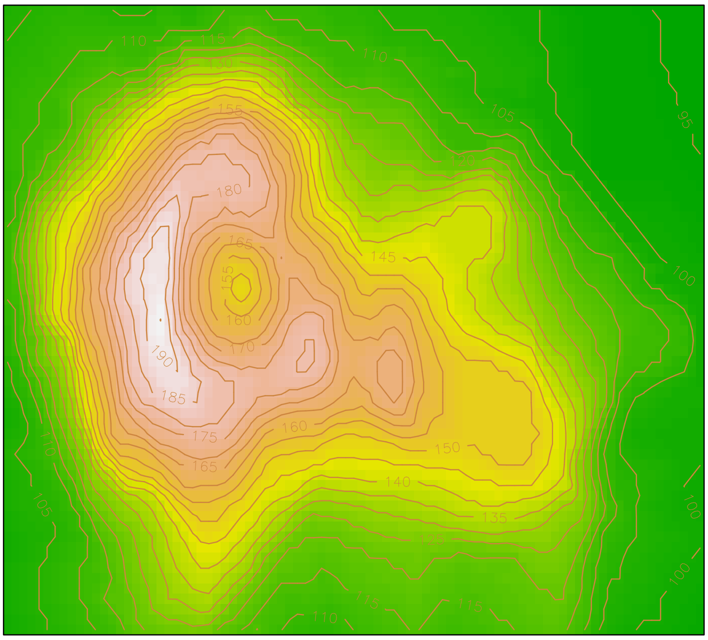
CSIRO Mathematical and Information Sciences

An Introduction to R: Software for Statistical Modelling & Computing



Course Materials and Exercises

Petra Kuhnert and Bill Venables

CSIRO Mathematical and Information Sciences
Cleveland, Australia

CSIRO

The information contained in this document is based on a number of technical, circumstantial or otherwise specified assumptions and parameters. The user must make its own analysis and assessment of the suitability of the information or material contained in or generated from the use of the document. To the extent permitted by law. CSIRO excludes all liability to any party for any expenses, losses, damages and costs arising directly or indirectly from using this document.

Contents

Elementary Introduction To R

Whirlwind Tour of R

The following examples provide a summary of analyses conducted in R. Results are not shown in this section and are left for the reader to verify.

Example 1: The Whiteside Insulation Data

Description
The Whiteside Insulation dataset is described in detail in Appendix I. The dataset consists of the weekly gas consumption and average external temperature records at a house in south-east England taken over two heating seasons:

- 26 weeks before cavity-wall insulation was installed
- 30 weeks after cavity-wall insulation was installed

The aim of the experiment was to examine the effect of insulation on gas consumption.

The dataset consists of 56 rows and three columns that contain information on:

- `Insul`: Insulation (before/after)
- `Temp`: Temperature in degrees Celsius
- `Gas`: Gas consumption in 1000s of cubic feet

Exploratory Analysis
Prior to modelling, an exploratory analysis of the data is often useful as it may highlight interesting features of the data that can be incorporated into a statistical analysis.

Figure 1 is the result of a call to the high level lattice function `xyplot`. The plot produces a scatterplot of gas consumption versus the average external temperature for each treatment type before insulation and similarly, after insulation).

Statistical Modelling

Based on the exploratory plots shown in Figure 1, it seems appropriate to fit straight lines through the points and examine whether these lines are different for varying treatment levels.

The analyses suggest that a straight line relationship is suitable for the data at each treatment level. In fact, nearly 95% of the variation is explained for the model using data prior to the insulation being installed, while approximately 81% of the variation was explained by the model incorporating data post insulation. Slopes for both models are very similar.

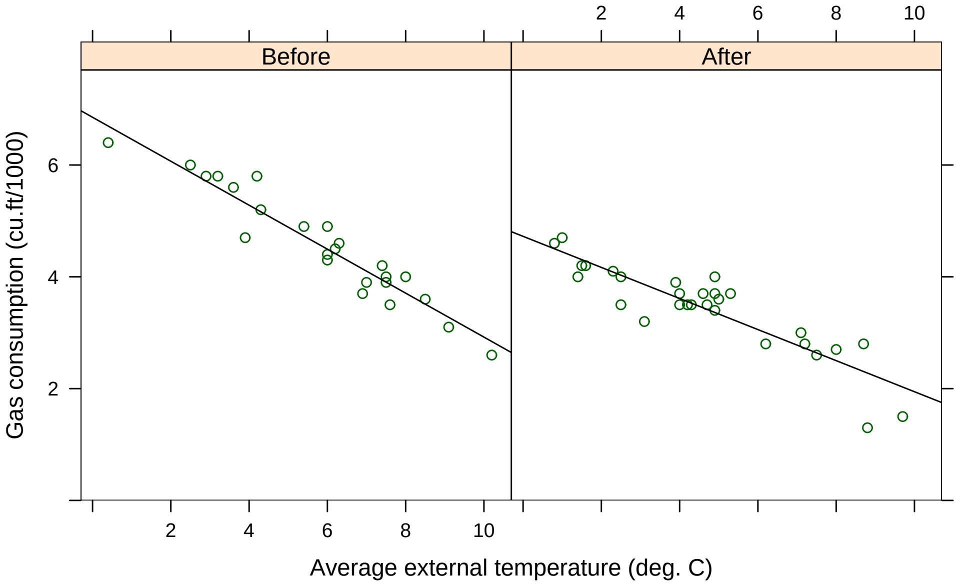


Figure 1: Scatterplot of gas consumption versus average external temperature for the two treatment levels: prior to insulation and post insulation. Least square lines are overlayed.

We can consider fitting a single model using both datasets. To check for curvature, we can introduce a polynomial term for the slopes. The results indicate that second order polynomial terms are not required in the model.

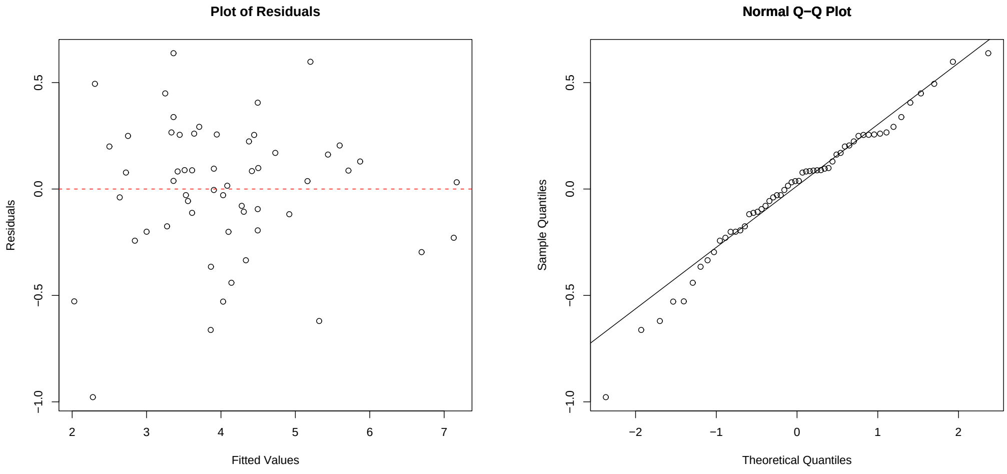
In the previous model fit where both lines were fitted in the one model, there is a suggestion that the lines may be parallel. To test this theory, we can fit a model with an overall treatment effect and an overall effect of temperature.

This model (not shown) suggests that there is a marked decrease in gas consumption after insulation was installed compared to having no insulation. The model also suggests that as the average external temperature increased, gas consumption decreased by a factor of 0.33.

Although the terms in the model are significant, the level of variation explained is lower than the model where both lines were fitted (∼91%). We can test whether separate regression lines fitted in the one model may be more appropriate using an analysis of variance.

It is useful to check the fit of the model using some diagnostic plots which examine the residuals with the assumptions of the model.

Figure 2 shows residual plots from the model where both lines were fitted. Residual plots indicate that the fit of the model is reasonable as both plots show no obvious departures from Normality.



(a) (b)

Figure 2: Residual plots of fitted model to the Whiteside Gas Consumption dataset. Figure (a) displays a plot of the residuals versus fitted values. Figure (b) presents a Normal quantile-quantile plot.

Example 2: Cars and Fuel Economy

The `Cars93` dataset is described in detail in Appendix I and by Lock (1993). It consists of a random list of passenger car makes and models (93 rows and 23 columns of data).

The information collected can be broken up into the type of data each variable represents. These are described below.

- **Factors:** AirBags, Cylinders, DriveTrain, Make, Man.trans.avail, Manufacturer, Model, Origin, Type
- **Numeric:**
  - **Integer:** MPG.city, MPG.highway, Luggage.room, Length, Horsepower, Passengers, Rev.per.mile, RPM, Turn.circle, Weight, Wheelbase, Width.
  - **Double:** EngineSize, Fuel.tank.capacity, Max.Price, Min.Price, Price, Rear.seat.room

We can produce a scatterplot of some of the data in this data frame. An interesting plot that can be produced is a scatterplot to investigate the relationship between gallons per 100 miles and weight. The plot is shown in Figure 3.

The scatterplot shown in Figure 3(a) suggests a possible linear relationship between gallons per 100 miles and weight. Note, the response is easier to model linearly if we use gallons per mile rather than miles per gallon. Before rushing into a formal analysis, we can investigate this assumption graphically, by reproducing the scatterplot with least square

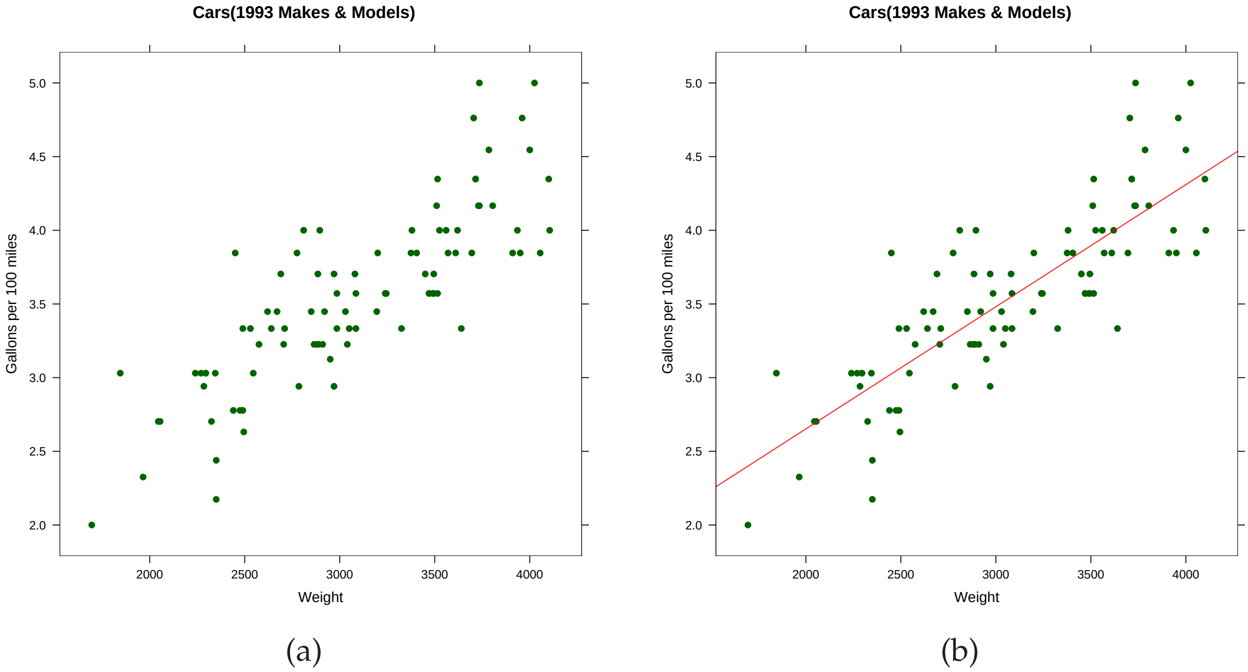


Figure 3: Scatterplot of gallons per 100 miles and weight of 1993 model cars (a) without and (b) with a least squares line overlayed

lines overlayed. Figure 3(b) shows a reasonable linear relationship between gallons per 100 miles and weight.

We can model this relationship more formally using least squares regression. The model (not shown) suggests a strong linear relationship between the weight of each vehicle and mileage. In fact, for every one pound increase, the gallons per 100 miles is expected to increase by a factor of 0.000829.

Examining the residuals from this fit (Figure 4) reveal a fairly reasonable fit with residuals that are behaving well with respect to Normal distribution assumptions.

We now extend the analysis to examine the relationship between mileage, weight and type of vehicle. Figure 5 shows a scatterplot of the data broken up by type of vehicle into six separate panels. This plot was produced using the following code: From Figure 5 we can see that as weight increases (for all types), the fuel consumption increases also. We also note some large variation between car types. To investigate these relationships further we fit a random effects model that has

- weight as the main predictor
- a term that incorporates type of vehicle
- a random intercept term associated with Manufacturer

The results from the model indicate that both type of car and weight are important pre-

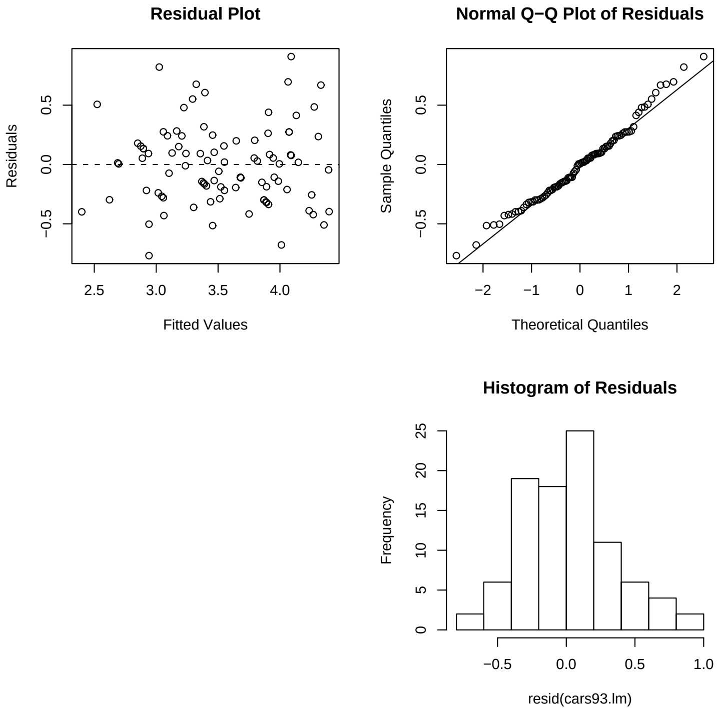


Figure 4: Residuals from fitted linear model to the Cars93 dataset.

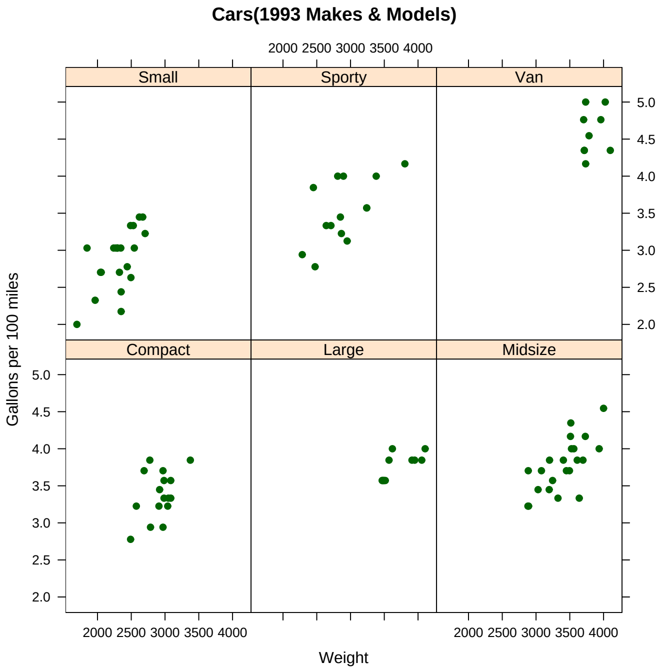


Figure 5: Panel plot of the Cars93 data showing scatterplots of gallons per 100 miles versus weight broken up by vehicle type.

dictors for gallons per 100 miles. The baseline type is `Compact`, (panel 4 in Figure 5). With respect to compact vehicles, vans and to some extent, sporty vehicles cost more to run. This can be seen in the scatterplots shown in Figure 5.

The random effect predictions are shown below

Honda	-0.223489568	Eagle	0.022092001
Geo	-0.202257478	Subaru	0.031985280
BMW	-0.160629835	Cadillac	0.036147200
Saturn	-0.103301920	Acura	0.047566558
Mazda	-0.098418520	Mercedes-Benz	0.051035604
Pontiac	-0.095375212	Chrysler	0.052572242
Suzuki	-0.088319971	Audi	0.054595702
Oldsmobile	-0.082034099	Mitsubishi	0.076673770
Chevrolet	-0.080333517	Hyundai	0.092313779
Toyota	-0.079260798	Ford	0.105985086
Lincoln	-0.059304804	Volkswagen	0.107058971
Buick	-0.057098894	Infiniti	0.107264569
Nissan	-0.052045909	Mercury	0.122912226
Chrylser	-0.021220000	Dodge	0.142520788
Plymouth	-0.007800792	Saab	0.157016239
Volvo	0.016483859	Lexus	0.186667440

The random effect predictions show some variation (but only slight). There appears to be two different types of vehicles:

- economical: good fuel consumption e.g. Honda, Geo, BMW, Saturn, and Mazda
- expensive: higher fuel consumption per mile e.g. Lexus, Saab, Dodge, Mercury

The estimate of the variance for the random effects terms is 0.0237. The errors relating to variation not accounted for in the model is almost negligible (0.062).

We may choose to investigate the prices of cars instead of the mileage for different makes and models. A simple way to view this relationship is through a boxplot of the data split by Manufacturer.

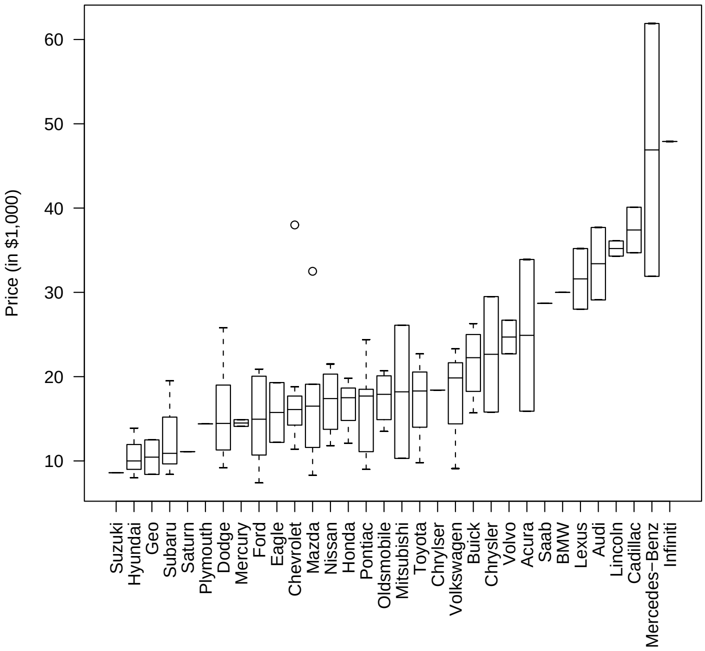


Figure 6: Distribution of car prices by manufacturer sorted by the median.

Example 3: Images of Volcanic Activity

The volcano dataset is a digitized map of a New Zealand volcano compiled by Ross Ihaka. The dataset is described in Appendix I and consists of a matrix with 87 rows and 61 columns. The topographic features of the dataset are plotted in Figure 7(a) using colours that range from white through to red. Figure 7(b) is the result of setting a user defined colour scheme.

Example 4: Coastline and Maps

Quite frequently you will need to produce a map for a project that you are working on. These types of activities are typically done in a GIS system. However, R offers some very useful functions for producing maps, given that you have the co-ordinates.

The following figure plots the main land and surrounding islands of Moreton Bay, in South East Queensland Australia. The geographical locations were obtained using *Coastline Extractor* (Signell, 2005), a web based tool for extracting coastlines.

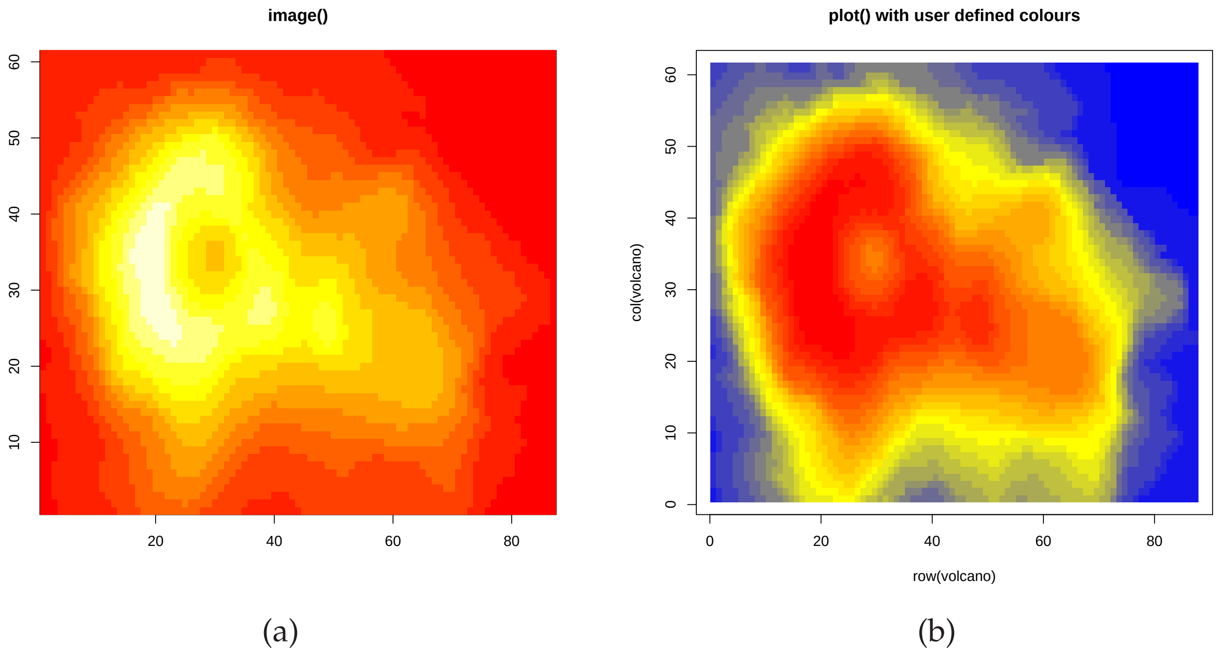


Figure 7: Image plots of volcanic activity on Mt Eden as produced by (a) `image` and (b) using user defined colours.

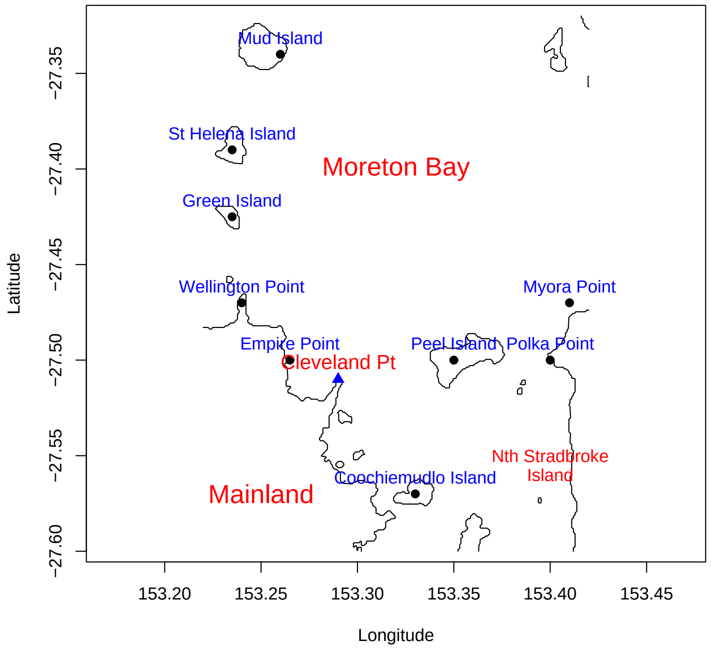


Figure 8: Map of Moreton Bay and surrounding islands.

R and the Tinn-R Editor

What is R?

Software Facilities
R provides a suite of software facilities for

- Reading and manipulating data
- Computation
- Conducting statistical analyses and
- Displaying the results

Implementation of the S Language
R is an implementation of the S language, a language for manipulating objects. For more details on the S language, readers are referred to Becker et al. (1988) and Venables & Ripley (2002).

R: A Programming Environment
R is a programming environment for data analysis and graphics. The *S Programming* book by Venables & Ripley (2000) provides a comprehensive overview of programming principles using S and R. The language was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics at the University of Auckland. Since its birth, a number of people have contributed to the package.

R: Platform for Development and Implementation of New Algorithms
R provides a platform for the development and implementation of new algorithms and technology transfer. R can achieve this in three ways

- functions that make use of existing algorithms within R
- functions that call on external programs written in either C or Fortran
- packaged up pieces of code that have specific classes attached to handle printing, summarising and the plotting data.

Obtaining R

Latest Copy
The latest copy of R (Version 2.1.0) can be downloaded from the CRAN (Comprehensive R Archive Network) website: *http://lib.stat.cmu.edu/R/CRAN/*.

R Packages
R packages can also be downloaded from this site or alternatively, they can be obtained

via R once the package has been installed. A list of R packages accompanied by a brief description can be found on the website itself.

R FAQ
In addition to these files, there is a manual and a list of frequently asked questions (FAQ) that range from basic syntax questions and help on obtaining R and downloading and installing packages to programming questions.

R Mailing Lists
Details of relevant mailing lists for R are available on http://www.R-project.org/mail.html

- **R-announce**: announcements of major developments of the package
- **R developments R-packages**: announcements of new R packages
- **R-help**: main discussion list
- **R-devel**: discussions about the future of R

R Manuals

There are a number of R manuals in `pdf` format provided on the CRAN website. These manuals consist of:

- **R Installation and Administration**: Comprehensive overview of how to install R and its packages under different operating systems.
- **An Introduction to R**: Provides an introduction to the language.
- **R Data Import/Export**: Describes import and export facilities.
- **Writing R Extensions**: Describes how you can create your own packages.
- **The R Reference Index**: Contains printable versions all of the R help files for standard and recommended packages

R Reference Material

There are a number of introductory texts and more advanced reference material that can help you with your journey through R. Below is a shortened list of key references. Those printed in red correspond to reference material that specifically focuses on S-Plus but has references to R or can be used as reference material for the R programming language.

- **Introductory texts**
  - Introductory Statistics with R by Peter Dalgaard

  - Linear Models with R by Julian Faraway
  - Data Analysis and Graphics using R: An Example Based Approach by John Maindonald and John Braun
  - Modern Applied Statistics with S-Plus by Bill Venables and Brian Ripley
  - Statistical Computing: An Introduction to Data Analysis using S-Plus by Michael Crawley

- **Programming**
  - S Programming by Bill Venables and Brian Ripley
- **Advanced Topics**
  - Mixed-Effects Models in S and S-Plus by Jose Pinheiro and Douglas Bates

How Does R Work and How Do I Work with it?

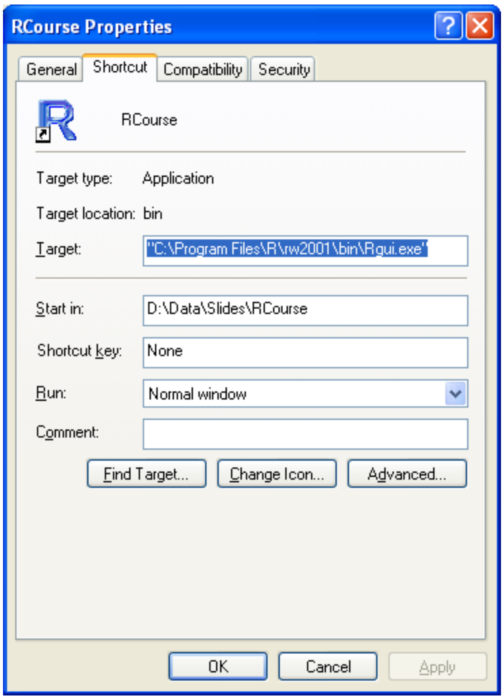
Dedicated Folder
R works best if you have a dedicated folder for each separate project. This is referred to as the *working folder*. The intension is to put all data files in the working folder or in sub-folders of it. This makes R sessions more manageable and it avoids objects getting messed up or mistakenly deleted.

Starting R
R can be started in the working folder by one of three methods:

1. Make an R shortcut which points to the folder (See Figure 9) and double-clicking on the R icon.
2. Double-click on the `.RData` file in the folder. This approach assumes that you have already created an R session.
3. Double-click any R shortcut and use `setwd(dir)`

In the windows version of R, the software can be started in either *multiple* or *single* windows format. Single windows format looks and behaves similar to a unix environment. Help and graphics screens are brought up as separate windows when they are called. In a multiple environment, graphics and help windows are viewed within the R session. This type of configuration can be set in the `Rgui Configuration Editor` by going to `Edit->GUI Preferences`. The 'look and feel' of your R session can also be changed using this screen.

R Commands
Any commands issued in R are recorded in an `.Rhistory` file. In R, commands may be



RCourse

Target type: Application

Target location: bin

Target: "C:\Program Files\R\rw2001\bin\Rgui.exe"

Start in: D:\Data\Slides\RCourse

Shortcut key: None

Run: Normal window

Comment:

Figure 9: The *Properties* dialog box for an R session.

recalled and reissued using the up- and down- arrow in an obvious way. Recalled commands may be edited in a *Windows familiar* fashion with a few extras. Flawed commands may be abandoned either by hitting the escape key (<`Esc`>) or (<`Home Ctrl-K`>) or (<`Home #`>).

Copying and pasting from a *script* file can be achieved by using the standard shortcut keys used by any Windows program: (<`Ctrl-C`>,<`Ctrl-V`>).

Copying and pasting from the history window is more suitable for recalling several commands at once or multiple-line commands.

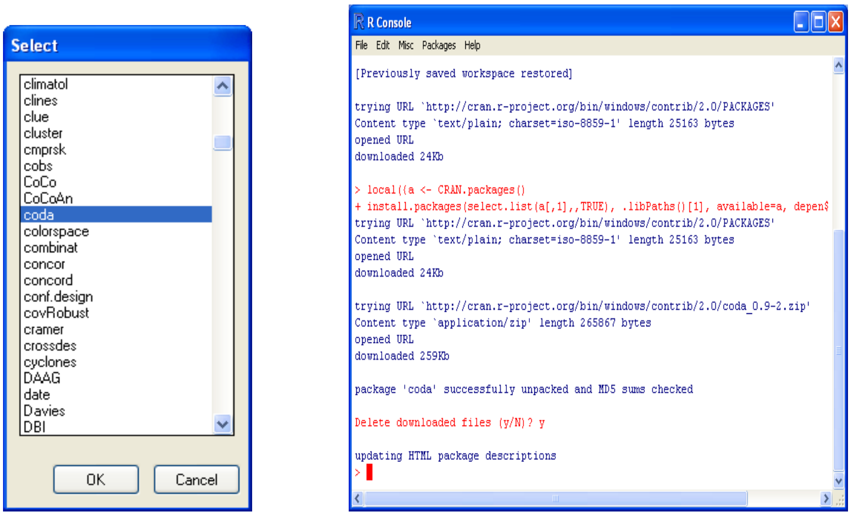
To ensure that a history of your commands are saved the savedhistory() function can be used explicitly. To have access to what you did during the last session, a history of previously used commands can be loaded using the `loadhistory()` function.

R Objects
R by default, creates its objects in memory and saves them in a single file called `.RData`. R objects are automatically saved in this file.

Quitting R
To quit from R either type `q()` in the R console or commands window or alternatively just kill the window. You will be prompted whether you want to save your session. Most times you will answer **yes** to this.

