldpar <- par(cex=1.5)
plot(distance ~ stretch, data=elasticband)
par(oldpar)         # Restores the earlier settings
```

Inside a function one can specify, e.g.

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

Type in `help(par)` to get details of available parameter settings.

Multiple plots on the one page

The parameter `mfrow` can be used to configure the graphics sheet so that subsequent plots appear row by row, one after the other in a rectangular layout, on the one page.
For a column by column layout, use `mfcol` instead. Figure 2.3 uses a two by two layout to show plots of `brain` versus `body` with four different transformations, for data in the `DAAG::MASS::Animals` data frame:

```
## Code
par(mfrow=c(2,2))
with(MASS::Animals, {
    plot(body, brain)
    plot(sqrt(body), sqrt(brain))
    plot(body^0.1, brain^0.1)
    plot(log(body),log(brain))
})
```

Note the use of braces (`{`,`}`) to bracket together the four plot statements, as a way to ensure that in all four cases the variable names refer to columns of the `MASS::Animals` data frame.

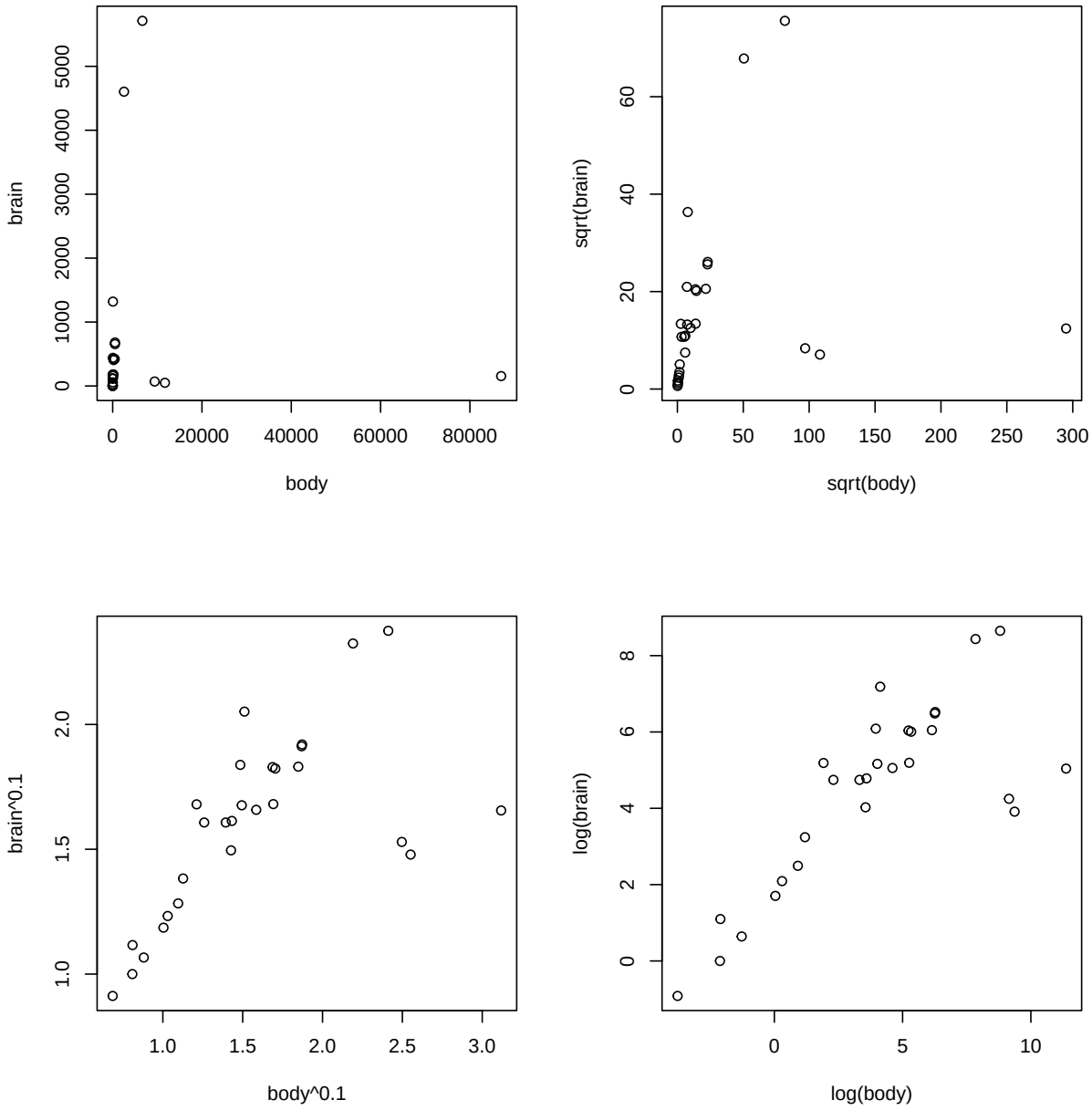

Figure 2.3: Plots of `brain` versus `body` for 26 different animals, with four different transformations.

The shape of the graph sheet

One may for example want the individual plots to be rectangular rather than square. The R for Windows functions `win.graph()` or `x11()` that set up the Windows screen take the parameters width (in inches), height (in inches) and pointsize (in 1/72 of an inch). The setting of `pointsize` (default =12) determines character heights. It is the relative sizes of these parameters that matter for screen display or for incorporation into Word and similar programs. Graphs can be enlarged or shrunk by pointing at one corner, holding down the left mouse button, and pulling.

2.3 Adding points, lines, and text

Use the function `points()` to add points. Use `lines()` to join successive lines to points.

Adding a specified line to plots

Use the function `abline()` for this. The parameters may be an intercept and slope, or a vector that holds the intercept and slope, or an lm object. Alternatively it is possible to draw a horizontal line (`h` =), or a vertical line (`v` =).

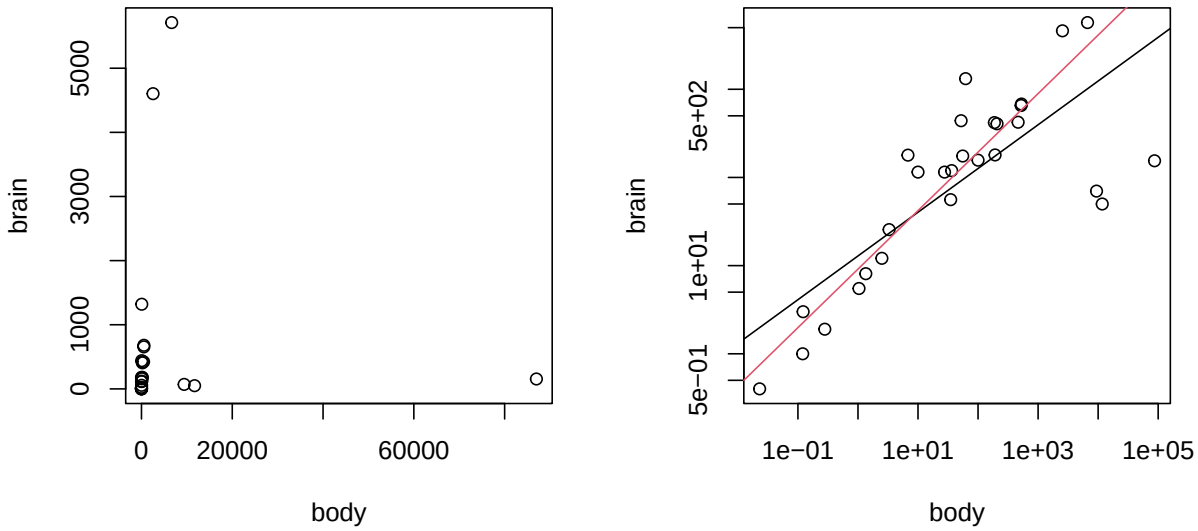

Figure 2.4: Plots of brain weight (gm) against body weight (kg).
Panel A makes it clear why a logarithmic or other power transformation is needed.
Panel B adds both a least squares regression line and a robust regression line.

Figure 2.4 shows plots of brain weight (gm) against body weight (kg), for 28 species of land animals. Logarithmic scales on both axes result in a meaningful plot, shown in Panel B. Two

lines have been added – a least squares regression line (in black), and a robust regression line (in red).

The robust regression line makes a more sense than the least squares line. There are a number of animals with large body weight that have a larger brain weight than the red line would suggest.

Code for Panel B is:

```
Animals.lm <- lm(log10(brain) ~ log10(body), data=Animals)
Animals.rlm <- MASS::rlm(log10(brain) ~ log10(body), data=Animals)
abline(Animals.rlm, col=2)
```

Adding text

Here is a simple example that uses the function `text()` to add text labels to the points on a plot. Data are

```
primates <- DAAG::primates
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
## Observe that the row names store labels for each row
```

Figure 2.5 (a) would be adequate for identifying points, but is not a presentation quality graph. Figure 2.5 (b) uses the `xlab` (x-axis) and `ylab` (y-axis) parameters to specify meaningful axis titles. It uses the parameter setting `pos=4` to move the labeling to the right of the points. It sets `pch=16` to make the plot character a heavy black dot. This helps make the points stand out against the labeling.

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

```
text(x=Bodywt, y=Brainwt, labels=row.names(primates), pos=2)
```

Here is the R code for Figure 2.5:

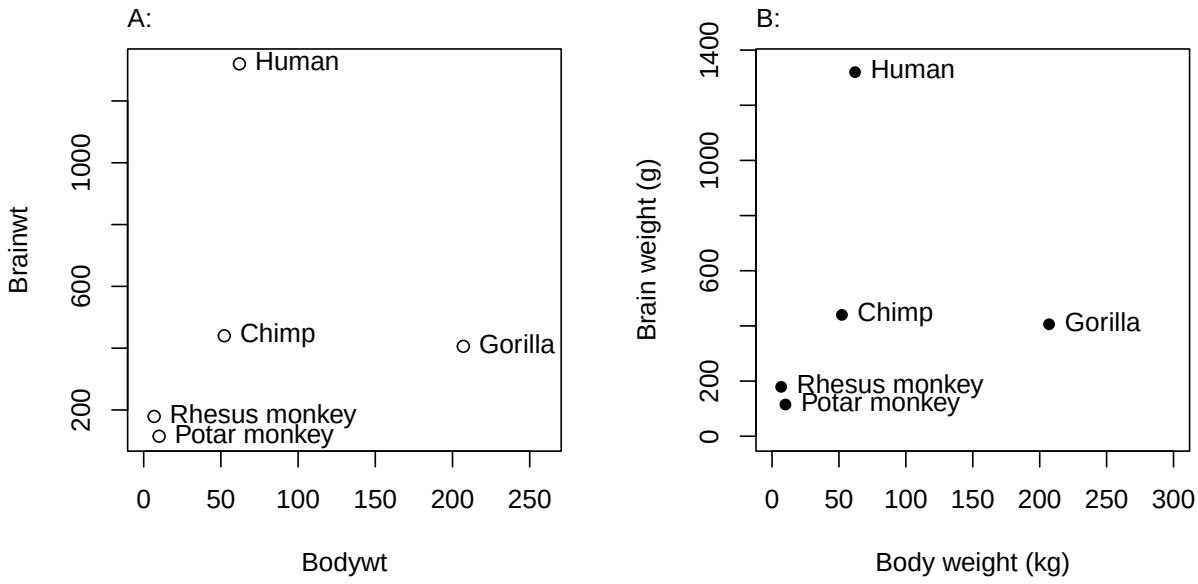

Figure 2.5: Plots of the primates data, with labels on points. Panel B improves on Panel A

Adding Text in one of the margins

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

Identification and Location on the Figure Region

The functions `identify()` and `locator()` can be used for this purpose. Draw the graph first, then call one or other of these functions.

Use `locator()` to show the co-ordinates of points, and `identify()` to label points. Position the cursor near the point whose co-ordinates are required, and click the left mouse button. prints out the co-ordinates of points. One positions the cursor at the location for which coordinates are required, and click the left mouse button. Depending on the screen device, a click with a mouse button other than the first, or pressing the ESC key terminates the process and returns a list that has the co-ordinates of the points. Or, if the setting of the parameter `n` (by default 360) is reached first, that terminates the process and returns the list.

The function `identify()` requires specification of a vector `x`, a vector `y`, and a vector of text strings that are available for use a labels. The parameter `n` is set to the number of points. Click to the left or right, and slightly above or below a point, depending on the preferred positioning of the label. When labeling is terminated, the row numbers of the observations that have been labelled are printed on the screen, in order. Thus try:

```
plot(brain ~ body, data=MASS::Animals)
with(primates, identify(Bodywt, Brainwt, rownames(primates)))
```

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

2.4 Plots that show the distribution of data values

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

Histograms and density plots

The shapes of histograms depend on the placement of the breaks, as Figure 2.6 illustrates:

```
par(mfrow=c(1,3))
possum <- DAAG::possum
with(possum, {
here <- sex == "f"
hist(totlngth[here], breaks = 72.5 + (0:5) * 5, ylim = c(0, 22),
        xlab="Total length", main ="A: Breaks at 72.5, 77.5, ...")

plot(density(totlngth[here]),type="l", main="B: Density curve")
dens <- density(totlngth[here])
xlim <- range(dens$x)
ylim <- range(dens$y)
hist(totlngth[here], breaks = 72.5 + (0:5) * 5, probability = TRUE,
   xlim = xlim, ylim = ylim, xlab="Total length",
   main="C: Breaks at 73, 78, ...")
lines(dens)
})
```

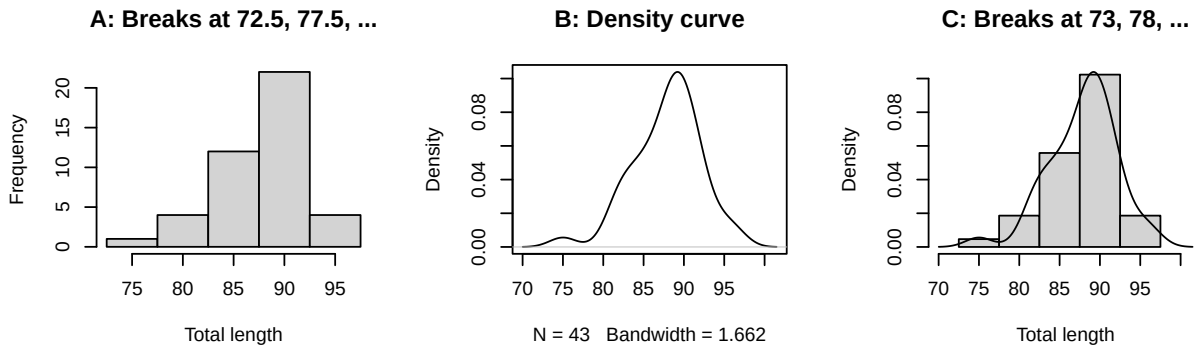

Figure 2.6: Panel A shows a histogram with a frequency scale. Panel B is drawn with a density scale, so that a density curve can be readily superimposed. Panel C has a different choice of breakpoints, so that the histogram gives a rather different impression of the distribution of the data. The density curve is superimposed.

Here is the code used to plot the histogram in Figure 2.6 A.

```
with(possum, {
  here <- sex == "f"
  hist(totlngth[here], breaks = 72.5 + (0:5) * 5, ylim = c(0, 22),
          xlab="Total length", main ="A: Breaks at 72.5, 77.5, ...")
  })
```

Density plots, now that they are available, are often a preferred alternative to a histogram. A histogram may be viewed as a crude form of density plot. Density plots do not depend on a choice of breakpoints. The choice of width and type of window, controlling the nature and amount of smoothing, does affect the appearance of the plot, making it more or less smooth. The following will give a density plot:

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

In Figure 2.6 B the y-axis for the histogram is labelled so that the area of a rectangle is the density for that rectangle, i.e., the frequency for the rectangle is divided by the width of the rectangle. This gives a scale that is appropriate for superimposing a density curve estimate. The only difference between Figure 2.6 C and Figure 2.6 B is that a different choice of breakpoints is used for the histogram, so that the histogram gives a rather different impression of the distribution of the data. Code for Figure 2.6 B is:

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

Dotcharts and stripcharts

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

```
dotchart(islands)  # vector of named numeric values, in the datasets package
```

Unfortunately there are many names, and there is substantial overlap. Either enlarge the graph to occupy a large display screen, or plot with perhaps `cex=0.5`, so that the names can be distinguished.

Note also stripcharts, obtained using `stripchart()`. Try, e.g.:

```
stripchart(decrease ~ treatment, log = "x", data = OrchardSprays)
```

Boxplots

Figure 2.7 shows a boxplot of the female possum lengths:

Code for the boxplot, without the annotation, is

```
totlngth <- subset(DAAG::possum, sex=="f")$totlngth
boxplot(totlngth[here], horizontal=TRUE, xlab="Total length (cm)")
rug(totlngth, side=1)   # Add a 'rug' that shows the location of points
```

Figure 2.7 adds interpretative information. Also a 'rug', showing the location of individual data points, has been added.

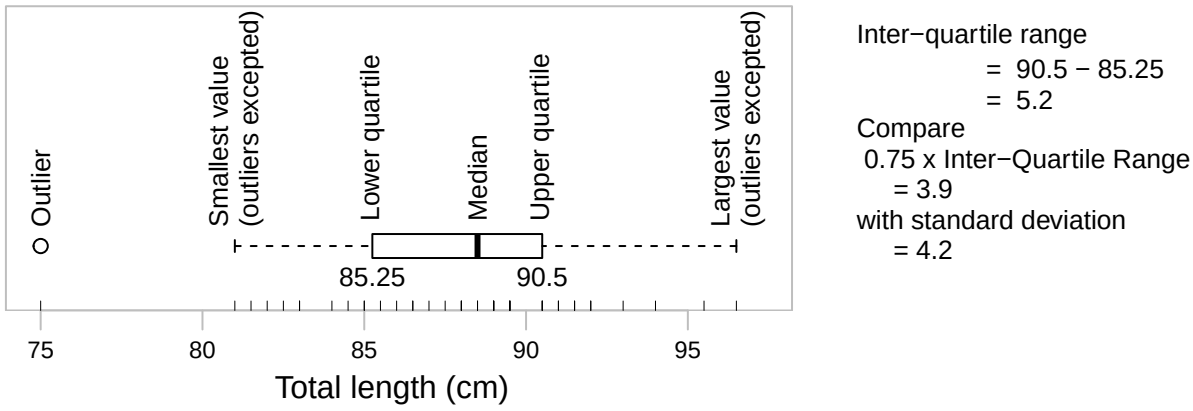

Figure 2.7: Boxplot of female possum lengths, with additional labeling information

Normal probability plots

Type `qqnorm(y)` to obtain a normal probability plot of the elements of `y`.
The points of a normal probability plot will on average lie on a straight line if the distribution is Normal. In order to calibrate the eye to recognize plots that indicate non-normal variation, it is helpful to do several normal probability plots for random samples of the relevant size from a normal distribution. Look ahead to Figure 2.12, which uses the *lattice* function `qqmath()`, with a much more effective graph than is readily obtained using base graphics functions.

2.5 Scatterplot smoothing

The function `panel.smooth()` plots points, then adds a smooth curve through the points. As an example, consider the `MASS:forbes` data frame that has 17 observations of barometric pressure (`bp`, inches of mercury) and boiling point of water (`bp`, degrees Fahrenheit), at various locations in the Alps. Figure 2.8 plots the data, and adds a smooth curve. A dashed line has been added for comparison.

The parameter `span`, specifying the proportion of points that influence the smoothed value at each point, can be used to control the smoothness. The default is $\frac{2}{3}$.

2.6 Lattice graphics

Lattice plots allow the use of the layout on the page to reflect meaningful aspects of data structure. The *lattice* package sits on top of the *grid* package. To use lattice graphics, both these packages must be installed.

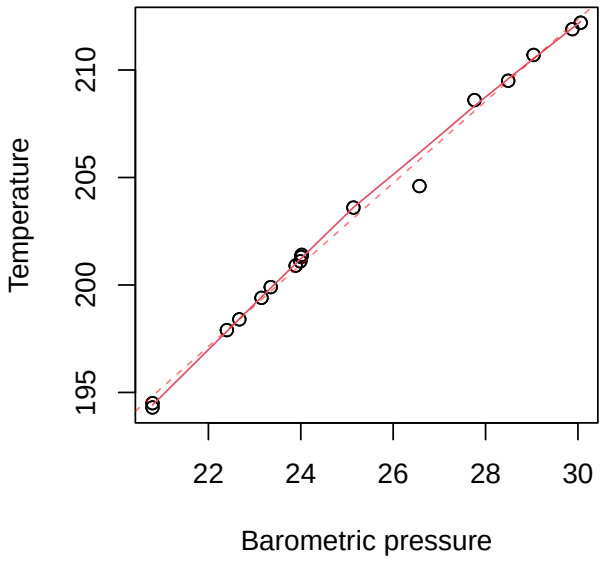

Figure 2.8: Boiling point of water, versus measured barometric pressure

Providing it is installed, the *grid* package will be attached automatically when lattice is attached.

The *lattice* package has now to an extent been overtaken by the *ggplot2* package, with its richer and more complex language syntax. A number of packages have been built on top of *ggplot2*, aimed at providing specific types of graph, that allow users to work around much of the complexity. Functions in *lattice* and *ggplot2* return a graphics object, i.e., instructions for creating a plot, rather than a plot. This can be successively updated using the `update()` function, and/or new *layers* added, before using the generic `print()` (or `plot()` has the same effect) function or to produce a plot. Explicit use of `print()` or `plot()` is not required – all that is needed is to return type the object name on the command line, effectively sending its instructions to the command line.

The function name `print` emphasizes that the main part of the work required to produce a graph has already been done before the `print()` function is called – all that is then required is take the instructions that have been embedded in the graphics object and use them to show a graph on the screen or on the printed page. Readers who wish to pursue use of *ggplot2* and of packages built on *ggplot2* are referred to the extensive tutorial resources available on the web.

Examples that Present Panels of Scatterplots – Using `xyplot()`

The basic function for creating panels of scatterplots is `xyplot()`. As an example, consider data, in the data frame `DAAG::tinting`, that are from an experiment that investigated the

effects of tinting of car windows on visual recognition performance, primarily for visual recognition tasks that would be performed through side windows. Variables are `csoa` (critical stimulus onset asynchrony, i.e. the time in milliseconds required to recognize an alphanumeric target), `it` (inspection time, i.e., the time required for a simple discrimination task) and `age`, with ordered factors `tint` (level of tinting: `no`, `lo`, `hi`), `target` (`contrast`: `locon`, `hicon`), and `agegp` (1 = young, in the early 20s; 2 = an older participant, in the early 70s), and the factor `sex` (1 = male, 2 = female). Figure 2.9 shows the style of graph that one can get from xyplot(). The different symbols are different contrasts.

```
Loading required package: lattice
```

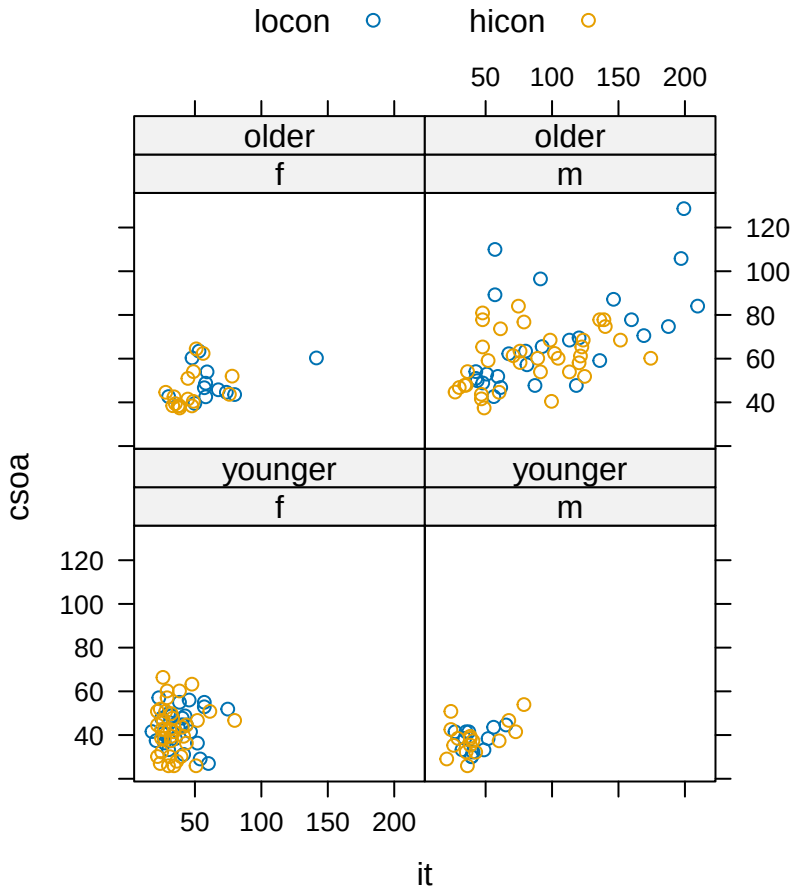

Figure 2.9: cap16

```
## Code
library(latticeExtra)
xyplot(csoa ~ it | sex * agegp, data = DAAG::tinting,
```

```
                panel = panel.superpose,
       groups = target, auto.key = list(columns = 2))
```

Addition of the argument `groups=target` has the result that two different colors of symbol are used to distinguish between low contrast and high contrast targets. To use different symbols, add an argument in the style of `par.settings=simpleTheme(pch=c(1,16))`. Additionally, or instead, it would make sense to use lighter and darker colors for the two levels of contrast. Code is easier to follow if one starts with the graphics object needed for Figure 2.9 and then updates it:

```
gph <- xyplot(csoa ~ it | sex * agegp, data = DAAG::tinting,
              panel = panel.superpose, groups = target)
update(gph, auto.key = list(columns = 2),
       par.settings=simpleTheme(pch=16, col=c("lightblue3","blue")))
  # As the result goes to the command line, it is `printed`
update(gph, groups = target, auto.key = list(columns = 2),
       par.settings=simpleTheme(pch=16, col=c("lightblue3","blue")))
```

A striking feature is that the very high values, for both csoa and it, occur only for elderly males. It is apparent that the long response times for some of the elderly males occur, as we might have expected, with the low contrast target.

Figure 2.10 puts smooth curves through the data, separately for the two target types:

```
xyplot(csoa~it|sex*agegp, data=DAAG::tinting, groups=target,
      type=c("p","smooth"))
```

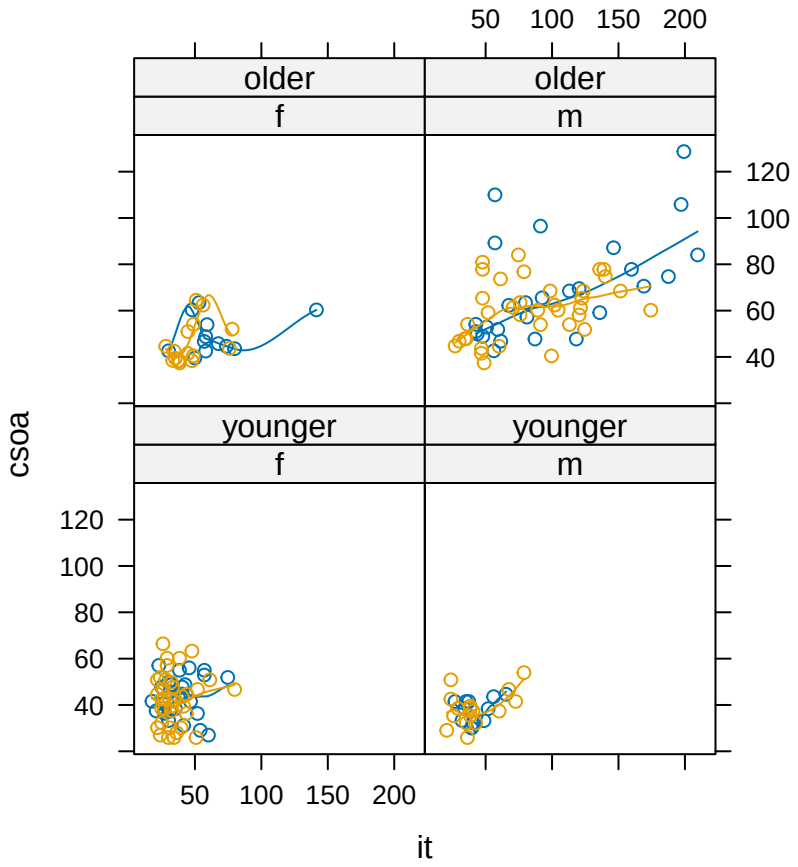

Figure 2.10: Smooth curves have been added to the plots, separately for the two types of target

The relationship between `csoa` and `it` seems much the same for both levels of contrast. Finally, we do a plot (Figure 2.11) that uses different symbols (in black and white) for different levels of tinting. The longest times are for the high level of tinting.

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

Plotting columns in parallel

Use the parameter outer to control whether the columns appear on the same or separate panels. If on the same panel, it is desirable to use auto.key to give a simple key. The following use the dataset `grog` from the *DAAGxtras* package:

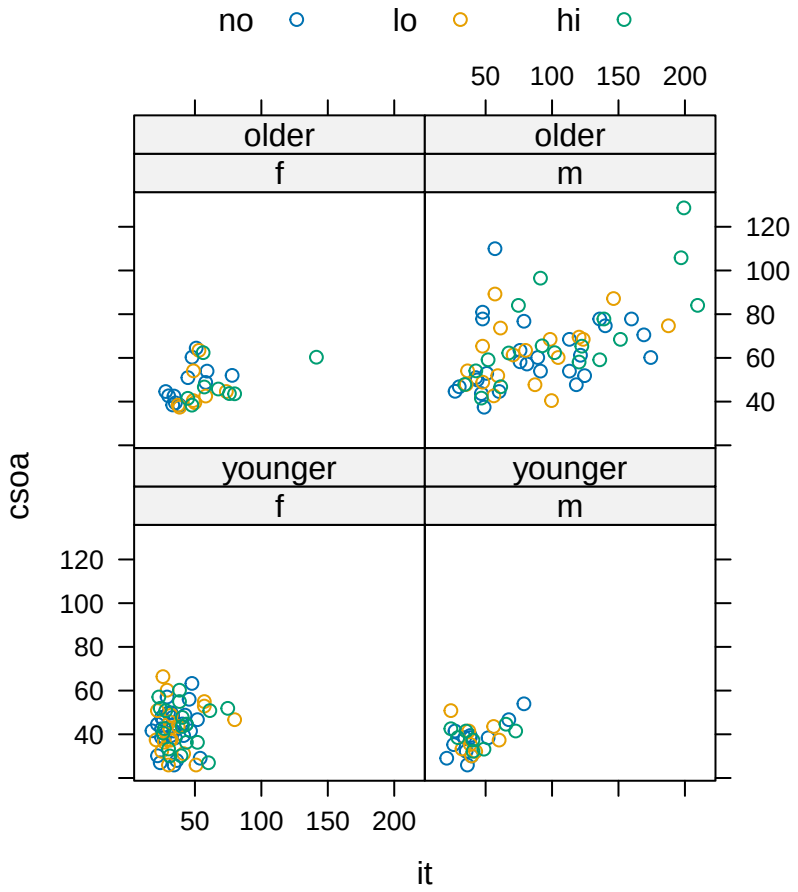

Figure 2.11: Panels show "csoa versus it, for each combination of females/males and elderly/young. The different levels of tinting (no, +=low, >=high) are shown with different symbols.

```
library(DAAG)
xyplot(Beer+Spirit+Wine ~ Year | Country, outer=TRUE,
        data=grog)
xyplot(Beer+Spirit+Wine ~ Year, groups=Country, outer=TRUE,
        data=grog)
xyplot(Beer+Spirit+Wine ~ Year | Country, outer=FALSE,
        data=grog, auto.key=list(columns=3),
        par.settings=simpleTheme(pch=16, cex=2) )
```

In the final plot, note the use of simpleTheme() as a simple mechanism for controlling common parameter settings. Use of the parameter par.settings makes the change for the current plot only. Use trellis.par.set() to make the changes for the duration of the current device, unless reset.

```
### Fixed, sliced and free scales
jobs <- DAAG::jobs
## scale="fixed"
xyplot(BC+Alberta ~ Date, data=jobs, outer=TRUE)
## scale="sliced" - different slices of same scale
xyplot(BC+Alberta ~ Date, data=jobs, outer=TRUE,
       scales=list(y=list(relation="sliced")) )
## scale="free" - independent scales
xyplot(BC+Alberta ~ Date, data=jobs, outer=TRUE,
       scales=list(y=list(relation="free")) )
```

Data, compared with simulated normal data

Is the sample data consistent with random normal data?

There is one unusually small value. Otherwise the points for the female possum lengths are as close to a straight line as in many of the plots for random normal data. Code is:

The idea is an important one. In order to judge whether data are normally distributed, examine a number of randomly generated samples of the same size from a normal distribution. It is a way to train the eye, to give an idea of the extent to which departures from linearity may be explained as random variation The mean and standard deviation is best matched to that of the data that are under investigation.

For comparison, for readers who are interested, here is code that plots the same data using base graphics:

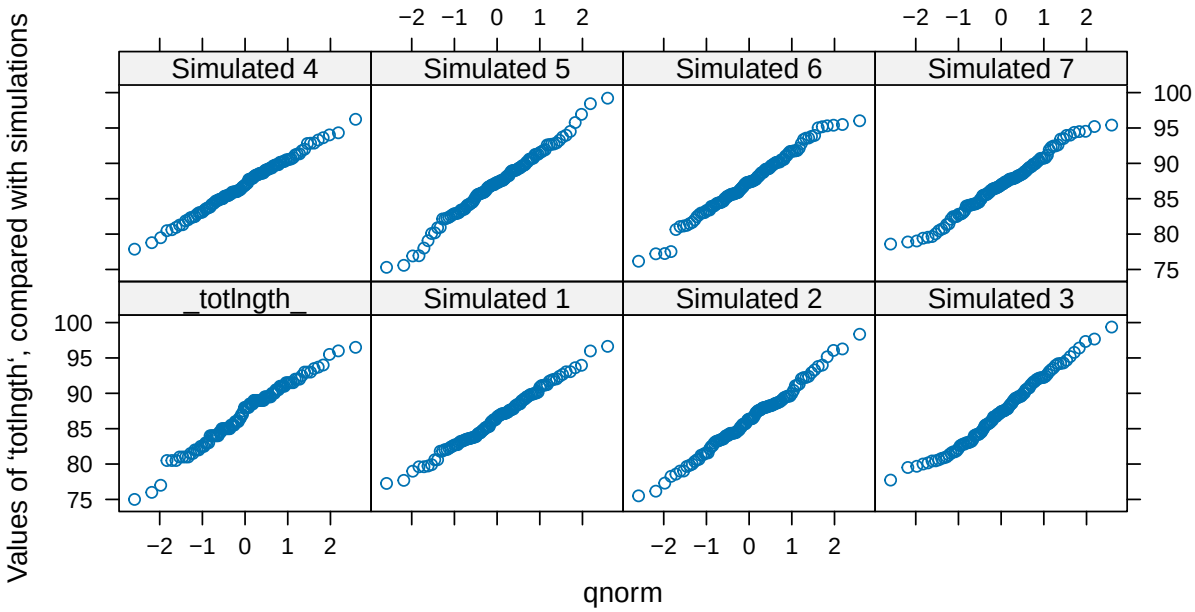

Figure 2.12: Normal probability plots. If data are from a normal distribution then points should fall, on average, along a line. The plot in the top left shows the 43 lengths of female possums. The other plots are for independent normal random samples of size 43.