Installing and Loading R Packages

The installation and loading of R packages can be done within R by going up to the `Packages` menu and clicking on `Install package(s) from CRAN`. A dialog box will appear with a list of available packages to install. Select the package or packages required and then click on `OK`. Figure 10 shows a list of packages from the CRAN website. In this example, the CODA package is selected to be installed. Alternatively, the `install`.packages() function can be used from the command line. (Note, in the latest version of R (2.1.0), you may be prompted to select a download site.)



```
[Previously saved workspace restored]

trying URL `http://cran.r-project.org/bin/windows/contrib/2.0/PACKAGES'
Content type `text/plain; charset=iso-8859-1' length 25163 bytes
opened URL
downloaded 24Kb

> local({a <- CRAN.packages()
+ install.packages(select.list(a[,1],,TRUE), .libPaths()[1], available=a, depen$
trying URL `http://cran.r-project.org/bin/windows/contrib/2.0/PACKAGES'
Content type `text/plain; charset=iso-8859-1' length 25163 bytes
opened URL
downloaded 24Kb

trying URL `http://cran.r-project.org/bin/windows/contrib/2.0/coda_0.9-2.zip'
Content type `application/zip' length 265867 bytes
opened URL
downloaded 259Kb

package 'coda' successfully unpacked and MD5 sums checked

Delete downloaded files (y/N)? y

updating HTML package descriptions
>
```

(a) (b)

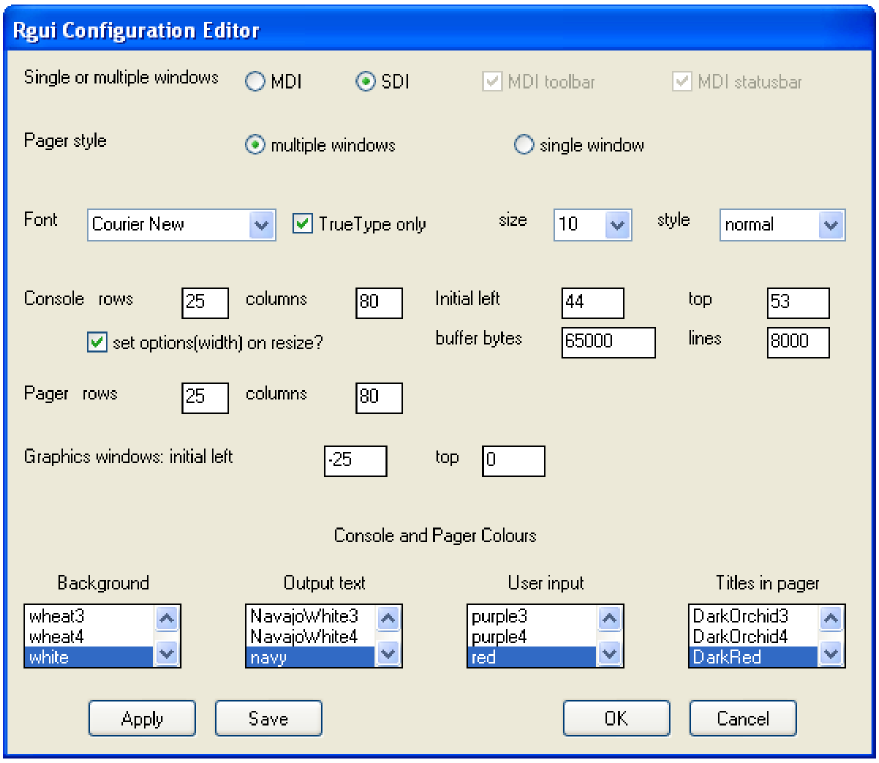
Figure 10: List of packages available from CRAN. The CODA package is selected from the *Select* dialog in Figure (a). Progress of the installation is summarised in the R console in Figure (b) and downloaded files are deleted.

Once installed, these packages can be loaded into R. Go to the `Packages` menu and select `Load package`. Select the package that is required and click on `OK`. These packages should be loaded into your current R session. Alternatively, the functions `library()` or `require()` can be used to load *installed* packages into R. The `require()` function is generally used inside functions as it provides a warning rather than an error (a feature of the `library()` function) when a package does not exist.

Updating R packages can be achieved either through the menu or by using the function `update.packages()` at the command line. If packages cannot be downloaded directly, the package should be saved as a zip file locally on your computer and then installed using the `install.packages()` function or using the options from the menu.

Customisation

Changes to the R console can be made through the `Edit` menu under `GUI preferences`. The dialog box shown in Figure 11 highlights the options available for changing how R *looks and feels*.



Rgui Configuration Editor

Single or multiple windows ○ MDI ● SDI ☑ MDI toolbar ☑ MDI statusbar

Pager style ● multiple windows ○ single window

Font Courier New ☑ TrueType only size 10 style normal

Console rows 25 columns 80 Initial left 44 top 53

☑ set options(width) on resize? buffer bytes 65000 lines 8000

Pager rows 25 columns 80

Graphics windows: initial left -25 top 0

Console and Pager Colours

Background	Output text	User input	Titles in pager
wheat3	NavajoWhite3	purple3	DarkOrchid3
wheat4	NavajoWhite4	purple4	DarkOrchid4
white	navy	red	DarkRed

Figure 11: GUI preferences for changing the look and feel of the R console.

If global actions are required, actions that need to be taken every time R is used on a particular machine may be set in a file `R_Home/etc/Rprofile.site`. Actions that happen automatically every time this working folder is used can be set by defining a `.First` function. For example,

```
> .First <- function() {
    require(MASS)
    require(lattice)
    options(length=99999)
    loadhistory()
  }
```

To implement any actions at the end of a session in this working folder a .Last function may be set up. For example,

```
.Last <- function()
savehistory("My.Rhistory")
```

The Rprofile.site file: An Example

An example of actions that you may want executed every time an R session begins is shown in the following script:

```
# When quitting using q(), save automatically without prompting
q <- function(save="yes",status = 0, runLast = TRUE)
       .Internal(quit(save, status, runLast))

# Things you might want to change
# options(width = 80,papersize = "a4",editor = "notepad")
# options(pager = "internal")

# to prefer Compiled HTML help
# options(chmhelp = TRUE)

# to prefer HTML help
# options(html = TRUE)

# to prefer Windows help
# options(winhelp = TRUE)

# Allows update.packages (for example) to proceed directly without
# prompting for the CRAN site.
options(show.signif.stars = FALSE,length = 999999)
options("CRAN = http://cran.au.r-project.org/")
options(repos = c(CRAN = getOption("CRAN"),
                  CRANextra = "http://www.stats.ox.ac.uk/pub/RWin"))

# Put working directory in the top border of the R console window
utils:::setWindowTitle(paste("-",getwd()))
```

What Editor can I use with R

Tinn-R is a free editor (distributed under the GNU Public License) that runs under Windows (9X/Me/2000/XP). Tinn stands for *Tinn is Not Notepad* and unlike notepad it allows syntax highlighting of R (in *.R, *.r, *.Q or *.q files). When an R session is open, Tinn-R includes an additional menu and toolbar and it allows the user to interact with R by submitting code in whole or in part.

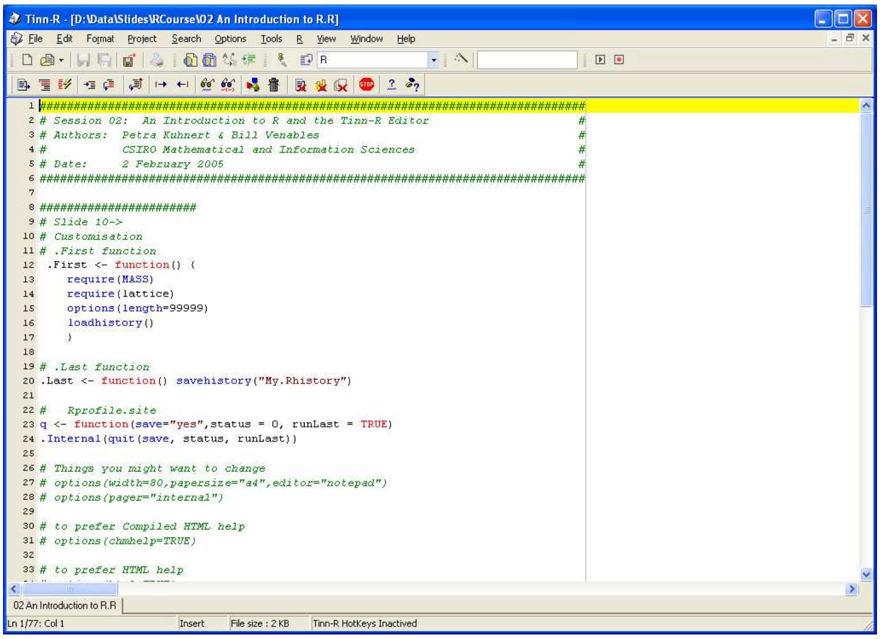
The software is available from: *http://www.sciviews.org/Tinn-R/*

The Tinn-R Editor: A Demonstration

The Tinn-R editor provides editing capabilities superior to that of the Windows notepad editor. A sample session is shown in Figure 12. The `File`, `Edit`, `Search`, `View`, `Window` and `Help` menus are standard for most Windows applications. However, Tinn-R offers a few extra features that make editing R scripts easier.

- The `Format` menu item helps with formatting and editing a file. In particular, it helps with *bracket* matching, a useful feature when writing programs.
- The `Project` menu allows you to set up a project containing more than one piece of code. This can be useful if you need to separate your code into components rather than placing each component in the one file.
- The `Options` menu allows you to change the look of the Tinn-R editor and how it deals with syntax highlighting.
- The `Tools` menu allows you to define macros and record sessions
- The `R` menu is useful for interacting with an R session when one is made available.

It is useful before writing a script for the first time within Tinn-R to edit the options. Figure 13 displays the list of application options available within the Tinn-R editor. A


Tinn-R - [D:\Data\Slides\RCourse\02 An Introduction to R.R]
File Edit Format Project Search Options Tools R View Window Help
R
02 An Introduction to R.R
Ln 1/77: Col 1
Insert
File size : 2 KB
Tinn-R HotKeys Inactived

```
################################################################################
# Session 02:  An Introduction to R and the Tinn-R Editor                       #
# Authors:  Petra Kuhnert & Bill Venables                                       #
#           CSIRO Mathematical and Information Sciences                         #
# Date:     2 February 2005                                                     #
################################################################################

#######################
# Slide 10->
# Customisation
# .First function
 .First <- function() {
    require(MASS)
    require(lattice)
    options(length=99999)
    loadhistory()
    }

# .Last function
.Last <- function() savehistory("My.Rhistory")

#   Rprofile.site
q <- function(save="yes",status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))

# Things you might want to change
# options(width=80,papersize="a4",editor="notepad")
# options(pager="internal")

# to prefer Compiled HTML help
# options(chmhelp=TRUE)

# to prefer HTML help
```

Figure 12: The Tinn-R Editor

useful option to select is *Active Line Highlighting*. This helps when sending scripts line by line.

Depending on the file being edited, Tinn-R offers syntax highlighting. See Figure 14 for an example. This is useful for writing scripts in R as it highlights reserved words and comments.

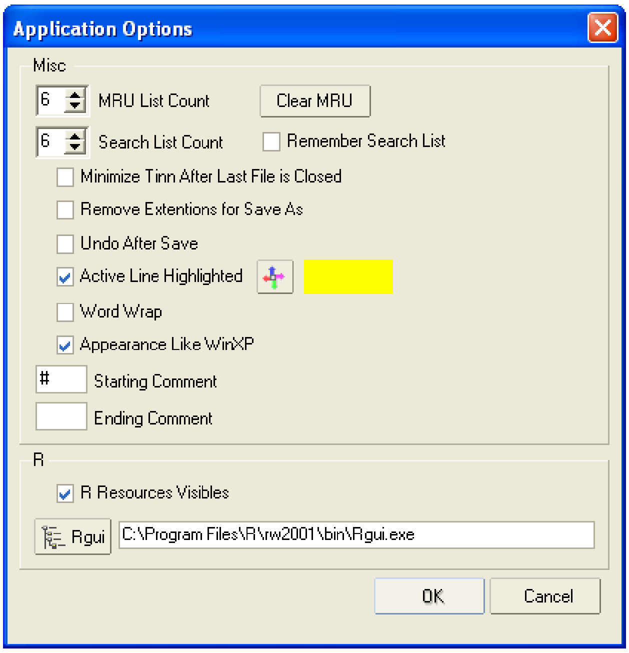
In Figure 14, a selection of the script is highlighted and then submitted to R from within the Tinn-R editor. Text can be submitted by simple highlighting, line-by-line or by sending the entire file.

Hotkeys may be set up in Tinn-R for the fast execution of commands. Figure 15 illustrates how a *hotkey* can be set up for use with R. The `Alt+S` key has been reserved for sending parts of highlighted scripts.

The R Language: Basic Syntax

It is important to learn some basic syntax of the R programming language before launching in to more sophisticated functions, graphics and modelling. Below is a compilation of some of the basic features of R that will get you going and help you to understand the R language.

R prompt
The default R prompt is the greater-than sign (>)



Application Options

Misc

6 MRU List Count

6 Search List Count

□ Remember Search List

□ Minimize Tinn After Last File is Closed

□ Remove Extentions for Save As

□ Undo After Save

☑ Active Line Highlighted

□ Word Wrap

☑ Appearance Like WinXP

Starting Comment

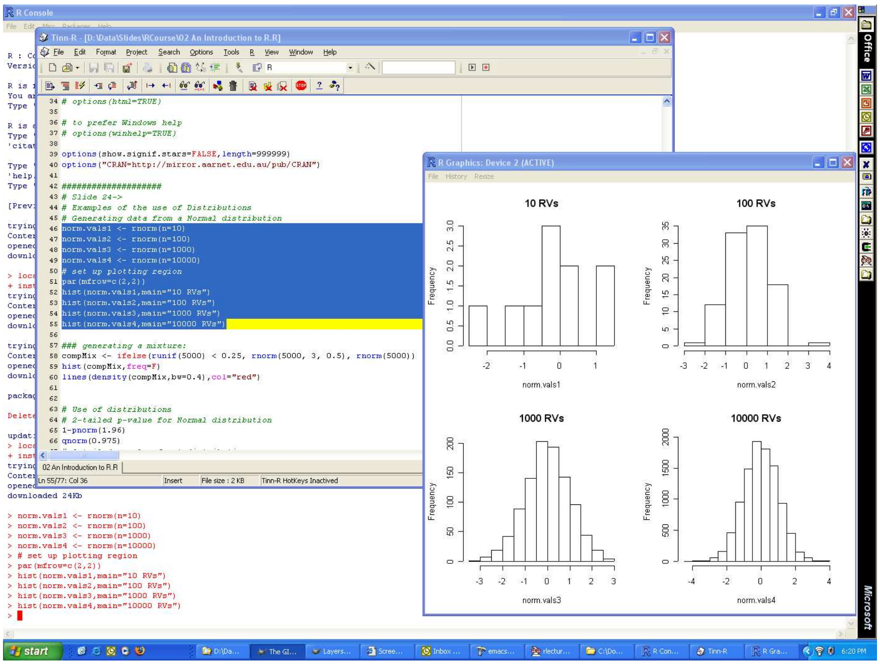
Ending Comment

R

☑ R Resources Visibles

Rgui C:\Program Files\R\rw2001\bin\Rgui.exe

Figure 13: Options within Tinn-R



Tinn-R - [D:\Data\Slides\RCourse\02 An Introduction to R.R]

```
# options(html=TRUE)

# to prefer Windows help
# options(winhelp=TRUE)

options(show.signif.stars=FALSE,length=999999)
options("CRAN=http://mirror.aarnet.edu.au/pub/CRAN")

####################
# Slide 24->
# Examples of the use of Distributions
# Generating data from a Normal distribution
norm.vals1 <- rnorm(n=10)
norm.vals2 <- rnorm(n=100)
norm.vals3 <- rnorm(n=1000)
norm.vals4 <- rnorm(n=10000)
# set up plotting region
par(mfrow=c(2,2))
hist(norm.vals1,main="10 RVs")
hist(norm.vals2,main="100 RVs")
hist(norm.vals3,main="1000 RVs")
hist(norm.vals4,main="10000 RVs")

### generating a mixture:
compMix <- ifelse(runif(5000) < 0.25, rnorm(5000, 3, 0.5), rnorm(5000))
hist(compMix,freq=F)
lines(density(compMix,bw=0.4),col="red")


# Use of distributions
# 2-tailed p-value for Normal distribution
1-pnorm(1.96)
qnorm(0.975)
```

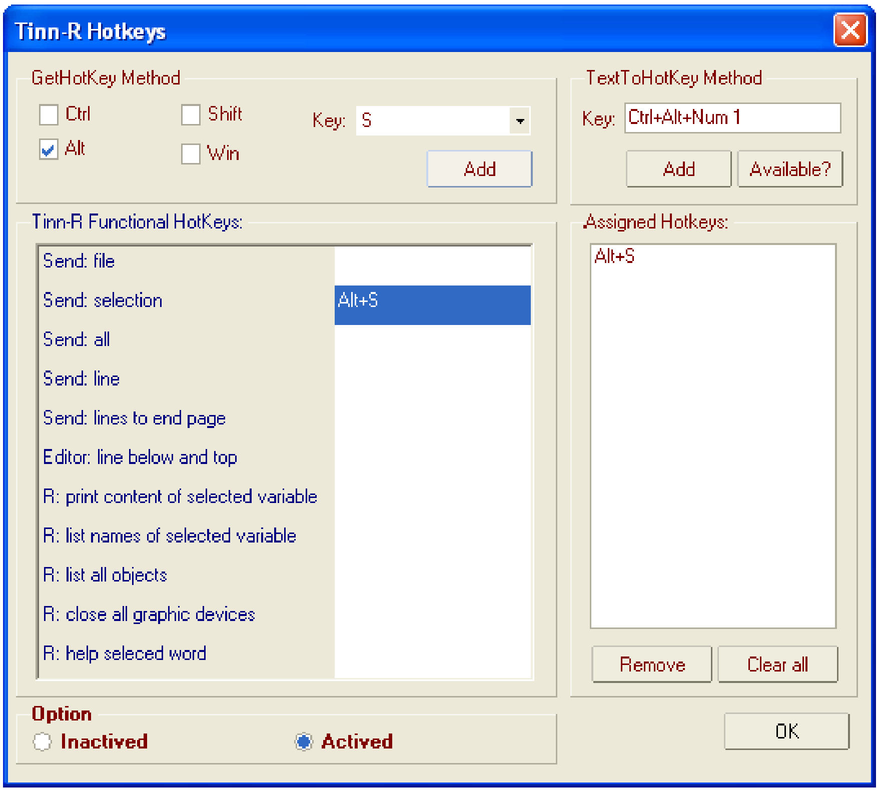
Ln 55/77: Col 36 Insert File size : 2 KB Tinn-R HotKeys Inactived

R Console

```
downloaded 24Kb

> norm.vals1 <- rnorm(n=10)
> norm.vals2 <- rnorm(n=100)
> norm.vals3 <- rnorm(n=1000)
> norm.vals4 <- rnorm(n=10000)
> # set up plotting region
> par(mfrow=c(2,2))
> hist(norm.vals1,main="10 RVs")
> hist(norm.vals2,main="100 RVs")
> hist(norm.vals3,main="1000 RVs")
> hist(norm.vals4,main="10000 RVs")
>
```

Figure 14: Example Tinn-R session. A selection of text is highlighted, sent to the R console and run to produce a series of plots.



Tinn-R Hotkeys

GetHotKey Method

□ Ctrl □ Shift

☑ Alt □ Win

Key: S

TextToHotKey Method

Key: Ctrl+Alt+Num 1

Tinn-R Functional HotKeys:

Send: file	
Send: selection	Alt+S
Send: all	
Send: line	
Send: lines to end page	
Editor: line below and top	
R: print content of selected variable	
R: list names of selected variable	
R: list all objects	
R: close all graphic devices	
R: help seleced word	

Assigned Hotkeys:

Alt+S

Option

○ Inactived ● Actived

Figure 15: Setting *hot keys* in Tinn-R.

```
> 2 * 4
[1] 8
```

Continuation prompt
If a line is not syntactically complete, a continuation prompt (+) appears

```
> 2 *
+  4
[1]  8
```

Assignment Operator
The assignment operator is the left arrow ($< -$) and assigns the value of the object on the right to the object on the left

```
> value <- 2 * 4
```

The contents of the object value can be viewed by typing value at the R prompt

```
> value
[1] 8
```

Last Expression
If you have forgotten to save your last expression, this can be retrieved through an internal object `.Last.value`

```
> 2 * 4
[1]  8
> value <- .Last.value
> value
[1]  8
```

Removing Objects
The functions `rm()` or `remove()` are used to remove objects from the working directory

```
> rm(value)
> value
Error: Object ’value’ not found
```

Legal R Names
Names for R objects can be any combination of letters, numbers and periods (.) but they must not start with a number. R is also case sensitive so

```
> value
[1] 8
```

is different from

```
> Value
Error: Object ’Value’ not found
```

Finding Objects
R looks for objects in a sequence of places known as the *search path*. The search path is a sequence of *environments* beginning with the *Global Environment*. You can inspect it at any time (and you should) by the `search()` function (or from the Misc menu). The `attach()` function allows copies of objects to be placed on the search path as individual components. The `detach()` function removes items from the search path.

Looking at the Search Path: An Example

```
> attach(Cars93)
> search()
 [1] ".GlobalEnv"       "Cars93"           "package:methods"
 [4] "package:graphics" "package:utils"    "package:RODBC"
 [7] "package:stats"    "package:MASS"     "Autoloads"
[10] "package:base"
> objects(2)
 [1] "AirBags"            "Cylinders"          "DriveTrain"
 [4] "EngineSize"         "Fuel.tank.capacity" "Horsepower"
....
[19] "Price"              "Rear.seat.room"     "Rev.per.mile"
[22] "RPM"                "Turn.circle"        "Type"
[25] "Weight"             "Wheelbase"          "Width"
> names(Cars93)
 [1] "Manufacturer"       "Model"              "Type"
 [4] "Min.Price"          "Price"              "Max.Price"
 [7] "MPG.city"           "MPG.highway"        "AirBags"
....
```

```
[22] "Turn.circle"        "Rear.seat.room"     "Luggage.room"
[25] "Weight"             "Origin"             "Make"
```

```
> find(Cars93)
[1] "package:MASS"
```

Assignments to Objects

Avoid using the names of built-in functions as object names. If you mistakenly assign an object or value to a built-in function and it is passed to another function you may get a warning but not always. . . things may go wrong.

R has a number of built-in functions. Some examples include `c`, `T`, `F`, `t`. An easy way to avoid assigning values/objects to built-in functions is to check the contents of the object you wish to use. This also stops you from overwriting the contents of a previously saved object.

```
> Value
# Object with no Error: Object "Value" not found
# value assigned

> value   # Object with a
[1] 8   # a value assigned

> T   # Built in R Value
[1]  TRUE

> t   # Built in R
function (x)   # function
UseMethod("t")
<environment:  namespace:base>
```

Spaces
R will ignore extra spaces between object names and operators

```
> value <-  2  * 4
[1]  8
```

Spaces cannot be placed between the $<$ and $-$ in the assignment operator

```
> value < -2 * 4
[1] FALSE
```

Be careful when placing spaces in character strings

```
> value <- "Hello World"
```

is different to

```
> value <- 'Hello  World'
```

Getting Help
To get help in R on a specific function or an object or alternatively an operator, one of the following commands can be issued:

```
> ?function
> help(function)
```

or click on the Help menu within R.

To get help on a specific topic, either one of the following will suffice

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

or click on the Help menu within R

The R Language: Data Types

There are four atomic data types in R.

- Numeric

  ```
  > value <- 605
  > value
  [1]  605
  ```

- Character

  ```
  > string <- "Hello World"
  > string
  [1]  "Hello World"
  ```

- Logical

  ```
  > 2 < 4
  [1]  TRUE
  ```

- Complex number

  ```
  > cn <- 2 + 3i
  > cn
  [1]  2+3i
  ```

The attribute of an object becomes important when manipulating objects. All objects have two attributes, the `mode` and their `length`.

The R function `mode` can be used to determine the mode of each object, while the function `length` will help to determine each object's length.

```
> mode(value)
[1]  "numeric"
> length(value)
[1]  1

> mode(string)
[1]  "character"
> length(string)
[1]  1

> mode(2<4)
[1]  "logical"

> mode(cn)
[1]  "complex"
> length(cn)
[1]  1
```

```
> mode(sin)
[1]  "function"
```

NULL objects are empty objects with no assigned mode. They have a length of zero.

```
> names(value)
[1]  NULL
```

The R Language: Missing, Indefinite and Infinite Values

In many practical examples, some of the data elements will not be known and will therefore be assigned a missing value. The code for missing values in R is `NA`. This indicates that the value or element of the object is unknown. Any operation on an `NA` results in an `NA`.

The `is.na()` function can be used to check for missing values in an object.

```
> value <- c(3,6,23,NA)
> is.na(value)
[1]  FALSE FALSE FALSE TRUE

> any(is.na(value))
[1]  TRUE

> na.omit(value)
[1] 3 6 23

> attr(,"na.action")
[1] 4

> attr(,"class")
[1] "omit"
```

Indefinite and Infinite values (`Inf`, `-Inf` and `NaN`) can also be tested using the `is.finite`, `is.infinite`, `is.nan` and `is.number` functions in a similar way as shown above.

These values come about usually from a division by zero or taking the log of zero.

```
> value1 <- 5/0
> value2 <- log(0)
> value3 <- 0/0
> cat("value1 = ",value1,"  value2 = ",value2,
    "  value3 = ",value3,"\n")
value1 =  Inf   value2 =  -Inf   value3 =  NaN
```

Arithmetic and Logical Operators
The last few sections used a variety of arithmetic and logical operators to evaluate expressions. A list of arithmetic and logical operators are shown in Tables 1 and 2 respectively.

Table 1: Arithmetic Operators

Operator	Description	Example

$+$	Addition	`> 2+5`
		`[1] 7`

$-$	Subtraction	`> 2-5`
		`[1] -3`

$\times$	Multiplication	`>2*5`
		`[1] 10`

/	Division	`> 2/5`
		`[1] 0.4`

$\wedge$	Exponentiation	`> 2∧5`
		`[1] 32`

%/%	Integer Divide	`> 5%/%2`
		`[1] 2`

%%	Modulo	`> 5%%2`
		`[1] 1`

Distributions and Simulation

There are a number of distributions available within R for simulating data, finding quantiles, probabilities and density functions. The complete list of distributions are displayed in Table 3. Other less common distributions, which are found in developed packages (not included with the original distribution) are also displayed in this table.

Table 2: Logical Operators

Operator	Description	Example
==	Equals	> value1 [1] 3 6 23 > value1==23 [1] FALSE FALSE TRUE
! =	Not Equals	> value1 != 23 [1] TRUE TRUE FALSE
<	Less Than	> value1 < 6 [1] TRUE FALSE FALSE
>	Greater Than	> value1 > 6 [1] FALSE FALSE TRUE
<=	Less Than or Equal To	> value1 <= 6 [1] TRUE TRUE FALSE
>=	Greater Than or Equal To	> value1 >= 6 [1] FALSE FALSE TRUE
&	Elementwise And	> value2 [1] 1 2 3 > value1==6 & value2 <= 2 [1] FALSE TRUE FALSE
|	Elementwise Or	> value1==6 | value2 <= 2 [1] TRUE TRUE FALSE
&&	Control And	> value1[1] <- NA > is.na(value1) && value2 == 1 [1] TRUE
|	Control Or	> is.na(value1) || value2 == 4 [1] TRUE
xor	Elementwise Exclusive Or	> xor(is.na(value1), value2 == 2) [1] TRUE TRUE FALSE
!	Logical Negation	> !is.na(value1) [1] FALSE TRUE TRUE

Table 3: Probability Distributions in R

R Function	Distribution	Parameters	Package
`beta`	Beta	shape1,shape2	stats
`binom`	Binomial	size,prob	stats
`cauchy`	Cauchy	location,scale	stats
`chisq`	(non-central) Chi-squared	df,ncp	stats
`dirichlet`	Dirichlet	alpha	MCMCpack
`exp`	Exponential	rate	stats
`f`	F	df1,df2	stats
`gamma`	Gamma	shape,rate	stats
`geom`	Geometric	prob	stats
`gev`	Generalized Extreme Value	xi,mu,sigma	evir
`gpd`	Generalized Pareto	xi,mu,beta	evir
`hyper`	Hypergeometric	m,n,k	stats
`invgamma`	Inverse Gamma	shape,rate	MCMCpack
`iwish`	Inverse Wishart	v,S	MCMCpack
`logis`	Logistic	location,scale	stats
`lnorm`	Log Normal	meanlog,sdlog	stats
`multinom`	Multinomial	size,prob	stats
`mvnorm`	Multivariate Normal	mean,sigma	mvtnorm
`mvt`	Multivariate-t	sigma,df	mvtnorm
`nbinom`	Negative Binomial	size,prob	stats
`norm`	Normal	mean,sd	stats
`pois`	Poisson	lambda	stats
`signrank`	Wilcoxon Signed Rank Statistic	n	stats
`t`	Student-t	df	stats
`unif`	Uniform	min,max	stats
`weibull`	Weibull	shape,scale	stats
`wilcox`	Wilcoxon Rank Sum Statistic	m,n	stats
`wish`	Wishart	v,S	MCMCpack

In R, each distribution has a name prefixed by a letter indicating whether a probability, quantile, density function or random value is required. The prefixes available are shown in more detail below:

- p: probabilities (distribution functions)
- q: quantiles (percentage points)
- d: density functions (probability for discrete RVs)
- r: random (or simulated) values

The following example illustrates how we can simulate data from a Normal distribution using the rnorm function.

```
> norm.vals1 <- rnorm(n=10)
> norm.vals2 <- rnorm(n=100)
> norm.vals3 <- rnorm(n=1000)
> norm.vals4 <- rnorm(n=10000)
```

The first object, norm.vals1 generates 10 random values from a Normal distribution with a default mean of 0 and default standard deviation of 1. If values were required from a Normal distribution with a different mean and/or standard deviation then these arguments would need to be explicitly specified. The second, third and fourth objects generate random values from the same distribution with the same mean and standard deviations but with varying sample sizes.

The result of the simulated data is shown graphically in Figure 16 using the following R code:

```
# set up plotting region
> par(mfrow=c(2,2))
# produce plots
> hist(norm.vals1,main="10 RVs")
> hist(norm.vals2,main="100 RVs")
> hist(norm.vals3,main="1000 RVs")
> hist(norm.vals4,main="10000 RVs")
```

As the sample sizes increase the shape of the distribution looks more like a Normal distribution. It is difficult to tell if the object norm.vals1 has been generated from a Normal distribution with a mean of zero and a standard deviation of zero. This can be confirmed by looking at summary statistics from this object as the mean and standard deviation are not close to 0 or 1 respectively.

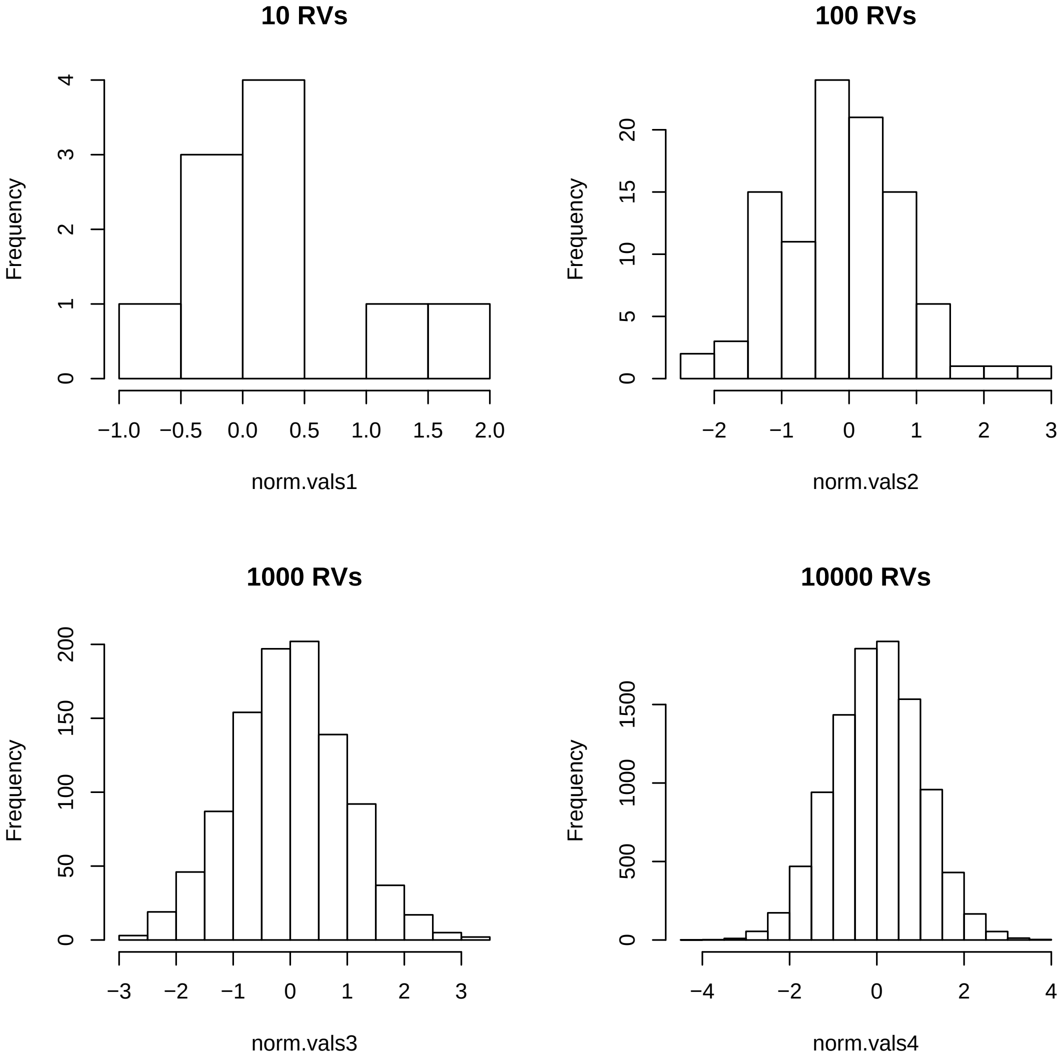


Figure 16: Histograms of simulated data from Normal distributions with a mean of 0 and standard deviation of 1.

```
> c(mean(norm.vals1),sd(norm.vals1))
[1]  0.2461831  0.7978427
```

The interpretation of the Central Limit theorem is appropriate here for this example. The theorem states that as the sample size n taken from a population with a mean μ and variance σ^2 approaches infinity, then the statistics from the sampled distribution will converge to the theoretical distribution of interest.