```
par(mfrow=c(2,4), mgp=c(1.8,0.5,0), mar=c(3.1,3.6,2.6,1.6))              # A 2 by 4 layout of
gps <- unique(dat$gp)
for (val in gps){
  x <- subset(dat, gp==val)$y
qqnorm(x, xlab="", ylab="Length", main=levs)
}
```

Adding new layers

The *latticeExtra* package provides wide-ranging abilities for overlaying or underlaying an existing graphics object. Or, given an initial graphics object, a separate graphics object can be converted to a layer that is then added to the initial object.

In each instance, conditioning variables can be added. In most cases, a groups parameter can be specified, i.e., the plot is repeated for the groupings within the one panel. The data on athletes in the Australian Institute of Sport, in the dataset `DAAG::ais` will be used for the example that now follows. The following show haemoglobin count versus red blood cell count, distinguished by sport within panel, with separate panels for females and males:

```
library(latticeExtra)
aisBS <- subset(DAAG::ais, sport %in% c("B_Ball", "Swim"))
basic1 <-  xyplot(hc ~ rcc | sex, groups=sport[drop=TRUE],  data=aisBS)
basic2 <- update(basic1,
      par.settings=simpleTheme(pch = c(1,3), lty=1:2, lwd=1.5),
        # Plot characters 1 and 3 distinguish the groups
      strip=strip.custom(factor.levels=c("Female","Male")),
        # Label the panels "Female" and "Male", not "f" and "m"
      scales=list(tck=0.5), auto.key=list(columns=2))
```

Printing the object `basic1` would give a very basic plot. The object `basic2` sets separate line types for males and females, creates thicker lines, specefies strip level names that are different from the group level names, reduces the length of the axis ticks, and adds a key.

The `xyplot()` function has provision for the addition of separate lines for the two sports, but not for the parallel lines that are preferred. The following creates a new layer. The values `x` and `y` will be taken from the data used for the object `basic1`, when the new layer is added. Code for the new layer is:

```
layer2 <- layer(parallel.fit <-
                    fitted(lm(y ~ groups[subscripts] + x)),
                panel.superpose(x, parallel.fit, type = "r", ...))
```

The following `print`s the graph:

```
basic2+layer2
```

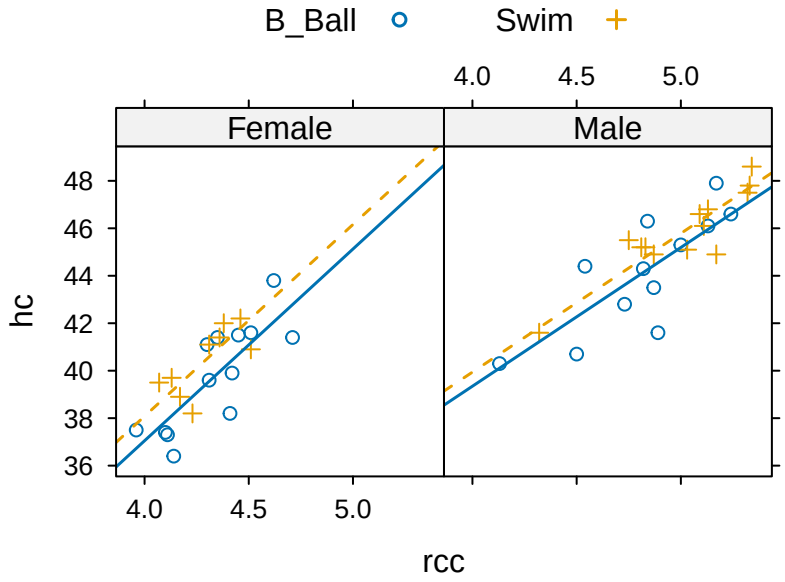

Figure 2.13: paste(cap20)

```
### An incomplete list of _lattice_ Functions  {-}
splom( ~ data.frame)                    # Scatterplot matrix
bwplot(factor ~ numeric , . .)          # Box and whisker plot
qqnorm(numeric , . .)           # normal probability plots
dotplot(factor ~ numeric , . .)         # 1-dim. Display
stripplot(factor ~ numeric , . .)       # 1-dim. Display
barchart(character ~ numeric , . .)
histogram( ~ numeric , . .)
densityplot( ~ numeric , . .)           # Smoothed version of histogram
qqmath(numeric ~ numeric , . .)         # QQ plot
splom( ~ dataframe, . .)            # Scatterplot matrix
parallel( ~ dataframe, . .)     # Parallel coordinate plots
cloud(numeric ~ numeric * numeric, . .)           # 3-D plot
contourplot(numeric ~ numeric * numeric, . .)   # Contour plot
levelplot(numeric ~ numeric * numeric, . .)       # Contour plot variant
```

2.7 Using mathematical expressions in plots

The following is a simple example. For this purpose, an expression has a much extended syntax, relative to that for a mathematical expression. See `?plotmath`.

‘

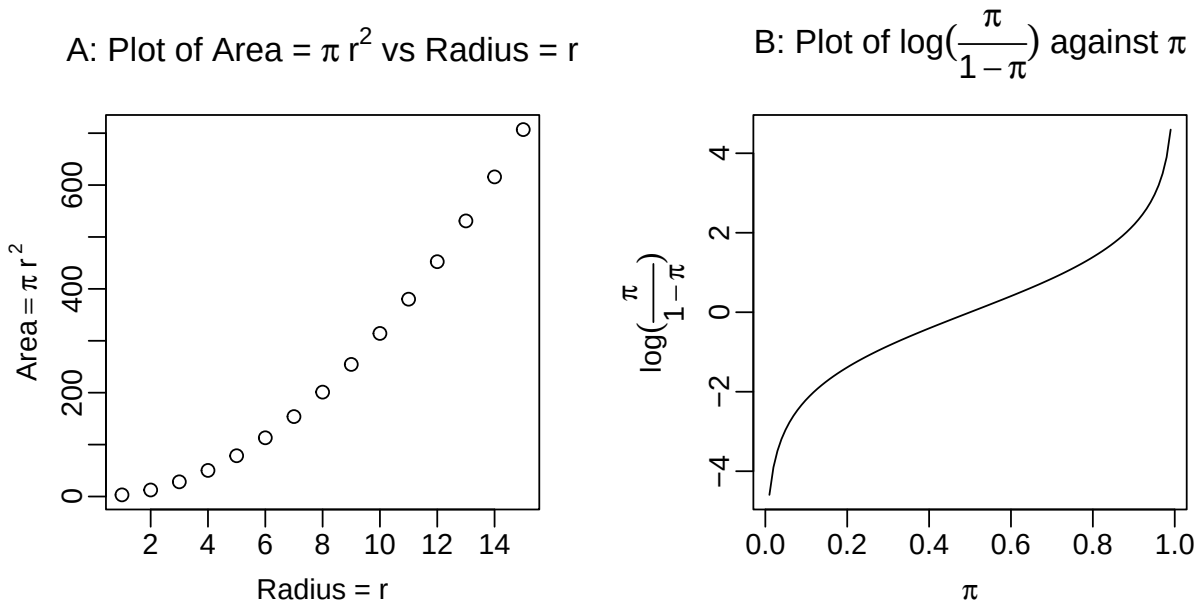

Figure 2.14: Examples where one or both axis labels are mathematical expressions,

```
par(mgp=c(1.8,0.5,0), mfrow=c(1,2))
r <- 1:15
y <- pi*r^2
plot(r, y, xlab=expression("Radius = "*r), ylab=expression(Area == pi*~r^~2))
title(main=expression("A: Plot of Area = "*pi*~r^2 *" vs Radius = "*r))
curve(log(x/(1-x)), from=0.01, to=0.99, xlab=expression(pi),
      ylab=expression(log(frac(pi,1-pi))))
title(main=expression("B: Plot of "*log(frac(pi,1-pi))*" against "*pi))
```

For *lattice* and *ggplot2* graphics as well as base graphics, expressions of the type shown can appear anywhere in place of a character vector.

Notice that in `ylab = expression(Area == pi*r^2)`, there is a double equals sign (`==`), although what will appear on the plot has a single equals sign. See `?plotmath` for detailed information on the plotting of mathematical expressions. Notice that `*` juxtaposes both text and mathematical symbols. Notice also that `~` been used to insert space before r and `phantom(,)` between r and the superscript 2. The final plot from `demo(graphics)` demonstrates some of the possibilities for plotting mathematical symbols.

2.8 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, 2 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.

2.9 Exercises

1. 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 . (Alternatively work with the vector `LakeHuron` from the datasets package, that has mean heights for 1875-1972 only.)

a) Plot mean.height against year.

b) Use the identify function to label points for the years that 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 (depending on the operating system) a mouse button other than the left, or press ESC.

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.

2. Plot the graph of `log(brain weight)` versus `log(body weight)`, for the data set Animals from the MASS package. Use the row labels to label the points corresponding to the three clear outliers.

3. Check the distributions of head lengths (`hdlngth`) in the possum 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(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?

4. 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.

5. Use `mfrow()` to set up the layout for a 3 by 4 array of plots. In the top 4 rows, show normal probability plots for four separate 'random' samples of size 10, all from a normal distribution. In the middle 4 rows, display plots for samples of size 100. In the bottom four rows, display plots for samples of size 1000. Comment on how the appearance of the plots changes as the sample size changes.

6. The function `runif()` generates 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?

*7. If you find exercise 6 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.

8. For the first two columns of the data frame hills, examine the distribution using: (a) histogram; (b) density plots; (c) normal probability plots.
Repeat (a), (b) and (c), now working with the logarithms of the data values.

9. This and remaining exercises ask for the use of *lattice* functions. The following data gives milk volume (g/day) for smoking and nonsmoking mothers :

```
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 (use `bwplot()`); (ii) using a dotplot form of display (use `dotplot()`).

10. For the `possum` data set, use *lattice* functions to generate the following plots:

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

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

c) density plots of `hdlngth` – use `densityplot()`. Investigate the effect of varying the density bandwidth (`bw`).

11. The following exercises, all using *lattice* functions, relate to the data frame `DAAG::possum`:

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

(b) Construct a contour plot of `chest` versus `belly` and `totlngth` – use `levelplot()` or `contourplot()`.

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

(d) Use `qqmath()` to construct normal probability plots for `hdlgth`, for each separate level of `sex` and `Pop`. Does it appear that the distribution of `hdlgth` varies with the level of these other factors.

12. The dataframe `airquality` (*datasets* package) has columns `Ozone`, `Solar.R`, `Wind`, `Temp`, `Month` and `Day`. Use `xyplot()` to plot `Ozone` against `Solar.R` for each of the three temperature ranges, and for each of three wind ranges.

2.10 References and reading

The web page https://www.eecs.yorku.ca/~papaggel/courses/eecs6414/index.html
has links to many different collections of information on statistical graphics.

Chang (2013) . R graphics cookbook. O'Reilly.

Murrell (2011) . R Graphics. Chapman and Hall/CRC.

3 Multiple Linear Regression

3.1 Linear model objects

We begin with the straight line regression example that appeared earlier, in Section 1.1. Look ahead to Figure 3.1, where a regression line has been added.

The code for the regression calculation is:

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

The object `distance ~ stretch` is a *model formula*, used to specify details of the model that is to be fitted. Other model formulae will appear in the course of this chapter.

```
elasticband <- DAAG::elasticband
plot(distance ~ stretch, data=DAAG::elasticband)
abline(elastic.lm)
```

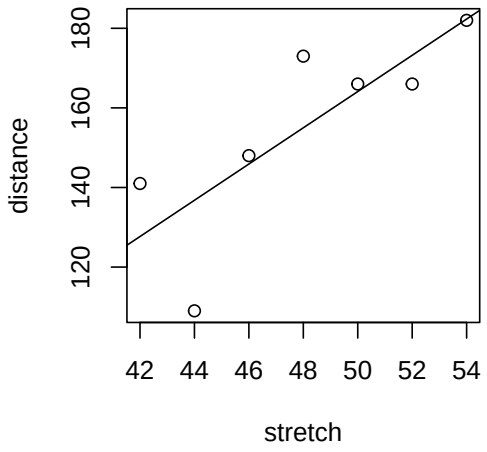

Figure 3.1: Plot of distance versus stretch for the elastic band data, with fitted least squares line

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 output documents the model formula that was used:

```
summary(elastic.lm, digits=4)

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

Residuals:
      1       2       3       4       5       6       7
 2.1071 -0.3214 18.0000  1.8929 -27.7857 13.3214 -7.2143

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -63.571     74.332  -0.855   0.4315
stretch        4.554      1.543   2.951   0.0319

Residual standard error: 16.33 on 5 degrees of freedom
Multiple R-squared:  0.6352,    Adjusted R-squared:  0.5622
F-statistic: 8.706 on 1 and 5 DF,  p-value: 0.03186
```

An `lm` object is a list of named elements. Names of `elastic.lm` are:

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

A number of functions are available for extracting information that might be required from the list. This is or most purposes preferable to extracting elements from the list directly. Examples are:

```
coef(elastic.lm)
(Intercept)     stretch
 -63.571429    4.553571
```

The function most often used to inspect regression output is `summary()`. It extracts the information that users are most likely to want. In section 5.1, we had

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

There is a plot method for *lm* objects that gives the diagnostic information shown in Figure 3.2.

To get Figure 3.2, type:

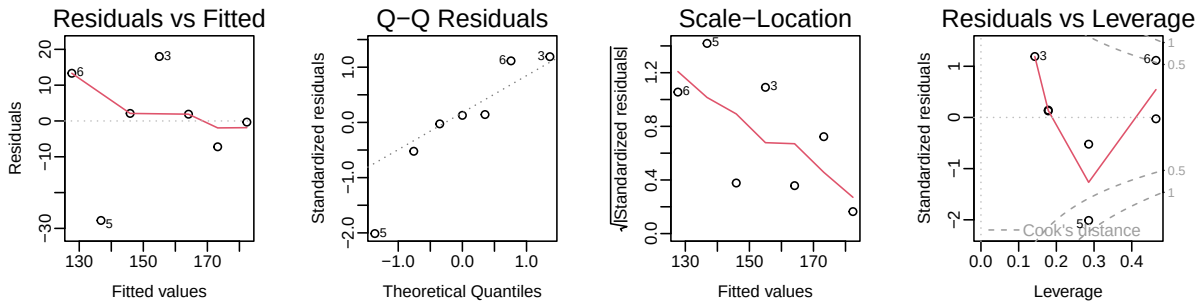

Figure 3.2: Diagnostic plot of lm object, obtained by `plot(elastic.lm)`.

```
opar <- par(mar=c(4.6,3.6,2.1,2.1), mgp=c(2,.5,0), mfrow=c(1,4))
plot(elastic.lm)
par(opar)
```

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 for the data frame, then the row numbers are used.]

3.2 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 involve 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 `X` matrix, with the `y`-vector alongside, is:

```
cbind(model.matrix(elastic.lm), distance=elasticband$distance)
  (Intercept) stretch distance
1           1      46      148
2           1      54      182
3           1      48      173
4           1      50      166
5           1      44      109
6           1      42      141
7           1      52      166
```

The model matrix relates to the part of the model that appears to the right of the equals sign. The straight line model is $y = a + bx +$ residual, which we write as

$$y = a \times 1 + b \times x + \text{residual}$$

The following are the fitted values and residuals that we get with the estimates of `a` (= -63.6) and `b` (= 4.55) from the least squares regression

1	Stretch (mm)	(Fitted)	(Observed)	(Residual)
−63.6	4.55×	−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
…	…	…	…	…

The symbol $\hat{y}$ [pronounced y-hat] is commonly used fot fitted values of the variable y.

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

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

where e is a random variable with mean 0. The `X` matrix then consists of a single column, holding the values of `x`.

Model formulae more generally

Model formulae take forms such as:

```
y ~ x+z             # Fit y as a linear combination of x and z
y~x + fac + fac:x
  # 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.
Notice the similarity between model formulae and the formulae that can be used for specifying plots. Thus, recall the graph formula for a `coplot` that gives a plot of `y` against `x` for each different combination of levels of `fac1` (across the page) and `fac2` (up the page) is:

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

Manipulating Model Formulae

Model formulae can be assigned, e.g.

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

or

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

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(elasticband)
[1] "stretch"  "distance"
```

```
nam <- names(elasticband)
formds <- formula(paste(nam[1],"~",nam[2]))
lm(formds, data=elasticband)

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

Coefficients:
(Intercept)     distance
    26.3780       0.1395
```

Note that graphics formulae can be manipulated in exactly the same way as model formulae.

3.3 Multiple Linear Regression – Examples

The data frame Rubber

The dataset `MASS::Rubber` is from the accelerated testing of tyre rubber. The variables are `loss` (the abrasion loss in gm/hr), `hard` (hardness in 'Shore' units), and `tens` (tensile strength in kg/sq m).[1] Figure 3.3 shows a scatterplot matrix for the variables:

Code is:

[1]The original source is Davies et al. (1947) , Table 6.1, p. 119.

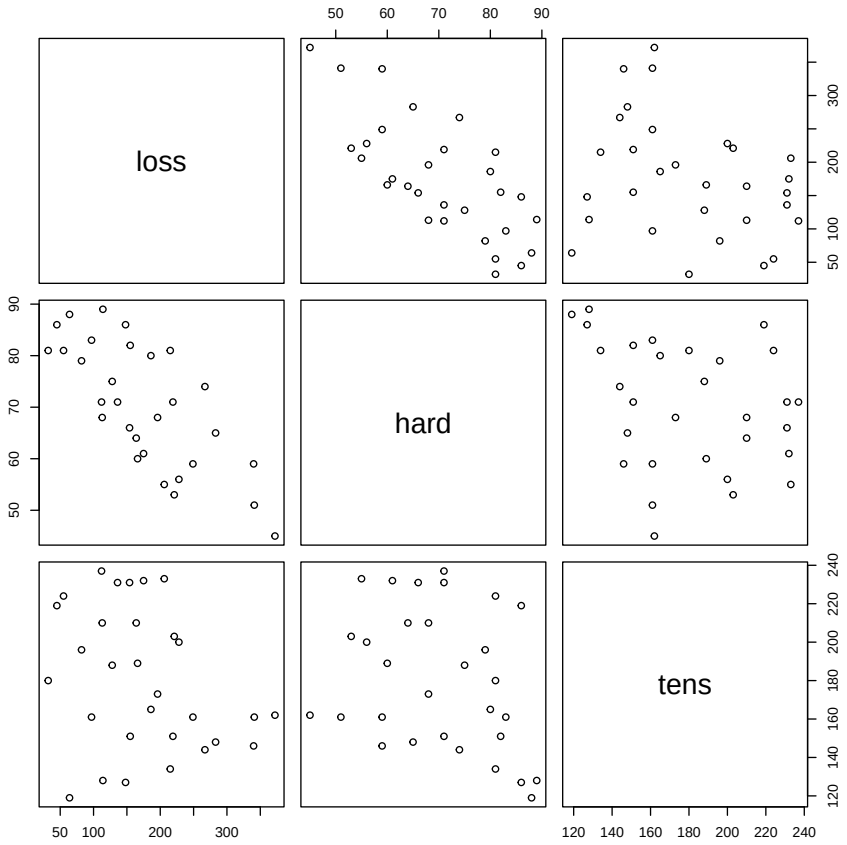

Figure 3.3: Scatterplot matrix for the data frame `MASS::Rubber`

```
pairs(MASS::Rubber)
```

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

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

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

Residuals:
    Min      1Q  Median      3Q     Max
-79.385 -14.608   3.816  19.755  65.981

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 885.1611    61.7516  14.334 3.84e-14
hard         -6.5708     0.5832 -11.267 1.03e-11
tens         -1.3743     0.1943  -7.073 1.32e-07

Residual standard error: 36.49 on 27 degrees of freedom
Multiple R-squared:  0.8402,    Adjusted R-squared:  0.8284
F-statistic:    71 on 2 and 27 DF,  p-value: 1.767e-11
```

In addition to the use of `plot()` with `lm` objects, note the use of `termplot()`.

```
par(mfrow=c(1,2))
termplot(Rubber.lm, partial=TRUE, smooth=panel.smooth)
```

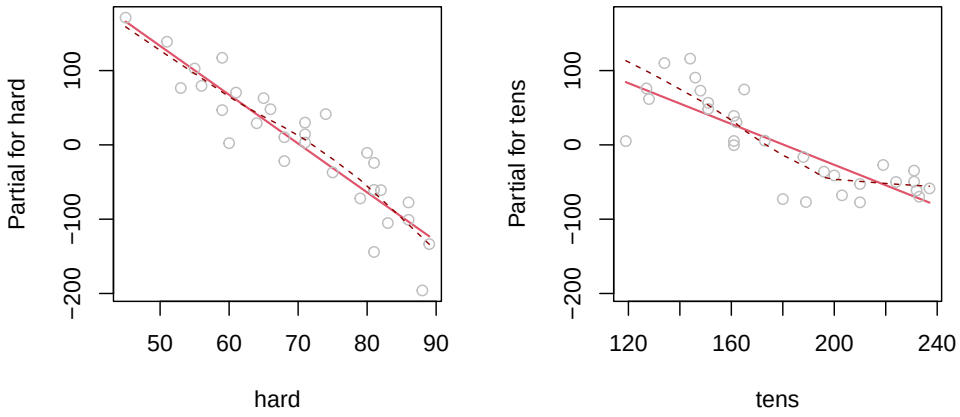

Figure 3.4: Plot, obtained with termplot(), showing the contribution of each of the two terms in the model, at the mean of the contributions for the other term. A smooth curve has, in each panel, been fitted through the partial residuals. There is a clear suggestion that, at the upper end of the range, the response is not linear with tensile strength.

Figure @ref(fig:fig24) used the following code:

```
par(mfrow=c(1,2))
termplot(Rubber.lm, partial=TRUE, smooth=panel.smooth)
```

This plot raises interesting questions.

3.3.1 Weights of Books

The books to which the data in the data set `DAAG::oddbooks` refer were chosen to cover a wide range of weight to height ratios. The use of the data to fit regression models illustrates how biases that affect the collection of observational data can skew results.

Code is:

```
pairs(DAAG::oddbooks)
```

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.

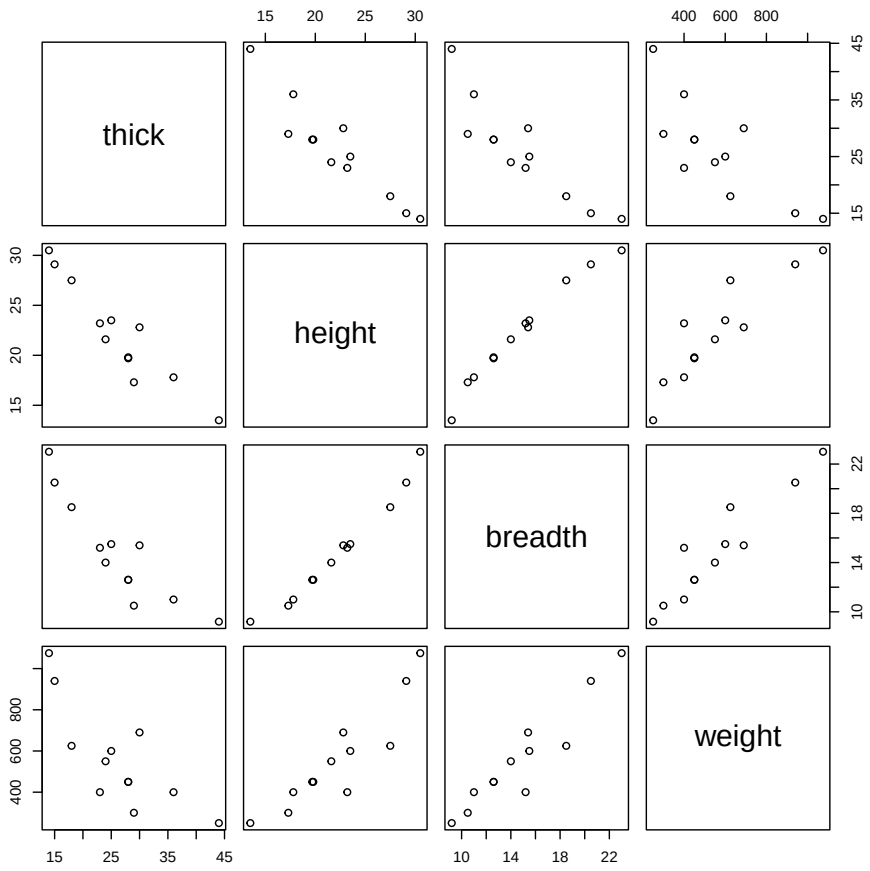

Figure 3.5: Scatterplot matrix for the data frame `DAAG::oddbooks`

The design of the data collection really is important for the interpretation of coefficients from a regression equation. Even where regression equations from observational data appear to work well for predictive purposes, the individual coefficients may be misleading. This is more than an academic issue, as the analyses in Lalonde (1986) demonstrate. They had data from experimental "treatment" and "control" groups, and also from two comparable non-experimental "controls". The regression estimate of the treatment effect, when comparison was with one of the non-experimental controls, was statistically significant but with the wrong sign! The regression should be fitted only to that part of the data where values of the covariates overlap substantially. Dehejia and Wahba demonstrate the use of scores ("propensities") to identify subsets that are defensibly comparable. The propensity is then the only covariate in the equation that estimates the treatment effect. It is impossible to be sure that any method is giving the right answer.

Assuming a uniform density, the geometry suggests

$$\text{weight} = \text{density} \times \text{thick} \times \text{height} \times \text{breadth}$$

On a logarithmic scale, this transforms to

$$\begin{aligned} \log(\text{weight}) &= \log(\text{density}) + \log(\text{thick}) + \log(\text{height}) + \log(\text{breadth}) \\ &= \log(\text{density}) + \log(\text{volume}) \end{aligned}$$

The following ignores what the geometry suggests

```
logbooks <- log(DAAG::oddbooks) # We might expect weight to be
                                # proportional to thick * height * width
## Regress log(weight) on log(thick) + log(height) + log(breadth)
logbooks.lm3<-lm(weight~thick+height+breadth,data=logbooks)
  # NB: `weight` is really `log(weight)`,
  # and similarly for other variables
cat(capture.output(summary(logbooks.lm3, digits=2))[-(1:8)], sep='\n')
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.7191     3.2162  -0.224    0.829
thick         0.4648     0.4344   1.070    0.316
height        0.1537     1.2734   0.121    0.907
breadth       1.8772     1.0696   1.755    0.117

Residual standard error: 0.1611 on 8 degrees of freedom
Multiple R-squared:  0.8978,    Adjusted R-squared:  0.8595
F-statistic: 23.43 on 3 and 8 DF,  p-value: 0.000257
```

Notice that all the coefficients are at the level of statistical error, but even so the overall fit (p=0.000257) is clearly good.

Now regress on logVolume = log(volume)

```
## Regress log(weight) on log(thick) + log(height) + log(breadth)
logVolume <- with(logbooks, thick+height+breadth)
logbooks.lm <- lm(weight~logVolume, data=logbooks)
cat(capture.output(summary(logbooks.lm, digits=2))[-(1:8)], sep='\n')
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -8.942      2.731  -3.274  0.00837
logVolume      1.696      0.305   5.562  0.00024

Residual standard error: 0.2228 on 10 degrees of freedom
Multiple R-squared:  0.7557,    Adjusted R-squared:  0.7313
F-statistic: 30.94 on 1 and 10 DF,  p-value: 0.00024
```

The model still does relatively well on the data used. Note however that the coefficient of `logVolume` differs from the expected 1.0 by 2.28 standard errors.

Figure @ref(fig:fig25) made it clear that all three variables are highly correlated. We now try adding each one in turn to the model regressed log(weight) on logVolume.

```
print(add1(logbooks.lm, scope=~.+thick+height+breadth, test='F'), digits=4)
Single term additions

Model:
weight ~ logVolume
        Df Sum of Sq    RSS    AIC F value  Pr(>F)
<none>               0.4966 -34.22
thick    1    0.2739 0.2227 -41.84  11.072 0.00884
height   1    0.2437 0.2528 -40.32   8.677 0.01634
breadth  1    0.2872 0.2094 -42.58  12.347 0.00658
print(add1(logbooks.lm, scope=~.+thick+height+breadth, test='F'), digits=3)
Single term additions

Model:
weight ~ logVolume
        Df Sum of Sq   RSS   AIC F value Pr(>F)
<none>               0.497 -34.2
thick    1     0.274 0.223 -41.8   11.07 0.0088
height   1     0.244 0.253 -40.3    8.68 0.0163
breadth  1     0.287 0.209 -42.6   12.35 0.0066
```

The smallest value of the AIC statistic is preferred, though given the small degrees of freedom for the residual, not too much can be made of the magnitude of the reduction in AIC. The preferred model (which is also the model that gives the smallest `Pr(>F)`) is then:

```
addBreadth.lm <- update(logbooks.lm, formula=.~.+breadth)
cat(capture.output(summary(addBreadth.lm, digits=2))[-(1:8)], sep='\n')
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.9192     2.9508  -0.311  0.76252
logVolume     0.4564     0.4100   1.113  0.29452
breadth       1.1562     0.3291   3.514  0.00658

Residual standard error: 0.1525 on 9 degrees of freedom
Multiple R-squared:  0.897, Adjusted R-squared:  0.8741
F-statistic: 39.19 on 2 and 9 DF,  p-value: 3.611e-05
```

Once account has been taken of `breadth`, `volume` does not make any clearly useful contribution to predicting `weight`. This is a result that has no, or very limited, applicability outside of the circumstances that generated this dataset.

3.4 Polynomial and Spline Regression

Linear regression is linear in the explanatory terms that are supplied. These can include, for example polynomial terms. Note that polynomial curves of high degree are in general unsatisfactory. Spline curves, constructed by joining low order polynomial curves (typically cubics) in such a way that the slope changes smoothly, are in general preferable.

The data frame `DAAG::seedrates` gives, for each of a number of different seeding rates, the number of barley grain per head.

Code is:

```
plot(grain ~ rate, data=DAAG::seedrates)   # Plot the data
seedrates.lm2 <- lm(grain ~ rate+I(rate^2), data=DAAG::seedrates)
with(data=DAAG::seedrates, lines(lowess(rate, fitted(seedrates.lm2))))
```

Spline Terms in Linear Models

The fitting of polynomial functions was a simple example of the use of linear models to fit terms that may be nonlinear functions of one or more of the variables. Spline functions variables extend this idea further. The splines considered here are constructed by joining together cubic

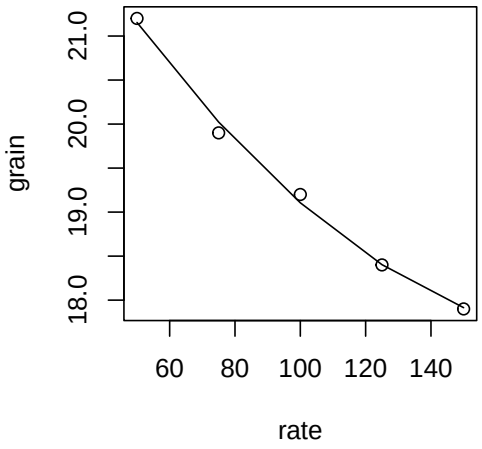

Figure 3.6: Number of grain per head versus seeding rate, for the barley seeding rate data, with fitted quadratic curve.

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 linear combinations of basis functions, where each basis function is a transformation of the variable.

The dataset `cars`, from the _datasets+ package, gives stopping distance (`dist`, in ft) versus `speed` (mph), for cars in the 1920s.

```
Loading required package: nlme
This is mgcv 1.9-0. For overview type 'help("mgcv-package")'.
```

Code is

```
par(mfrow=c(1,2))
library(mgcv)
cars3.gam <- gam(dist ~ s(speed, k=3, fx=T, bs="cr"), data=cars)
  # k=3 includes 1 degree of freedom for the intercept.
plot(cars3.gam, residuals=T, pch=1, shift=mean(predict(cars3.gam)),
     ylab="Stopping distance")
title(main="A: Regression spline smooth -- 3 df")
## Fit with automatic choice of smoothing parameter
cars.gam <- gam(dist ~ s(speed, k=10), data=cars)  # k=10 is default
plot(cars.gam, residuals=T, pch=1, shift=mean(predict(cars.gam)),
     shade=T)
lines(cars$speed, fitted(cars3.gam), col='red', lty=2)
title(main="B: Penalized spline smooth")
legend('topleft', lty=2, col='red', legend="Regression spline fit from Panel A", bty="n")
```

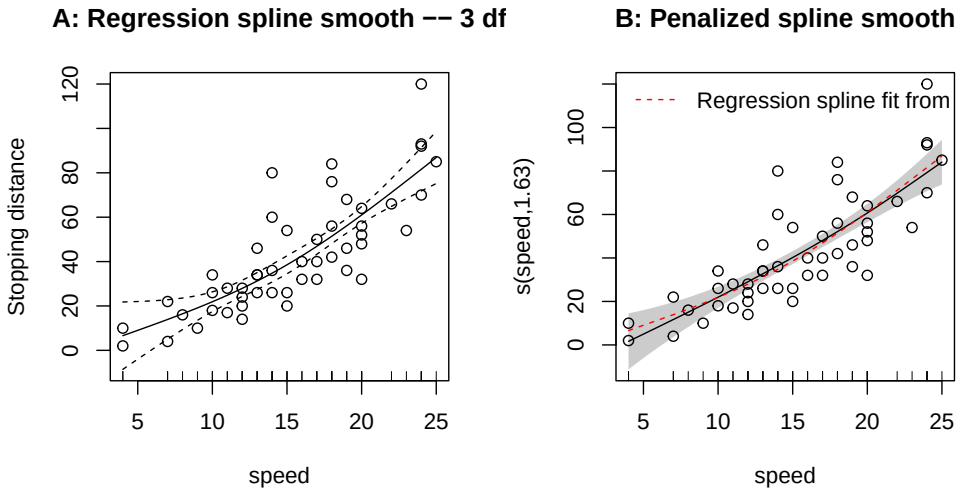

Figure 3.7: Stopping distance (`dist`) versus `speed`, for cars from the 1920s.

The Panel A choice of 3 degrees of freedom for a regression spline smooth is ad hoc. Better than such an ad hoc smooth is the penalized spline approach, which adds a penalty term that reflects model complexity to the residual sum of squares that is to be minimized. The name GAM (Generalized Additive Model) is used to refer to both types of model.

The Panel A regression spline smooth could alternatively be fitted as an `lm` style linear model. Fitting using the function `mgcv::gam()`, and specifying `k=3` and `fx=T` to obtain a regression spline fixed degrees of freedom fit, has the advantage that the function `plot.gam()` can then be used to obtain a graph that shows 2 standard error bounds.

3.5 Using Factors in R Models

Factors are crucial for specifying R models that include categorical or *factor* variables. Consider data from an experiment that compared houses with and without cavity insulation . While one would not usually handle these calculations using an lm model, it makes a simple example to illustrate the choi,ce 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
        # `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
-4409.0  -979.1   131.9  1575.0  3690.0

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)         7890.1      873.8   9.030  5.8e-07
insulationwithout   3102.9     1196.4   2.593   0.0223

Residual standard error: 2312 on 13 degrees of freedom
Multiple R-squared:  0.341, Adjusted R-squared:  0.2903
F-statistic: 6.726 on 1 and 13 DF,  p-value: 0.02228
```

The p-value is 0.022, which may be taken as weak evidence that we can distinguish between the two types of houses. The factor levels are by default taken in alphabetical order, with the initial level as the baseline. Thus, `with` comes before `without`, and is the baseline. Hence:

Average for Insulated Houses = 7980
Estimate for uninsulated houses = 7980 + 3103 = 10993
Standard error of difference = 1196.

It often helps to keep in mind 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 withouts:

7980	3103	Add to get	Compare with	Residual
1	1	7980+3103=10993	10225	10225-10993
1	1	7980+3103=10993	10689	10689-10993
. . .	. . .	. . .	. . .	. . . .

7980	3103	Add to get	Compare with	Residual
1	0	7980+0	9708	9708-7980
1	0	7980+0	6700	6700-7980

Type

```
model.matrix(kWh ~ insulation)
   (Intercept) insulationwithout
1            1                  1
2            1                  1
3            1                  1
4            1                  1
5            1                  1
6            1                  1
7            1                  1
8            1                  1
9            1                  0
10           1                  0
11           1                  0
12           1                  0
13           1                  0
14           1                  0
15           1                  0
attr(,"assign")
[1] 0 1
attr(,"contrasts")
attr(,"contrasts")$insulation
[1] "contr.treatment"
```

and check that it gives the above model matrix.

Alternative Choices of Contrasts

The X matrix can take different forms, depending on the choice of contrasts. One obvious alternative to that shown is One obvious alternative in the present example is to make `without` the first factor level, so that it becomes the baseline or reference level. For this, specify:

```
insulation <- relevel(insulation, ref="without")
  # Make `without' the baseline
```

Another possibility is 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 from use of the 'sum' contrasts :

```
options(contrasts = c("contr.sum", "contr.poly"), digits = 2)
   #  Try the `sum' contrasts
insulation <- factor(insulation, levels=c("without", "with"))
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)     9442        598   15.78  7.4e-10
insulation1     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 DF,  p-value: 0.0223
```

Here is the interpretation:

Average of (mean for "without", "mean for with") = 9442
Estimate for uninsulated houses (the first level) = 9442 + 1551 = 10993 As *effects* sum to one, the effect for the 2^{nd} level ('with') is -1551.
Thus the estimate for insulated houses (1^{st} level) = 9442 - 1551 = 7980.

The sum contrasts are sometimes called "analysis of variance" contrasts. It is possible to 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 and rarely helpful, even though S-PLUS uses them as the default. Novices should avoid them .