To illustrate this, if we calculate the mean and standard deviation of `norm.vals4`, the object where we generated 10,000 random values from a $\mathrm{N}(0, 1)$ distribution, we find that the summary statistics are close to the actual values.

```
> c(mean(norm.vals4),sd(norm.vals4))
[1] 0.004500385 1.013574485
```

For larger simulations, these are closer again,

```
> norm.vals5 <- rnorm(n=1000000)
> c(mean(norm.vals5),sd(norm.vals5))
[1] 0.0004690608 0.9994011738
```

We can also overlay a density on top of a histogram summarising the data. This can be useful to display the features in the histogram and to identify intersection points where components in the mixture distribution meet. We illustrate this through the generation of a mixture of two Normal distributions using the following piece of R code. Figure 17 displays the two-component mixture with the density overlayed.

```
# Generating a two component mixture
> compMix <- ifelse(runif(5000) < 0.25,rnorm(5000,3,0.5),rnorm(5000))
# Plotting
> hist(comp,freq=F)
> lines(density(comp,bw=0.4),col="red")
```

R can also be used to evaluate probabilities or quantiles from distributions. This is a useful mechanism for determining p-values instead of searching through statistical tables and they can be easily achieved using the `p`(dist) and `q`(dist) functions. Some examples are shown below.

```
# 2-tailed p-value for Normal distribution
> 1-pnorm(1.96)
[1] 0.0249979
```

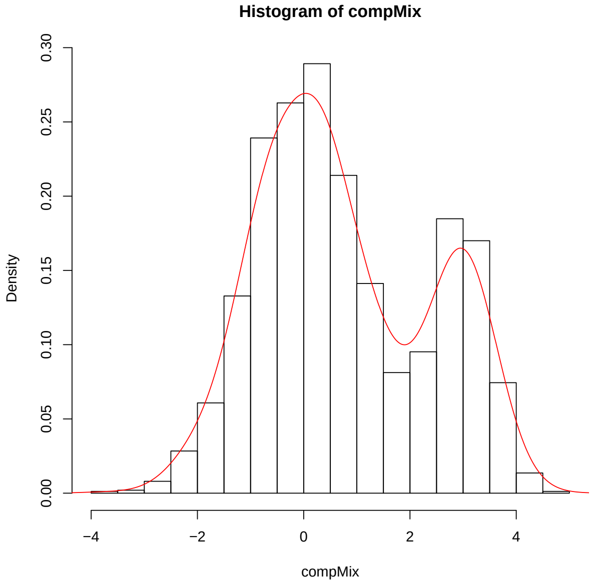


Figure 17: Histograms of two-component mixture model generated from Normal distributions. The density is overlayed in red.

```
> qnorm(0.975) # quantile
[1] 1.959964

# 2-tailed p-value for t distribution
> 2*pt(-2.43,df=13)
[1] 0.0303309
> qt(0.025,df=13)
[1] -2.160369  # quantile

#p-value from a chi-squared distribution with 1 degree of freedom
> 1-pchisq(5.1,1)
[1] 0.02392584
> qchisq(0.975,1)
[1] 5.023886   # quantile
```

R Objects

Data Objects in R

The four most frequently used types of data objects in R are vectors, matrices, data frames and lists.

A **vector** represents a set of elements of the same mode whether they are logical, numeric (integer or double), complex, character or lists.

A **matrix** is a set of elements appearing in rows and columns where the elements are of the same mode whether they are logical, numeric (integer or double), complex or character.

A **data frame** is similar to a matrix object but the columns can be of different modes.

A **list** is a generalisation of a vector and represents a collection of data objects.

Creating Vectors

`c` Function

The simplest way to create a vector is through the concatenation function, `c`. This function binds elements together, whether they are of character form, numeric or logical. Some examples of the use of the concatenation operator are shown in the following script.

```
> value.num <- c(3,4,2,6,20)

> value.char <- c("koala","kangaroo","echidna")

> value.logical.1 <- c(F,F,T,T)

# or

> value.logical.2 <- c(FALSE,FALSE,TRUE,TRUE)
```

The latter two examples require some explanation. For logical vectors, `TRUE` and `FALSE` are logical values and `T` and `F` are variables with those values. This is the opposite for S-PLUS. Although they have a different structure, logical vectors can be created using either value.

`rep` and `seq` Functions

The `rep` function replicates elements of vectors. For example,

```
> value <- rep(5,6)

> value

[1]  5  5  5  5  5  5
```

replicates the number 5, six times to create a vector called value, the contents of which are displayed.

The `seq` function creates a regular sequence of values to form a vector. The following script shows some simple examples of creating vectors using this function.

```
> seq(from=2,to=10,by=2)
[1]  2  4  6  8  10
> seq(from=2,to=10,length=5)
[1]  2  4  6  8  10
> 1:5
[1]  1  2  3  4   5
> seq(along=value)
[1]  1  2  3  4  5  6
```

`c, rep` and `seq` Functions

As well as using each of these functions individually to create a vector, the functions can be used in combination. For example,

```
> value <- c(1,3,4,rep(3,4),seq(from=1,to=6,by=2))
> value
[1]  1 3 4 3 3 3 3 1 3 5
```

uses the `rep` and `seq` functions inside the concatenation function to create the vector `value`.

It is important to remember that elements of a vector are expected to be of the same mode. So an expression

```
> c(1:3,"a","b","c")
```

will produce an error message.

`scan` Function

The `scan` function is used to enter in data at the terminal. This is useful for small datasets but tiresome for entering in large datasets. A more comprehensive summary of how data is read from files will be discussed in the session on 'importing and exporting'. An example of reading data in from the terminal is shown below.

```
> value <- scan()
1:  3  4  2  6  20
```

```
6:
> value
[1]  3  4  2  6  20
```

Basic Computation with Numerical Vectors
Computation with vectors is achieved using an element-by-element operation. This is useful when writing code because it avoids ’for loops’. However, care must be taken when doing arithmetic with vectors, especially when one vector is shorter than another. In the latter circumstance, short vectors are recycled. This could lead to problems if ’recycling’ was not meant to happen. An example is shown below.

```
> x <- runif(10)
> x
[1] 0.3565455 0.8021543 0.6338499 0.9511269
[5] 0.9741948 0.1371202 0.2457823 0.7773790
[9] 0.2524180 0.5636271
> y < 2*x + 1 # recycling short vectors
> y
[1] 1.713091 2.604309 2.267700 2.902254 2.948390
[6] 1.274240 1.491565 2.554758 1.504836 2.127254
```

Some functions take vectors of values and produce results of the same length. Table 4 lists a number of functions that behave this way.

Other functions return a single value when applied to a vector. Some of these functions are summarised in Table 5.

The following script makes use of some of this functionality.

```
> z <- (x-mean(x))/sd(x) # see also ’scale’
> z
[1] -0.69326707  0.75794573  0.20982940  1.24310440
[5]  1.31822981 -1.40786896 -1.05398941  0.67726018
[9] -1.03237897 -0.01886511
> mean(z)
[1] -1.488393e-16
> sd(z)
```

Table 4: Functions that produce results of the same length.

Function	Description
cos, sin, tan	Cosine, Sine, Tangent
acos, asin, atan	Inverse functions
cosh, sinh, tanh	Hyperbolic functions
acosh, asinh, atanh	Inverse hyperbolic functions
log	Logarithm (any base, default is natural logarithm)
log10	Logarithm (base 10)
exp	Exponential (e raised to a power)
round	Rounding
abs	Absolute value
ceiling, floor, trunc	Truncating to integer values
gamma	Gamma function
lgamma	Log of gamma function
sqrt	Square root

Table 5: Functions that produce a single result.

Function	Description
sum	Sum elements of a vector
mean	arithmetic mean
max, min	Maximum and minimum
prod	Product of elements of a vector
sd	standard deviation
var	variance
median	50th percentile

```
[1]  1
```

Laboratory Exercise: Try the first three examples from *Lab 2*

Creating Matrices

`dim` and `matrix` functions

The `dim` function can be used to convert a vector to a matrix

```
> value <- rnorm(6)
> dim(value) <- c(2,3)
> value
           [,1]       [,2]       [,3]
[1,]  0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805  1.8176161
```

This piece of script will fill the columns of the matrix. To convert back to a vector we simply use the dim function again.

```
> dim(value) <- NULL
```

Alternatively we can use the matrix function to convert a vector to a matrix

```
> matrix(value,2,3)
          [,1]       [,2]       [,3]
[1,] 0.7093460 -0.8643547 -0.1093764
[2,] -0.3461981 -1.7348805 1.8176161
```

If we want to fill by rows instead then we can use the following script

```
> matrix(value,2,3,byrow=T)
         [,1]       [,2]       [,3]
[1,] 0.709346 -0.3461981 -0.8643547
[2,] -1.734881 -0.1093764 1.8176161
```

`rbind` and `cbind` Functions

To bind a row onto an already existing matrix, the rbind function can be used

```
> value <- matrix(rnorm(6),2,3,byrow=T)
> value2 <- rbind(value,c(1,1,2))
> value2
           [,1]       [,2]      [,3]
[1,]  0.5037181  0.2142138 0.3245778
[2,] -0.3206511 -0.4632307 0.2654400
[3,]  1.0000000  1.0000000 2.0000000
```

To bind a column onto an already existing matrix, the `cbind` function can be used

```
> value3 <- cbind(value2,c(1,1,2))
           [,1]       [,2]      [,3] [,4]
[1,]  0.5037181  0.2142138 0.3245778    1
[2,] -0.3206511 -0.4632307 0.2654400    1
[3,]  1.0000000  1.0000000 2.0000000    2
```

`data.frame` Function

The function `data.frame` converts a matrix or collection of vectors into a data frame

```
> value3 <- data.frame(value3)
> value3
          X1         X2        X3 X4
1  0.5037181  0.2142138 0.3245778  1
2 -0.3206511 -0.4632307 0.2654400  1
3  1.0000000  1.0000000 2.0000000  2
```

Another example joins two columns of data together.

```
> value4 <- data.frame(rnorm(3),runif(3))
> value4
    rnorm.3.  runif.3.
1 -0.6786953 0.8105632
2 -1.4916136 0.6675202
3  0.4686428 0.6593426
```

Row and column names are already assigned to a data frame but they may be changed using the `names` and `row.names` functions. To view the row and column names of a data frame:

```
> names(value3)
[1] "X1" "X2" "X3" "X4"
> row.names(value3)
[1] "1" "2" "3"
```

Alternative labels can be assigned by doing the following

```
> names(value3) <- c("C1","C2","C3","C4")
> row.names(value3) <- c("R1","R2","R3")
```

Names can also be specified within the `data.frame` function itself.

```
> data.frame(C1=rnorm(3),C2=runif(3),row.names=c("R1","R2","R3")
           C1        C2
R1 -0.2177390 0.8652764
R2  0.4142899 0.2224165
R3  1.8229383 0.5382999
```

Manipulating Data: An Example

The iris dataset (`iris3`) is a three dimensional dataset described in Appendix I. One dimension is represented for each species: Setosa, Versicolor and Virginica. Each species has the sepal lengths and widths, and petal lengths and widths recorded.

To make this dataset more manageable, we can convert the three-dimensional array into a d-dimensional data frame.

To begin with, we examine the names of the three-dimensional array.

```
> dimnames(iris3)
[[1]]
NULL


[[2]]
```

```
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."


[[3]]
[1] "Setosa"     "Versicolor" "Virginica"
```

We see that the first dimension has not been given any names. This dimension corresponds to the row names of the dataset for each species. The second dimension corresponds to the explanatory variables collected for each species. The third dimension corresponds to the species.

Before coercing this three dimensional array into a two dimensional data frame, we first store the species name into a vector.

```
> Snames <- dimnames(iris3)[[3]]
```

We now convert the three dimensional array into a 150× 3 matrix and coerce the matrix into a data frame.

```
> iris.df <- rbind(iris3[,,1],iris3[,,2],iris3[,,3])
> iris.df <- as.data.frame(iris.df)
```

Now we check the column names of the data frame.

```
> names(iris.df)
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
```

Using the `Snames` vector, we create a species factor and bind it to the columns of `iris.df`.

```
> iris.df$Species <- factor(rep(Snames,rep(50,3)))
```

To check that we have created the data frame correctly, we print out the first five rows of the data frame.

```
> iris.df[1:5,]
  Sepal L. Sepal W. Petal L. Petal W. Species
1      5.1      3.5      1.4      0.2  Setosa
2      4.9      3.0      1.4      0.2  Setosa
3      4.7      3.2      1.3      0.2  Setosa
4      4.6      3.1      1.5      0.2  Setosa
5      5.0      3.6      1.4      0.2  Setosa
```

A pairwise plot of the data (Figure 18) can be produced using the `pairs` function in the following way.

```
> pairs(iris.df[1:4],main = "Anderson’s Iris Data",
    pch = 21,bg = c("red","green3","blue")[unclass(iris$Species)])
```

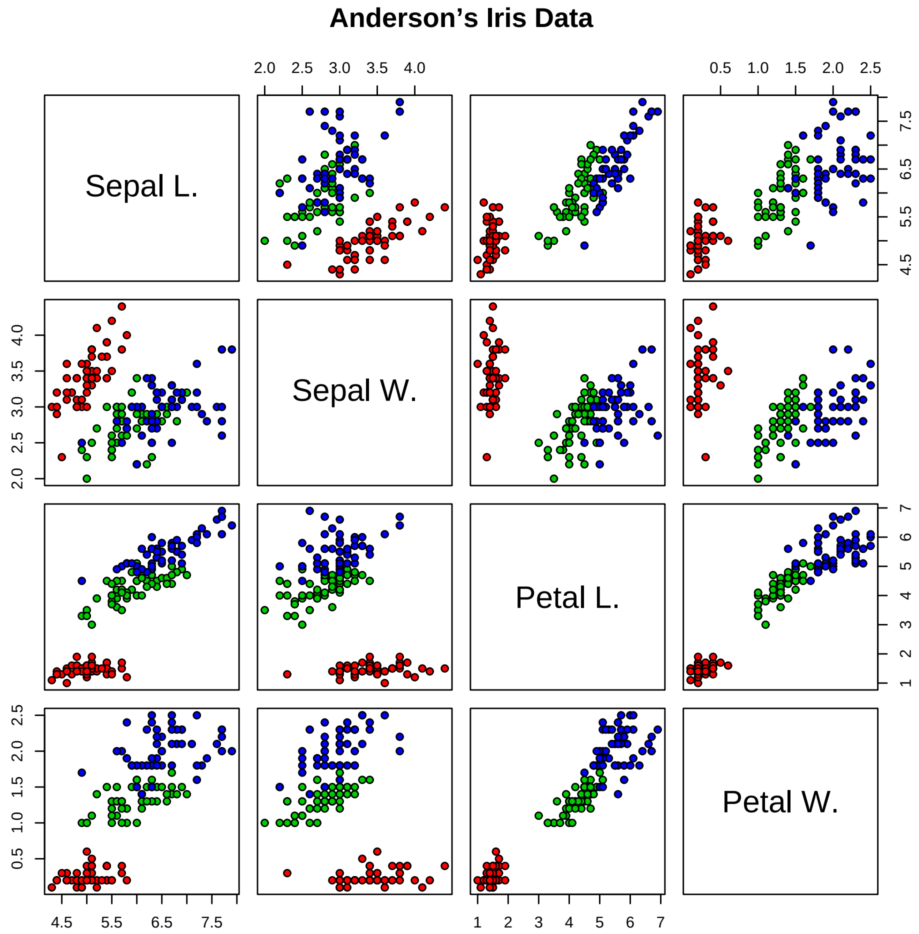


Figure 18: Pairwise plot of the iris data frame

Accessing Elements of a Vector or Matrix

Accessing elements is achieved through a process called *indexing*. Indexing may be done by

- a vector of positive integers: to indicate inclusion

- a vector of negative integers: to indicate exclusion
- a vector of logical values: to indicate which are in and which are out
- a vector of names: if the object has a names attribute

For the latter, if a zero index occurs on the right, no element is selected. If a zero index occurs on the left, no assignment is made. An empty index position stands for *the lot*!

Indexing Vectors

The first example involves producing a random sample of values between one and five, twenty times and determining which elements are equal to 1.

```
> x <- sample(1:5, 20, rep=T)
> x
[1] 3 4 1 1 2 1 4 2 1 1 5 3 1 1 1 2 4 5 5 3
> x == 1
 [1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE
[10]  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[19] FALSE FALSE
> ones <- (x == 1) # parentheses unnecessary
```

We now want to replace the ones appearing in the sample with zeros and store the values greater than 1 into an object called `y`.

```
> x[ones] <- 0
> x
 [1] 3 4 0 0 2 0 4 2 0 0 5 3 0 0 0 2 4 5 5 3
> others <- (x > 1) # parentheses unnecessary
> y <- x[others]
> y
[1] 3 4 2 4 2 5 3 2 4 5 5 3
```

The following command queries the `x` vector and reports the position of each element that is greater than 1.

```
> which(x > 1)
[1]  1  2  5  7  8 11 12 16 17 18 19 20
```

Indexing Data Frames

Data frames can be indexed by either row or column using a specific name (that corresponds to either the row or column) or a number. Some examples of indexing are shown below.

Indexing by column:

```
> value3
           C1         C2        C3 C4
R1  0.5037181  0.2142138 0.3245778  1
R2 -0.3206511 -0.4632307 0.2654400  1
R3  1.0000000  1.0000000 2.0000000  2
> value3[, "C1"] <- 0
> value3
   C1         C2        C3 C4
R1  0  0.2142138 0.3245778  1
R2  0 -0.4632307 0.2654400  1
R3  0  1.0000000 2.0000000  2
```

Indexing by row:

```
> value3["R1", ] <- 0
> value3
   C1         C2        C3 C4
R1  0  0.0000000 0.0000000  0
R2  0 -0.4632307 0.2654400  1
R3  0  1.0000000 2.0000000  2
> value3[] <- 1:12
> value3
  C1 C2 C3  C4
R1 1  4  7  10
R2 2  5  8  11
R3 3  6  9  12
```

To access the first two rows of the matrix/data frame:

```
> value3[1:2,]
   C1 C2 C3 C4
R1  1  4  7 10
R2  2  5  8 11
```

To access the first two columns of the matrix/data frame:

```
> value3[,1:2]
   C1 C2
R1  1  4
R2  2  5
R3  3  6
```

To access elements with a value greater than five we can use some subsetting commands and logical operators to produce the desired result.

```
> as.vector(value3[value3>5])
[1]  6  7  8  9 10 11 12
```

Lists

Creating Lists

Lists can be created using the `list` function. Like data frames, they can incorporate a mixture of modes into the one list and each component can be of a different length or size. For example, the following is an example of how we might create a list from scratch.

```
> L1 <- list(x = sample(1:5, 20, rep=T),
          y = rep(letters[1:5], 4), z = rpois(20, 1))
> L1
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1


$y
 [1]  "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
 [13] "c" "d" "e" "a" "b" "c" "d" "e"


$z
 [1] 1 3 0 0 3 1 3 1 0 1 2 2 0 3 1 1 0 1 2 0
```

There are a number of ways of accessing the first component of a list. We can either access it through the name of that component (if names are assigned) or by using a number corresponding to the position the component corresponds to. The former approach can be performed using subsetting ([[]]) or alternatively, by the extraction operator ($). Here are a few examples:

```
> L1[["x"]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
> L1[[1]]
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

To extract a sublist, we use single brackets. The following example extracts the first component only.

```
> L1[1]
$x
 [1] 2 1 1 4 5 3 4 5 5 3 3 3 4 3 2 3 3 2 3 1
```

Working with Lists

The length of a list is equal to the number of components in that list. So in the previous example, the number of components in `L1` equals 3. We confirm this result using the following line of code:

```
> length(L1)
[1]  3
```

To determine the names assigned to a list, the `names` function can be used. Names of lists can also be altered in a similar way to that shown for data frames.

```
> names(L1) <- c("Item1","Item2","Item3")
```

Indexing lists can be achieved in a similar way to how data frames are indexed:

```
>  L1$Item1[L1$Item1>2]
[1]  4  3  4  5  3  3  3  5  3  3  5
```

Joining two lists can be achieved either using the concatenation function or the append function. The following two scripts show how to join two lists together using both functions.

Concatenation function:

```
> L2 <- list(x=c(1,5,6,7),
     y=c("apple","orange","melon","grapes"))
> c(L1,L2)
$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a" "b"
[13]"c" "d" "e" "a" "b" "c" "d" "e"
$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
$x
[1] 1 5 6 7
$y
[1] "apple"  "orange" "melon"  "grapes"
```

Append Function:

```
> append(L1,L2,after=2)
$Item1
[1] 2 4 3 4 1 5 3 1 1 2 3 3 5 2 1 3 2 3 5 1
$Item2
[1] "a" "b" "c" "d" "e" "a" "b" "c" "d" "e" "a"
[12]"b" "c" "d" "e" "a" "b" "c" "d" "e"
$x
[1] 1 5 6 7
```

```
$y
[1] "apple"  "orange" "melon"  "grapes"
$Item3
[1] 0 0 2 1 1 0 2 0 0 1 1 1 0 0 1 1 1 3 0 2
```

Adding elements to a list can be achieved by

- adding a new component name:

  ```
  > L1$Item4 <- c("apple","orange","melon","grapes")
  # alternative way
  > L1[["Item4"]] <- c("apple","orange","melon","grapes")
  ```

- adding a new component element, whose index is greater than the length of the list

  ```
   L1[[4]] <- c("apple","orange","melon","grapes")
  > names(L1)[4] <- c("Item4")
  ```

There are also many functions within R that produce a list as output. Examples of these functions include `spline()`,`density()` and `locator()`.

Example: Cars93 Dataset

The Cars93 dataset was used in Session 1 to illustrate some modelling and graphical features of R. We use this dataset again to demonstrate the use of lists.

The script that appears below produces a density plot of vehicle weight using splines of different bin widths. The bin widths used are 500 and 1000 respectively and they change the level of smoothness assigned to each density.

The spline function returns a list of densities (y) corresponding to bin values (x). These can be passed to the plot routine to produce a line graph of the density.

A rug plot is produced beneath the graph to indicate actual data values. A legend describing the two lines on the plot is produced for clarity.

Figure 19 displays the density plot produced from the script below. (Note, this plot does not reflect the *mono* family option that appears in the slides.)

```
> attach(Cars93)
> windows()
> par(family="mono")
```

```
> dw5 <- spline(density(Weight, width=500))  # list
> dw10 <- spline(density(Weight,width=1000)) # list
> rx <- range(dw5$x,dw10$x)
> ry <- range(dw5$y,dw10$y)
> par(mar=c(5,5,2,2)+0.1) -> oldpar
> plot(dw5,type="n",xlim=rx,ylim=ry,cex=1.5,
xlab="Weight",ylab="Density")
> lines(dw5,lty=1,col="blue")
> lines(dw10,lty=2,col="red")
> pu <- par("usr")[3:4]  # actual y limits
> segments(Weight,pu[1],Weight,0,col="green")
> legend(locator(1),c("500kg window",
"1000kg   window"),lty=1:2)
> detach("Cars93")
```

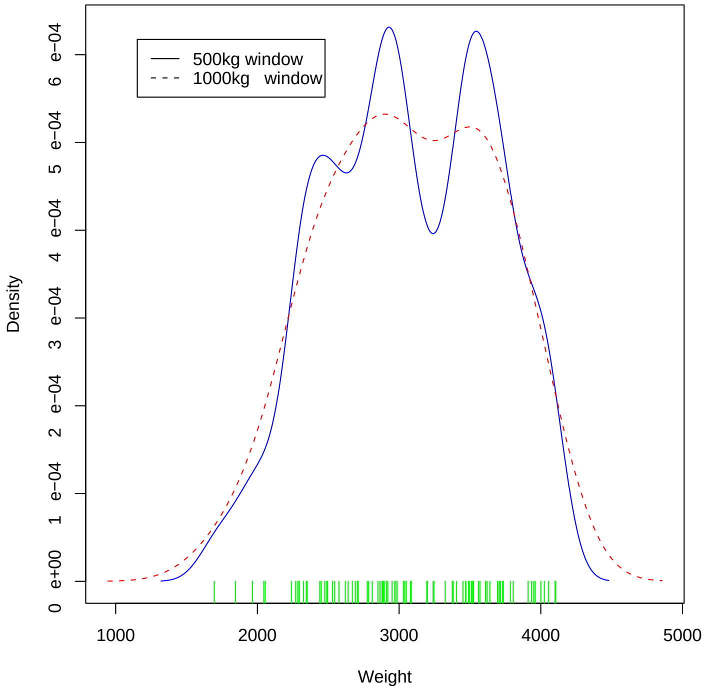


Figure 19: Density plot of vehicle weight from the Cars93 dataset.

Graphics: An Introduction

Anatomy of a Plot

High level plotting commands generate figures.

A figure consists of a *plot region* surrounded by margins:

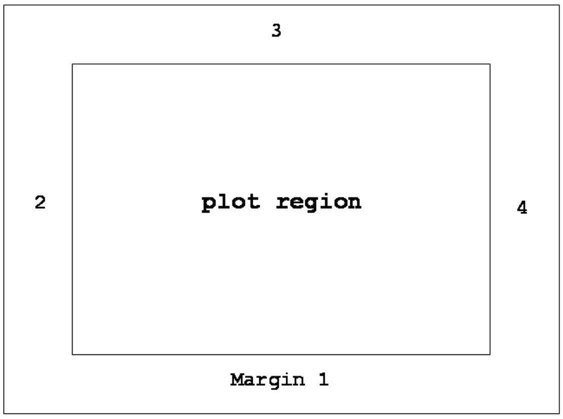


Figure 20: Anatomy of a Figure

The size of the margins is controlled by the argument `mai`. The value of `mai` is a vector `c(bottom,left,top,right)` of length 4 whose values are the widths, in inches, of the corresponding margin sides.

A typical call to `par()` to set the margins might be

```
par(mai=c(5,5,8,5)/10)
```

which allows `0.8in` at the top and `0.5in` on all other sides.

Figure Margins

Axes, axis labels and titles all appear in the margins of the figure.

Each margin is considered to have a number of *text lines* (not necessarily a whole number):

- Lines specified at 0 correspond to the edge of a plotted region (where the axis lines are drawn).
- Higher line numbers are further away from the plot.

The graphical parameter `mar` defines how many lines appear in each of the four margins. So `mar` is an alternative way of defining margins.

For any open graphics device there is a *standard font* which will be used for any characters if no other font is specified (`font`).

The standard font determines the width of the text lines in the margins.

If the font is *expanded* or `contracted` (`cex`) before `mar` is set, the text line width changes accordingly.

The `axis()` function draws an axis on the current plot. The `side` argument determines on which side it is to appear. Axes normally get drawn at line 0 but this may be changed with the `line` argument, or even inside the plot with the `pos` argument. If you wish to be specific about the positions of tick marks, use the `at` argument.

Margin Text

Axis labels can be created using the `xlab` and `ylab` graphics parameters when passed to functions such as `plot()`. To add such labels after the plot has been created, the title() function may be used.

An alternative approach to adding margin text is through the `mtext()` function:

```
> mtext("Label text",side=1,line=2)
```

The above piece of code will add text just below the x-axis. Using `side=3` is an alternative method for adding a plot title. Text is centred on the axis by default, but the `at` argument to `mtext()` can be used for more specific positioning.

Axes and tickmarks

The `axis()` function and others such as `plot()` or `tsplot()` use the following graphical parameters to allow control of the style of the axes:

- `axes`: should axes be drawn? (TRUE/FALSE)
- `bty`: controls the type of box which is drawn around plots
  - `bty="o"`: box drawn around plot (default)
  - `bty="l"`: L shaped axes drawn
  - `bty="7"`: part axes drawn on the left side and bottom of the plot. Lines drawn to the top and right side of the plot
  - `bty="c"`: C shaped axes drawn
  - `bty="u"`: U shaped axes drawn
  - `bty="]"`:] shaped axes drawn with part axis drawn on the left side of the plot
  - `bty="n"`: No box is drawn around plot
- `lab=c(nx,ny,len)`: modifies the way that axes are annotated. Defines the number of x and y tick intervals and the length (in characters) of the tick labels.

- las: style of the axis labels
  - las=0: always parallel to the axis (default)
  - las=1: always horizontal
  - las=2: always perpendicular to the axis
  - las=3: always vertical
- tck: length of tick marks as a fraction of the plotting region. Negative values refer to positions that fall outside the plotting region. Positive values indicate tick marks inside the plotting region.
- xaxs/yaxs: style of the axis interval calculation
  - "s" or "e": extreme than the range of the data
  - "i" or "r": inside the range of the data
  - "d": locks in the current axis

The Plot Region

Points within the plot region are accessed using *user co-ordinates*. The user co-ordinates are defined when a high level plot is created, or may be explicitly set with the usr graphics parameter. A setting

```
> par(usr=c(x.lo,x.hi,y.lo,y.hi))
```

means that x.lo, x.hi are the two extreme allowable plotting values in the x-direction and similarly in the y-direction.

When a graphics device is initialised, usr defaults to c(0,1,0,1). The frame() command (which starts a new empty figure) uses the old value of usr for the new plotting region.

Multiple Plots

There are two main ways of placing several plots on the one surface. The graphics parameter fig allows you to place several plots, possibly irregularly, on the one figure region.

It is also possible, and more common to have more than one figure to a page as a regular $n \times m$ array of figures. This behaviour is controlled by the mfrow or mfcol graphics parameter. For example

```
> par(mfrow=c(3,2))
```

will produce a plotting region with three rows and two columns.

Each high-level plotting command starts plotting on a new figure. When all figures are exhausted, a new page is generated. The mfg graphics parameter keeps track of the row and column of the current figure and the dimensions of the figure array. By setting this parameter unusual figure arrangements can be achieved.

Other Graphics Parameters

Some other useful graphics parameters include

- `ask=T`: R asks before producing the graphic. This is useful if you need to view multiple plots, one at a time.
- `new=T`: declares the current plot is unused (even if it is not). This means that R will not erase it before moving on to the next plot. This is useful for more fancy plots, where you may be producing a number of plots on the one figure.
- `fin`: gives the width and height of the current figure in inches.
- `din`: a read only parameter that returns the width and height of the current device surface in inches.

Overview of Graphics Functions

R has a variety of graphics functions. These are generally classed into

- High-level plotting functions that start a new plot
- Low-level plotting functions that add elements to an existing plot

Each function has its own set of arguments. The most common ones are

- `xlim`,`ylim`: range of variable plotted on the x and y axis respectively
- `pch`, `col`, `lty`: plotting character, colour and line type
- `xlab`, `ylab`: labels of x and y axis respectively
- `main`, `sub`: main title and sub-title of graph

General graphing parameters can be set using the `par()` function. For example, to view the setting for line type

```
> par()$lty
```

To set the line type using the par function

```
> par(lty=2)
```

Multiple plots per page can be achieved using the `mfrow` or `mfcol` argument to par. For example,