3.6 Multiple Lines – Different Regression Lines for Different Species

The terms that 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 that follows, we have weights for a porpoise species (Stellena styx) and for a dolphin species (Delphinus delphis).

```
dolphins <- data.frame(
  weight = c(35, 42, 71, 65, 63, 64, 45, 54, 59, 50,
42, 55, 37, 47, 40, 52),
  heart = c(245, 255, 525, 425, 425, 440,
350, 300, 350, 320, 240, 305, 220, 310, 210, 350),
species = rep(c("styx", "delph"), c(7,9))
)
```

Figure 3.8 shows a plot of the data, with separate lines fitted for the two species:

```
library(lattice)
xyplot(heart ~ weight, groups=species, auto.key=list(columns=2), data=dolphins,
       par.settings=simpleTheme(pch=c(16,17)), type=c("p","r"), scales=list(log=T))
```

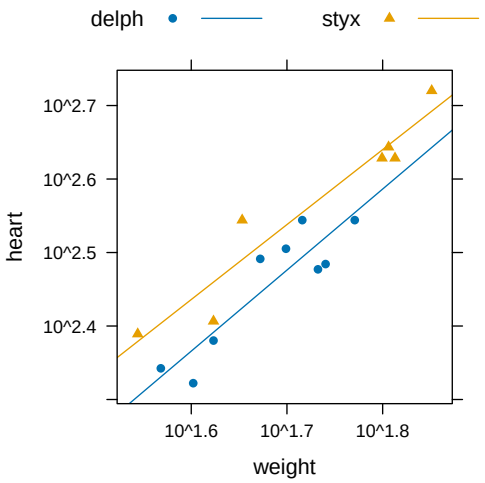

Figure 3.8: Height weight versus body weight, with lobgarithmic scales on both axes. Separate lines are fitted for the two dolphin species.

We take `x1` to be a variable that has the value 0 for Delphinus delphis, and 1 for Stellena styx. We take `x2` to be body weight. Possibilities we may want to consider are:

(a) A single line: `y = a + b x2`

(b) Two parallel lines: `y = a1 + a2 x1 + b x2`
[For the first group (Stellena styx; `x1` = 0) the constant term is `a1`, while for the second group (Delphinus delphis; `x1` = 1) the constant term is `a1 + a2`.]

(c) Two separate lines: `y = a1 + a2 x1 + b1 x2  + b2 x1 x2`
[For the first group (Delphinus delphis; `x1` = 0) the constant term is `a1` and the slope is `b1`. For the second group (Stellena styx; `x1` = 1) the constant term is `a1 + a2`, and the slope is `b1 + b2`.]

Now compare these three models, both with the AIC statistics, and with AICc which adjusts for small sample size. AIC is one of several alternative 'information' statistics.

```
cet.lm1 <- lm(log(heart) ~ log(weight), data = dolphins)
cet.lm2 <- lm(log(heart) ~ factor(species) + log(weight), data = dolphins)
cet.lm3 <- lm(log(heart) ~ factor(species) + factor(species)/log(weight), data = dolphins)
cbind(AIC(cet.lm1, cet.lm2, cet.lm3),
      AICc = sapply(list(cet.lm1, cet.lm2, cet.lm3), AICcmodavg::AICc))
        df AIC AICc
cet.lm1  3 -21  -19
cet.lm2  4 -28  -25
cet.lm3  5 -27  -21
```

The smallest value is best, in both cases. Both statistics favour the parallel lines model. The AICc statistic makes a much clearer case against fitting separate lines.

Selected rows of the model matrix are:

```
model.matrix(cet.lm2)[c(1,2,8,16), ]
   (Intercept) factor(species)1 log(weight)
1            1               -1         3.6
2            1               -1         3.7
8            1                1         4.0
16           1                1         4.0
```

Now try an analysis of variance comparison.

```
cet.lm3 <- lm(log(heart) ~ factor(species) + log(weight) +
    factor(species):log(weight), data=dolphins)
anova(cet.lm1,cet.lm2,cet.lm3)
Analysis of Variance Table

Model 1: log(heart) ~ log(weight)
Model 2: log(heart) ~ factor(species) + log(weight)
Model 3: log(heart) ~ factor(species) + log(weight) + factor(species):log(weight)
  Res.Df    RSS Df Sum of Sq    F Pr(>F)
1     14 0.1717
2     13 0.0959  1    0.0758 9.59 0.0093
3     12 0.0949  1    0.0010 0.12 0.7346
```

3.7 aov models (Analysis of Variance)

The class of models that 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 3.8 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()`, using the parameter `ref` to set the level that is wanted as the reference level.

Plant Growth Example

Figure 3.9 shows boxplot comparisons of plant weight.

```
opar <- par(mgp=c(2,0.5,0), mar=c(3.6,3.1,2.1,0.6))
with(PlantGrowth, boxplot(split(weight, group), horizontal=T))
par(opar)
```

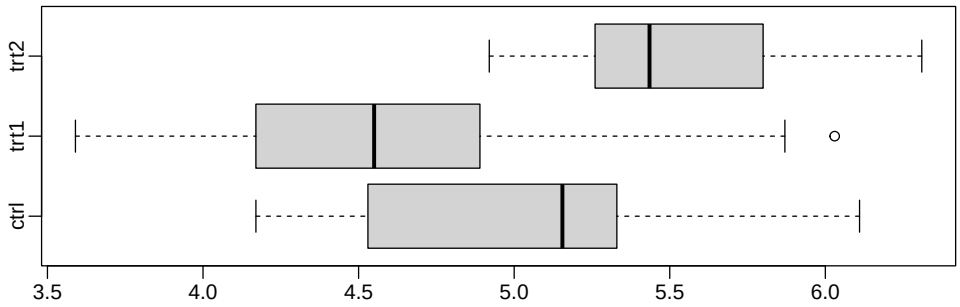

Figure 3.9: Boxplots of plant weight, by group

Now fit a model using `aov()`

```
PlantGrowth.aov <- aov(weight~group, data=PlantGrowth)
summary(PlantGrowth.aov)
            Df Sum Sq Mean Sq F value Pr(>F)
group        2   3.77   1.883    4.85  0.016
Residuals   27  10.49   0.389
summary.lm(PlantGrowth.aov)   # As from `lm` model fit

Call:
aov(formula = weight ~ group, data = PlantGrowth)

Residuals:
   Min     1Q Median     3Q    Max
-1.071 -0.418 -0.006  0.263  1.369

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    5.073      0.114   44.57   <2e-16
group1        -0.041      0.161   -0.25    0.801
group2        -0.412      0.161   -2.56    0.016

Residual standard error: 0.62 on 27 degrees of freedom
Multiple R-squared:  0.264, Adjusted R-squared:  0.21
F-statistic: 4.85 on 2 and 27 DF,  p-value: 0.0159
```

3.7.1 Dataset `MASS::cabbages` (Run code to get output)

Type `?MASS::cabbages` to get details of the data.

```
help(cabbages)          # cabbages is from the MASS package
names(cabbages)
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)
```

3.8 Shading of Kiwifruit Vines

These data (yields in kilograms) in the data frame `DAAG::kiwishade`, are from an experiment 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 <- DAAG::kiwishade
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)
Warning in aov(yield ~ block + shade + Error(block:shade), data = kiwishade):
Error() model is singular
summary(kiwishade.aov)

Error: block:shade
          Df Sum Sq Mean Sq F value Pr(>F)
block      2    172      86    4.12 0.0749
shade      3   1395     465   22.21 0.0012
Residuals  6    126      21

Error: Within
```

```
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals 36    439    12.2
coef(kiwishade.aov)
(Intercept) :
(Intercept)
         97

block:shade :
block1 block2 shade1 shade2 shade3
  0.81   1.81   3.67   6.70  -6.61

Within :
numeric(0)
```

3.9 Exercises

1. The datasets `DAAG::elastic1` and `DAAG::elastic2` were both obtained using the same apparatus, including the same rubber band, as the data frame `DAAG::elasticband`. The variable `stretch` is, in each case, the amount by which an elastic band was stretched over the end of a ruler, and `distance` the distance that the band traveled when released.

a) 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?

b) 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?

2. Enter the data frame beams, thus:

```
beams <- data.frame(
  strength = c(11.14, 12.74, 13.13, 11.51, 12.38,
               12.6, 11.13, 11.7, 11.02, 11.41),
  SpecificGravity = c(0.499, 0.558, 0.604, 0.441, 0.55,
                      0.528, 0.418, 0.48, 0.406, 0.467),
  moisture = c(11.1, 8.9, 8.8, 8.9, 8.8, 9.9, 10.7, 10.5,
               10.5, 10.7))
```

Regress `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?

3. Using the data frame `cars` (in the datasets package), 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?
4. Using the data frame hills (in package MASS), regress time on distance and climb. What can you learn from the diagnostic plots that 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?
5. Use the method of Section 3.5 to determine, formally, whether one needs different regression lines for the two data frames elastic1 and elastic2.

6.In Section 3.5, check the form of the model matrix (i) for fitting two parallel lines and (ii) for fitting two arbitrary lines, using the sum contrasts.

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 the data set `MASS::cement`, 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?

9. In the dataset `pressure` (datasets), 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?

10. *Repeat the analysis of the kiwishade data (section 5.8.2), 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.

3.10 References and reading

Cunningham (2021) . Causal inference. Yale University Press.

Faraway (2014) . Linear Models with R. Taylor & Francis.

Fox and Weisberg (2018) . An R and S-PLUS Companion to Applied Regression. Sage Books.

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach. Cambridge University Press.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.

J. Maindonald (1992) . Statistical design, analysis and presentation issues.

Muenchen (2011) . R for SAS and SPSS Users. Springer.

Tu and Gilthorpe (2011) . Statistical thinking in epidemiology. CRC Press.

Venables and Ripley (2002) . Modern Applied Statistics with S. Springer, NY.
[This assumes a fair level of statistical sophistication.
Explanation is careful, but often terse.

Wood (2017) . Generalized Additive Models. An Introduction with R. Chapman and Hall/CRC.

4 Generalized Linear and Additive Models

Generalized Linear Models (GLMs) extend multiple regression style models. Generalized Additive Models (GAMs) are a further extension.

4.1 Extending the Linear Model

The basic model formulation 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, the focus is on the form of the model prediction, leaving until later sections the discussion of different possibilities for the 'error' distribution.

The various other models we describe are, in essence, generalizations of the multiple regression model, fitted in earlier chapters using `lm()`.

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_p x_p + \epsilon$$

Generalized Linear Model (e.g. logit model) {-} To simplify the discussion, a single explanatory variable will be assumed.

$$y = g(b_0 + b_1 x_1) + \epsilon, \text{so that } \mathrm{E}[y] = g(b_0 + b_1 x_1)$$

Here `g()` is selected from one of a small number of options. For logit models, with μ the expected number of 'successes'. Then:

$$\mu = n\frac{\eta}{1+e^{\eta}}, \text{ where } \eta = b_0 + b_1 x_1$$

Here $\frac{\mu}{n}$ is an expected proportion.

We can turn this model around, and write

$$\eta = \log(\frac{\mu}{n+\mu}) = f(\frac{\pi}{1-\pi}), \text{ where } \pi = \frac{\mu}{n}$$

Here `f()` is the logit link.

Generalized Additive Models (to be introduced later)

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

Some terms may be smoothing functions, while others may be the usual linear model terms.

4.2 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. Logit models transform to 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

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

The linear model predicts, not p, but $log(\frac{p}{1-p})$.

The logit or log(odds) function turns expected proportions into values that may range from $-\infty$ to ∞. The values from the linear model may in principle vary across the whole real line. One needs a transformation, such as the logit, such that transformed values may extend outside the range from 0 to 1.

Among other link functions that are used with proportions, one of the commonest is the complementary log-log link. See `?make.link` for details of those that are available for use with GLM models, as fitted using `glm()`.

Anesthetic Depth Example

Thirty patients were given an anesthetic agent that was maintained at a pre-determined [alveolar] concentration for 15 minutes before making an incision . 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 anesthetic agent.

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

Table 4.1: Patients moving (0) and not moving (1), for each of six different alveolar concentrations.

	0.8	1	1.2	1.4	1.6	2.5
nomove=0	6	4	2	2	0	0
nomove=1	1	1	4	4	4	2

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 (divide by d.f.)	mean deviance (divide by d.f.)
Minimize residual sum of squares	Minimize 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:

```
anesthetic <- DAAG::anesthetic
anaes.logit <- glm(nomove ~ conc, family = binomial(link = logit),
                   data = anesthetic)
summary(anaes.logit)

Call:
glm(formula = nomove ~ conc, family = binomial(link = logit),
    data = anesthetic)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -6.469      2.418  -2.675  0.00748
conc           5.567      2.044   2.724  0.00645

(Dispersion parameter for binomial family taken to be 1)
```

```
    Null deviance: 41.455  on 29  degrees of freedom
Residual deviance: 27.754  on 28  degrees of freedom
AIC: 31.754

Number of Fisher Scoring iterations: 5
```

Figure 4.1 is a graphical summary of the results. The labeling on the y-axis is on the scale of the linear predictor (η),

```
termplot(anaes.logit, partial=T, pch=16, ylab="Probability", yaxt='n')
probVal <- c(c(0.02, seq(from=.1, to=.9, by=.2), 0.98, 0.998))
axis(2, at=log(probVal/(1-probVal)), labels=paste(probVal), las=1)
```

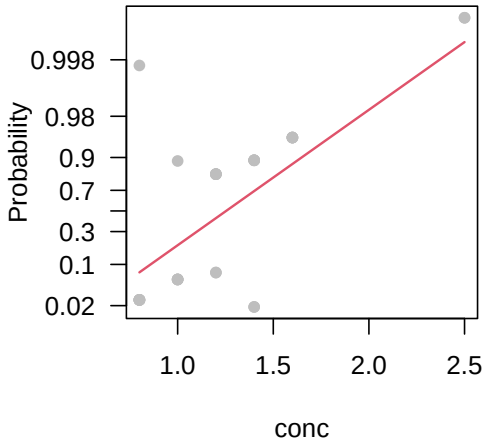

Figure 4.1: Plot, versus concentration, of log(odds) [= logit(proportion)] of patients not moving. The line is the estimate of the proportion of moves that is based on the fitted logit model.

Code is:

```
termplot(anaes.logit, partial=T, pch=16, ylab="Probability", yaxt='n')
probVal <- c(c(0.02, seq(from=.1, to=.9, by=.2), 0.98, 0.998))
axis(2, at=log(probVal/(1-probVal)), labels=paste(probVal), las=1)
```

With such a small sample size it is impossible to do much that is useful to check the adequacy of the model. Try also `plot(anaes.logit)`.

4.3 GLM models (Generalized Linear Regression Modelling)

In the above we had

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

The family parameter specifies the distribution for the dependent variable. An optional argument allows specification of the link function. Below we give further examples.

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 is log. The log link transforms from positive numbers to numbers in the range $-\infty$ to ∞.

The gaussian family

If no family is specified, then the family is taken to be gaussian.
The default link is then the identity, effectively giving an `lm` model. This way of formulating a linear model has the advantage that one is not restricted to what is effectively the identity link.

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

4.4 Generalized Additive Models (GAMs)

In most circumstances, rather than adding regression spline terms to `lm` style linear models, it is better to move directly to using the penalized spline approach, with automatic choice of smoothing parameter, that is implemented in the *mgcv* package. Note however the risk of over-fitting is there is a correlation structure in the data, such as is likely to be present for time sereis.

Dewpoint Data

The data set dewpoint 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 for a generalized additive model:

```
dewpoint <- DAAG::dewpoint
library(mgcv)
Loading required package: nlme
This is mgcv 1.9-0. For overview type 'help("mgcv-package")'.
dewpoint.gam <- gam(dewpt ~ s(mintemp) + s(maxtemp),
                        data = dewpoint)
summary(dewpoint.gam, digits=3)

Family: gaussian
Link function: identity

Formula:
dewpt ~ s(mintemp) + s(maxtemp)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.12500    0.04215   311.4   <2e-16

Approximate significance of smooth terms:
             edf Ref.df      F p-value
s(mintemp) 3.220  4.042 3992.2  <2e-16
s(maxtemp) 6.252  7.420  794.3  <2e-16

R-sq.(adj) =  0.996   Deviance explained = 99.7%
GCV = 0.14971  Scale est. = 0.12794   n = 72
```

Model Summaries

Type:

```
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 object. The output may not be what you might expect. So be careful!

4.5 Further types of model

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 modeled using this same methodology. The R survival package has state of the art abilities for survival analysis.

Nonlinear Models

The function `nls()` (non-linear least squares) can be used to obtain a least squares fit to a non-linear function.

4.6 Further Elaborations

Generalized Linear Models were developed in the 1970s. They unified a wide range of what had earlier been treated as distinct methods, and have now become a stock-in-trade of statistical analysts. Their practical implementation took advantage of new computational abilities that 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* and *lme4* packages implement such extensions, for a wide class of nonlinear models as well for linear models. Each such new development builds on the theoretical and computational tools that have arisen from earlier developments. Powerful 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.

4.7 Exercises

1. Fit a Poisson regression model to the data in the data frame `DAAG::moths`, Allow different intercepts for different habitats. Use `log(meters)` as a covariate. Why `log(meters)`?

4.8 References and reading

Faraway (2016) . Extending the Linear Model with R. Taylor & Francis.

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach. Cambridge University Press.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.

Venables and Ripley (2002) . Modern Applied Statistics with S. Springer, NY.

Wood (2017) . Generalized Additive Models. An Introduction with R. Chapman and Hall/CRC.

5 Regular Time Series in R

Time points that are close together in time commonly show a sequential (usually, +ve) correlation.

Any process that evolves in time is likely to have a sequential correlation structure. The value at the current time is likely to be correlated with the value at the previous time, and perhaps with values several time points back. The discussion that follows will explore implications for data analysis.

R's `acf()` and `arima()` functions are useful tools for exploring time series.

The data series `Erie`, giving levels of Lake Erie from 1918 to 2009, will be used as an example from which to start the discussion.[1]

The series is available as the column `Erie` in the multivariate time series object `DAAG::greatLakes`.

```
Erie <- DAAG::greatLakes[,"Erie"]
attributes(Erie)    # Erie is a time series, for years 1918:2009
$tsp
[1] 1918 2009    1

$class
[1] "ts"
```

2 shows a plot of the series.

Code is:

```
par(mfrow=c(1,2))
# Panel A
plot(Erie, xlab="", fg="gray", ylab="Level (m)")
mtext(side=3, line=1, adj=0,
      "A: Lag plots, at lags of 1, 2 and 3")
## Panel B
acf(Erie, main="", fg="gray")
mtext(side=3, line=1, adj=0, "B: Autocorrelation function")
```

[1] See `?greatlakes` for details of the source of the data.

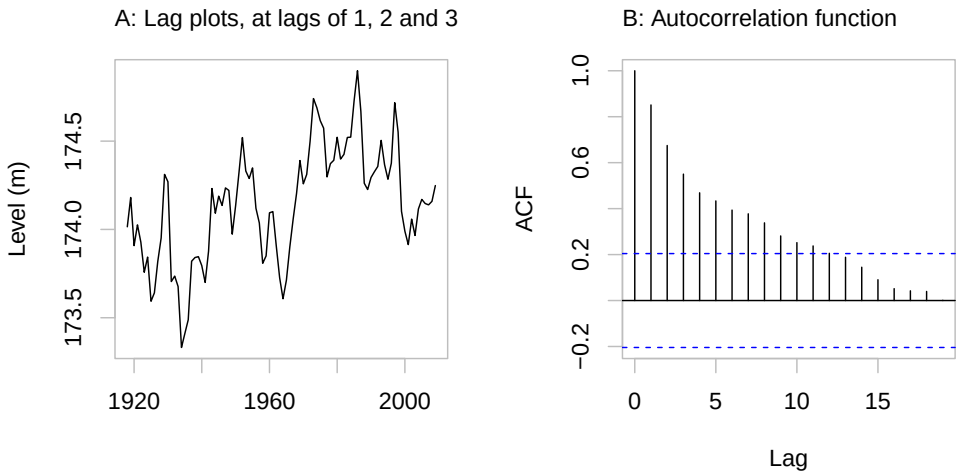

Figure 5.1: Panel A plots Lake Erie levels (m). Panel B shows a consistent pattern of decreasing autocorrelation at successive lags.

Figure 5.2 shows a lag plot of the series.

```
cap33 <- "Lag plots for Lake Erie levels vs levels at lags 1, 2 and 3
  respectively."
```

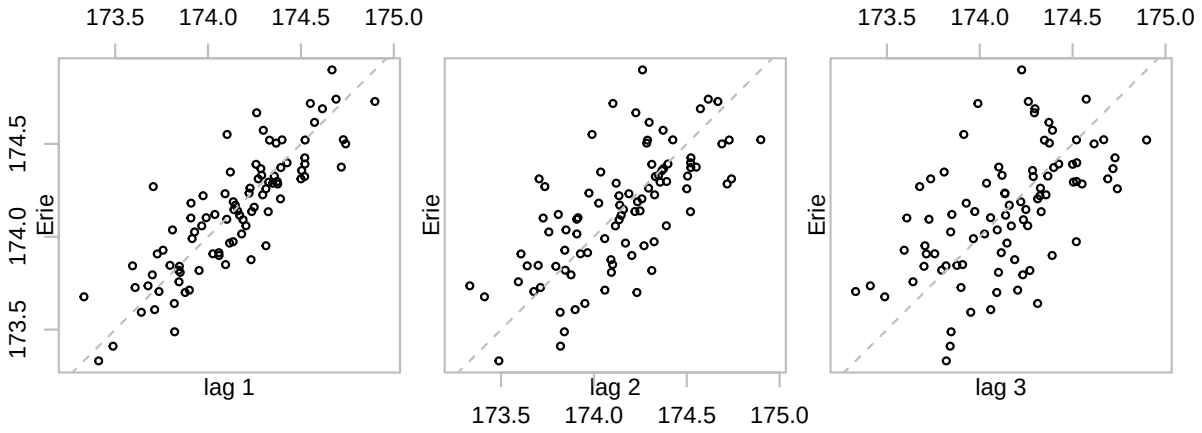

Figure 5.2: Lag plots for Lake Erie levels vs levels at lags 1, 2 and 3 respectively.

Where values of covariates are available that largely or partly explain the dependence, it may make sense to account for these in the model. The details of how this should be done will depend on the intended use of the model.

2 and Figure 5.2 provide a good starting point for investigation of the correlation structure.

There is a strong correlation at lag 1, a strong but weaker correlation at lag 2, and a noticeable correlation at lag 3. Such a correlation pattern is typical of an autoregressive process where most of the sequential dependence can be explained as a flow-on effect from a dependence at lag 1.

In an autoregressive time series, an independent error component, or *innovation* is associated with each time point.[2] For an order p autoregressive time series, the error for any time point is obtained by taking the innovation for that time point, and adding a linear combination of the innovations at the p previous time points. (For the present time series, initial indications are that $p = 1$ might capture most of the correlation structure.)

Patterns that are repeatable

Smoothing terms can be fitted to the pattern apparent in serially correlated data, leaving {*errors*} *that are pretty much uncorrelated. Such a pattern is in general, however, unrepeatable. It gives little clue of what may happen the future. A re-run of the process (a new* {realization}) will produce a different series, albeit one that shows the same general tendency to move up and down.

What sorts of patterns may then be repeatable? Indications that a pattern may be repeatable include:

- A straight line trend is a good starting point for some limited extrapolation. But think: Is it plausible that the trend will continue more than a short distance into the future?

- There may be a clear pattern of seasonal change, e.g., with seasons of the year or (as happens with airborne pollution) with days of the week. If yearly seasonal changes persist over different years, or weekly day-of-the-week changes persist over different weeks, these effects can perhaps be extrapolated with some reasonable confidence.

- There is a regression relationship that seems likely to explain future as well as current data.

An ideal would be to find a covariate or covariates than can largely explain the year to year changes. For this series, this does not seem a possibility. In the absence of identifiable direct cause for the year to year changes, a reasonable recourse is to look for a correlation structure that largely accounts for the pattern of the year to year change.

[2] An autoregressive model is a special case of an Autoregressive Moving Average (ARMA) model.

5.1 Smooth, with automatic choice of smoothing parameter

Figure 5.3 uses the abilities of the *mgcv* package, assuming independently and identically distributed data (hence, no serial correlation!) to make an automatic choice of the smoothing parameter. As the curve is conditional on a particular realization of the process that generated it, its usefulness is limited. It does not separate systematic effects from effects due to processes that evolve in time. The curve is not repeatable.

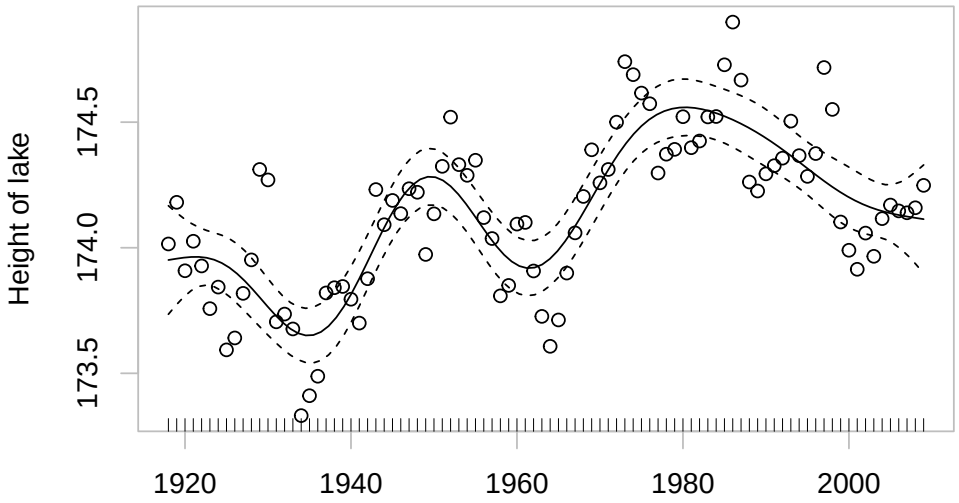

Figure 5.3: GAM smoothing term, fitted to the Lake Erie Data. Most of the autocorrelation structure has been removed, leaving residuals that are very nearly independent.

Code is:

```
suppressPackageStartupMessages(library(mgcv))
df <-  data.frame(height=as.vector(Erie),  year=time(Erie))
obj <- gam(height ~ s(year), data=df)
plot(obj, fg="gray", shift=mean(df$height), residuals=TRUE, pch=1,
     xlab="", ylab="Height of lake")
```

The pointwise confidence limits are similarly conditioned, relevant perhaps for interpolation given this particular realization. All that is repeatable, given another realization, is the process that generated the curve, not the curve itself.

5.2 Fitting and use of an autoregressive model

Several different types of time series models may be used to model the correlation structure, allowing realistic estimates of the lake level a short time ahead, with realistic confidence bounds

around those estimates. For the Lake Erie data, an autoregressive correlation structure does a good job of accounting for the pattern of change around a mean that stays constant.

Figure 5.2 suggested that a correlation between each year and the previous year accounted for the main part of the autocorrelation structure in Figure 2. An AR1 model (autoregressive with a correlation at lag 1 only), which we now fit, formalizes this.

```
ar(Erie, order.max=1)

Call:
ar(x = Erie, order.max = 1)

Coefficients:
     1
0.8512

Order selected 1  sigma^2 estimated as  0.02906
```

The one coefficient that is now given is the lag 1 correlation, equaling 0.851.

1) investigates how repeated simulations of this process, with a lag 1 correlation of 0.0.85, compare with

2. This illustrates the point that a GAM smooth will extract, from an autoregressive process with mean 0, a pattern that is not repeatable when the process is re-run.

Code is:

```
par(mfrow=c(2,3))
for (i in 1:6){
ysim <- arima.sim(list(ar=0.85), n=200)
df <- data.frame(x=1:200, y=ysim)
df.gam <- gam(y ~ s(x), data=df)
plot(df.gam, fg="gray", ylab=paste("Sim", i), residuals=TRUE)
}
```

The curves are different on each occasion. For generalization beyond the particular realization that generated them, they serve no useful purpose.

Once an autoregressive model has been fitted, the function `forecast()` in the *forecast* package can be used to predict future levels, albeit with very wide confidence bounds. For this, it is necessary to refit the model using the function `arima()`. An arima model with order (1,0,0) is an autoregressive model with order 1.

Code is:

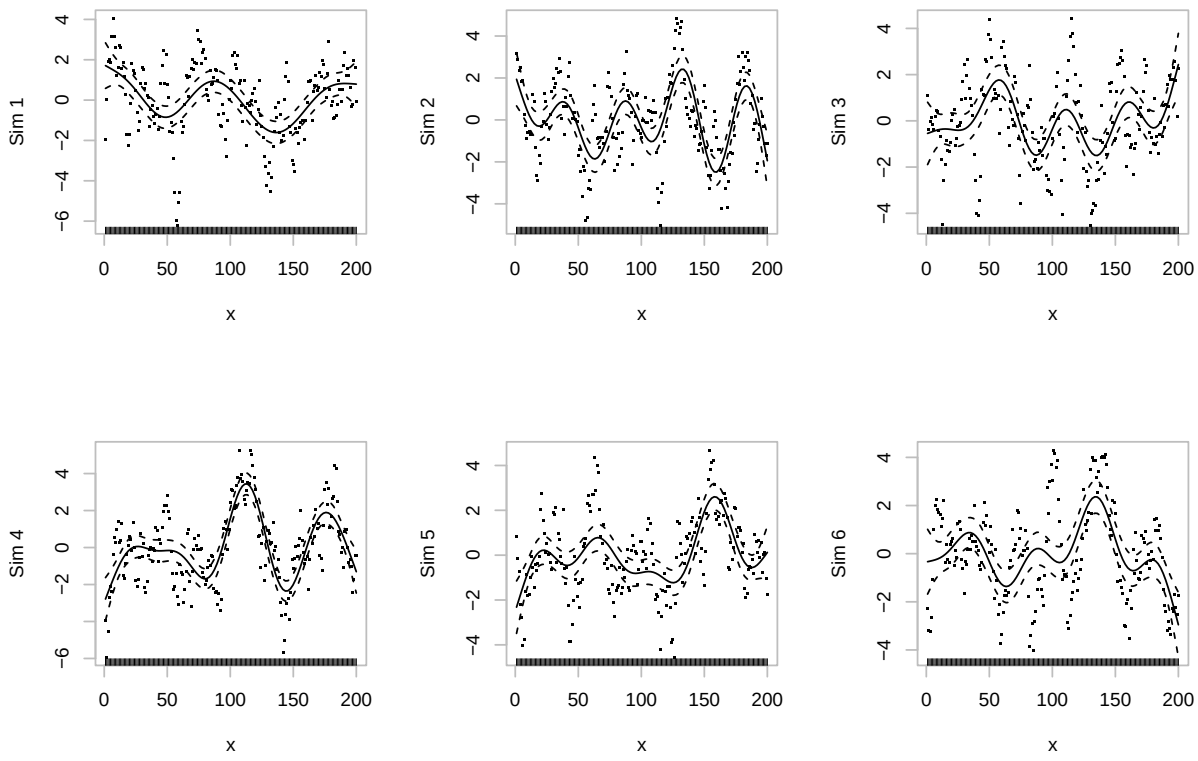

Figure 5.4: The plots are from repeated simulations of an AR1 process with a lag 1 correlation of 0.85. Smooth curves, assuming independent errors, have been fitted.

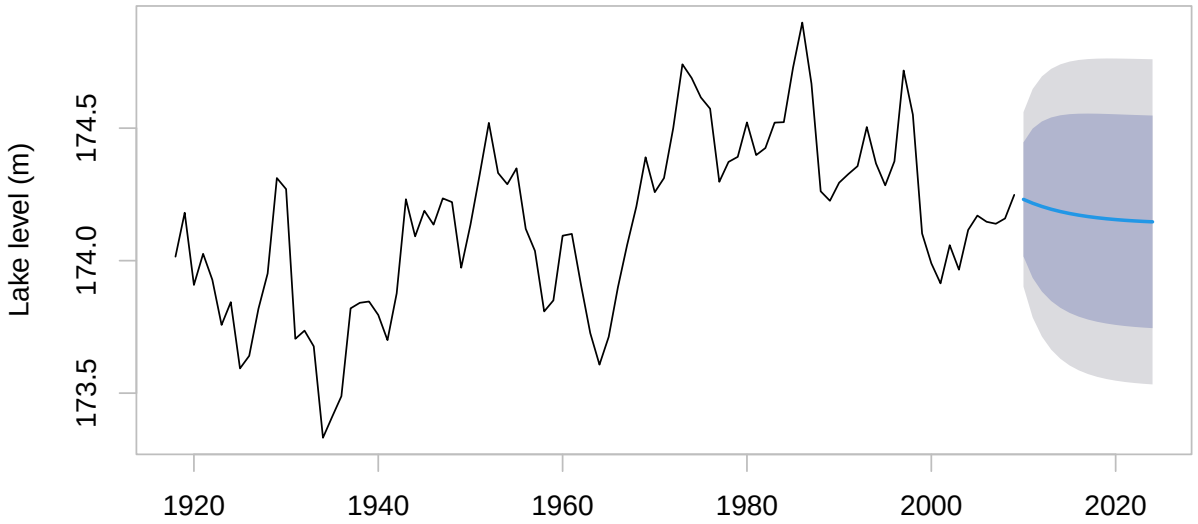

Figure 5.5: Predictions, 15 years into the future, of lake levels (m). The shaded areas give 80% and 95% confidence bounds.

```
erie.ar <- arima(Erie, order=c(1,0,0))
suppressPackageStartupMessages(library(forecast))
fc <- forecast(erie.ar, h=15)
plot(fc, main="", fg="gray",  ylab="Lake level (m)")
  # 15 time points ahead
```

This brief excursion into a simple form of time series model is designed only to indicate the limitations of automatic smooths, and to give a sense of the broad style of time series modeling. The list of references at the end of the chapter has details of several books on time series.

5.3 Regression with time series errors

Figure 5.6 shows the estimated contributions of the two model terms, in a fit of annual rainfall in the Murray-Darling basin of Australia as a sum of smooth functions of `Year` and `SOI`.

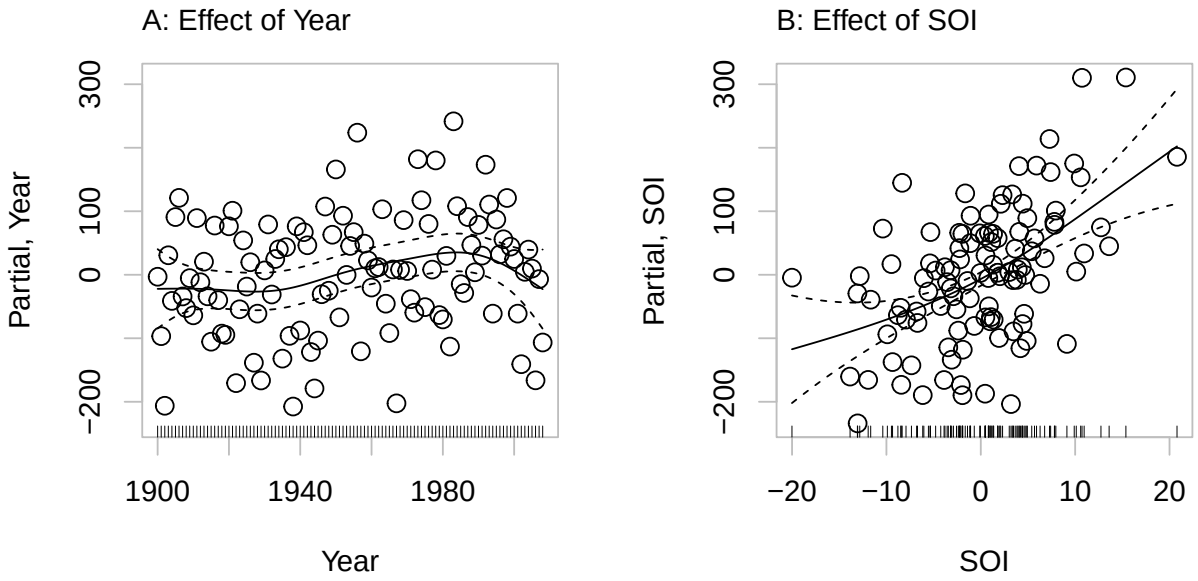

Figure 5.6: Estimated contributions of model terms to `mdbRain`, in a GAM model that adds smooth terms in `Year` and `Rain`. The dashed curves show pointwise 2-SE limits, for the fitted curve.