```
# 2x2 plotting region where plots
# appear by row
> par(mfrow=c(2,2))
# 2x2 plotting region where plots
# appear by column
> par(mfcol=c(2,2))
```

The `make.high()` function produces a number of high-level graphics ranging from dotcharts, histograms, boxplots and barplots for one dimensional data, scatterplots for two-dimensional data and contour, image plots and perspective mesh plots for three dimensional data. Figure 21 displays the results from running this function.

Laboratory Exercise

Try editing this function using the Tinn-R editor and changing some of the input parameters to these graphical functions.

The `make.low()` function produces a number of low-level graphics. These include plotting points, symbols, lines, segments or text on a graph, producing a box around a plot or joining line segments to create a polygon. These types of graphics are often useful for enhancing the feature of an existing plot. Figure 22 displays the results from running this function.

Laboratory Exercise

Try editing this function using the Tinn-R editor and changing some of the input parameters such as the type of symbol or plotting character to these graphical functions.

Displaying Univariate Data

Graphics for univariate data are often useful for exploring the location and distribution of observations in a vector. Comparisons can be made across vectors to determine changes in location or distribution. Furthermore, if the data correspond to times we can use time series methods for displaying and exploring the data.

Graphical methods for exploring the distributional properties of a vector include

- `hist` (histogram)
- `boxplot`

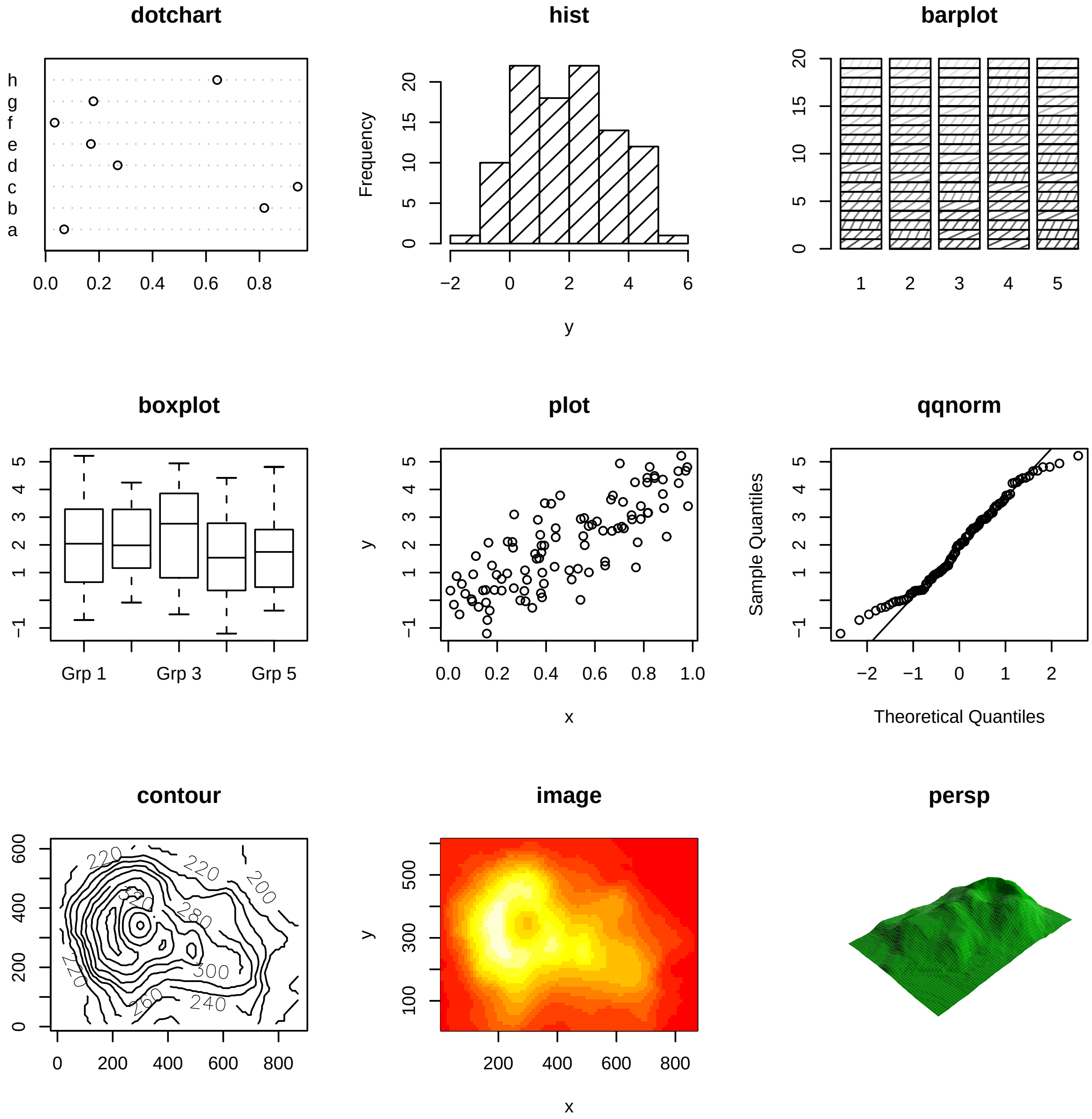


Figure 21: Examples of high level plotting functions

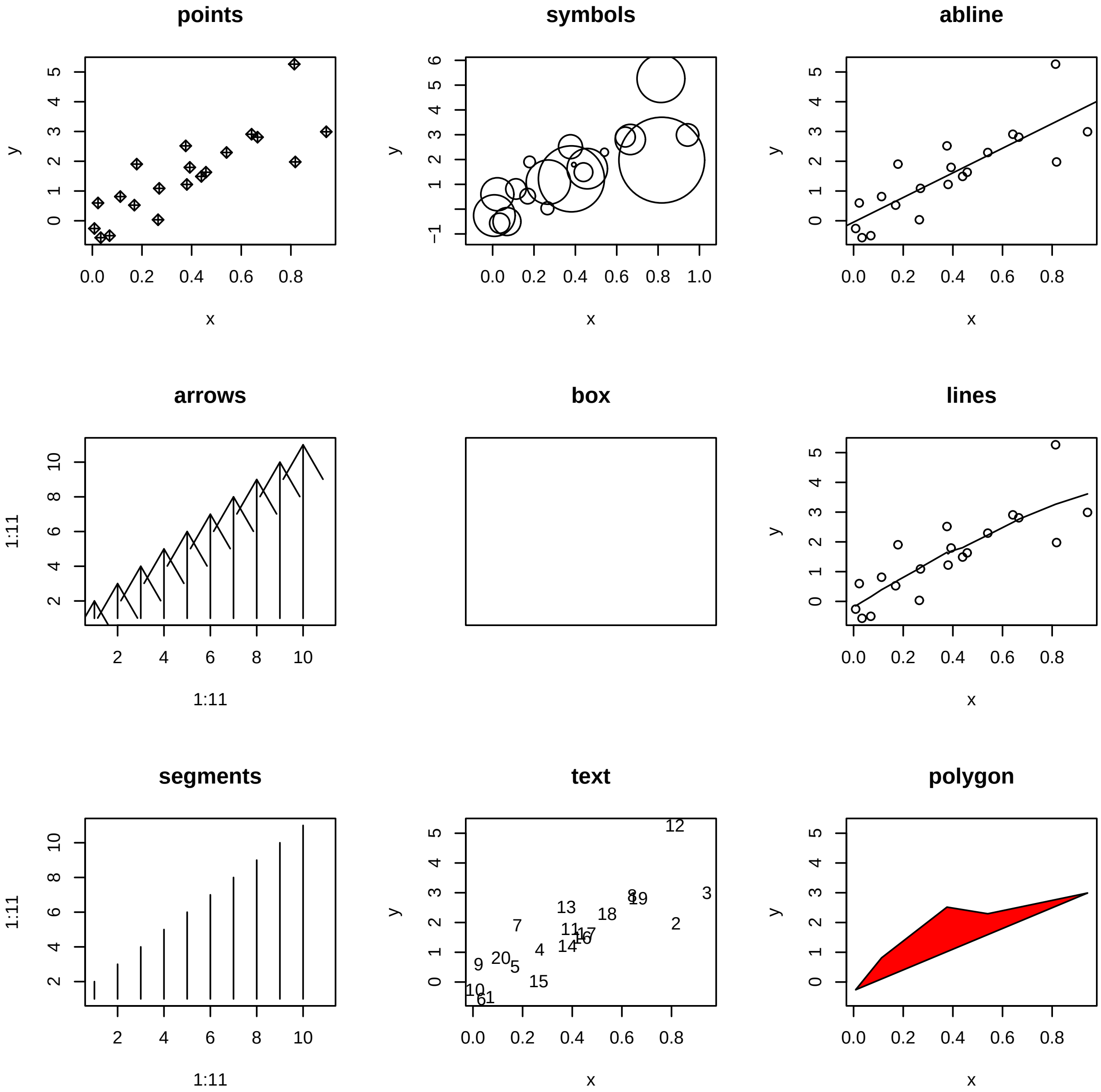


Figure 22: Examples of high level plotting functions

- `density`
- `qqnorm` (Normal quantile plot) and
- `qqline`

The `Cars93` dataset will be used to illustrate some of these plots. The following script sets up a 2×2 plotting region and produces a histogram, boxplot, density plot and Normal scores plot of the `MPG.highway` vector.

```
> attach(Cars93)
> par(mfrow=c(2,2))
# Histogram
> hist(MPG.highway,xlab="Miles per US Gallon",
main="Histogram")
# Boxplot
> boxplot(MPG.highway,main="Boxplot")
# Density
> plot(density(MPG.highway),type="l",
xlab="Miles per US Gallon",main="Density")
# Q-Q Plot
> qqnorm(MPG.highway,main="Normal Q-Qplot")
> qqline(MPG.highway)
```

The resulting plot is shown in Figure 23 and shows a distribution that is skewed heavily towards the right. This is visible in all four plots and it is particularly highlighted in the Normal scores plot shown in the bottom right hand corner by a set of points that deviate significantly from the line.

To make this variable more normal we could consider using a transformation, say logs, of the data:

```
> log(MPG.highway)
```

If we reproduce these plots on the newly transformed dataset we see that the distributions look a little better but there is still some departure from Normality present. (See Figure 24.)

Histograms

Histograms are a useful graphic for displaying univariate data. They break up data into cells and display each cell as a bar or rectangle, where the height is proportional to the

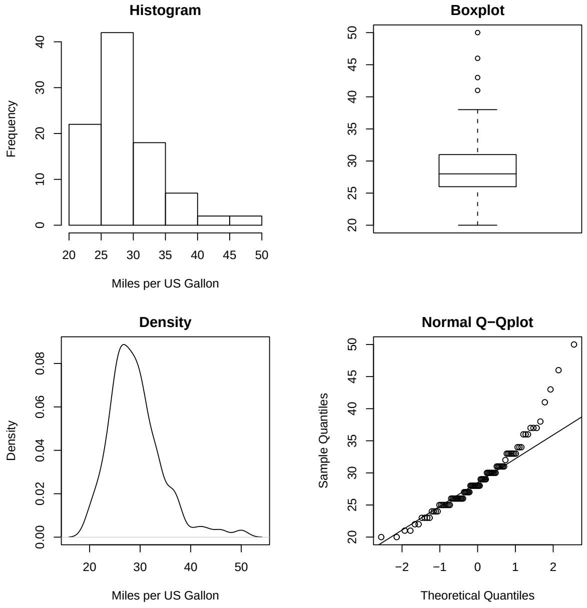


Figure 23: Distribution summaries of miles per gallon (highway)

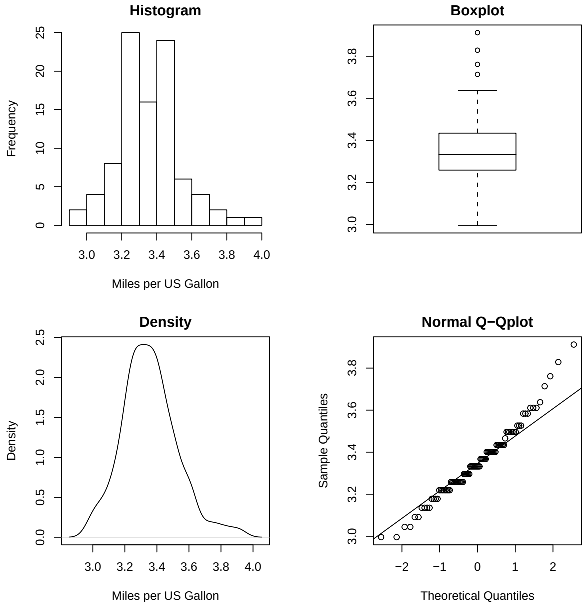


Figure 24: Distribution summaries of miles per gallon (highway)

number of points falling within each cell. The number of breaks/classes can be defined if required. The following shows example code for producing histograms. The second histogram drawn in Figure 25 specifies break points.

```
> par(mfrow=c(1,2))
> hist(MPG.highway,nclass=4,main="Specifying the Number of Classes")
> hist(MPG.highway,breaks=seq(from=20,to=60,by=5),
main="Specifying the Break Points")
> par(mfrow=c(1,1))
```

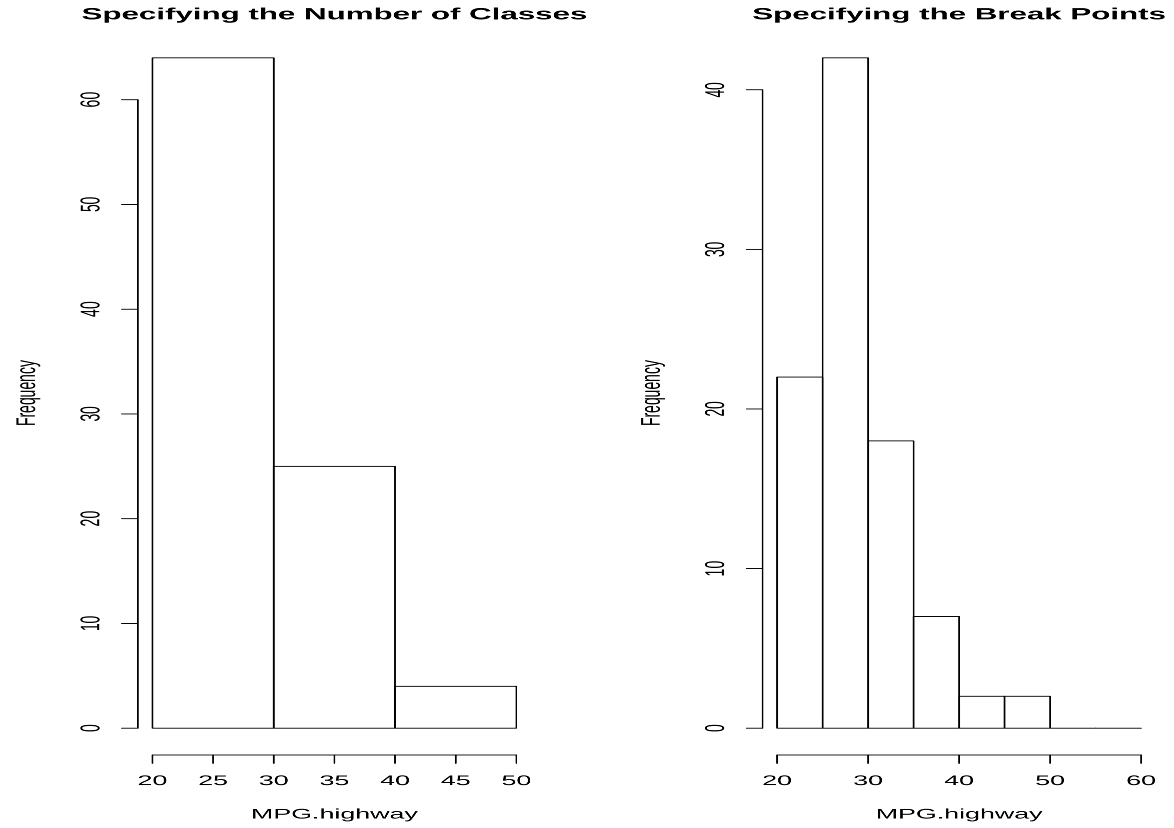


Figure 25: Examples of histograms produced on the Cars93 data: (a) no breakpoints specified and (b) breakpoints specified

Boxplots

Boxplots summarise the data and display these summaries in a box and whisker formation. They represent useful summaries for one dimensional data.

The box represents the inter-quartile range (IQR) and shows the median (line), first (lower edge of box) and third quartile (upper edge of box) of the distribution. Minimum and maximum values are displayed by the whiskers (lines that extend from the box to the minimum and maximum points).

If the distance between the minimum value and the first quartile exceeds 1.5 × IQR then the whisker extends from the lower quartile to the smallest value within 1.5 × IQR. Extreme points, representing those beyond this limit are indicated by points. A similar procedure is adopted for distances between the maximum value and the third quartile.

Figure 26 shows the result from producing a boxplot in R using the `boxplot` function. A summary of the data produces the following statistics:

```
> summary(MPG.highway}
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
20.00   26.00   28.00   29.09   31.00   50.00
```

These can be visualised on the plot in Figure 26.

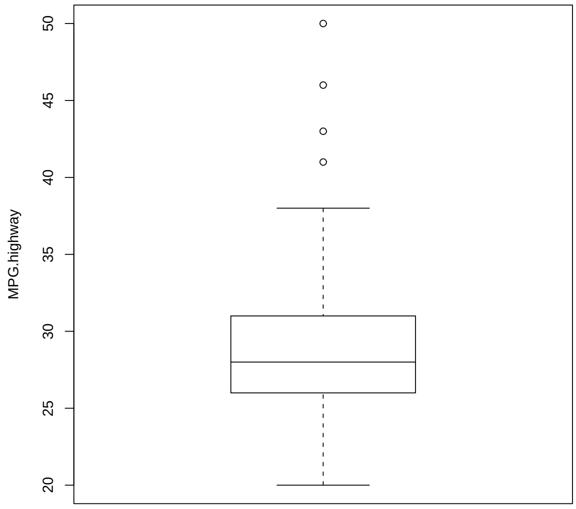


Figure 26: An example boxplot produced on the Cars93 data.

Densities

Densities can be used to compute smoothed representations of the observed data. The function `density` produces kernel density estimates for a given kernel and bandwidth. By default, the Gaussian kernel is used but there is an array of other kernels available in R. Look up the R help on `density` and see what options are available.

The bandwidth controls the level of smoothing. By default, this represents the standard deviation of the smoothing kernel but this too, can be changed depending on your requirements.

The following script produces a range of smoothed densities for the `MPG.highway` variable in the Cars93 dataset.

```
> par(mfrow=c(2,2))
> plot(density(MPG.highway),type="l",
main="Default Bandwidth)
> plot(density(MPG.highway,bw=0.5),type="l",
main="Bandwidth=0.5")
> plot(density(MPG.highway,bw=1),type="l",
main="Bandwidth=1")
> plot(density(MPG.highway,bw=5),type="l",
main="Bandwidth=5")
> par(mfrow=c(1,1))
```

The plots shown in Figure 27 show the result of running this script. The first plot in the top left hand corner of the figure is a density produced using the default bandwidth. This plot is fairly smooth, showing the skewed nature of the data. The plot produced using a bandwidth of 0.5 is a very rough representation of the data and does not accurately portray the features of the data. The density corresponding to a bandwidth of 1 provides a slightly higher level of smoothing but still appears too rough. The final plot, showing a density with a bandwidth of 5 is probably too smooth as it does not highlight the skewed nature of the data.

Quantile-Quantile Plots

Quantile-quantile plots are useful graphical displays when the aim is to check the distributional assumptions of your data. These plots produce a plot of the quantiles of one sample versus the quantiles of another sample and overlays the points with a line that corresponds to the theoretical quantiles from the distribution of interest. If the distributions are of the same shape then the points will fall *roughly* on a straight line.

Extreme points tend to be more variable than points in the centre. Therefore you can expect to see slight departures towards the lower and upper ends of the plot.

The function `qqnorm` compares the quantiles of the observed data against the quantiles from a Normal distribution. The function `qqline` will overlay the plot of quantiles with a line based on quantiles from a theoretical Normal distribution.

Figure 28 shows a Normal scores plot for the `MPG.highway` variable using the `qqnorm` and `qqline` functions in R. This plot shows some departure from Normality since the

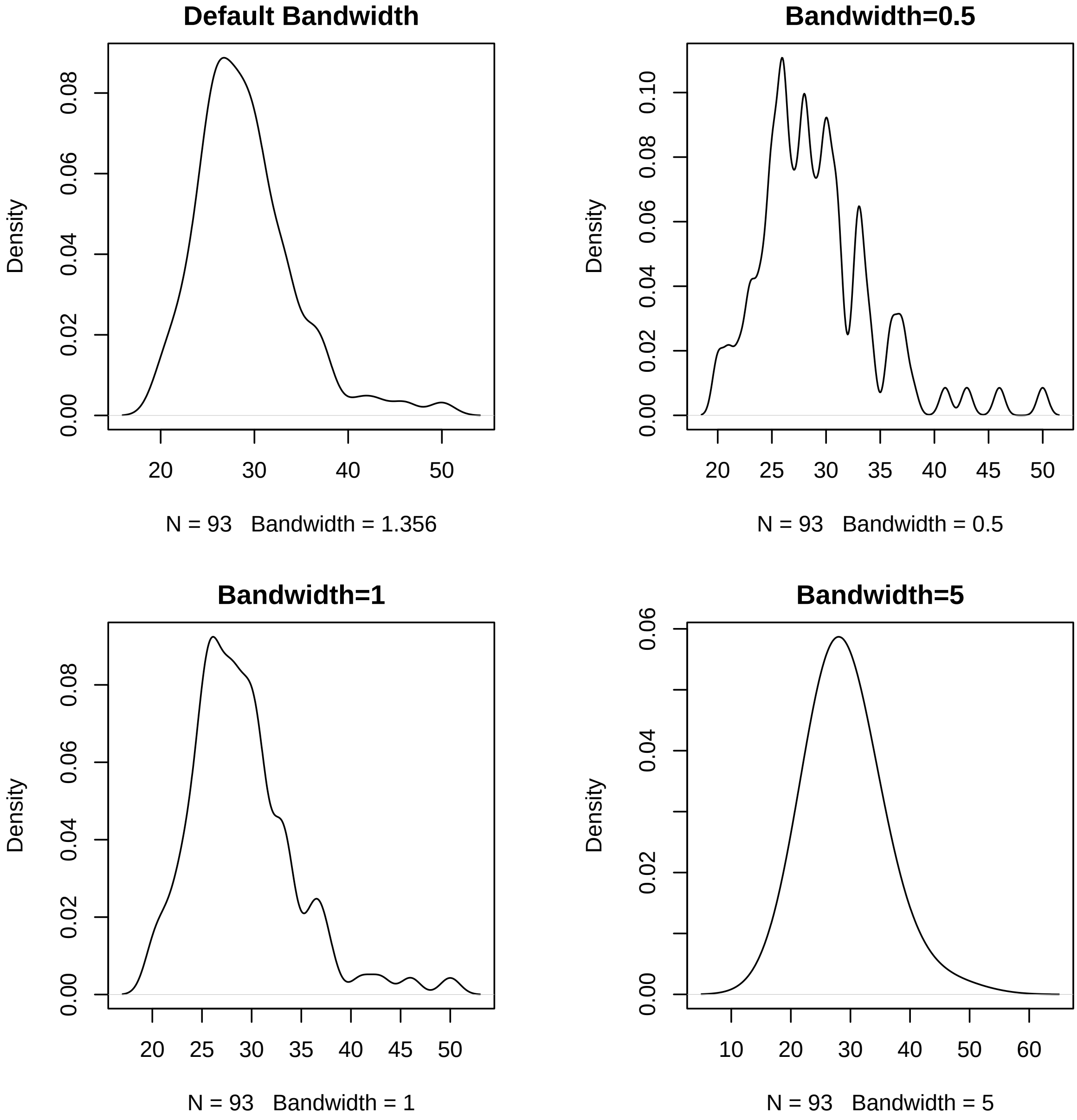


Figure 27: Density plots of `MPG.highway` data produced using (a) the default bandwidth, (b) a bandwidth of 0.5, (c) a bandwidth of 1 and (d) a bandwidth of 5

extreme points towards the upper end of the plot fall away from the plotted *theoretical* line.

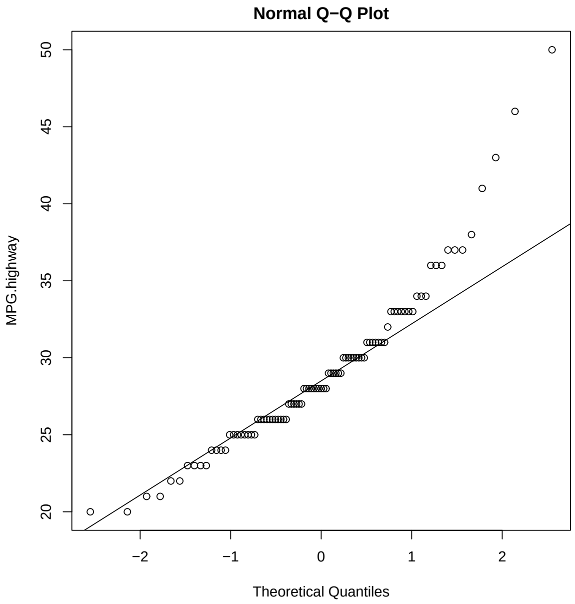


Figure 28: Normal scores plot of the `MPG.highway` data.

To compare a sample of data with other distributions, the `qqplot` function can be used. The following script generates data from a Poisson distribution and compares the data against the Normal distribution and Poisson distribution. Figure 29 presents the results of these comparisons.

```
# Generating Data from Poisson Distribution
> x <- rpois(1000,lambda=5)
> par(mfrow=c(1,2),pty="s")
# Comparing against a Normal
> qqnorm(x,ylab="x")
> qqline(x)
# Comparing against a Poisson
> qqplot(qpois(seq(0,1,length=50),
lambda=mean(x)),x,
xlab="Theoretical Quantiles",ylab="x")
> title(main="Poisson Q-Q Plot")
> par(mfrow=c(1,1))
```

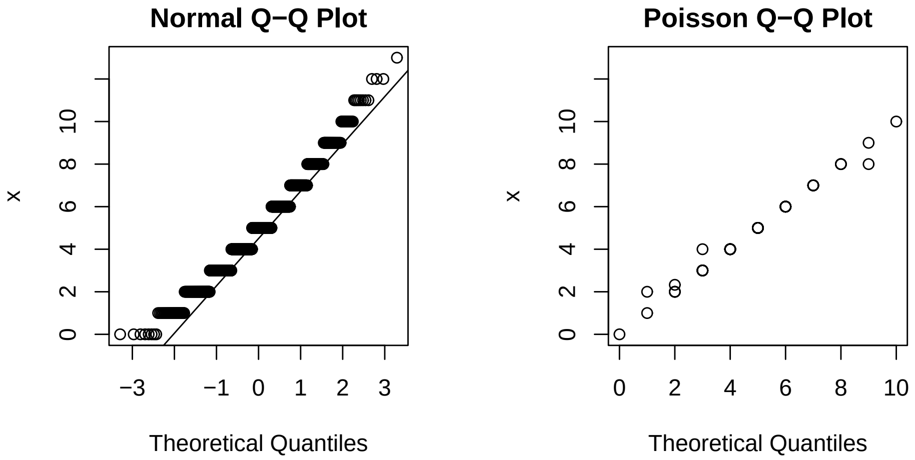


Figure 29: Quantile-Quantile plots of sample data with (a) the Normal distribution and (b) the Poisson distribution

Comparing Groups

There may be instances where you want to compare different groupings to investigate differences between location and scale and other distributional properties. There are a couple of graphical displays to help with these types of comparisons.

- Multiple histograms plotted with the same scale
- Boxplots split by groups
- Quantile-Quantile plots
  These plots enable the comparison of quantiles between two samples to determine if they are from similar distributions. If the distributions are similar, then the points

should lie in a straight line (roughly). Gaps between tick mark labels indicate differences in location and scale for the two datasets.

The following script produces histograms, boxplots and quantile-quantile plots to enable comparison between two variables in the `Cars93` database.

```
# Set up plotting region
> par(mfcol=c(2,2))

# Produce histograms to compare each dataset
> hist(MPG.highway,
xlim=range(MPG.highway,MPG.city))
> hist(MPG.city,xlim=range(MPG.highway,MPG.city))

# Produce boxplot split by type of driving
> boxplot(list(MPG.highway,MPG.city),
names=c("Highway","City"),
main="Miles per Gallon")

# Q-Q plot to check distribution shape and scale
> qqplot(MPG.highway,MPG.city, main="Q-Q Plot")
> par(mfrow=c(1,1))
```

Figure 30 shows the result from running this script in R. The plots show some differences between variables.

Working with Time Series Objects

Time series objects can be plotted using special plotting functions, which are available in the `stats` package. This is a standard package that is loaded when an R session begins.

To illustrate the plotting of time series objects, we investigate the `ldeaths` dataset. This is a time series object that reports the monthly deaths from bronchitis, emphysema and asthma for both males and females in the UK between 1974 and 1979.

To verify that it is a time series object we can use the `is.ts` function

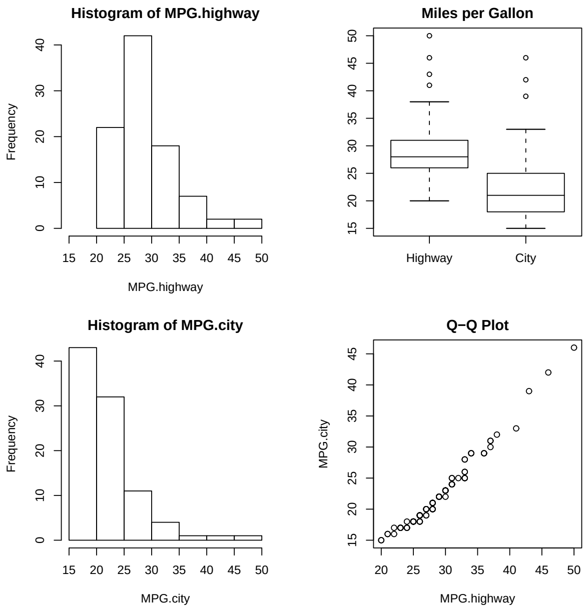


Figure 30: Comparison between the `MPG.highway` and `MPG.city` variables in the Cars93 database.

```
> is.ts(ldeaths)
[1] TRUE
```

Typing in `ldeaths` at the R prompt provides information about the time series

```
> ldeaths
      Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
1974 3035 2552 2704 2554 2014 1655 1721 1524 1596 2074 2199 2512
1975 2933 2889 2938 2497 1870 1726 1607 1545 1396 1787 2076 2837
1976 2787 3891 3179 2011 1636 1580 1489 1300 1356 1653 2013 2823
1977 3102 2294 2385 2444 1748 1554 1498 1361 1346 1564 1640 2293
1978 2815 3137 2679 1969 1870 1633 1529 1366 1357 1570 1535 2491
1979 3084 2605 2573 2143 1693 1504 1461 1354 1333 1492 1781 1915
```

Plots of time series objects can be obtained via the `plot.ts()` function.

```
> plot.ts(ldeaths)
```

Figure 31 displays the resulting plot and shows a strong seasonal component with a high number of deaths occurring in January and February.

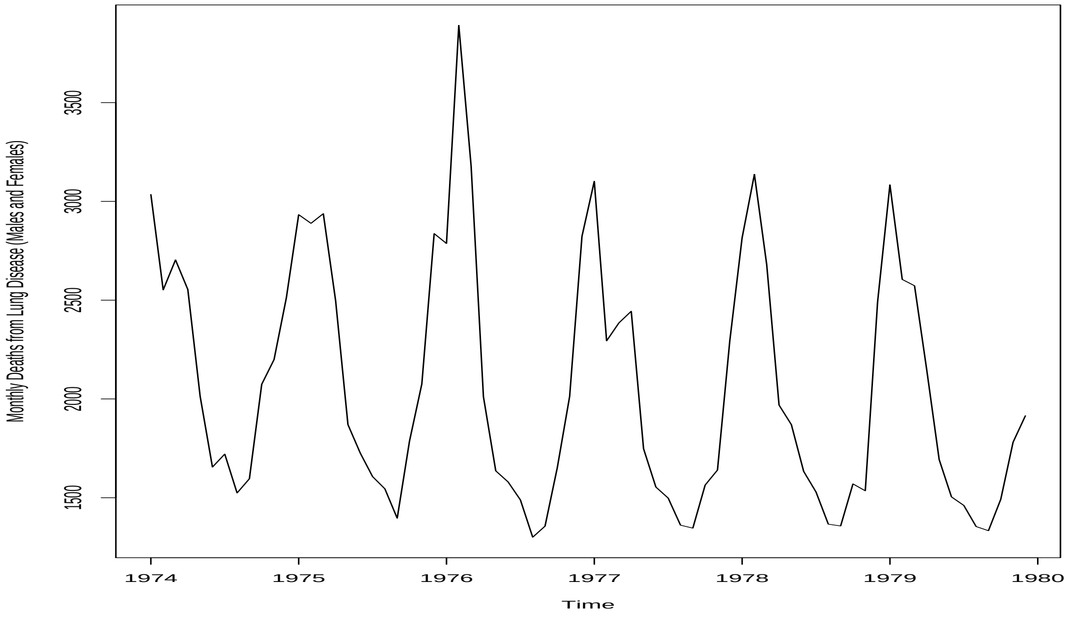


Figure 31: Time series plot showing the strong seasonal component of the `ldeaths` dataset.

The correlation at successive lags can be investigated using the `acf` function. This is useful when we want to try and understand the components of the time series and the dependencies over time. Three types of plots can be produced

- covariance computed at different lags (`type="covariance"`)
- correlations computed at different lags (`type="correlation"`)
- partial correlations computed at different lags (`type="partial"`)

Autocorrelations and partial autocorrelations are the two most useful plots for assessing serial correlation, determining an appropriate model and what parameters go into the model. Partial autocorrelations are an extension of autocorrelations that partial out the correlations with all elements within the lag. In other words, the dependence is on the intermediate elements. If partial autocorrelations are requested for a lag of 1 then this is equivalent to the autocorrelations.

Correlation functions produced for the `ldeaths` dataset are shown in Figure 32. All three plots show a strong seasonal pattern that will need to be accommodated in the time series model. A confidence interval (shown in blue) is also plotted to help with the choice of model and associated parameters of the model.

Correlation plots were produced using the following code:

```
> par(mfrow=c(3,1))
```

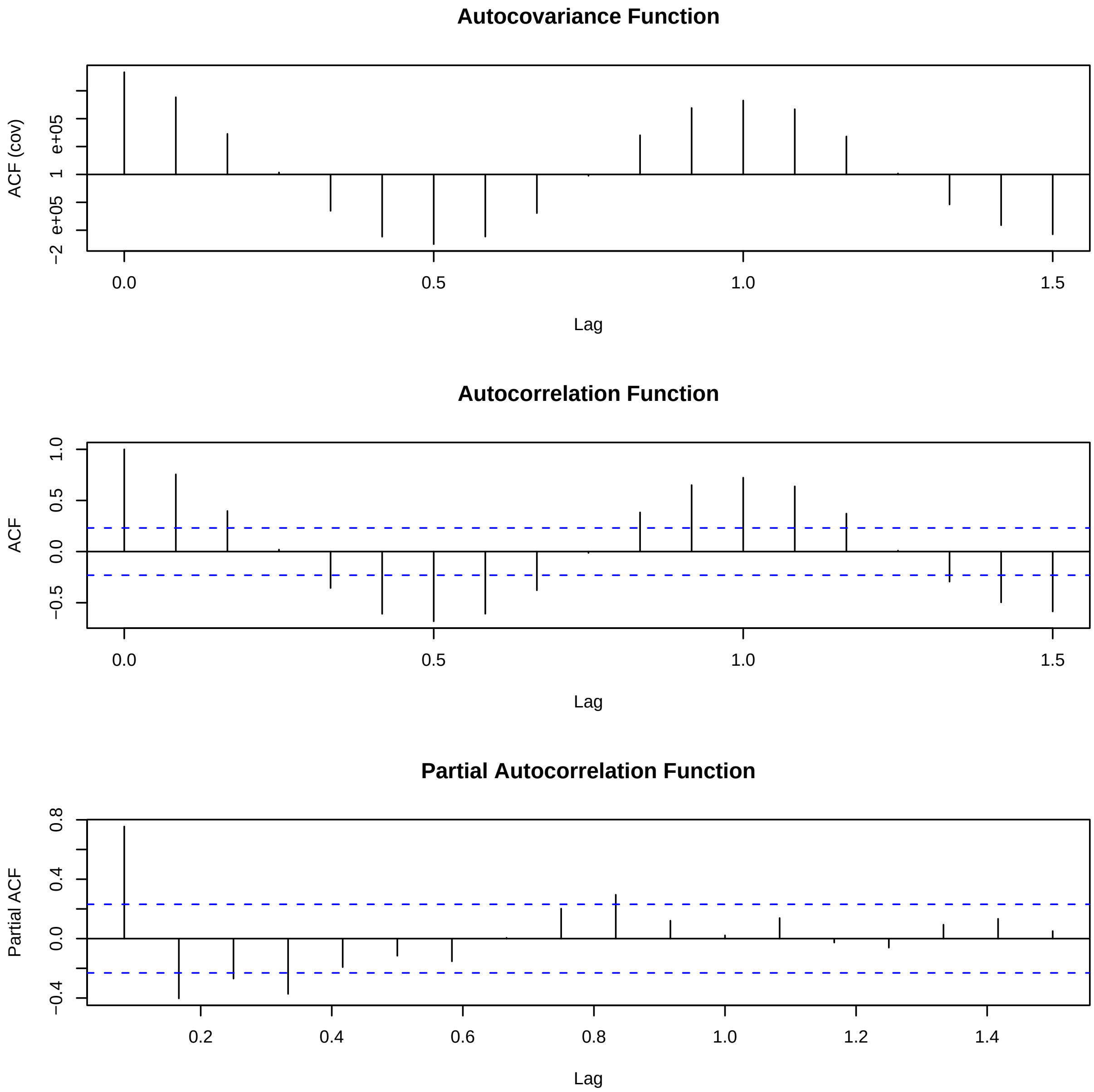


Figure 32: Correlation and Covariance functions of the `ldeaths` time series.