Code is:

```
par(mfrow=c(1,2))
mdbRain.gam <- gam(mdbRain ~ s(Year) + s(SOI), data=DAAG::bomregions)
plot(mdbRain.gam, residuals=TRUE, se=2, fg="gray",
```

```
     pch=1, select=1, cex=1.35, ylab="Partial, Year")
mtext(side=3, line=0.75, "A: Effect of Year", adj=0)
plot(mdbRain.gam, residuals=TRUE, se=2, fg="gray",
     pch=1, select=2, cex=1.35, ylab="Partial, SOI")
mtext(side=3, line=0.75, "B: Effect of SOI", adj=0)
```

The left panel indicates a consistent pattern of increase of rainfall with succeeding years, given an adjustment for the effect of SOI. Errors from the fitted model are consistent with the independent errors assumption. The model has then identified a pattern of increase of rainfall with time, given SOI, that does seem real. It is necessary to warn against reliance on extrapolation more than a few time points into the future. While the result is consistent with expected effects from global warming, those effects are known to play out very differently in different parts of the globe.

Investigation of the residual error structure

Sequential correlation structures are often effective, with data collected over time, for use in modeling departure from iid errors. Where there is such structure structure in the data, the methodology will if possible use a smooth curve to account for it.

The residuals can be checked to determine whether the fitted curve has removed most of the correlation structure in the data. Figure 5.7 shows the autocorrelation function of the residuals, followed by autocorrelation functions for several series of independent random normal numbers. Apart from the weakly attested correlation at a lag of 12 years, which is a commonplace of weather data, the pattern of sequential correlation is not much different from what can be expected in a sequence of independent random normal numbers.

Code is:

```
mdbRain.gam <- gam(mdbRain ~ s(Year) + s(SOI), data=DAAG::bomregions)
n <-  dim(DAAG::bomregions)[1]
acf(resid(mdbRain.gam), ylab="MDB series")
for(i in 1:5)acf(rnorm(n), ylab=paste("Sim",i),
                 fg="gray", col="gray40")
```

5.4 *Box-Jenkins ARIMA Time Series Modeling

Models that are closely analogous to ARIMA models had been used earlier in control theory. ARIMA models are feedback systems! From the perspective of the Box-Jenkins ARIMA (Autoregressive Integrated Moving Average) approach to time series models, autoregressive models

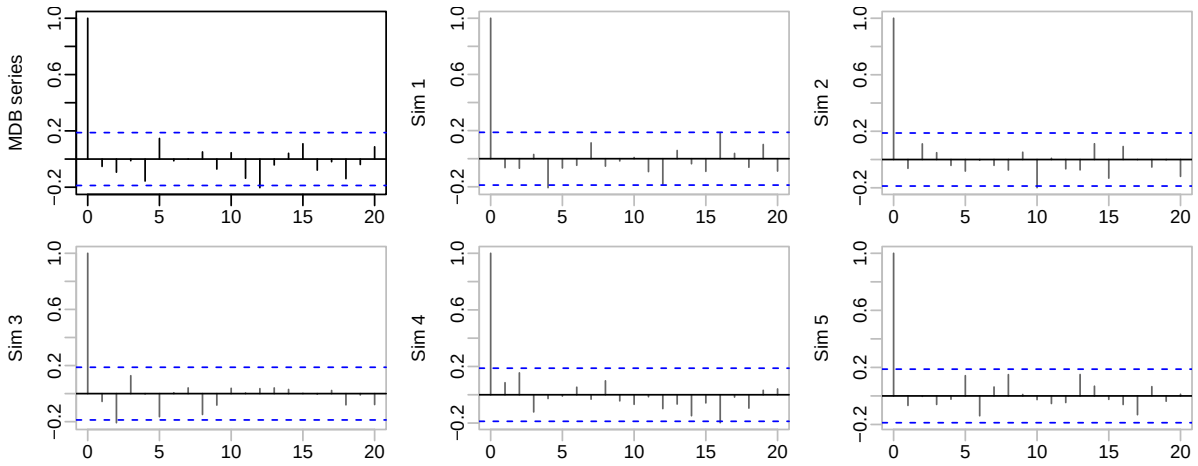

Figure 5.7: The top left panel shows the autocorrelations of the residuals from the model `mdbRain.gam`. The five remaining panels are the equivalent plots for sequences of independent random normal numbers.

are a special case. Many standard types of time series can be modeled very satisfactorily as ARIMA processes.

Exercise

The simulations in Figure 5.7 show a pattern of variation that seems not too different from that in the actual series. Modeling of the process as an ARMA or ARIMA process (i.e., allow for a moving average term) may do even better. Use the `auto.arima()` function in the *forecast* package to fit an ARIMA process:

5.5 Count Data with Poisson Errors

Data are for aircraft accidents, from the website https://www.planecrashinfo.com/. The 1920 file has accidents starting from 1908. The full data are in the dataset `gamclass::airAccs`. Data are a time series.
Serious accidents are sufficiently uncommon that joint occurrences, or cases where one event changes the probability of the next, are likely to be uncommon.

Such issues as there are with sequential correlation can be ameliorated by working with weekly, rather than daily, counts.

Figure 5.8 shows a fitted smooth curve, with pointwise confidence bounds, from a GAM smoothing model that was fitted to the weekly counts.

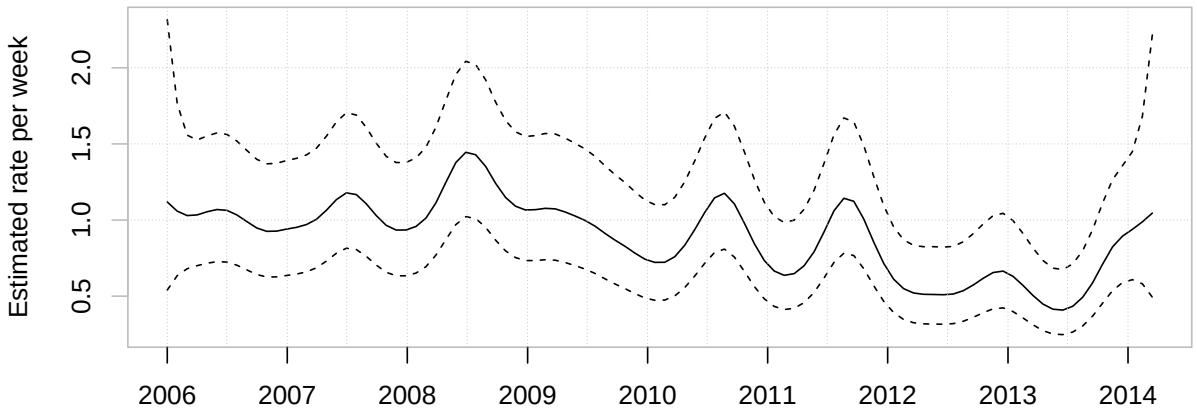

Figure 5.8: Estimated number of events (aircraft crashes) per week, versus time. The yearly tick marks are for January 1 of the stated year.

The function `gamclass::eventCounts()` was used to create weekly counts of accidents from January 1, 2006:

```
## Code
airAccs <- gamclass::airAccs
fromDate <- as.Date("2006-01-01")
dfWeek06 <- gamclass::eventCounts(airAccs, dateCol="Date",
                  from=fromDate, by="1 week", prefix="num")
dfWeek06$day <- julian(dfWeek06$Date, origin=fromDate)
```

Code for Figure 5.8 is then.

```
## Code
suppressPackageStartupMessages(library(mgcv))
year <- seq(from=fromDate, to=max(dfWeek06$Date), by="1 year")
at6 <- julian(seq(from=fromDate, to=max(dfWeek06$Date),
                  by="6 months"), origin=fromDate)
atyear <- julian(year, origin=fromDate)
dfWeek06.gam <- gam(num~s(day, k=200), data=dfWeek06, family=quasipoisson)
avWk <- mean(predict(dfWeek06.gam))
plot(dfWeek06.gam, xaxt="n", shift=avWk, trans=exp, rug=FALSE,
     xlab="", ylab="Estimated rate per week", fg="gray")
axis(1, at=atyear, labels=format(year, "%Y"), lwd=0, lwd.ticks=1)
abline(h=0.5+(1:4)*0.5, v=at6, col="gray", lty=3, lwd=0.5)
# mtext(side=3, line=0.75, "A: Events per week, vs date", adj=0)
```

The argument `k` to the function `s()` that sets up the smooth controls the temporal resolution.

A large `k` allows, if the data seem to justify it, for fine resolution. A penalty is applied that discriminates against curves that are overly 'wiggly'.

Not all count data is suitable for modeling assuming a Poisson type rare event distribution. For example, the dataset `DAAG::hurricNamed` has details, for the years 1950-2012, of US deaths from Atlantic hurricanes. For any given hurricane, deaths are not at all independent rare events.

5.6 Exercises

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

5.7 References and reading

See the vignette that accompanies the *forecast* package.

Hyndman and Athanasopoulos (2021) . Forecasting: principles and practice. OTexts.

6 Tree-based models

6.1 Decision Tree models (Tree-based models)

Tree-based classification, as for example implemented in the *rpart* (recursive partitioning) package, can be used for multivariate supervised classification (discrimination) or for tree-based regression. Tree-based methods are in general more suited to binary regression and classification than to regression with an ordinal or continuous dependent variable.

Such "Classification and Regression Trees" (CART), may be suitable for regression and classification problems when there are extensive data. An advantage of such methods is that they automatically handle non-linearity and interactions. Output includes a "decision tree" that is immediately useful for prediction. The `MASS:fgl` glass fragment data will be used for an example. If high levels of accuracy are important and obtaining a single decision tree is not a priority, the ''random forests" approach that will be described below is usually to be preferred.

Figure 6.1 shows an initial tree, before pruning.

Code is:

```
library(rpart)
fgl <- MASS::fgl
set.seed(31)    ## Use to reproduce output shown.
# Use fgl: Forensic glass fragment data; from MASS package
glass.tree <- rpart(type ~ RI+Na+Mg+Al+Si+K+Ca+Ba+Fe, data=fgl)
plot(glass.tree);  text(glass.tree)
```

Now check how cross-validated predictive accuracy changes with the number of splits. The column `xerror` is the one to check. Error values must be multiplied by the root node error to get an absolute error value.

```
printcp(glass.tree, digits=3)

Classification tree:
rpart(formula = type ~ RI + Na + Mg + Al + Si + K + Ca + Ba +
    Fe, data = fgl)
```

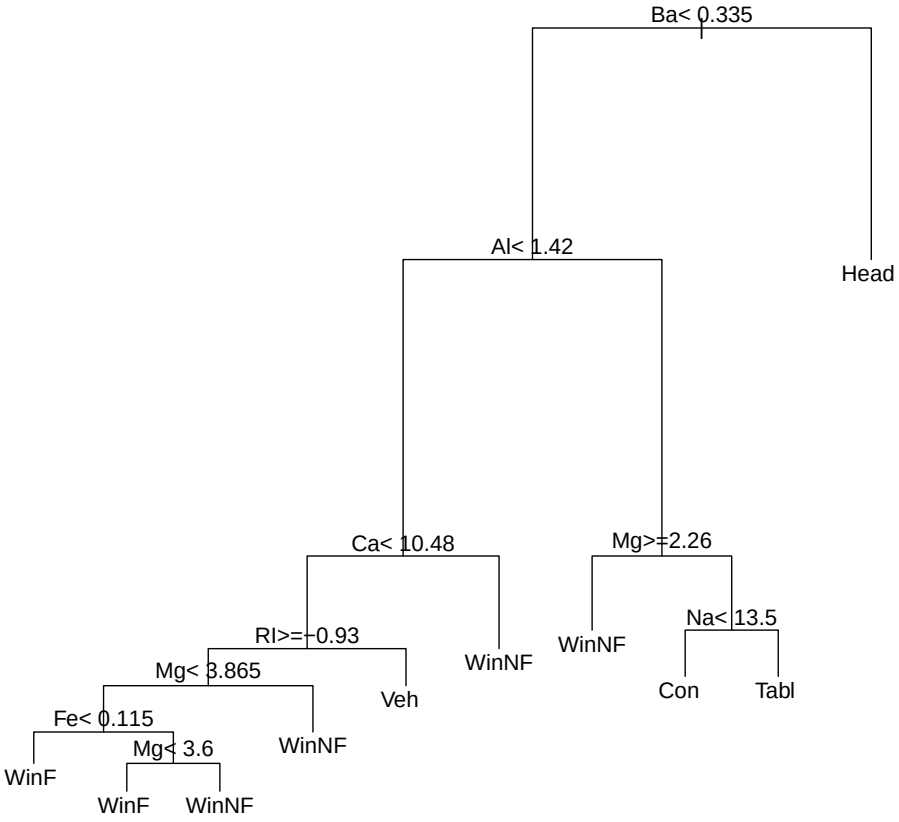

Figure 6.1: Initial tree for predicting `type` for the forensic glass data.

```
Variables actually used in tree construction:
[1] Al Ba Ca Fe Mg Na RI

Root node error: 138/214 = 0.645

n= 214

      CP nsplit rel error xerror   xstd
1 0.2065      0     1.000  1.043 0.0497
2 0.0725      2     0.587  0.601 0.0517
3 0.0580      3     0.514  0.572 0.0512
4 0.0362      4     0.457  0.493 0.0494
5 0.0326      5     0.420  0.507 0.0497
6 0.0109      7     0.355  0.464 0.0485
7 0.0100      9     0.333  0.471 0.0487
```

The optimum number of splits, as indicated by this output (this may change from one run to the next) is 7. The function `prune()` should be used to prune the splits back to this number. For this purpose, set `cp` to a value between that for `nsplit=7` and that for `nsplit=5`.

```
printcp(prune(glass.tree, cp = 0.011))

Classification tree:
rpart(formula = type ~ RI + Na + Mg + Al + Si + K + Ca + Ba +
    Fe, data = fgl)

Variables actually used in tree construction:
[1] Al Ba Ca Mg Na RI

Root node error: 138/214 = 0.64486

n= 214

        CP nsplit rel error  xerror     xstd
1 0.206522      0   1.00000 1.04348 0.049733
2 0.072464      2   0.58696 0.60145 0.051652
3 0.057971      3   0.51449 0.57246 0.051156
4 0.036232      4   0.45652 0.49275 0.049357
5 0.032609      5   0.42029 0.50725 0.049733
6 0.011000      7   0.35507 0.46377 0.048534
```

To use single tree methods effectively, one needs to be familiar with approaches for such

pruning, involving the use of cross-validation to obtain error estimates. 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.

6.1.1 The random forests approach

The random forests approach, implemented in the *randomForests* package, involves generating trees for repeated bootstrap samples (by default, 500) from the data, with data that is excluded from the bootstrap sample used to make, in each case, a prediction for that tree. The final prediction is based on a vote over all trees This is simplest for classification trees. The stopping criterion for individual trees is, unlike the case for single tree methods, not of great importance. Predictive accuracy is typically much better than for single tree methods.

6.2 Exercises

1. The `MASS::Aids2` dataset contains 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.
2. Compare the effectiveness of `rpart()` with that of `randomForest()`, for discriminating between plagiotropic and orthotropic species in the data set `DAAG::leafshape`.

6.3 References and reading

See the vignettes that accompany the *rpart* package.

Liaw and Wiener (2002) . Classification and Regression by randomForest. R News.

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach. Cambridge University Press.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.

7 Multivariate 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 initial set of variables by a small number of "principal components" that together may explain most of the variation in the data. The first principal component is the component (linear combination of the initial variables) 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 remaining variation, and so on.

The measure of variation used is the sum of the variances of variables, perhaps after scaling so that they each have variance one. An analysis that works with the unscaled variables, and hence with the variance-covariance matrix, gives a greater weight to variables that have a large variance. The common alternative — scaling variables so that they each have variance equal to one – is equivalent to working with the correlation matrix.

With biological measurement data, it is usually desirable to begin by taking logarithms. The standard deviations then measure the logarithm of relative change. Because all variables measure much the same quantity (i.e. relative variability), and because the standard deviations are typically fairly comparable, scaling to give equal variances is unnecessary.

The data set `DAAG::possum` has nine morphometric measurements on each of 102 mountain brushtail possums, trapped at seven sites from southern Victoria to central Queensland . 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. Taking logarithms of these data does not make much difference to the appearance of the plots. This is because the ratio of largest to smallest value is relatively small, never more than 1.6, for all variables.

Scatterplot matrix possibilities include:

```
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 (Figure 7.1) at the view of the data that comes from plotting the second principal component against the first:

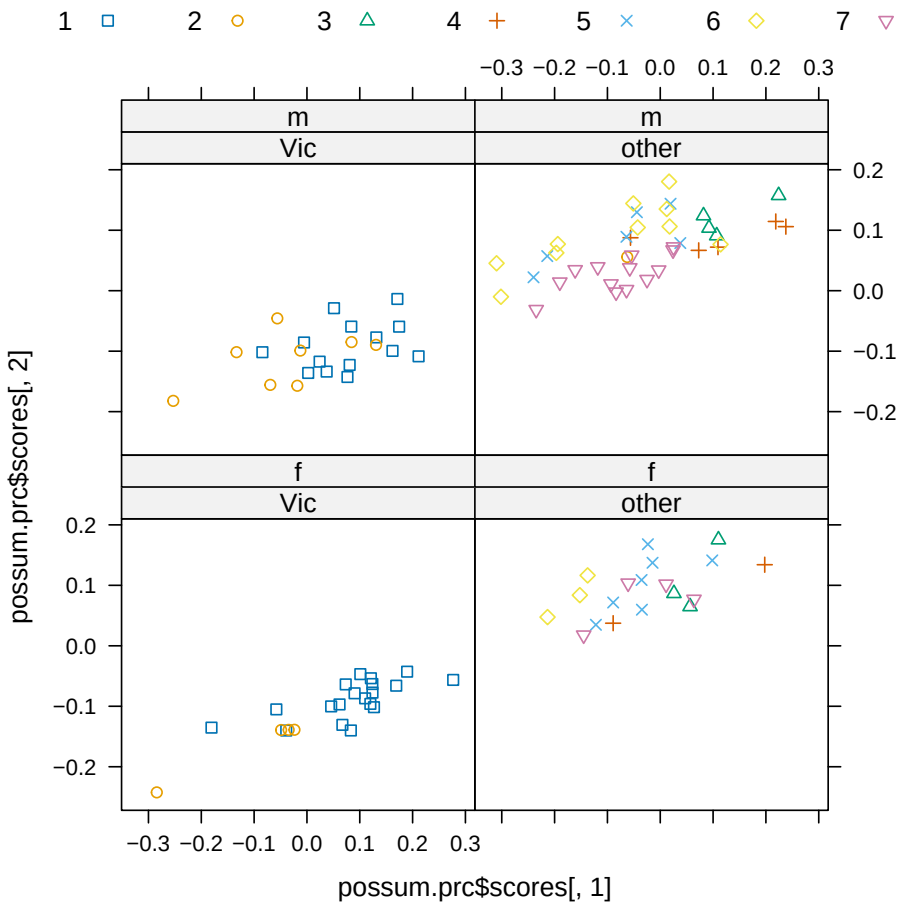

Figure 7.1: Second principal component versus first principal component, by population and by sex, for the possum data.

Code is:

```
possum <- DAAG::possum
here <- !is.na(possum$footlgth)    # We need to exclude missing values
possum.prc <- princomp(log(possum[here,6:14]))  # Principal components
# Print scores on second pc versus scores on first pc,
# by populations and sex, identified by site
library(lattice)
xyplot(possum.prc$scores[,2] ~  possum.prc$scores[,1]|
       possum$Pop[here]+possum$sex[here], groups=possum$site,
      par.settings=simpleTheme(pch=0:6), auto.key=list(columns=7))
```

7.2 Cluster Analysis

Cluster analysis is a form of unsupervised classification, 'unsupervised' because the clusters are not known in advance. There are two common 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 that shows the relationships between observations and between the clusters into which they are successively merged. A judgment is then needed on the point at which further merging is unwarranted.

In iterative relocation, the algorithm starts with an initial classification, typically from a prior use of a hierarchical agglomeration algorithm, that it then tries to improve.

The mva package has the function `dist()` that calculates distances, the function `hclust()` that does hierarchical agglomerative clustering with with one of several choices of methods, and the function `kmeans()` (k-means clustering) that implements iterative relocation.

7.3 Discriminant Analysis

We start with data that are classified into several groups, and want a rule that will allow us to predict the group to which a new data value will belong. This is a form of supervised classification – thr groups are known in advance. For example, we may wish to predict, based on prognostic measurements and outcome information for previous patients, which future patients are likely to remain free of disease symptoms for twelve months or more following treatment. Or we may wish to check, based on morphometric measurements, the extent of differences between animals from different sites. Can they be clearly distinguished, to an extent that they appear different species?

Calculations now follow for the possum data frame, using the `lda()` function from the *MASS* package. Our interest is in whether it is possible, on the basis of morphometric measurements, to distinguish animals from different sites. A cruder distinction is between populations, i.e. sites in Victoria (an Australian state) as opposed to sites in other states (New South Wales or Queensland). Because there is little effect on the distribution of variable values, it appears unnecessary to take logarithms. This will be discussed further below.

```
library(MASS)                 # Only if not already attached.
here<- !is.na(possum$footlgth)
possum.lda <- lda(site ~ hdlngth+skullw+totlngth+
   totlngth+footlgth+earconch+eye+chest+belly,data=possum, subset=here)
options(digits=4)
```

```
possum.lda$svd   # Examine the singular values
[1] 13.8327  3.9224  2.7004  1.4731  1.0494  0.3572
```

Figure 7.2 shows the scatterplot matrix of first three canonical variates.

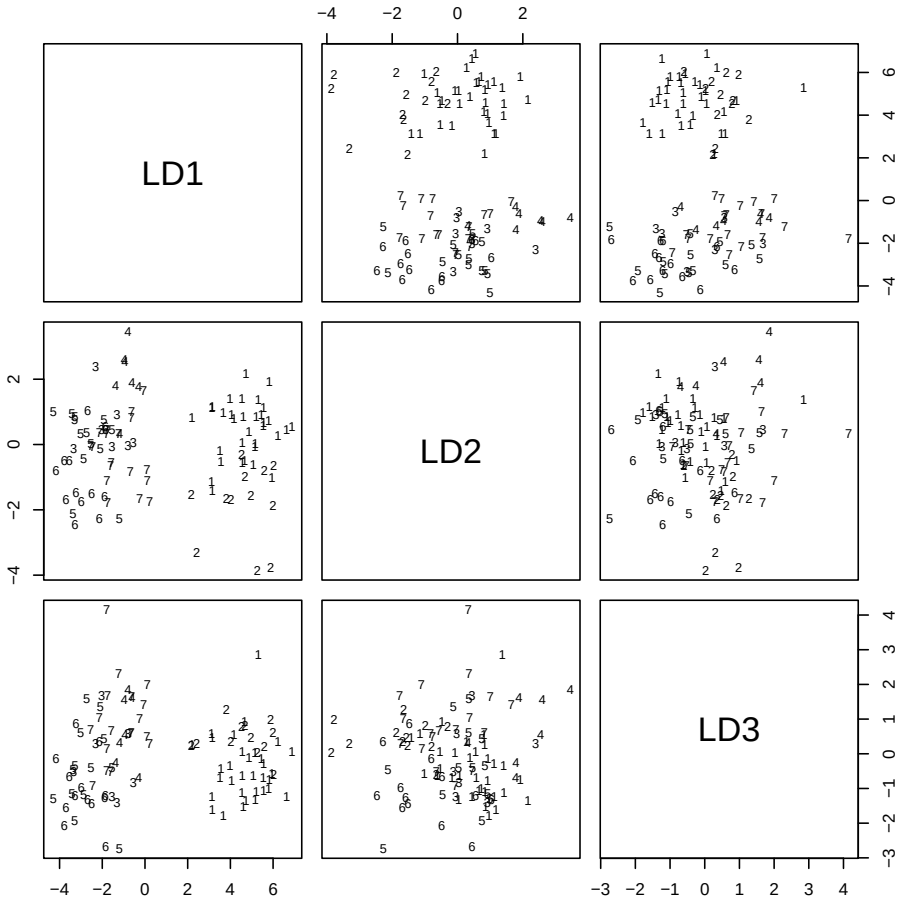

Figure 7.2: Scatterplot matrix of first three canonical variates.

Code is:

```
plot(possum.lda, dimen=3)
# Scatterplot matrix for scores on 1st 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.

Note that there may be interpretative advantages in taking logarithms of biological measurement data. The standard against which patterns of measurement are commonly compared is that of allometric growth, which implies a linear relationship between the logarithms of the measurements. Differences between different sites are then indicative of different patterns of allometric growth. The reader may wish to repeat the above analysis, but working with the logarithms of measurements.

7.4 Exercises

1. Using the data set `MASS::painters`, 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 colors or symbols to identify the different schools.
2. Using the columns of continuous or ordinal data in the `MASS::Cars93` dataset, 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 `MASS::lda()` to derive canonical discriminant scores, as in section 6.3.
4. Investigate discrimination between plagiotropic and orthotropic species in the data set `DAAG::leafshape`.

7.5 References and reading

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach. Cambridge University Press.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.

Venables and Ripley (2002) . Modern Applied Statistics with S. Springer, NY.

8 Multi-Level Models and 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 *nlme* package, handle models in which a repeated measures error structure is superimposed on a linear (`lme4()`) or non-linear (`nlme()`) model. The `lme()` function is broadly comparable to Proc Mixed in the widely used SAS statistical package. The function `lme()` has associated with it important abilities for diagnostic checking and other insight.

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 realization 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 realizations of a series that is typically quite short.

8.1 Multi-level models – examples

The Kiwifruit Shading Data, Again

Refer back to Section 3.8 for details of the data. The fixed effects are `block` and treatment (`shade`). The random effects are block (though making block a random effect is optional, for purposes of comparing treatments), `plot` within `block`, and units within each block/plot combination. Here is the analysis:

```
library(nlme)
kiwishade <- DAAG::kiwishade
kiwishade$plot <- factor(paste(kiwishade$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:    1.478623 3.490381

Fixed effects:  yield ~ shade
                 Value Std.Error DF  t-value p-value
(Intercept)  100.20250  1.761617 36 56.88098  0.0000
shadeAug2Dec   3.03083  1.867621  6  1.62283  0.1558
shadeDec2Feb -10.28167  1.867621  6 -5.50522  0.0015
shadeFeb2May  -7.42833  1.867621  6 -3.97743  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
-2.41538976 -0.59814252 -0.06899575  0.78046182  1.58909233

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 5190.560  <.0001
shade           3     6   22.211  0.0012
intervals(kiwishade.lme)
Approximate 95% confidence intervals

 Fixed effects:
                 lower       est.      upper
```

```
(Intercept)   96.629775 100.202500 103.775225
shadeAug2Dec  -1.539072   3.030833   7.600738
shadeDec2Feb -14.851572 -10.281667  -5.711762
shadeFeb2May -11.998238  -7.428333  -2.858428

 Random Effects:
  Level: block
                     lower     est.    upper
sd((Intercept)) 0.5475859 2.019373 7.446993
  Level: plot
                     lower     est.    upper
sd((Intercept)) 0.3700762 1.478623 5.907772

 Within-group standard error:
   lower     est.    upper
2.770652 3.490381 4.397072
```

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

	Variance component	Notes
block	$2.0192^2 = 4.076$	Three blocks
plot	$1.4792^2 = 2.186$	4 plots per block
residual (within group)	3.4902^2=12.180	4 vines (subplots) per plot

The above gives the information for an analysis of variance table. We have:

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

Now see where these same pieces of information appeared in the analysis of variance table of Section 3.8:

```
kiwishade.aov<-aov(yield~block+shade+Error(block:shade),data=kiwishade)
Warning in aov(yield ~ block + shade + Error(block:shade), data = kiwishade):
Error() model is singular
summary(kiwishade.aov)

Error: block:shade
          Df Sum Sq Mean Sq F value  Pr(>F)
block      2  172.3    86.2   4.118 0.07488
shade      3 1394.5   464.8  22.211 0.00119
Residuals  6  125.6    20.9

Error: Within
          Df Sum Sq Mean Sq F value Pr(>F)
Residuals 36  438.6   12.18
```

The Tinting of Car Windows

In Section 2.6 we encountered data from an experiment that aimed to model the effects of the tinting of car windows on visual performance . 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 2.6 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):

```
tinting <- DAAG::tinting
itstar.lme<-lme(log(it)~tint*target*agegp*sex,
  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 all these models have been fitted by maximum likelihood. This allows 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 appears adequate. As a preliminary to examining the first order interactions individually, we re-fit the model used for `it2.lme`, 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)             3.61907    0.1301 145  27.817 5.30e-60
tint.L                  0.16095    0.0442 145   3.638 3.81e-04
tint.Q                  0.02096    0.0452 145   0.464 6.44e-01
targethicon            -0.11807    0.0423 145  -2.789 5.99e-03
agegpolder              0.47121    0.2329  22   2.023 5.54e-02
sexm                    0.08213    0.2329  22   0.353 7.28e-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:agegpolder       0.13075    0.0492 145   2.658 8.74e-03
tint.Q:agegpolder       0.06972    0.0520 145   1.341 1.82e-01
tint.L:sexm            -0.09794    0.0492 145  -1.991 4.83e-02
```

```
tint.Q:sexm              0.00542    0.0520 145   0.104 9.17e-01
targethicon:agegpolder -0.13887    0.0584 145  -2.376 1.88e-02
targethicon:sexm         0.07785    0.0584 145   1.332 1.85e-01
agegpolder:sexm          0.33164    0.3261  22   1.017 3.20e-01
```

Because tint is an ordered factor, polynomial contrasts are used.

The Michelson Speed of Light Data

The `MASS::michelson` dataframe has columns `Speed`, `Run`, and `Expt`, for five experiments of 20 runs each. A plot of the data seems consistent with sequential dependence within runs, possibly with random variation between runs.

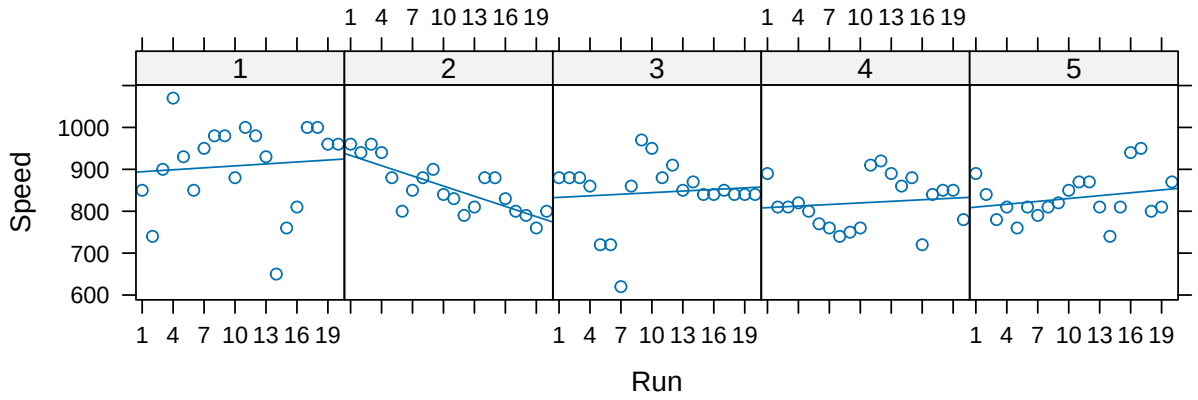

Figure 8.1: Plots show speed of light estimates against run number, for each of five experiments.

Code is:

```
michelson <- MASS::michelson
lattice::xyplot(Speed~Run|factor(Expt), layout=c(5,1),
                data=michelson, type=c('p','r'),
                scales=list(x=list(at=seq(from=1,to=19, by=3))))
```

We try a model that allows the estimates to vary linearly with Run (from 1 to 20), with the slope varying randomly between experiments. We assume an autoregressive dependence structure of order 1. We allow the variance to change from one experiment to another.

To test whether this level of model complexity is justified statistically, one needs to compare models with and without these effects, setting `method="ML"` in each case, and compare the likelihoods.

```
michelson <- MASS::michelson
library(nlme)
michelson$Run <- as.numeric(michelson$Run)   # Ensure Run is a variable
mich.lme1 <- lme(fixed = Speed ~ Run, data = michelson,
      random =  ~ Run| Expt, correlation = corAR1(form =  ~ 1 | Expt),
      weights = varIdent(form =  ~ 1 | Expt), method='ML')
mich.lme0 <- lme(fixed = Speed ~ Run, data = michelson,
      random =  ~ 1| Expt, correlation = corAR1(form =  ~ 1 | Expt),
      weights = varIdent(form =  ~ 1 | Expt), method='ML')
anova(mich.lme0, mich.lme1)
          Model df  AIC  BIC logLik   Test  L.Ratio p-value
mich.lme0     1  9 1121 1144   -551
mich.lme1     2 11 1125 1153   -551 1 vs 2 2.63e-08       1
```

The simpler model is preferred. Can it be simplified further?

8.2 References and reading

See the vignettes that accompany the *lme4* package.

J. Maindonald and Braun (2010) . Data Analysis and Graphics Using R — An Example-Based Approach. Cambridge University Press.

J. Maindonald, Braun, and Andrews (2024, forthcoming) . A Practical Guide to Data Analysis Using R. An Example-Based Approach. Cambridge University Press.

Pinheiro and Bates (2000) . Mixed effects models in S and S-PLUS. Springer.

References

Braun, W. John, and Duncan J. Murdoch. 2021. *A First Course in Statistical Programming with R*. 3rd ed. Cambridge University Press.

Chang, Winston. 2013. *R Graphics Cookbook*. 1st ed. O'Reilly.

Cunningham, Scott. 2021. *Causal Inference*. Yale University Press. https://mixtape.scunning.com/index.html.

Dalgaard, Peter. 2008. *Introductory Statistics with R*. 2nd ed. Springer.

Davies, Owen L et al. 1947. "Statistical Methods in Research and Production."

Faraway, Julian J. 2014. *Linear Models with R*. 2nd ed. Taylor & Francis Ltd.

———. 2016. *Extending the Linear Model with r*. 2nd ed. Taylor & Francis Inc.

Fox, John, and Sanford Weisberg. 2018. *An R Companion to Applied Regression*. Sage publications. http://socserv.socsci.mcmaster.ca/jfox/Books/Companion.

Hyndman, Rob J, and George Athanasopoulos. 2021. *Forecasting: Principles and Practice*. 3rd ed. OTexts. https://otexts.com/fpp2/.

Liaw, A., and M. Wiener. 2002. "Classification and Regression by randomForest." *R News* 2 (3): 18–22.

Maindonald, John. 1992. "Statistical Design, Analysis, and Presentation Issues." *New Zealand Journal of Agricultural Research* 35 (2): 121–41.

Maindonald, John H, and W. John Braun. 2022. *DAAG: Data Analysis and Graphics Data and Functions*. https://gitlab.com/daagur.

Maindonald, John, and John Braun. 2010. *Data Analysis and Graphics Using r: An Example-Based Approach*. Cambridge University Press.

Maindonald, John, W John Braun, and Jeffrey Andrews. 2024, forthcoming. *A Practical Guide to Data Analysis Using r. An Example-Based Approach.* Cambridge University Press. https://jhmaindonald.github.io/PGRcode.

Matloff, Norman. 2011. *The Art of r Programming*. No Starch Inc.

Muenchen, R. A. 2011. *R for SAS and SPSS Users*. 2nd ed. Springer. http://r4stats.com/books/r4sas-spss/.

Murrell, Paul. 2009. *Introduction to Data Technologies*. 1st ed. CRC Press.

———. 2011. *R Graphics*. 2nd ed. Chapman; Hall/CRC. http://www.e-reading.org.ua/bookreader.php/137370/Murrell_-_R_Graphics.pdf.

Pinheiro, J. C., and D. M. Bates. 2000. *Mixed Effects Models in S and S-PLUS*. Springer.

Ripley, Brian. 2023. *MASS: Support Functions and Datasets for Venables and Ripley's MASS*. http://www.stats.ox.ac.uk/pub/MASS4/.

Sarkar, Deepayan. 2023. *Lattice: Trellis Graphics for r*. https://lattice.r-forge.r-project.org/.

Tu, Y., and M. S. Gilthorpe. 2011. *Statistical Thinking in Epidemiology*. CRC Press.

Venables, W. N., and B. D. Ripley. 2002. *Modern Applied Statistics with S*. 4th ed. Springer.
Wickham, H. 2016. *R for Data Science*. O'Reilly.
Wood, S. N. 2017. *Generalized Additive Models. An Introduction with r*. 2nd ed. Chapman; Hall/CRC.