```
> acf(ldeaths,type="covariance")
> acf(ldeaths,type="correlation")
> acf(ldeaths,type="partial")
> par(mfrow=c(1,1))
```

Plotting multiple time series can be achieved using the `ts.plot` function. To illustrate this, we use the `ldeaths`, `mdeaths` and `fdeaths` datasets which represent the monthly deaths for both sexes, for males and females respectively. The script for producing such a plot is shown below. Figure 33 displays the resulting plot.

```
> ts.plot(ldeaths,mdeaths,fdeaths,
gpars=list(xlab="year", ylab="deaths", lty=c(1:3)))
> legend(locator(1),c("Overall Deaths",
"Male Deaths","Female Deaths"),lty=1:3,bty="n")
```

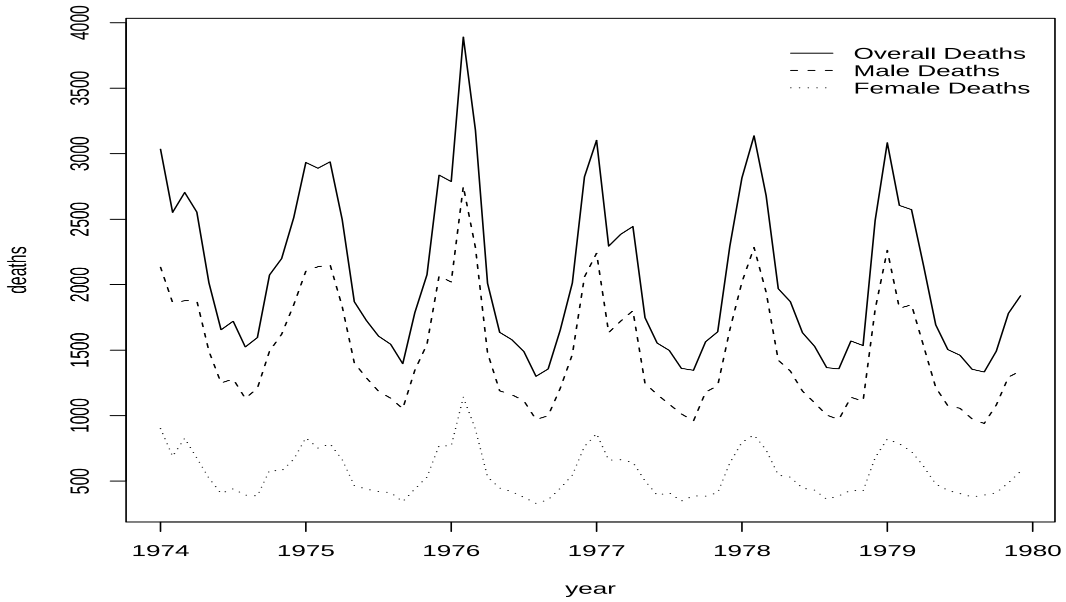


Figure 33: Time series plots of reported monthly deaths of lung disease.

Displaying Bivariate Data

The easiest way to display bivariate data is through a scatterplot using the `plot` function. The `type` argument allows you to produce different types of plots

- `type="p"`: plots a character at each point
- `type="l"`: plots a line connecting each point
- `type="b"`: plots both lines and characters
- `type="o"`: plots lines and characters that overlay the lines
- `type="s"`: plots stair steps
- `type="h"`: plots histogram-like vertical lines
- `type="n"`: no points or lines are plotted

Figure 34 shows a number of plots produced using the `ldeaths` dataset for different settings of `type`. The script that produced this plot is shown below.

```
# Producing scatterplots of different types
> par(mfrow=c(4,2))
> plot(ldeaths,type="p",main='pty="p"')
> plot(ldeaths,type="l",main='pty="l"')
> plot(ldeaths,type="b",main='pty="b"')
> plot(ldeaths,type="o",main='pty="o"')
> plot(ldeaths,type="s",main='pty="s"')
> plot(ldeaths,type="h",main='pty="h"')
> plot(ldeaths,type="n",main='pty="n"')
> par(mfrow=c(1,1))
```

Adding Points, Text, Symbols & Lines

Adding points, text, symbols and lines to an existing plot is simple to do and will be demonstrated using the `Cars93` dataset.

Points can be added to an existing plot using the `points` function. See the following script for an example of how to do this.

```
# Set up plotting region
> plot(MPG.highway,Price,type="n",
xlim=range(MPG.highway,MPG.city),
   xlab="miles per gallon")
> points(MPG.highway,Price,col="red",pch=16)
> points(MPG.city,Price,col="blue",pch=16)
```

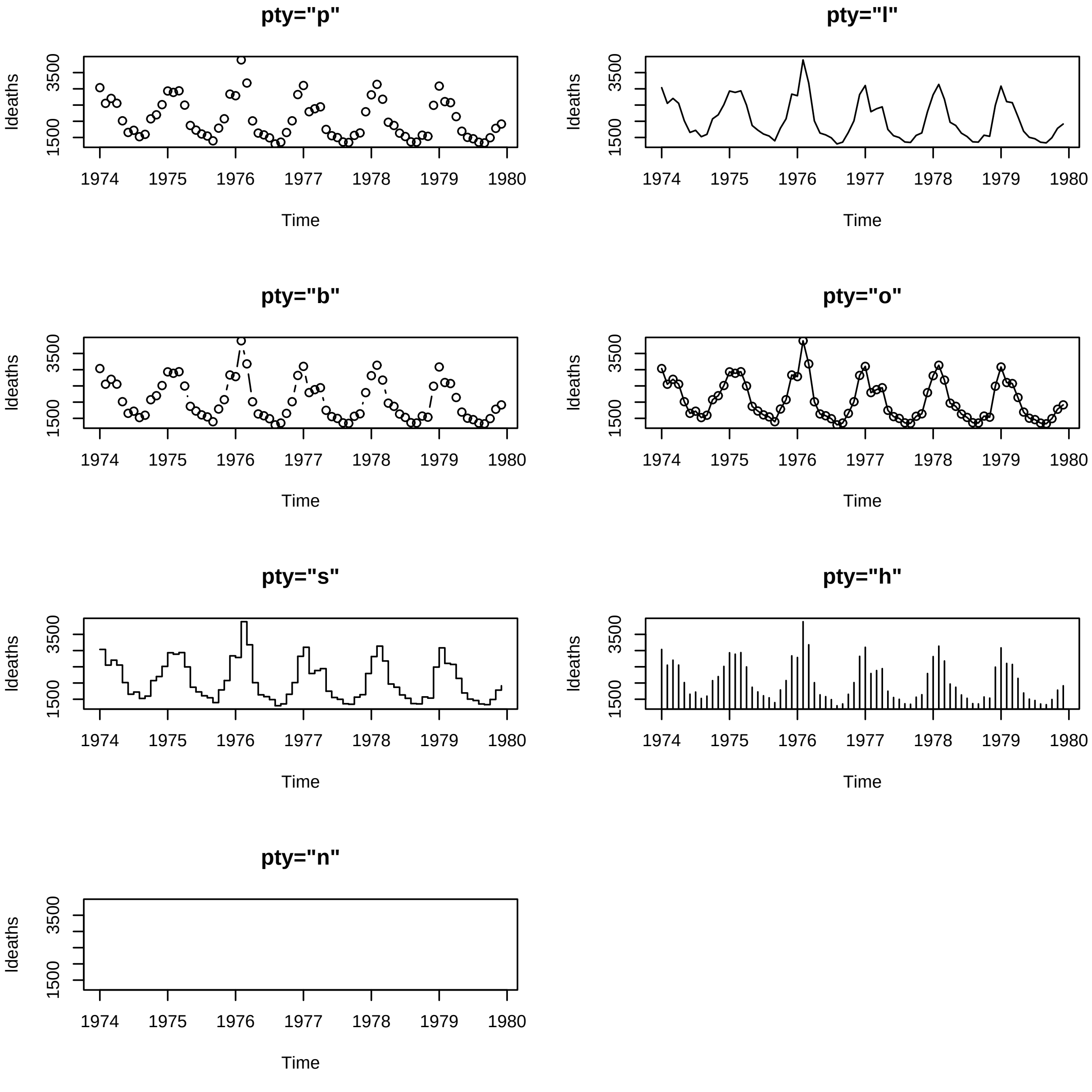


Figure 34: Time series plots of reported monthly deaths of lung disease using different plotting characters.

```
> legend(locator(1),c("Highway","City"),
col=c("red","blue"),pch=16,bty="n")
```

Figure 35(a) shows the result from this script and displays a plot of the price of vehicles versus the mile per gallon for highway driving (red) and city driving (blue).

We may wish to add text to the plot shown in Figure 35(a) to identify specific vehicles. We can do this using the `text` function and this is demonstrated in the following script using only the first ten rows of the data. Figure 35(b) is the resulting plot.

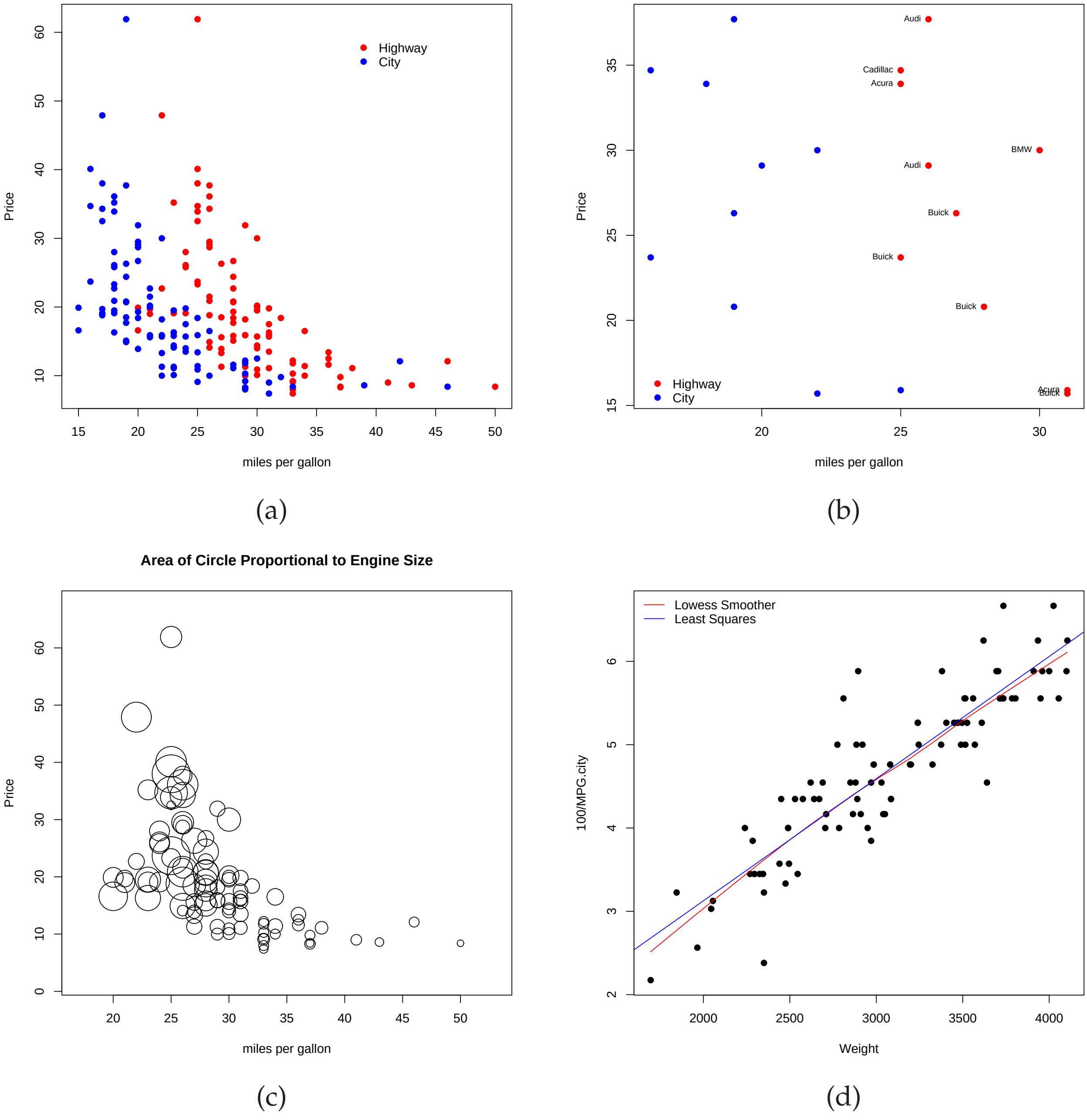
```
> plot(MPG.highway[1:10],Price[1:10],type="n",
      ylab="Price",xlim=range(MPG.highway[1:10],
      MPG.city[1:10]),xlab="miles per gallon")
> points(MPG.highway[1:10],Price[1:10],col="red",pch=16)
> points(MPG.city[1:10],Price[1:10],col="blue",pch=16)
> legend(locator(1),c("Highway","City"),
 col=c("red","blue"),pch=16,bty="n")
# label highway data
> text(MPG.highway[1:10],Price[1:10],Manufacturer[1:10],
  cex=0.7,pos=2)
```

Of course there may be a time where we want to select points interactively. These may be outlying points for example in a residual plot. We can do this using the `identify` function shown in the following script.

```
> identify(c(MPG.city[1:10],MPG.highway[1:10]),
    rep(Price[1:10],2),rep(Manufacturer[1:10],2),pos=2)
```

Instead of adding points to a plot, we may wish to add a symbol that represents the size of another variable in the database. For the `Cars93` dataset, it may be interesting to look at price versus miles per gallon according to engine size of the vehicle and produce a plot such as the one shown in Figure 35(c). This plot indicates that the cars with the bigger engine tend to be more expensive in price and have a lower miles per gallon ratio than other cars with smaller engines and lower in price. The script used to produce this plot is shown below.

```
symbols(MPG.highway,Price,circles=EngineSize,
xlab="miles per gallon",ylab="Price",inches=0.25,
main="Area of Circle Proportional to Engine Size")
```



(a) (b)

(c) (d)

Figure 35: Plots that show how to (a) add points, (b) add text, (c) add symbols and (d) add lines to a plot.

Finally, we may wish to add lines to an existing plot. This can be achieved using the `lines` function, which adds a line connected by specified points, or the `abline` function, which adds a vertical, horizontal or straight line with specific intercept and slope. To illustrate this concept, we produce a plot of gallons per 100 miles versus weight with three different lines overlayed: (1) a lowess smoother, (2) a least squares fit using `lines` and (3) a least squares fit using `abline`. The resulting plot is shown in Figure 35(d) using the script set out below.

```
> with(Cars93, {
plot(Weight,100/MPG.city,pch=16)
lines(lowess(Weight,100/MPG.city),col="red")
lines(lsfit(Weight,100/MPG.city),col="blue")
abline(coef(lsfit(Weight,100/MPG.city)),col="blue")
xy <- par("usr")[c(1,4)]
legend(xy[1], xy[2],
c("Lowess Smoother","Least Squares"),
col=c("red","blue"),lty=1,bty="n")
})
```

Labelling and Documenting Plots

R contains a number of functions for providing labels and documentation for plots. Some of these may have been mentioned before but here they are again.

- `title`: allows you to place a title, labels for the x and y axes and a subtitle
- `legend`: produces a legend for a plot at a specific location, unless the locator function has been used
- `mtext`: allows you to place text into the margins of a plot

The functions `legend` and `title` have been used frequently throughout this presentation. The function `mtext` has also been mentioned but now, here is an example.

For large datasets it is often easier to view the correlations of the covariates as an image instead of viewing them as text. To illustrate this concept, we use the `iris` dataset.

To recap, the iris dataset consists of the sepal length and width and petal length and width measurements for three species of Iris: Setosa, Versicolor and Virginica. The `cor` function will calculate Pearson correlations between variables. Applying this function to the iris dataset gives:

```
> cor(iris[,-5])
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

For small datasets, output to the screen if fine but for larger datasets this becomes a little monotonous. Alternatively, we could visually display the correlations as an image, where black pixels represent correlations of 1 and white pixels represent correlations of -1. Here's how we do it . . .

```
> image(1:4,1:4,cor(iris[,-5]),
      col=gray(seq(from=100,to=0,length=100)/100),
      axes=F,xlab="",ylab="")
> mtext(side=1,text=names(iris[,-5]),
   at=seq(from=1,to=4,length=4),line=1,cex=0.8)
> mtext(side=2,text=names(iris[,-5]),
   at=seq(from=1,to=4,length=4),line=1,cex=0.8)
> title(main="Image Plot of Correlations (Iris Data)")
```

Notice how we produce the image without any axes. The reason for this is to stop R from printing numbers and tick marks on the x and y axes because we want the column names to appear instead. To ensure that the column names are printed, we use the `mtext` function and then add a title to the plot. The resulting plot is shown in Figure 36.

Displaying Higher Dimensional Data

Pairwise Plots

The `pairs` function is a useful high-level plotting function for displaying and exploring multivariate data. It produces a scatterplot between all possible pairs of variables in a dataset and for each variable, it uses the same scale. This is useful if you are looking for patterns in your data.

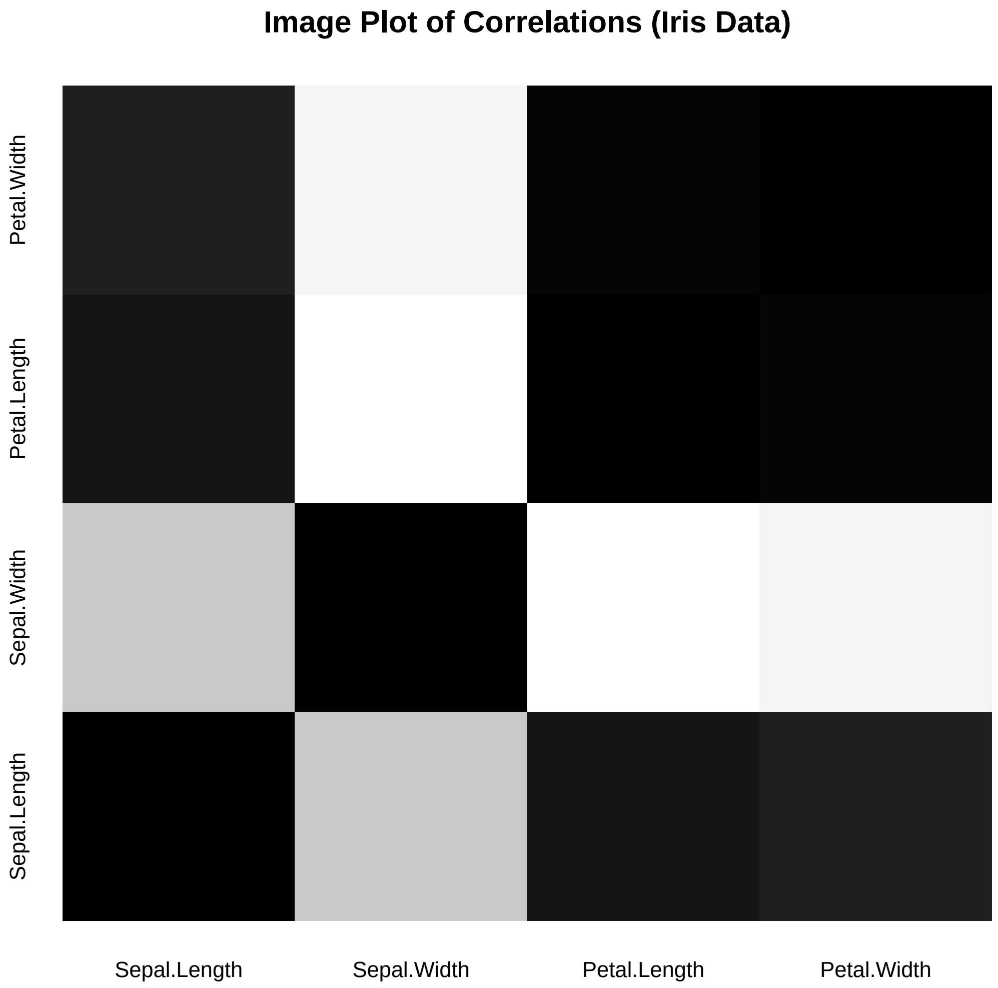


Figure 36: Image plot of correlations for the iris dataset.

In a previous session a pairs plot was produced for the iris data. We now apply this function to the `state.x77` dataset which provides information on 50 states of the USA. The script below shows how you can produce such a plot and Figure 37 shows the resulting figure.

```
> pairs(state.x77[,1:5],
    main = "Information from 50 States of America",pch = 16)
```

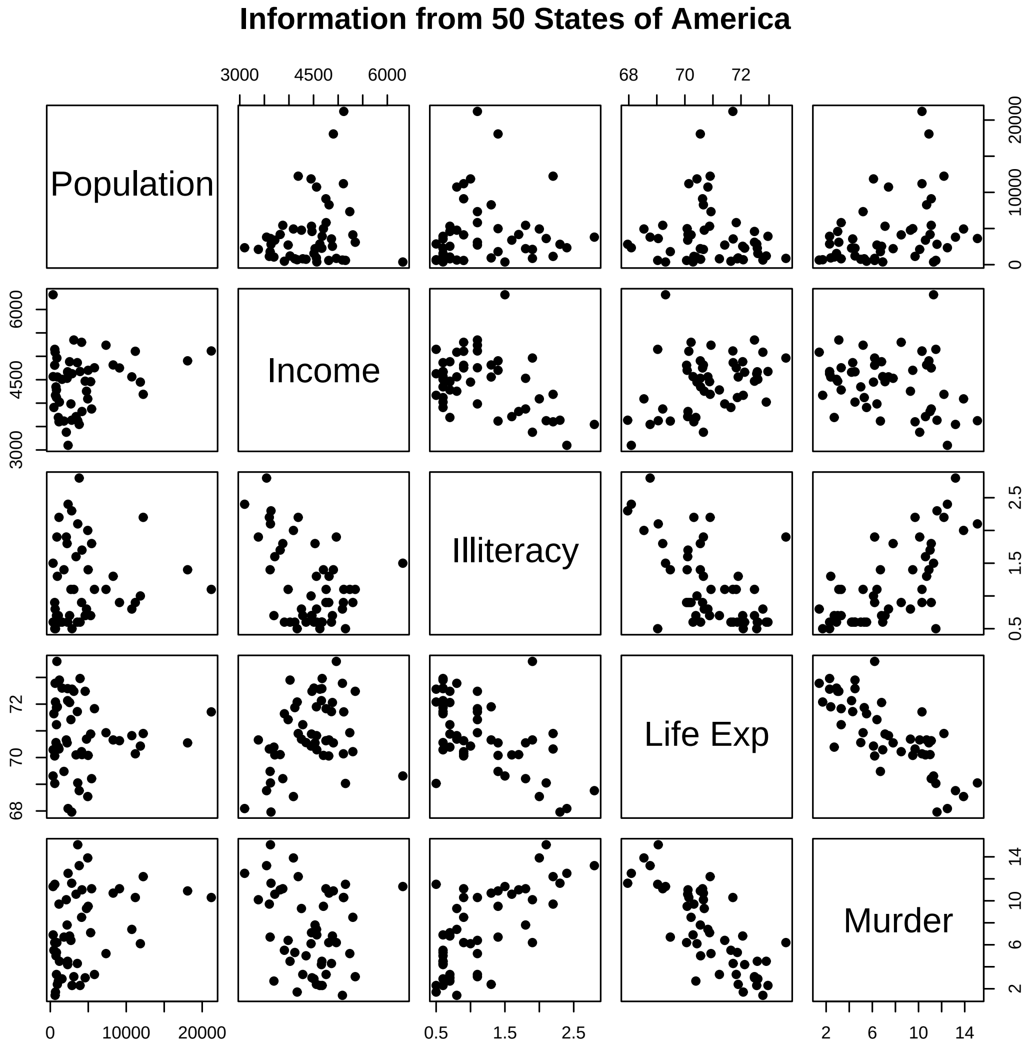


Figure 37: Pairwise scatter plots of statistics reported from the US

For something a little more sophisticated, we can overlay the pairs plot with a smoother to look for any trends in the data. We can achieve this using the following script. Figure 38 plots the result.

```
> pairs(state.x77[,1:5],
main = "Information from 50 States of America",
        pch = 16,panel=panel.smooth)
```

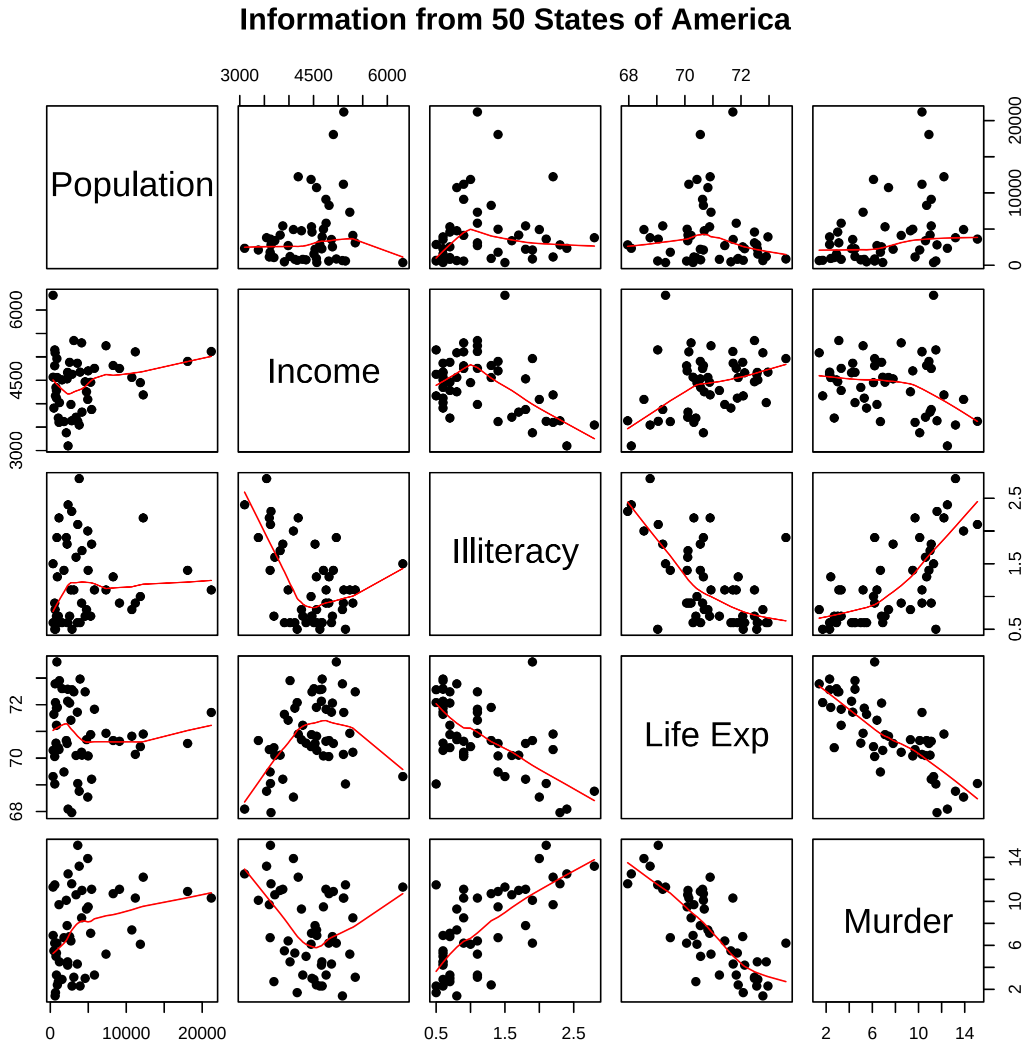


Figure 38: Pairwise scatter plots of statistics reported from the US. A scatterplot smoother is overlayed.

Star and Segment Plots

These type of plots are useful for exploring multivariate data. Each point on the star represents a variable and the length of each radial is proportional to the value of that variable. Similar stars therefore indicate similar cases.

To illustrate this concept, the Motor Vehicle Performance (1974 US dataset) will be used.

The aim is to investigate the similarity between makes and models using seven variables:

Miles/US gallon (mpg)   number of cylinders (cyl)   displacement (cub. in.)
gross horsepower (hp)   rear axle ratio (drat)   weight (wt)
quarter mile time (qsec)

Figure 39 displays a star plot using the `stars` function outlined in the script below. (Note the specification of the key location in the script. This places the key in the bottom right hand corner of the plot.) Of the different makes and models displayed, there are some similarities between vehicles. For example, the Cadillac Fleetwood and Lincoln Continental appear to be similar in terms of all of the variables. The Mercedes vehicles appear to be similar with the exception to the 240D series. Although the 280C series is similar to the 230 and 280 series, there are slight differences in terms of the rear axle ratio and mileage.

```
> stars(mtcars[,1:7],key.loc=c(14,1.8),
        main="Motor Vehicle Performance",
        flip.labels=FALSE)
```

A segment plot shown in Figure 40 is an alternative multivariate plot that tries to summarise the features of each vehicle using segments from a circle. To obtain this plot, the `draw.segments=T` must be specified.

Overlaying Figures

Figures can be overlayed using one of two features in R. The first uses the `add=T` option, which is available in some plotting functions. This will add certain features to an existing graph. For example, in the script below, a contour plot showing topographic information about a New Zealand volcano is overlayed on an image. Figure 41 shows the resulting graphic.

```
> z <- volcano
> x <- 10*(1:nrow(z))   # 10m spacing (S to N)
> y <- 10*(1:ncol(z))   # 10m spacing (E to W)
> image(x,y,z,main="Mt Eden")
> contour(x,y,z,add=T)
```

Alternatively, figures can be overlayed using the `new=T` option within the `par` function. This was described earlier in the notes. To recap, this feature, when used in R, can overlay a plot using the same axis setup as the first plot. This feature is also useful for producing multiple figures as shown in the following piece of code.

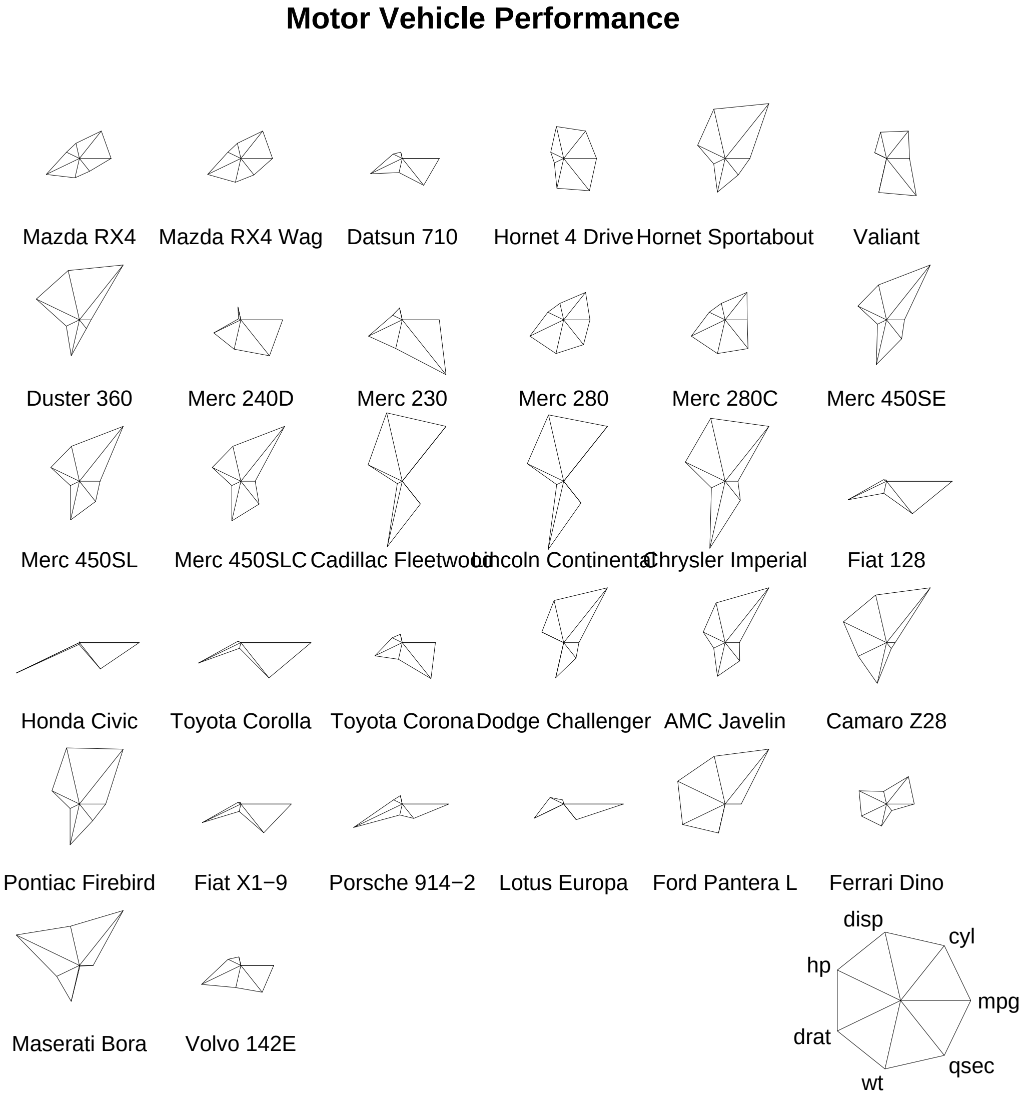


Figure 39: Star plot of the motor vehicle performance dataset

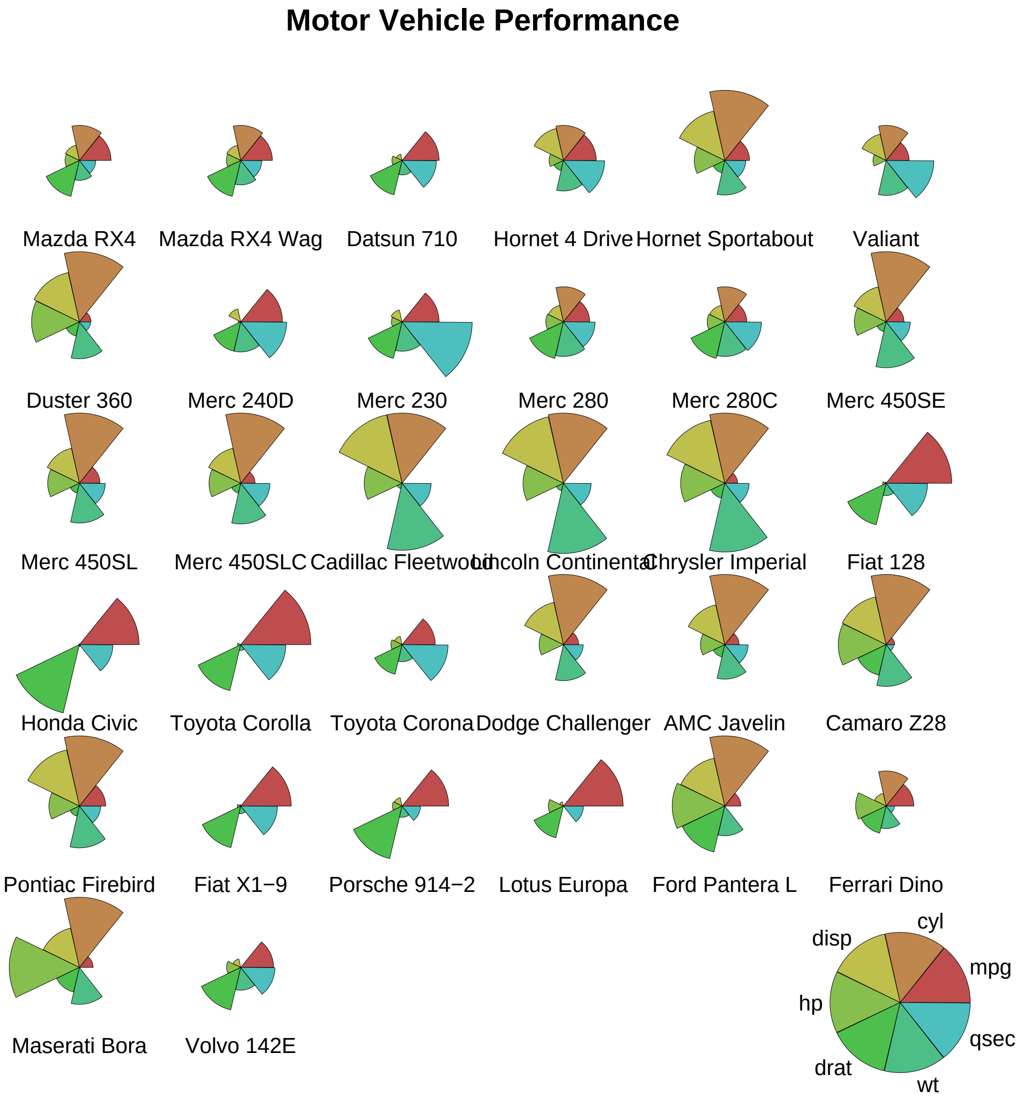


Figure 40: Segment plot of the motor vehicle performance dataset

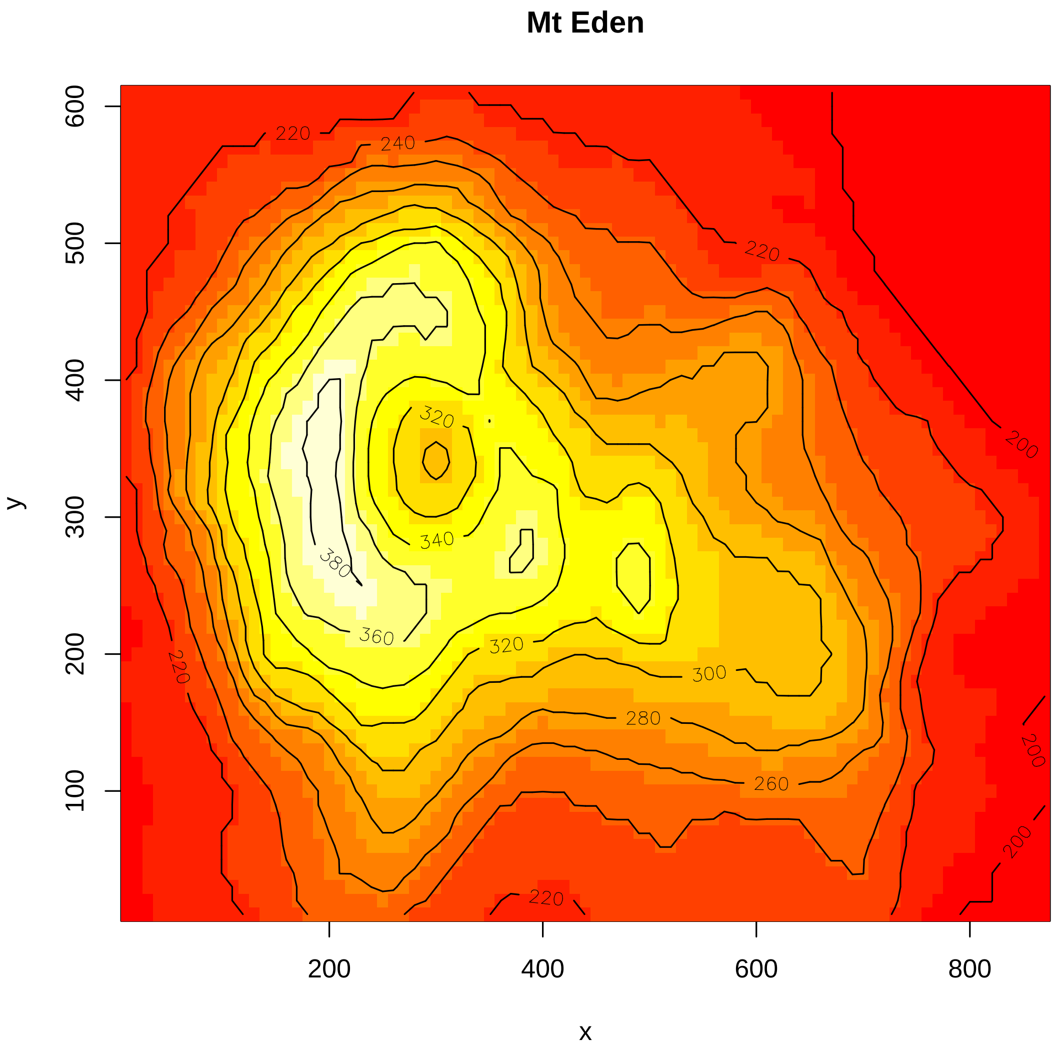


Figure 41: Image plot of the topographic of a New Zealand volcano with contours overlayed.

```
> x <- rnorm(1000,1,1)
> y <- -1 + 4*x + rnorm(1000)
> frame()
> par(fig=c(0,1,0.4,1),mar=c(1,4,4,4))
> plot(x,y,xlab="",ylab="y",pch=16,col="gray",axes=F)
> abline(coef(lm(y˜x)),col="red")
> box()
> axis(side=2)
> par(new=T,fig=c(0,1,0,0.4),mar=c(5,4,0.5,4))
> hist(x,xlab="x",main="",ylab="Frequency")
```

In this script, we generate some data and store it in `x`. We then generate a response `y` based on a linear relationship with some error. To set up the first plotting region we use the `frame()` function. The plot we would like to produce needs to be plotted in two components. The first is a plot of `y` versus `x` with a least squares line overlayed. The second is a histogram of `x` showing the distribution of the explanatory variable. In order to produce the second plot, `new=T` was specified to trick R into believing that it has gone to a new graphics window. This allows you to produce multiple graphs per plot. The resulting plot appears in Figure 42.

For something a little more fancy, we may want to overlay a series of plots to help with an analysis. The script called `Time_Series_Ex.R` produces Figure 43, a panel of figures that summarise some interesting features of a simulated time series. The figure comprises a time series plot of the data, illustration of the seasonal component that was estimated from a time series model, the estimated trend lines with 95% confidence limits and a plot of residuals. Notice how the axis at the bottom is the same axis used for all of the plots. The axis for each plot have also been alternated to appear on different sides, for clarity.

Laboratory Exercise

Step through this function in more detail

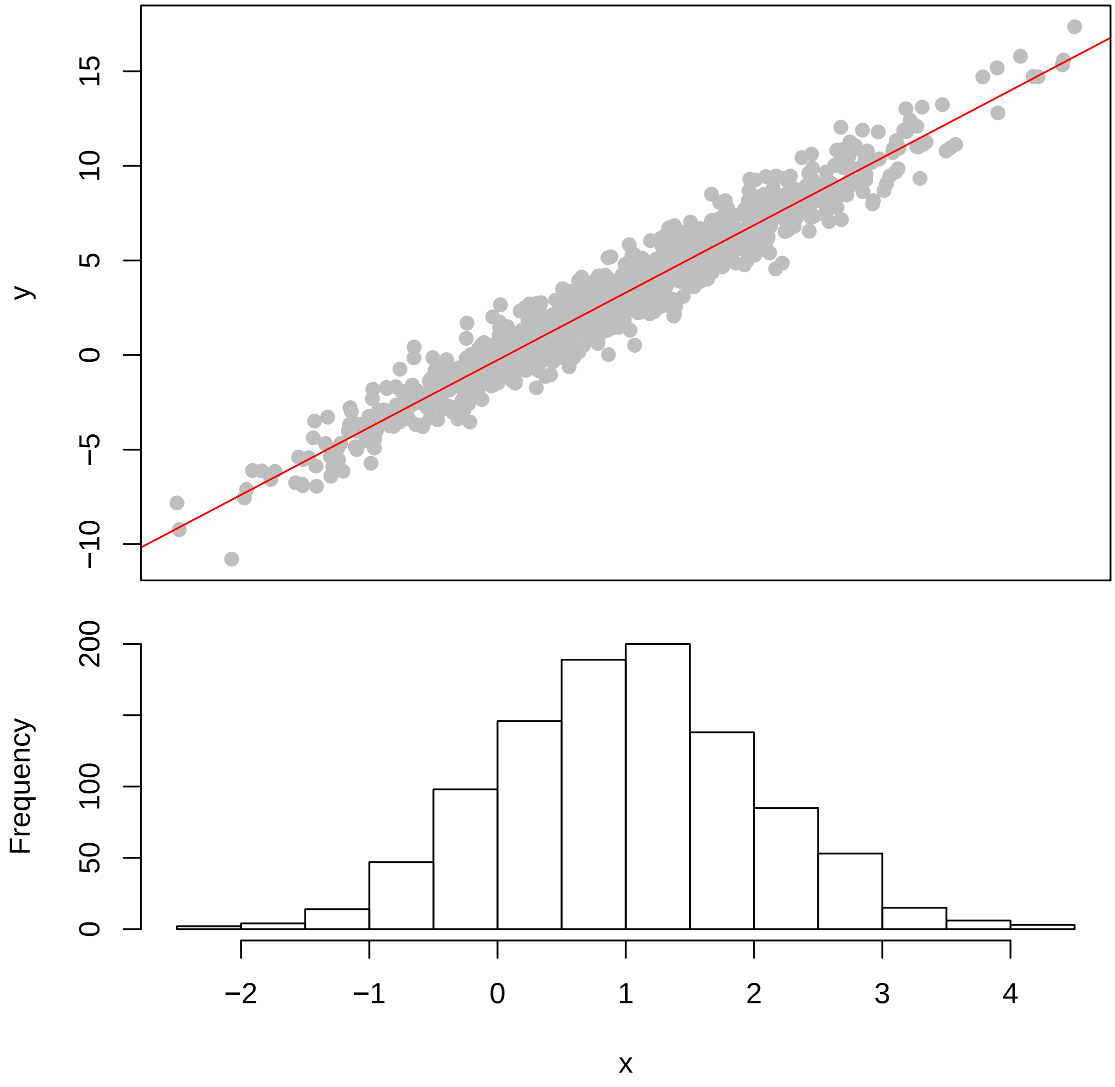


Figure 42: Example of overlaying figures

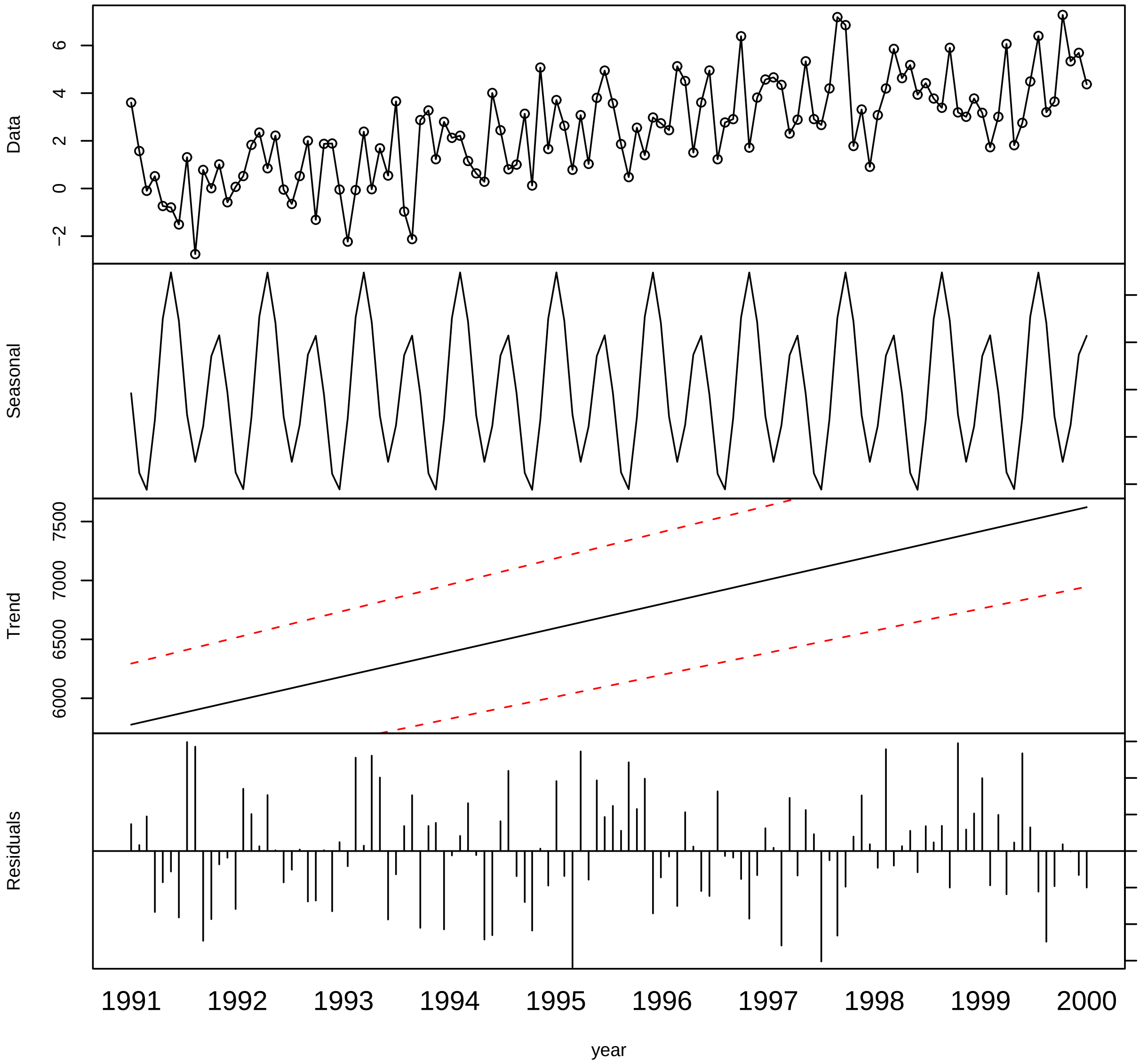


Figure 43: Example of overlaying figures

Manipulating Data

Sorting

order Function

Ordering is usually best done indirectly: Find an index vector that achieves the sort operation and use it for all vectors that need to remain together. The function `order` allows sorting with tie-breaking: Find an index vector that arranges the first of its arguments in increasing order. Ties are broken by the second argument and any remaining ties are broken by a third argument.

Example

```
> x <- sample(1:5, 20, rep=T)
> y <- sample(1:5, 20, rep=T)
> z <- sample(1:5, 20, rep=T)
> xyz <- rbind(x, y, z)
> dimnames(xyz)[[2]] <- letters[1:20]
> xyz
  a b c d e f g h i j k l m n o p q r s t
x 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
y 5 5 2 5 2 3 5 4 4 2 4 2 1 4 3 4 4 2 2 2
z 4 5 3 2 4 2 4 5 5 2 4 2 4 5 3 4 3 4 4 3
> o <- order(x, y, z)
> xyz[, o]
  m h j c i l e s p t f q d a g b r o k n
x 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5
y 1 4 2 2 4 2 2 2 4 2 3 4 5 5 5 5 2 3 4 4
z 4 5 2 3 5 2 4 4 4 3 2 3 2 4 4 5 4 3 4 5

> xyz # reminder
  a b c d e f g h i j k l m n o p q r s t
x 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
```

```
y 5 5 2 5 2 3 5 4 4 2 4 2 1 4 3 4 4 2 2 2
z 4 5 3 2 4 2 4 5 5 2 4 2 4 5 3 4 3 4 4 3
```

`sort` Function

The sort function can also be used to sort a vector or a list respectively into ascending or descending order

```
> sort(x)
[1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5
> sort(x,decreasing=T)
[1] 5 5 5 5 4 4 4 4 4 4 4 3 3 3 3 2 2 2 1 1
```

To sort a vector partially, use the partial argument

```
> sort(x,partial=c(3,4))
[1] 1 1 2 2 3 4 4 4 2 4 5 3 4 5 5 3 4 5 3 4
```

`rank` Function

To rank the values in a vector, the rank function can be used. Ties result in ranks being averaged by default but other options are available: taking the first occurrence, randomly selecting a value or selecting the maximum or minimum value.

```
> rank(x)
[1]  13.0 13.0 4.0 13.0  7.5 13.0 13.0  1.5 4.0  4.0
[11] 18.5  7.5 1.5 18.5 18.5  7.5 13.0 18.5 7.5 13.0
> rank(x, ties="first")     # first occurrence wins
 [1] 4 4 2 4 3 4 4 1 2 2 5 3 1 5 5 3 4 5 3 4
> rank(x, ties="random")     # ties broken at random
 [1] 16 15 5 14 9 12 11 1 3 4 19 6 2 17 18 8 13 20
[19] 7 10
> rank(x, ties="min")     # typical sports ranking
 [1] 10 10 3 10 6 10 10 1 3 3 17 6 1 17 17 6 10 17
[19] 6 10
```

Dates and Times

R has several mechanisms available for the representation of dates and times. The standard one however is the POSIXct/POSIXlt suite of functions and objects of (old) class POSIXct are numeric vectors with each component representing the number of seconds since the start of 1970. Such objects are suitable for inclusion in data frames, for example. Objects of (old) class POSIXlt are lists with the separate parts of the date/time held as separate components.

Conversion from one form to another

- The function `as.POSIXlt(obj)` converts from POSIXct to POSIXlt.
- The function `as.POSIXct(obj)` converts from POSIXlt to POSIXct.
- The function `strptime(char,form)` generates POSIXlt objects from suitable character string vectors where the format must be specified.
- The `format(obj,form)` generates character string vectors from POSIXlt or POSIXct objects, which also requires the output format to be specified.
- The function `as.character(obj)` also generates character string vectors like `format(,)`, but only to the ISO standard time/date format. For formatting details see help on `strptime`.

Example

On what day of the week were you born and for how many seconds have you lived?

```
> myBday <- strptime("18-Apr-1973", "%d-%b-%Y")

> class(myBday)

[1] "POSIXt"  "POSIXlt"

> myBday

[1] "1973-04-18"

> weekdays(myBday)

[1] Wednesday"


> Sys.time()

[1] "2005-01-19 12:08:12 E. Australia Standard Time"

> Sys.time() - myBday

Time difference of 11599.51 days
```

Arithmetic on POSIXt Objects

Some arithmetic operations are allowed on date/time objects (POSIXlt or POSIXct). These are

- obj + number
- obj - number
- obj1 <lop> obj2
- obj1 - obj2

In the first two cases, number represents the number of seconds and each date is augmented by this number of seconds. If you wish to augment by days you need to work with multiples of 60*60*24. In the second case <lop> is a logical operator and the result is a logical vector In the third case the result is a difftime object, represented as the number of seconds time difference.

Birthday example continued.

```
> as.numeric(Sys.time())
[1] 1106100492
> as.numeric(myBday)
[1]  0 0  0   18   3     73       3        107      0
> as.numeric(as.POSIXct(myBday))
[1] 103903200
> as.numeric(Sys.time()) - as.numeric(as.POSIXct(myBday))
[1] 1002197292
```

Tables

It can be quite useful to tabulate factors or find the frequency of an object. This can be achieved using the table function in R. The quine dataset consists of 146 rows describing childrens ethnicity (Eth), age (Age), sex (Sex), days absent from school (Days) and their learning ability (Lrn). Eth, Sex, Age and Lrn are factors. Days is a numeric vector. If we want to find out the age classes in the quine dataset we can do the following

```
> attach(quine)
> table(Age)
```

```
Age
F0 F1 F2 F3  # F0: primary, F1-F3: forms 1-3
27 46 40 33
```

If we need to know the breakdown of ages according to sex

```
>  table(Sex,Age)
   Age
Sex F0 F1 F2 F3
  F 10 32 19 19
  M 17 14 21 14
```

Split

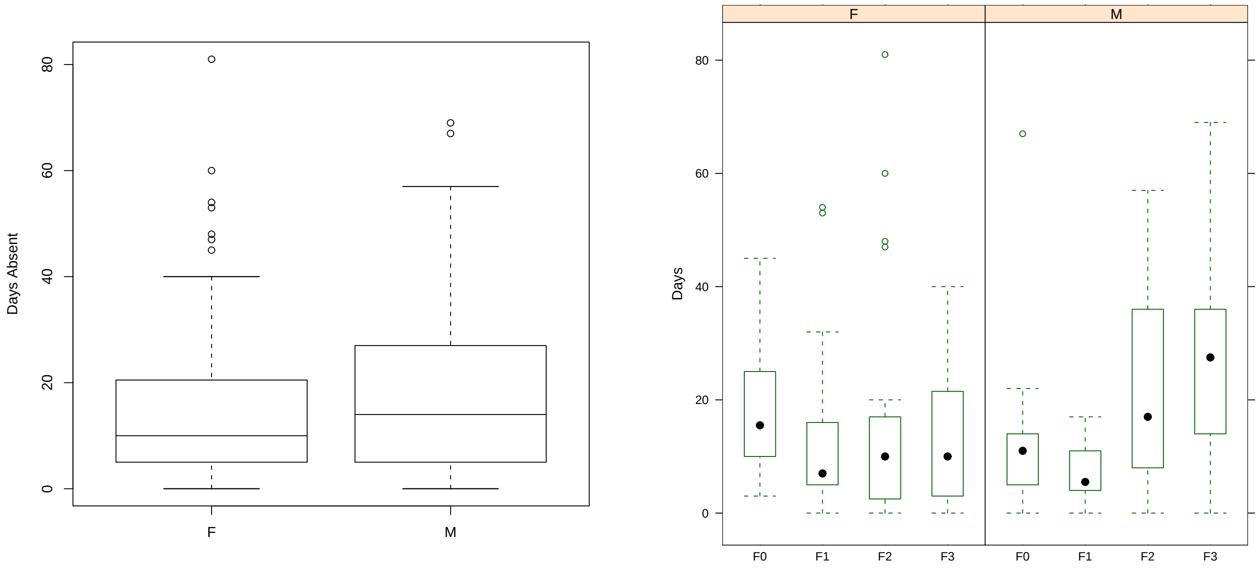
The split function divides the data specified by vector x into the groups defined by factor f This function can be useful for graphical displays of data If we want to obtain a summary of Days split by Sex we can write the following code.

```
> split(Days,Sex)
$F
[1]   3  5 11 24 45  5  6  6  9 13 23 25 32 53 54  5  5 11 17
[20] 19  8 13 14 20 47 48 60 81  2  0  2  3  5 10 14 21 36 40
[39] 25 10 11 20 33  5  7  0  1  5  5  5  5  7 11 15  5 14  6
[58]  6  7 28  0  5 14  2  2  3  8 10 12  1  1  9 22  3  3  5
[77] 15 18 22 37


$M
[1]   2 11 14  5  5 13 20 22  6  6 15  7 14  6 32 53 57 14 16
[20] 16 17 40 43 46  8 23 23 28 34 36 38  6 17 67  0  0  2  7
[39] 11 12  0  0  5  5  5 11 17  3  4 22 30 36  8  0  1  5  7
[58] 16 27  0 30 10 14 27 41 69
```

or for some nice graphical displays

```
> boxplot(split(Days,Sex),ylab="Days Absent")
> library(lattice) # trellis graphics
> trellis.par.set(col.whitebg())
> bwplot(Days ~ Age | Sex)  # implicit split
```



(a) (b)

Figure 44: Boxplots showing the breakdown of days for (a) males and females and (b) age classes.

`with`, `subset` and `transform` Functions

These functions operate on an object or elements within an object. No attachment of the dataset is necessary

- `with`: evaluates expressions constructed from the data

  ```
  > with(Cars93,plot(Weight,100/MPG.highway))
  ```

- `subset`: returns subsets of vectors or data frames that meet specific requirements

  ```
  > Vans <- subset(Cars93,Type=="Van")
  ```

- `transform`: transforms elements of an object

  ```
  > Cars93T <- transform(Cars93,WeightT=Weight/1000)
  ```

Vectorised Calculations

Those from a programming background may be used to operating on individual elements of a vector. However, R has the ability to perform vectorised calculations. This allows you to perform calculations on an entire vector/matrix/dataframe/list instead of the individual elements. Not all problems can be vectorised but most can once you know how.

Four members: `lapply`, `sapply`, `tapply`, `apply`

- `lapply`: takes any structure, gives a list of results
- `sapply`: like lapply, but simplifies the result if possible
- `apply`: only used for arrays
- `tapply`: used for ragged arrays: vectors with an indexing specified by one or more factors.

Those functions are used for efficiency and convenience.

The `apply` Function

This function allows functions to operate on successive sections of an array. To illustrate this, we compute the mean of each column of the iris data

```
> iris[1:4,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
> apply(iris[,-5],2,mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

The `tapply` Function

Ragged arrays represent a combination of a vector and a labelling factor or factors, where the group sizes are irregular. To apply functions to ragged arrays the tapply function is used, where the object, list of factors and function is supplied. We will illustrate this function using the quine dataset again

```
> quine[1:5,]
   Eth Sex Age Lrn Days
```

```
1   A   M  F0  SL    2
2   A   M  F0  SL   11
3   A   M  F0  SL   14
4   A   M  F0  AL    5
5   A   M  F0  AL    5
```

To calculate the average number of days absent for each age class we can use the tapply function

```
> tapply(Days,Age,mean)
      F0       F1       F2       F3
14.85185 11.15217 21.05000 19.60606
```

To perform this calculation for each gender we need to specify using the list function two factors: Sex and Age

```
> tapply(Days,list(Sex,Age),mean)
    F0       F1       F2       F3
F 18.70000 12.96875 18.42105 14.00000
M 12.58824  7.00000 23.42857 27.21429
```

The `lapply` and `sapply` Functions

- `lapply` and sapply operate on components of a list or vector
- `lapply` will always return a list
- `sapply` is a more user friendly version of lappy and will attempt to simplify the result into a vector or array

```
# Example 1: lapply
> l <- list(Sex=Sex,Eth=Eth)
> lapply(l,table)
$Sex
F  M
80 66
```

```
# Example 2: sapply
> l <- list(Sex=Sex,Eth=Eth)
> sapply(l,table)
   Sex  Eth
F   80   69
M   66   77
```

```
$Eth

A  N

69 77
```

The `apply` Functions versus Conventional Programming

Programming methods will be discussed in more detail in a later session but how do the family of apply functions compare with conventional programming methods? We can test the performance of these functions using the `system.time` function.

We consider the following problem: We wish to subtract the mean from each element in a 25000 x 4 matrix

```
> mat <- matrix(rnorm(100000),ncol=4)
```

Program 1: The programmer's approach (5 for loops)

```
program1 <- function(mat){
  col.scale <- matrix(NA,nrow(mat),ncol(mat))
  m <- rep(0,ncol(mat))                  # create a vector, m and
                                         #   set it to zero
  for(j in 1:ncol(mat)){
      for(i in 1:nrow(mat)) {
          m[j] <- m[j] + mat[i,j]        # compute the sum of
                                         #   elements in each
                                         #   column
      }
  }
  for(j in 1:ncol(mat))
      m[j] <- m[j]/nrow(mat)             # compute the mean
  for(i in 1:nrow(mat)){
      for(j in 1:ncol(mat)){
          col.scale[i,j] <- mat[i,j]-m[j] # centre each column by
                                         #  the mean
      }
```

```
   }
  col.scale                               # print the scaled matrix
}
```

Program 2: The programmer's approach (using a built-in function) (3 for loops)

```
program2 <- function(mat){
  col.scale <- matrix(NA,nrow(mat),ncol(mat))
  m <- NULL                               # initialise the vector,m
  for(j in 1:ncol(mat))
      m[j] <- mean(mat[,j])               # compute the mean of
                                          #   each column
  for(i in 1:nrow(mat)){
      for(j in 1:ncol(mat)){
          col.scale[i,j] <- mat[i,j]-m[j] # centre columns
      }
  }
  col.scale                               # print the scaled matrix
}
```

Program 3: R programming approach (No for loops)

```
program3 <- function(mat){
    apply(mat,2,scale,scale=F)  # apply to the matrix (mat)
                                #   the function scale
                                #   specifying that
                                #   centring only be
                                #   performed.
                                # print the scaled matrix
}
```

How does each program perform? The `system.time` function produces a vector of length 5 containing information about (1) user CPU, (2) system CPU, (3) Elapsed time, (4) subprocessor 1 time and (5) subprocessor 2 time. Only the first three are of real interest.

```
> system.time(v1 <- program1(mat)) # Slowest
[1]  2.33 0.04 2.37
> system.time(v2 <- program2(mat)) # 3.5x increase in CPU
[1]  0.68 0.00 0.72
> system.time(v3 <- program3(mat)) # 78x increase in CPU
[1]  0.03 0.00 0.03
> system.time(v4 <- scale(mat,scale=F)) # 233x increase in CPU
[1]  0.01 0.00 0.01
> check <- function(v1,v2) abs(diff(range(v1-v2))) # check
> check(v1,v2) + check(v2,v3) + check(v3,v4)
[1] 4.440892e-16
```

Which approach would you use?

Classical Linear Models

Statistical Models in R

The expression for fitting a multiple linear regression model is shown in the following equation. Here, y_i is the response recorded at the i-th observation, x_{ij} is the j-th explanatory variable recorded for the i-th observation, β_j are the regression coefficients and e_i is the error term that is independent and identically distributed (iid).

$$\begin{aligned} y_i &= \sum_{j=1}^{p} x_{ij}\beta_j + e_i \\ \text{where} & \\ e_i &\sim \text{NID}(0, \sigma^2) \end{aligned} \tag{1}$$

In matrix terms this expression would be written as

$$y = X\beta + e$$

where y is the response vector, β is the vector of regression coefficients, X is the model matrix or *design matrix* and e is the error vector.

Model Formulae

In R, the general form of this expression is: $\text{response} \sim \text{term}_i \pm \text{term}_2 \pm \ldots$. Extensions to this are summarised in Table 6. In this table, x refers to continuous variables while G refers to categorical variables. We will be using some of these expressions in the next few sessions of the course.

Generic Functions for Inference

Once the model is fitted in R, we examine the fit. The most common functions are described in Table 7.

Table 6: Model Formulae

Expression	Description
`y` $\sim$ `x`	Simple regression
`y` $\sim$ `1+x`	Explicit intercept
`y` $\sim$ `-1 + x`	Through the origin
`y` $\sim$ `x +` x^2	Quadratic regression
`y` $\sim$ `x1 + x2 + x3`	Multiple regression
`y` $\sim$ `G + x1 + x2`	Parallel regressions
`y` $\sim$ `G/(x1+x2)`	Separate regressions
`sqrt(y)` $\sim$ `x +` x^2	Transformed
`y` $\sim$ `G`	Single Classification
`y` $\sim$ `A+B`	Randomized block
`y` $\sim$ `B+N*P`	Factorial in blocks
`y` $\sim$ `x+B+N*P`	with covariate
`y` $\sim$ `.-X1`	All variables except `X1`
`.` $\sim$ `.+A:B`	Add interaction (update)
`Nitrogen` $\sim$ `Times*(River/Site)`	More complex design

Table 7: Common functions for inference

Expression	Description
`coef(obj)`	regression coefficients
`resid(obj)`	residuals
`fitted(obj)`	fitted values
`summary(obj)`	analysis summary
`predict(obj,newdata=ndat)`	predict for new data
`deviance(obj)`	residual sum of squares

Example: The Janka Data

Description and Exploratory Analysis

We illustrate linear modelling using a very simple example, the Janka Hardness data. This data is described in Appendix I and contains information on the hardness and density of 36 samples of Australian hardwoods. We are interested in building a prediction model for hardness using density.

Figure 45 is a plot of hardness versus density. It was produced using the following piece of code.

```
> janka <- read.csv("janka.csv")
> with(janka,plot(Density,Hardness,col="blue"))
```

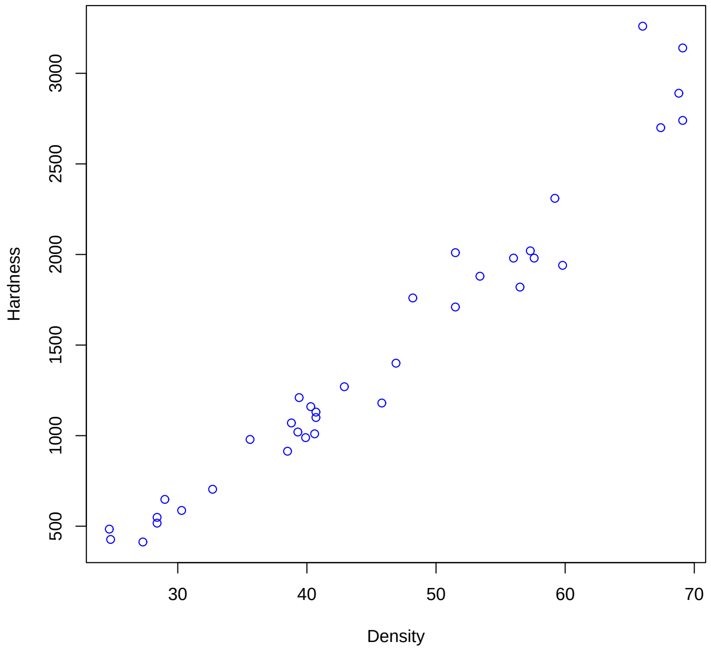


Figure 45: Hardness versus Density (Janka)

Initial Model Building

To begin with, we start with a linear or quadratic model (suggested by Williams) and start examining the fit. Note how the function I is used in the specification of this model.

This function is used to prevent the operator, ∧ being used as a formula operator. When used in a formula, the expression inside the brackets is evaluated first prior to the formula being evaluated.

```
> jank.1 <- lm(Hardness ~ Density, janka)
> jank.2 <- update(jank.1, . ~ . + I(Density^2))
> summary(jank.2)$coef
      Estimate  Std. Error    t value    Pr(>|t|)
(Intercept)  -118.0073759 334.9669049 -0.3522956 0.726856611
Density         9.4340214  14.9356203  0.6316458 0.531969926
I(Density^2)    0.5090775   0.1567210  3.2483031 0.002669045
```

The model suggests that a quadratic term for density is appropriate for this dataset. We should however, also check the need for a cubic term in the model. To achieve this, we simply add one more term to the model as follows.

```
> jank.3 <- update(jank.2, . ~ . + I(Density^3))
> summary(jank.3)$coef
        Estimate   Std. Error    t value  Pr(>|t|)
(Intercept)  -6.414379e+02 1.235655e+03 -0.5191076 0.6072576
Density       4.686373e+01 8.630180e+01  0.5430215 0.5908777
I(Density^2) -3.311734e-01 1.913986e+00 -0.1730281 0.8637192
I(Density^3)  5.958701e-03 1.352646e-02  0.4405220 0.6625207
```

The results indicate that a quadratic term is necessary, but a cubic term is not supported.

The regression coefficients should remain more stable under extensions to the model if we standardize, or even just mean-correct, the predictors. We can do this by subtracting off the mean as follows

```
> janka <- transform(janka,d=Density-mean(Density))
> jank.1 <- lm(Hardness ~ d, janka)
> jank.2 <- update(jank.1, .~.+I(d^2))
> jank.3 <- update(jank.2, .~.+I(d^3))
> summary(jank.1)$coef
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) 1469.472    30.5099  48.164        0
          d   57.507     2.2785  25.238        0
> summary(jank.2)$coef
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 1378.19661   38.93951 35.3933 0.000000
          d   55.99764    2.06614 27.1026 0.000000
     I(d^2)    0.50908    0.15672  3.2483 0.002669
> round(summary(jank.3)$coef, 4)
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1379.1028    39.4775 34.9339   0.0000
          d   53.9610     5.0746 10.6336   0.0000
     I(d^2)    0.4864     0.1668  2.9151   0.0064
     I(d^3)    0.0060     0.0135  0.4405   0.6625
```

Why is this so? Does it matter very much?

Centering results in properties of the coefficients that lie in the middle of the x range. When unentered, the coefficients have properties at unrealistic values of the density, near 0. In the middle of the x-range, predictions under any model are pretty stable but at density 0 they tend to fluctuate wildly. Therefore, always select a parametrisation of your model where the parameters mean something and where the estimates of the fitted function are stable.

Diagnostic Checks

We examine the fit of the model using a plot of residuals and a Normal scores plot shown in Figures 46 and 47 respectively. These residual plots are plotted using the trellis library, which will be discussed in more detail in Session 10.

```
> trellis.par.set(col.whitebg())
> require(MASS)
> xyplot(studres(jank.2) ~ fitted(jank.2),
aspect = 0.6,
   panel = function(x, y, ...) {
      panel.xyplot(x, y, col = "navy", ...)
      panel.abline(h = 0, lty = 4, col = "red")
  }, xlab = "Fitted values", ylab = "Residuals")

> qqmath(˜ studres(jank.2), panel =
function(x, y, ...) {
   panel.qqmath(x, y, col = "navy", ...)
   panel.qqmathline(y, qnorm, col = "red", lty=4)
  }, xlab = "Normal scores", aspect = 0.6,
        ylab = "Sorted studentized residuals")
```

The plot shows some departures from the Normality assumptions. This is particularly evident in the Normal scores plot shown in Figure 47 as the edges of the plot deviate from the theoretical line. We may need to consider a transformation for this dataset.

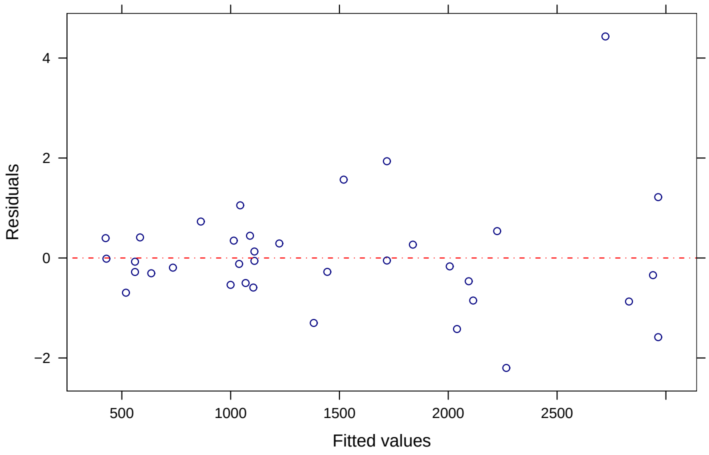


Figure 46: Residuals versus Fitted Values (janka)

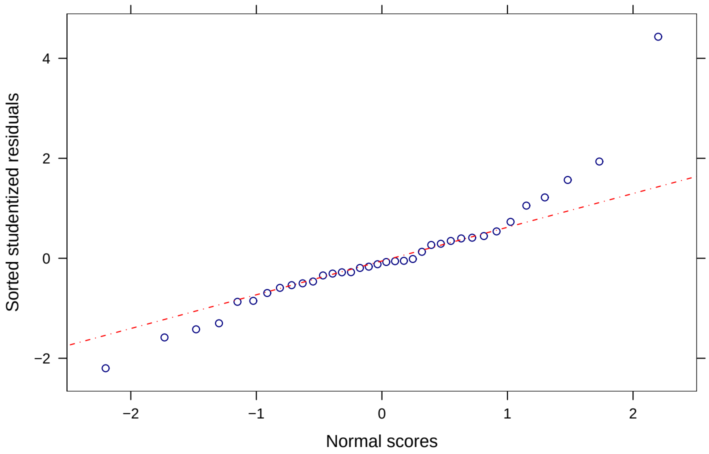


Figure 47: Sorted Studentized Residuals versus Normal Scores (janka)

Transformation

The Box-Cox family of transformations includes square-root and log transformations as special cases. The boxcox function in the MASS library allows the marginal likelihood function for the transformation parameter to be calculated and displayed. Its use is easy. (Note: it only applies to positive response variables.)

The following is an example of how to do this.

```
> require(MASS)     # necessary if the MASS library
    # has not been attached
> graphsheet()     # necessary if no graphics
    # device open.
> boxcox(jank.2,lambda = seq(-0.25, 1, len=20))
```

Figure 48 displays the marginal likelihood plot produced on the Janka fitted model object `jank.2`. The plot shows that an appropriate value for λ is approximately 0.17, with 95% confidence intervals that include zero. The plot indicates that a log transformation may be suitable for this data.

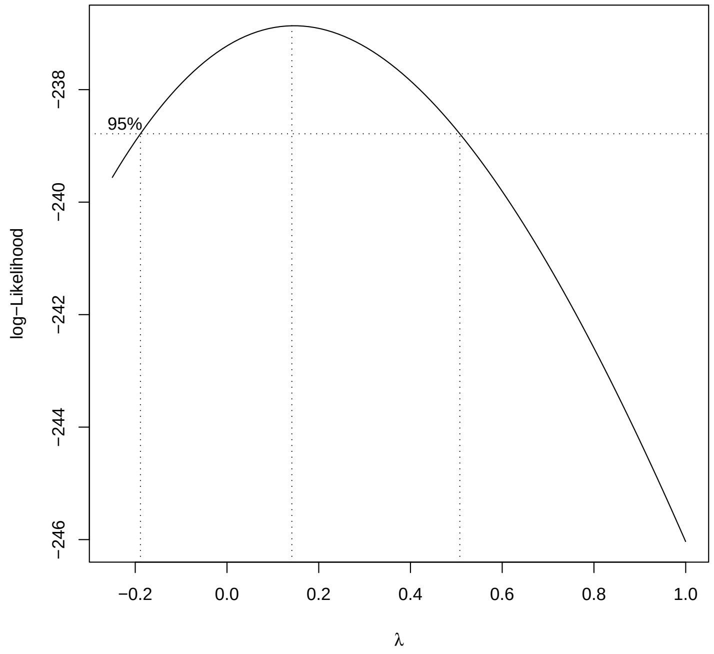


Figure 48: Box-Cox Transformation (janka)

A plot of the transformed data using the script below is shown in Figure 49. A quadratic relationship seems reasonable for this data under this transformation.

```
> with(janka,plot(Density, Hardness, log = "y"))
```

We now refit this model using a log transformation on the response, compute the residuals and fitted values and produce a corresponding plot of residuals, which is shown in Figure 50. The residual plot indicates a fairly good fit with no departure from Normal distribution assumptions.

```
> ljank.2 <- update(jank.2, log(.)˜.)
> round(summary(ljank.2)$coef, 4)
            Estimate Std. Error   t value Pr(>|t|)
(Intercept)  7.2299     0.0243  298.0154        0
          d  0.0437     0.0013   33.9468        0
     I(dˆ2) -0.0005     0.0001   -5.3542        0
> lrs <- studres(ljank.2)
> lfv <- fitted(ljank.2)
```

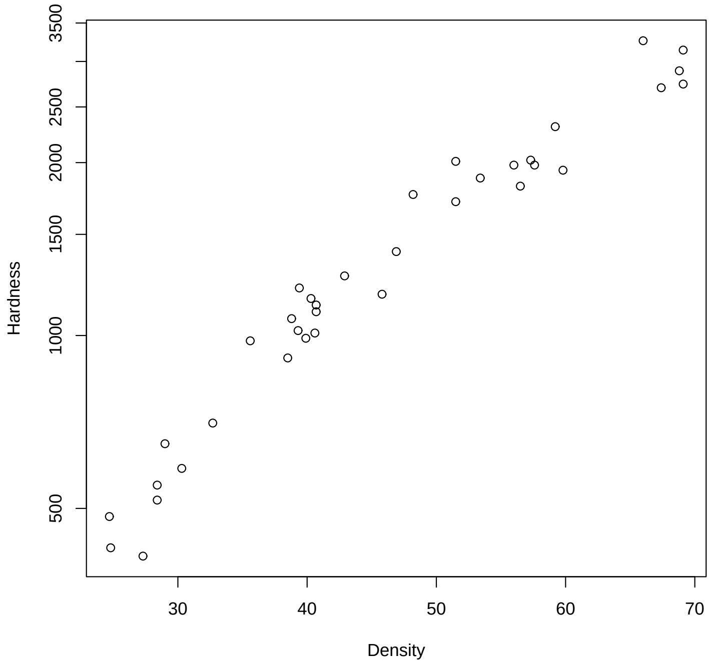


Figure 49: Plot of the transformed data (janka)

```
> xyplot(lrs ~ lfv, panel =
    function(x, y, ...) {
    panel.xyplot(x, y, pch=16,...)
    panel.abline(h=0, lty=4)
    }, xlab = "Fitted (log trans.)",
    ylab = "Residuals (log trans.)", col = "red")
```

Example: Iowa Wheat Yield Data

We use the Iowa Wheat Yield data as a second illustration of linear regression modelling. This dataset is described in Appendix I and contains information on the yield of wheat taken at different times throughout the year under different weather conditions.

```
> library(RODBC)
> iowheat <-sqlFetch(odbcConnectExcel("Iowa.xls"), "Iowa")
```

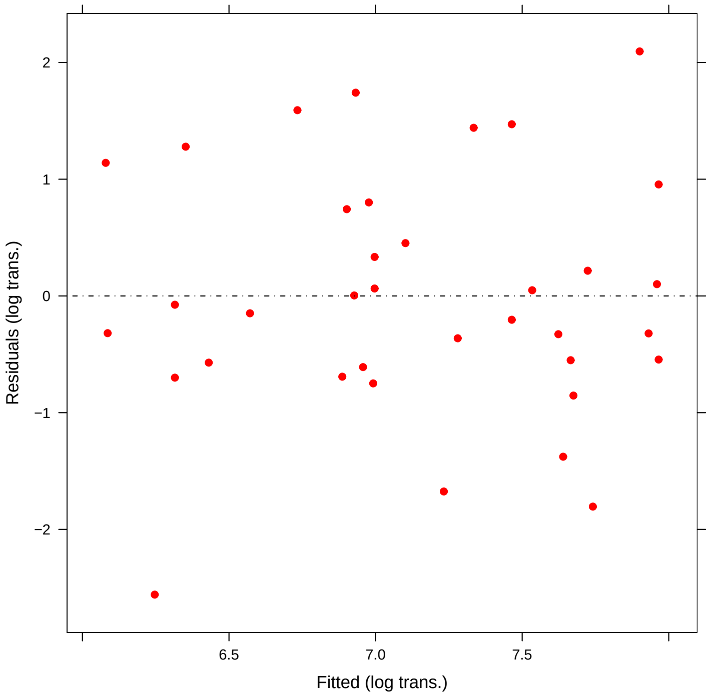


Figure 50: Log transformed residuals versus log transformed fitted values (janka)

```
> names(iowheat)
 [1] "Year"  "Rain0" "Temp1" "Rain1" "Temp2"
 [6] "Rain2" "Temp3" "Rain3" "Temp4" "Yield"
> bigm <- lm(Yield ˜ ., data = iowheat)
```

The above piece of code reads in the Iowa Wheat Yield dataset and fits a regression model using *all of the other variables* in the data frame as predictors.

From the full model, we now examine the effect of dropping each term individually using the `dropterm` function. The `addterm` function looks at the impact of adding in terms from a smaller model in a stepwise fashion.

```
> dropterm(bigm, test = "F")
Single term deletions


Model:
Yield ˜ Year + Rain0 + Temp1 + Rain1 + Temp2 + Rain2 +
  Temp3 + Rain3 + Temp4
```

```
       Df Sum of Sq    RSS    AIC F Value   Pr(F)
<none>              1404.8 143.79
  Year  1    1326.4 2731.2 163.73  21.715 0.00011
 Rain0  1     203.6 1608.4 146.25   3.333 0.08092
 Temp1  1      70.2 1475.0 143.40   1.149 0.29495
 Rain1  1      33.2 1438.0 142.56   0.543 0.46869
 Temp2  1      43.2 1448.0 142.79   0.707 0.40905
 Rain2  1     209.2 1614.0 146.37   3.425 0.07710
 Temp3  1       0.3 1405.1 141.80   0.005 0.94652
 Rain3  1       9.5 1414.4 142.01   0.156 0.69655
 Temp4  1      58.6 1463.5 143.14   0.960 0.33738

> smallm <- update(bigm, . ~ Year)
> addterm(smallm, bigm, test = "F")
Single term additions

Model:
Yield ~ Year
       Df Sum of Sq    RSS    AIC F Value   Pr(F)
<none>              2429.8 145.87
 Rain0  1    138.65 2291.1 145.93  1.8155 0.18793
 Temp1  1     30.52 2399.3 147.45  0.3816 0.54141
 Rain1  1     47.88 2381.9 147.21  0.6031 0.44349
 Temp2  1     16.45 2413.3 147.64  0.2045 0.65437
 Rain2  1    518.88 1910.9 139.94  8.1461 0.00775
 Temp3  1    229.14 2200.6 144.60  3.1238 0.08733
 Rain3  1    149.78 2280.0 145.77  1.9708 0.17063
 Temp4  1    445.11 1984.7 141.19  6.7282 0.01454
```

Alternatively, we could automate the process to eliminate variables in a backwards fashion. Notice how different approaches can lead to different solutions.

```
> stepm <- stepAIC(bigm,
scope = list(lower = ~ Year))
Start:  AIC= 143.79
....
Step:  AIC= 137.13
 Yield ~ Year + Rain0 + Rain2 + Temp4

        Df Sum of Sq    RSS    AIC
 <none> NA        NA 1554.6 137.13
- Temp4  1    187.95 1742.6 138.90
- Rain0  1    196.01 1750.6 139.05
- Rain2  1    240.20 1794.8 139.87
```

A Flexible Regression

The clearly useful predictor is `Year`, but `Rain0`, `Rain2` and `Temp4` may have some value. Based on these results, we now consider a flexible version of the linear regression using splines. The following script fits natural splines to `Rain0`, `Rain2`, `Temp4` and `Year`, each with 3 knots. Figure 51 displays the spline fits to each of the terms in the model. This plot was produced using `termplot`.

```
> require(splines)
> iowa.spline <- aov(Yield ~ ns(Rain0, 3) + ns(Rain2, 3) +
        ns(Temp4, 3) + ns(Year, 3),iowheat)
> par(mfrow=c(2,2))
> termplot(iowa.spline, se = TRUE, rug = TRUE,
partial=TRUE)
```

The plots, although interesting can be somewhat deceiving in terms of examining their contribution to the model as the scale of each plot is different for different variables. A more useful plot is one where the limits of the y-axis for each plot is constrained. The `tplot` function created by Bill Venables, is a variation on the `termplot` function. By constraining the y-axis we are able to see which predictors are really contributing to the fit of the model. Figure 52 displays the results. The big contributor to the model appears

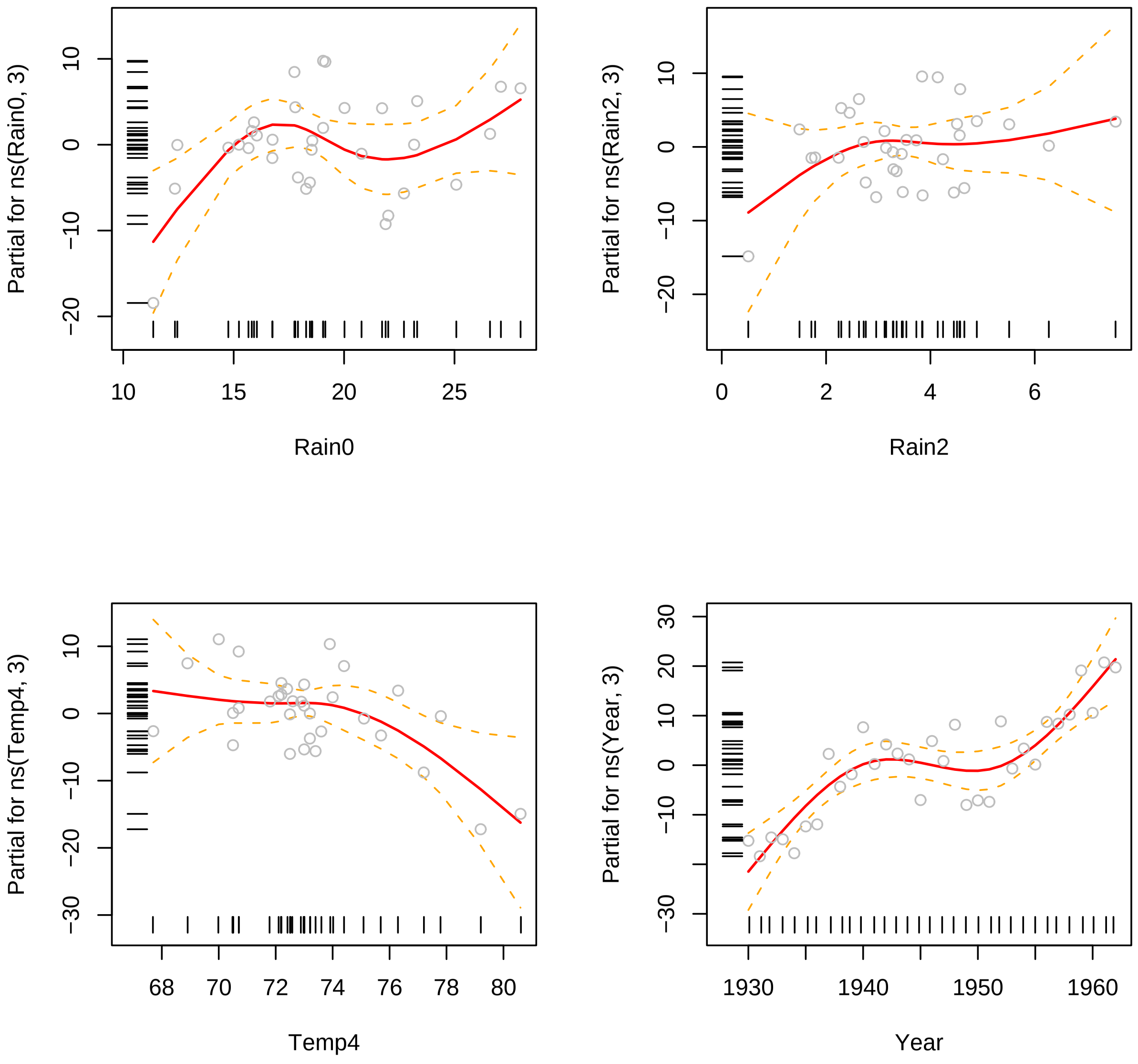


Figure 51: Plot of spline terms in regression model using the `termplot()` function.

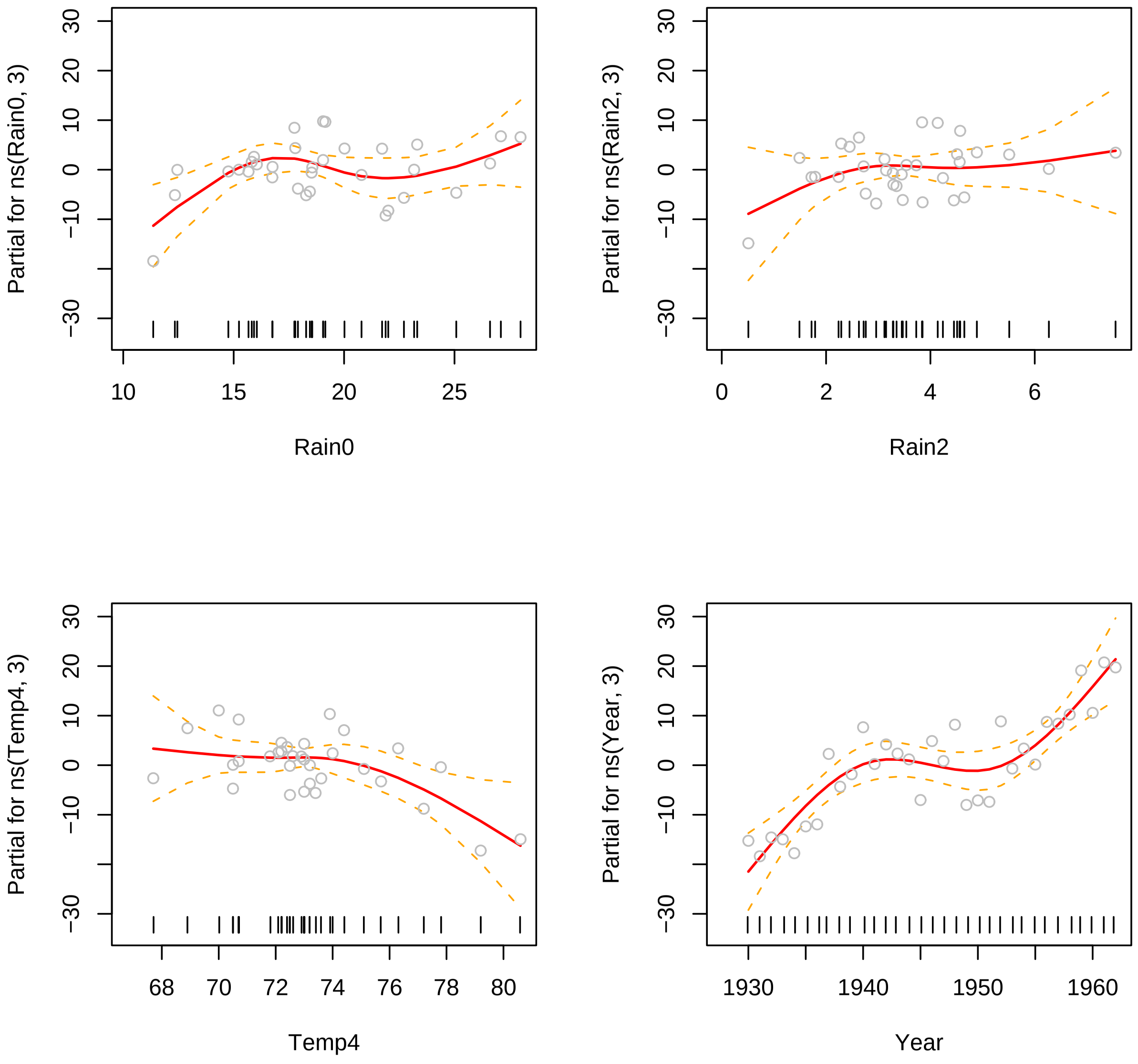


Figure 52: Plot of spline terms in regression model using the `tplot()` function

to be Year, showing an unusual drop and leveling off in yield just after 1945 before rising again after 1950. We think that this result is related to the impact of the war.

Example: Petroleum Data

Overview and Description

The petroleum data of Nilon L Prater is described in Appendix I and will be touched upon in a later session when we discuss random effects modelling in greater detail. The dataset consists of measurements on ten crude oil sources. Subsamples of crude oil (3-5) are refined to a certain end point and this is measured. The response is the yield of refined petroleum (as a percentage).

The question we are investigating is: How can petroleum yield be predicted from properties of crude and end point?

Exploratory Analysis

For this kind of grouped data a Trellis display, by group, with a simple model fitted within each group is often very revealing. Figure 53 displays the resulting plot.

```
> names(petrol)
[1] "No"  "SG"  "VP"  "V10" "EP"  "Y"
> xyplot(Y ~ EP | No, petrol, as.table = T,
    panel = function(x, y, ...) {
     panel.xyplot(x, y, ...)
      panel.lmline(x, y, ...)
            }, xlab = "End point", ylab = "Yield (%)",
         main = "Petroleum data of N L Prater")
```

Fixed Effects Model

Clearly a straight line model is reasonable. There is some variation between groups, but parallel lines is also a reasonable simplification. There also appears to be considerable variation between intercepts, though.

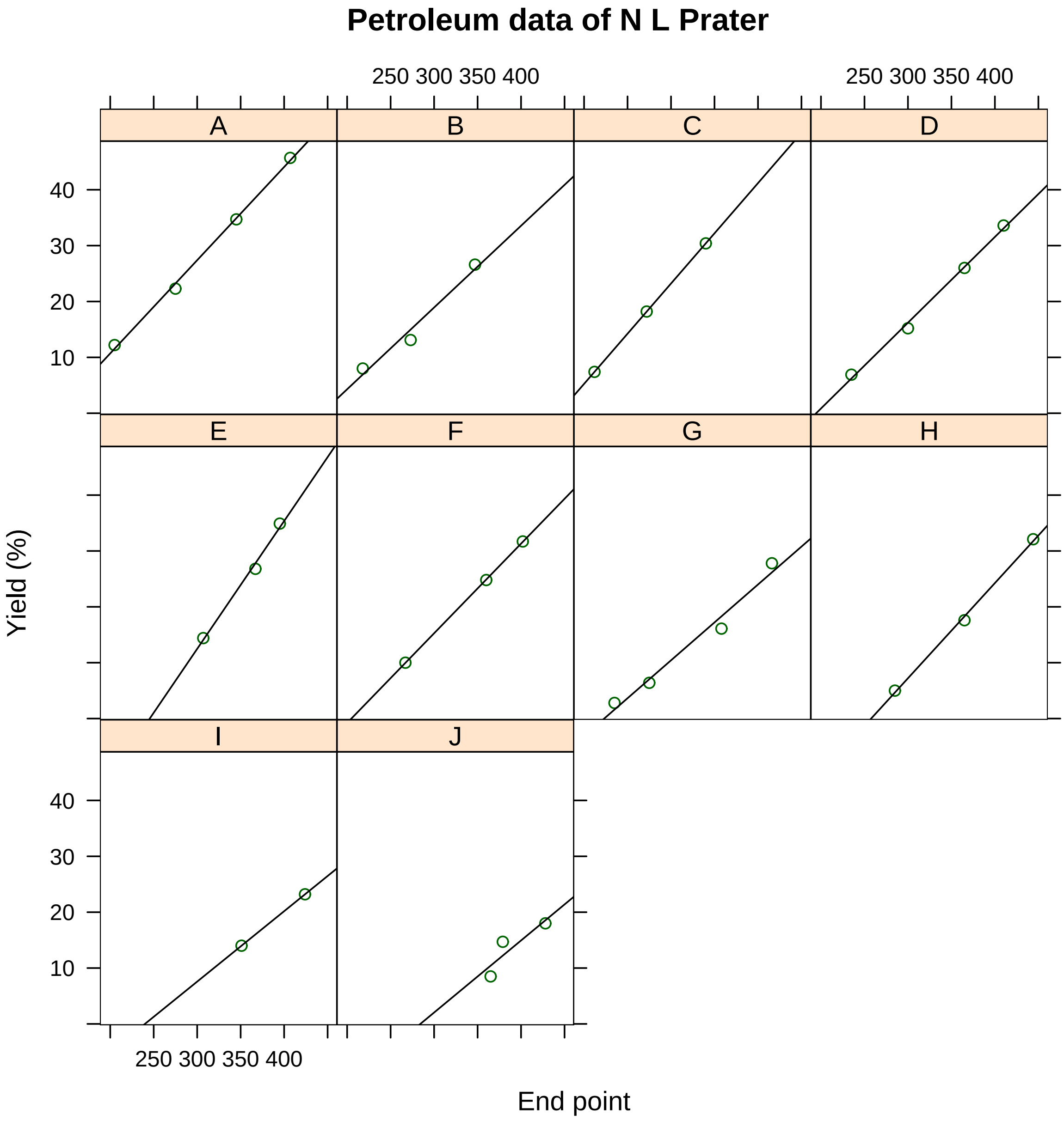


Figure 53: Plot of the data with a least squares line fitted.

We examine three types of models: (1) A model with interactions between crude oil and end point (pet.2), (2) a model with main effects terms only (pet.1) and (3) a model with just end point fitted (pet.0).

```
> pet.2 <- aov(Y ~ No*EP, petrol)
> pet.1 <- update(pet.2, .~.-No:EP)
> pet.0 <- update(pet.1, .~.-No)
> anova(pet.0, pet.1, pet.2)
Analysis of Variance Table


Model 1: Y ~ EP
Model 2: Y ~ No + EP
Model 3: Y ~ No * EP
  Res.Df     RSS Df Sum of Sq       F    Pr(>F)
1     30 1759.69
2     21   74.13  9   1685.56 74.1008 3.912e-09
3     12   30.33  9     43.80  1.9257    0.1439
```

The analysis of variance table reveals a substantial improvement in the fit of the model when main effect terms for both crude oil and end point are included in the model. Little improvement is noted when interaction terms are fitted as well.

Random Effects Model

Although we discuss random effects models a little later in the course, it is useful to touch on the topic here as there does appear to be some variation between intercepts (crude oil type) and slopes (end point).

We fit the random effects model using the lme function which is part of the nlme library. The result is shown below and indicates substantial variability between intercepts and minimal variation between slopes.

```
> require(nlme)
> pet.re1 <- lme(Y ~ EP, petrol, random = ~1+EP|No)
> summary(pet.re1)
.....
```

```
       AIC      BIC    logLik
  184.8296 193.2367 -86.41478

Random effects:
 Formula: ~1 + EP | No
 Structure: General positive-definite, Log-Cholesky parametrization
            StdDev     Corr
(Intercept) 5.05889184 (Intr)
EP          0.01139653 0.709
Residual    1.75086332

Fixed effects: Y ~ EP
                Value Std.Error DF   t-value p-value
(Intercept) -32.10612  2.414396 21 -13.29778       0
EP            0.15486  0.006434 21  24.06915       0
 Correlation:
   .....
```

The Shrinkage Effect

Having the random effect terms in the model offers some stability in the model. If we look at a plot of the least squares line (green) and fitted values from the random effects model (navy) in Figure 54 we see that the navy lines are much more stable than the green.

```
> B <- coef(pet.re1)
> B
 (Intercept)        EP
A   -24.03882 0.1707089
B   -28.89570 0.1598992
C   -27.19826 0.1666324
D   -30.91653 0.1566529
```

```
E   -31.52066 0.1598700
F   -31.54272 0.1565821
G   -34.07130 0.1474257
H   -36.12921 0.1495958
I   -36.91514 0.1436641
J   -39.83281 0.1375851
```

```
> xyplot(Y ˜ EP | No, petrol, as.table = T, subscripts = T,
    panel = function(x, y, subscripts, ...) {
     panel.xyplot(x, y, ...)
     panel.lmline(x, y, col="navy", ...)
     wh <- as(petrol$No[subscripts][1], "character")
     panel.abline(B[wh, 1], B[wh, 2], col = "green")
    }, xlab = "End point", ylab = "Yield (%)")
```

General Notes on Modelling

Some final notes . . .

1. Analysis of variance models are linear models but are usually fitted using aov rather than lm. The computation is the same but the resulting object behaves differently in response to some generics, especially summary.
2. Generalized linear modelling (logistic regression, log-linear models, quasi-likelihood etc) also use linear modelling formulae in that they specify the model matrix, not the parameters. Generalized additive modelling (smoothing splines, loess models etc) formulae are quite similar.
3. Non-linear regression uses a formula, but a completely different paradigm: the formula gives the full model as an expression, including parameters.

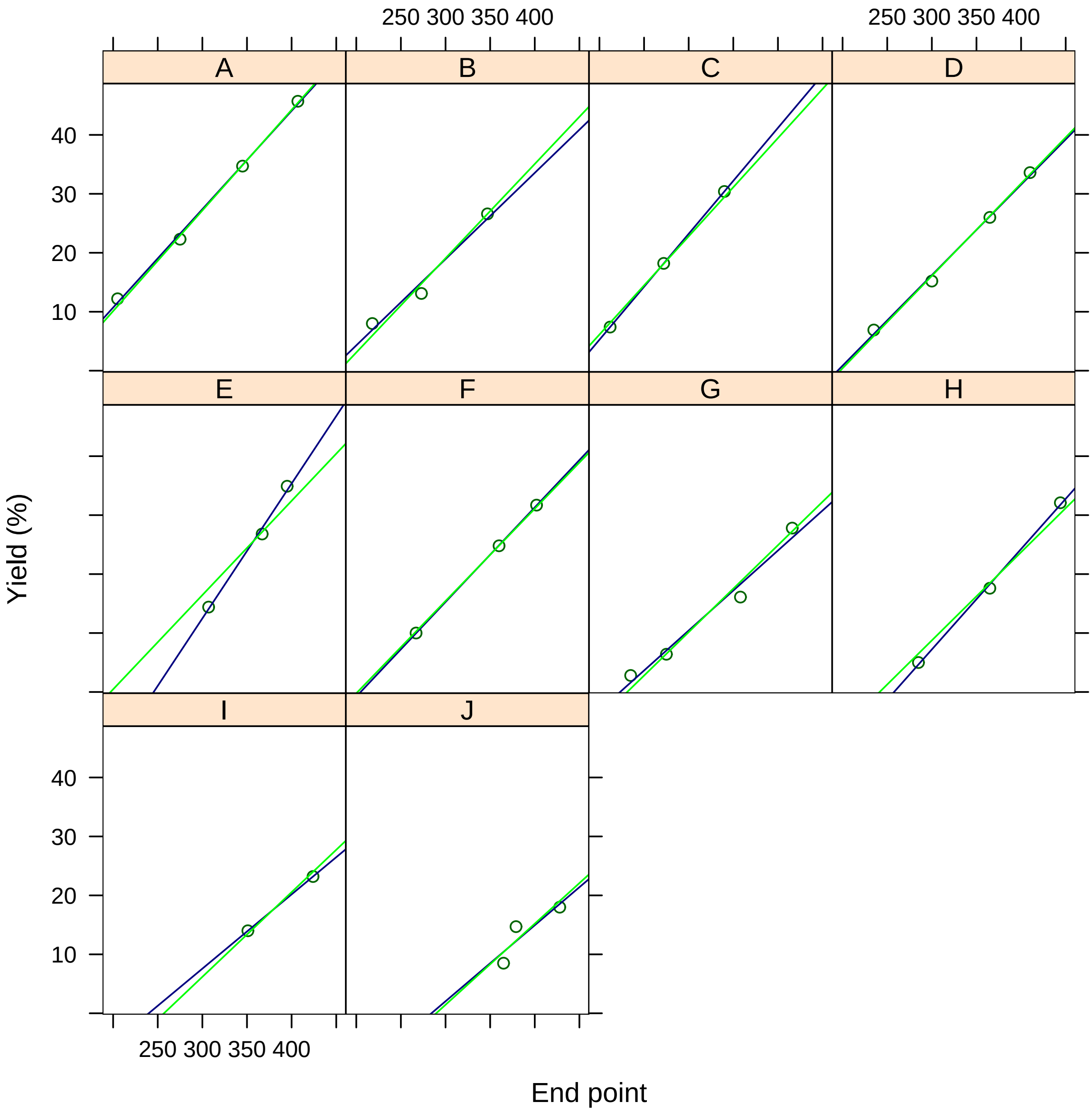


Figure 54: Plot of the data with a least squares (green) and random effect (navy) lines overlayed. Note how the random effect lines are less variable than the separate least squares lines.

Non-Linear Regression

Non-linear regression is a generalization of linear regression that was discussed in the last session. Normality and equal variance are retained but the linearity of parameters are relaxed somewhat.

Generally, non-linear regression arises from a fairly secure theory but it is not often appropriate for empirical work.

Estimation is still by least squares, that is, maximum likelihood, but the sum of squares surface is not necessarily quadratic.

The theory behind non-linear regression is approximate and relies on the sum of squared surface being nearly quadratic in a region around the minimum.

Example: Stormer Viscometer Data

We illustrate the concept of non-linear regression using an example, the Stormer Viscometer data.

Overview and Description

The stormer viscometer measures the viscosity of a fluid by measuring the time taken for an inner cylinder in the mechanism to perform a fixed number of revolutions in response to an actuating weight. The viscometer is calibrated by measuring the time taken with varying weights while the mechanism is suspended in fluids of accurately known viscosity. The data comes from such a calibration, and theoretical considerations suggest a non-linear relationship between time, weight and viscosity of the form

$$T = \frac{\beta \upsilon}{W - \theta} + \varepsilon \tag{2}$$

where β and θ are unknown parameters to be estimated and ε represents the error term. Appendix I describes the data in more detail.

Ignoring the error term and re-arranging the above expression gives

$$WT \simeq \beta \upsilon + \theta T \tag{3}$$

We consider fitting this relationship with ordinary least squares to get initial values for β and θ before launching into a non-linear regression analysis.

Fitting the Model and Looking at the Results

Fitting a linear model is very easy for this example. The first few lines extract the coefficients from the least squares fit. These represent starting values for the non-linear model.

The nls function is then used to fit the non-linear model expressed above using the starting values from the previous model. The results are presented in the summary below and indicate that both β and θ are significant terms in the model.

```
> b <- coef(lm(Wt*Time ˜ Viscosity + Time - 1,stormer))
> names(b) <- c("beta", "theta")
> b
  beta  theta
28.876 2.8437
> storm.1 <- nls(Time ˜ beta*Viscosity/(Wt - theta),
stormer, start=b, trace=T)
885.365 : 28.8755 2.84373
825.110 : 29.3935 2.23328
825.051 : 29.4013 2.21823

> summary(storm.1)
Formula: Time ˜ (beta * Viscosity)/(Wt - theta)

Parameters:
        Value Std. Error t value
 beta 29.4013     0.9155  32.114
theta  2.2183     0.6655   3.333

Residual standard error: 6.268 on 21 degrees of freedom

Correlation of Parameter Estimates:
       beta
```

```
theta -0.92
```

Self Starting Models

Self starting models allow the starting procedure to be encoded into the model function. This may seem somewhat arcane but very powerful.

The `selfStart` function takes on a `model` argument that defines a nonlinear model or a nonlinear formula, an `initial` argument, a function object that takes three arguments: `mCall`, `data` and `LHS` that represent a matched call to the function model, a data frame to interpret the variables in `mCall` and the expression from the left and side of the model formula in the call to `nls`, and a `parameters` argument which is a character vector specifying the terms on the right hand side of the model which are to be calculated. Refer to the help on self starting models for more information.

The `eval` function in the script below evaluates an expression, in this case `mCall` and `LHS`, two arguments passed to the `storm.init` function.

```
> require(MASS)
> storm.init <- function(mCall, data, LHS) {
      v <- eval(mCall[["V"]], data)
      w <- eval(mCall[["W"]], data)
      t <- eval(LHS, data)
      b <- lsfit(cbind(v, t), t * w, int = F)$coef
      names(b) <- mCall[c("b", "t")]
      b
    }
> NLSstormer <- selfStart( ~ b*V/(W-t),storm.init, c("b","t"))
> args(NLSstormer)
function(V, W, b, t)
NULL ...

> tst <- nls(Time ~ NLSstormer(Viscosity, Wt, beta,
theta), stormer, trace = T)
885.365 : 28.8755 2.84373
```

```
825.110 : 29.3935 2.23328
825.051 : 29.4013 2.21823
```

We can think about bootstrapping the model to obtain standard errors of the estimates. We can achieve this by constructing a matrix called B consisting of 500 rows and two columns. We then mean correct the residuals from the fitted model, sample from the residuals 500 times and add these residuals to the fitted values from the model. We then update the model using these new values as the response. To avoid any failures, we use the `try` function. `Try` evaluates an expression and traps any errors that may occur from the function that is being run.

We can then compute the means and standard deviations across the 500 bootstrap samples.

```
> tst$call$trace <- NULL
> B <- matrix(NA, 500, 2)
> r <- scale(resid(tst), scale = F)      # mean correct
> f <- fitted(tst)
> for(i in 1:500) {
    v <- f + sample(r, rep = T)
    B[i, ] <- try(coef(update(tst, v ˜ .))) # guard!
}
> cbind(Coef = colMeans(B), SD = sd(B))
          Coef        SD
[1,] 29.353547 0.8145761
[2,]  2.243293 0.5910415
```

Alternatively we could calculate the standard errors using a parametric approach as follows:

```
> cbind(Coef = coef(tst), SD = sqrt(diag(vcov(tst))))
           Coef        SD
beta  29.401257 0.9155336
theta  2.218274 0.6655216
```

A Bayesian Bootstrap is yet another alternative.

```
> b <- c(b = 29.1, th=2.21)
> n <- nrow(stormer)
> B <- matrix(NA, 1000, 2)
>
> for(i in 1:1000) {
    w <- rexp(n)
    B[i, ] <- try(coef(
     nls(~sqrt(w)*(Time - b*Viscosity/(Wt - th)),
     data = stormer, start = b)))
  }
>
> cbind(Coef = colMeans(B), SD = sd(B))
           Coef        SD
[1,] 29.378877 0.5587205
[2,]  2.251673 0.5822162
```

Example: Muscle Data

The muscle dataset is an old data from an experiment on muscle contraction in experimental animals. The data is explained and referenced in Appendix I.

The data consists of the variables: `Strip` (identifier of muscle), `Conc` (CaCl concentrations used to soak the section and `Length` (resulting length of muscle section for each concentration).

The model we are considering for this data is a non-linear model of the form

$$L = \alpha + \beta \exp(-C/\theta) + \text{error} \tag{4}$$

where α and β may vary with the animal but θ is constant. Note that α and β are (very many) linear parameters.

First Model: Fixed Parameters

Since there are 21 animals with separate alpha's and beta's for each, the number of parameters is 21+21+1=43, from 61 observations! We use the `plinear` algorithm since all parameters are linear with the exception to one.

```
> X <- model.matrix(~ Strip - 1, muscle)
> musc.1 <- nls(Length ~ cbind(X, X*exp(-Conc/th)),
muscle, start = list(th = 1), algorithm = "plinear",
  trace = T)
....
> b <- coef(musc.1)
> b
      th  .lin1  .lin2  .lin3  .lin4 .lin5 .lin6  .lin7
 0.79689 23.454 28.302 30.801 25.921  23.2 20.12 33.595
......
   .lin39 .lin40  .lin41  .lin42
  -15.897 -28.97 -36.918 -26.508
```

Conventional Fitting Algorithm

Parameters in non-linear regression may be indexed as follows

```
> b <- as.vector(b) # remove names attribute
> th <- b[1]
> a <- b[2:22]
> b <- b[23:43]
> musc.2 <- nls(Length ~ a[Strip] +
       b[Strip]*exp(-Conc/th),
       muscle, start = list(a = a, b = b, th = th),
       trace = T)
......
```

As we have some proper starting values, the algorithm converges in one step now. Note that with indexed parameters, the starting values must be given in a list (with names).

Plotting the Result

To plot the results, we use the `expand.grid` function to create a data frame from all combinations of the supplied vectors or factors.

```
> range(muscle$Conc)
[1] 0.25 4.00
> newdat <- expand.grid(
Strip = levels(muscle$Strip),
Conc = seq(0.25, 4, 0.05))
> dim(newdat)
[1] 1596    2
> names(newdat)
[1] "Strip" "Conc"
> newdat$Length <- predict(musc.2, newdat)
```

Trellis then comes to the rescue, plotting the predictions from the expanded grid to produce the plot shown in Figure 55.

```
> trellis.par.set(col.whitebg())
> xyplot(Length ˜ Conc | Strip, muscle, subscripts = T,
 panel = function(x, y, subscripts, ...) {
     panel.xyplot(x, y, ...)
     ws <- as(muscle$Strip[subscripts[1]], "character")
     wf <- which(newdat$Strip == ws)
     xx <- newdat$Conc[wf]
     yy <- newdat$Length[wf]
     llines(xx, yy, col = "red")
    }, ylim = range(newdat$Length, muscle$Length),
 as.table = T, col = "navy")
```

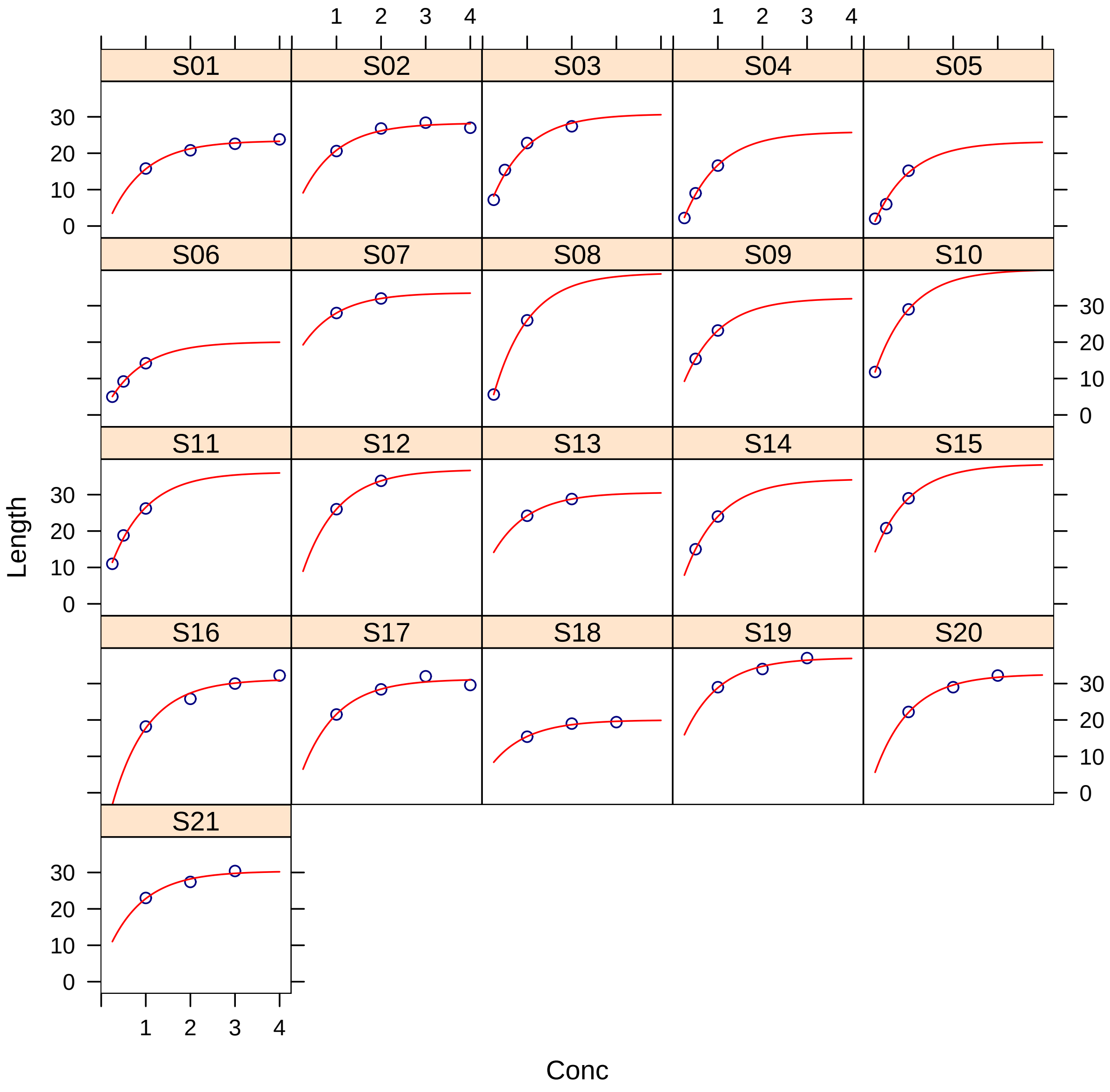


Figure 55: Predictions produced for the muscle dataset on an expanded grid.

A Random Effects Version

Random effect modelling of this dataset will be explored in a later session. However, we briefly touch upon the approach here. Random effects models allow the parametric dimension to be more easily controlled. We assume α and β are now random over animals

```
> musc.re <- nlme(Length ~ a + b*exp(-Conc/th),
fixed = a+b+th~1, random = a+b~1|Strip,
data = muscle, start = c(a = mean(a),
b = mean(b), th = th))
```

The vectors `a`, `b` and `th` come from the previous fit so there is no need to supply initial values for the random effects, (though you may).

We can produce a plot showing the predictions from the two models: fixed effects (gold) and random effects (navy), with points overlayed in hot pink. The resulting plot is shown in Figure 56.

```
> newdat$L2 <- predict(musc.re, newdat)
> xyplot(Length ~ Conc | Strip, muscle, subscripts = T,
par.strip.text=list(lines=1,cex=0.7),
 panel =function(x, y, subscripts, ...) {
     panel.xyplot(x, y, ...)
     ws <- as(muscle$Strip[subscripts[1]], "character")
     wf <- which(newdat$Strip == ws)
     xx <- newdat$Conc[wf]
     yy <- newdat$Length[wf]
     llines(xx, yy, col = "gold")
     yy <- newdat$L2[wf]
     llines(xx, yy, lty=4, col = "navy")
    },
  ylim = range(newdat$Length, muscle$Length),
as.table = T, col = "hotpink")
```

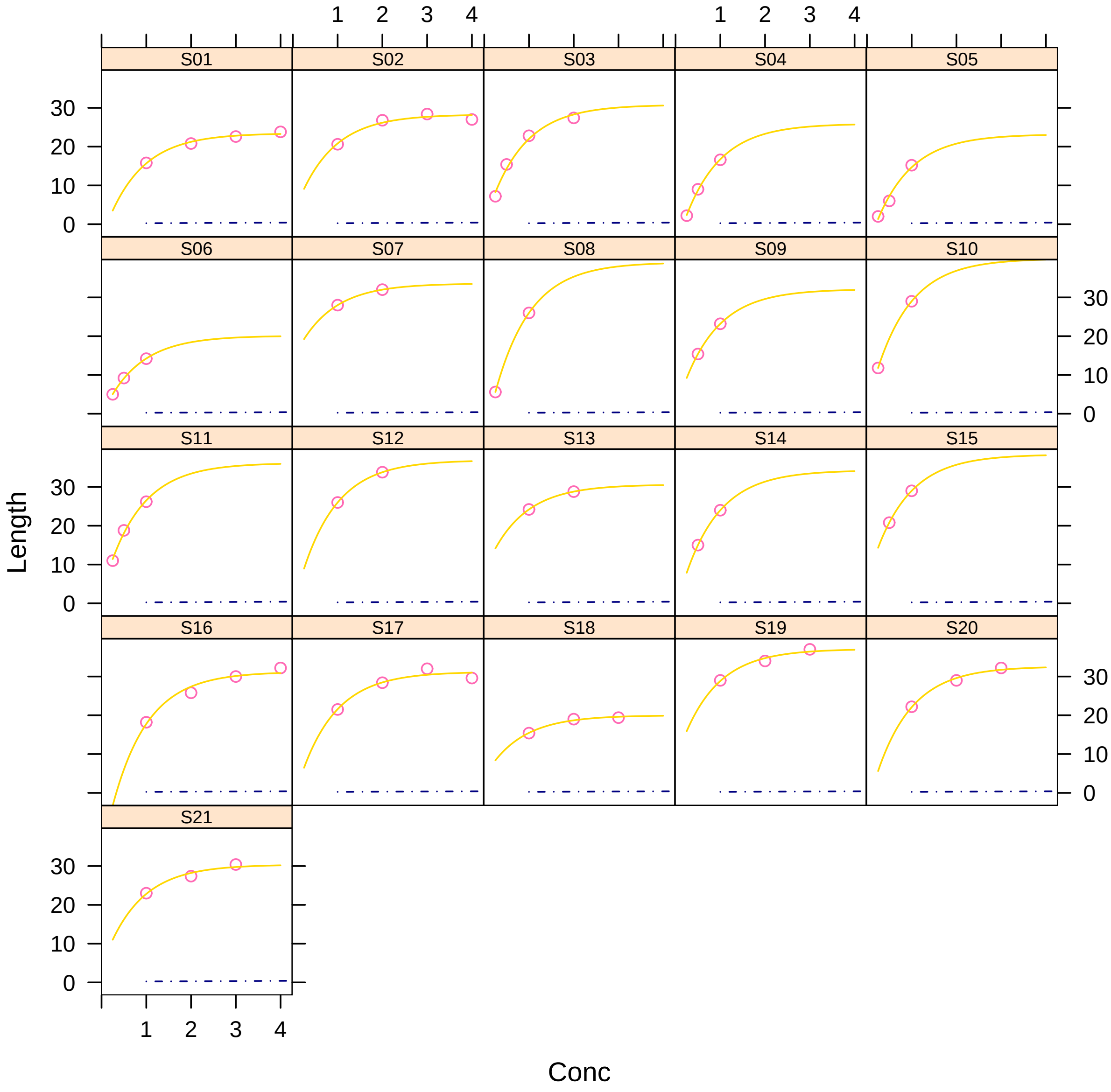


Figure 56: A Composite Plot

Final Notes

- Non-linear regression can be tricky. Some ingenuity in finding good starting values can be well rewarded.
- We have considered two bootstrapping techniques
  - bootstrapping mean-centred residuals and
  - using exponential weights.
- Ordinary bootstrapping by re-sampling the data can lead to problems in non-linear regression if influential points are omitted.
- Non-linear mixed models can require a great deal of computation. Large data sets can pose particularly difficult computational problems.

Generalized Linear Modelling

Methodology

Generalisations of Traditional Linear Models: A Roadmap

Figure 57 displays a roadmap of the different statistical models and their origins. Table 8 provides an explanation of acronyms to accompany the roadmap.

In the previous sessions we focussed on linear and non-linear models. This session we focus on generalized linear models. Future sessions will discuss the remaining modelling approaches shown in this diagram.

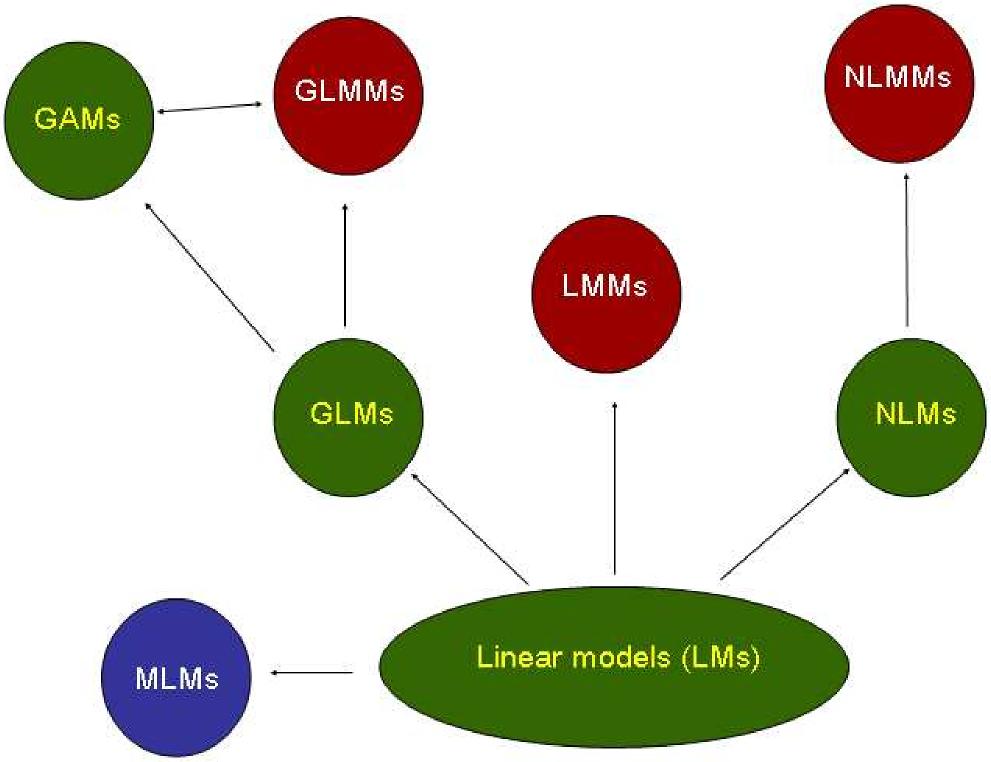


Figure 57: A roadmap of regression modelling techniques

Table 8: Explanation of Acronyms

Model	Acronym	R function
Linear Models	LM	`lm, aov`
Multivariate LMs	MLM	`manova`
Generalized LMs	GLM	`glm`
Linear Mixed Models	LMM	`lme, aov`
Non-linear Models	NLM	`nls`
Non-linear Mixed Models	NLMM	`nlme`
Generalized LMMs	GLMM	`glmmPQL`
Generalized Additive Ms	GAM	`gam`

Nature of the Generalization

If we consider a single response variable y and some candidate predictor variables $x_1, x_2, \ldots, x_p$, the distribution of y can only depend on the predictors through a single linear function:

$$\eta = b_1x_1 + b_2x_2 + \ldots + b_px_p$$

The distribution belongs to the GLM family of distributions, where there may (or may not) be an unknown scale parameter.

Distributions in the GLM Family

There are a number of distributions that we can consider apart from the Normal distribution that leads to a linear model. A summary of these distributions is displayed in Table 9.

Table 9: Distributions in the GLM Family

Distribution	Model
Normal	Ordinary linear models
Binomial	Logistic regression, probit
Poisson	Log-linear models
Gamma	Alternative to lognormal models
Negative Binomial	Take into account overdispersion

Link Functions

It is assumed that the linear predictor determines the mean of the response. The linear predictor is unbounded, but the mean of some of these distributions (e.g. binomial) is restricted. The mean is assumed to be a (monotone) function of the linear predictor and the inverse of this function is called the link function. Choosing a link is often the first problem in constructing a GLM.

Here are a few examples:

Normal → Identity link
Binomial → Logistic or Probit links
Poisson → Log or Square-root link
Gamma → log or inverse link

For the binomial distribution the response is taken as the proportion of cases responding. Thus the mean lies between 0 and 1 and the logistic link uses

$$\mu = \frac{\exp\eta}{1+\exp\eta}, \quad \eta = \log\frac{\mu}{1-\mu}$$

A question you may be wondering is why the link function is defined backwards. This is largely due to historical reasons. GLM theory was developed as a replacement for an older approximate theory that used transformations of the data. The link function is defined in the same sense, but the data are never transformed. The connection however, is assumed between parameters. The newer theory produces exact maximum likelihood estimates, but apart from the normal/identity case, inference procedures are still somewhat approximate.

Practice

Constructing GLMs in R is almost entirely analogous to constructing linear models. Estimation is by iteratively weighted least squares, so some care has to be taken that the iterative scheme has converged.

Some tools exist for manual and automated variable selection. There are differences however. For example, the residuals function distinguishes four types of residuals which all coincide in the case of linear models.

Example: Budworm Data

We illustrate fitting a GLM using the Budworm data. The data needs to be constructed and this is achieved using the following script.

```
> options(contrasts = c("contr.treatment", "contr.poly"))
> ldose <- rep(0:5, 2)
> numdead <- scan()
  1 4 9 13 18 20
  0 2 6 10 12 16

> sex <- factor(rep(c("M", "F"), each = 6))
> SF <- cbind(numdead, numalive = 20 - numdead)
> Budworms <- data.frame(ldose, sex)
> Budworms$SF <- SF
> rm(sex, ldose, SF)
```

An Initial Model

We fit an initial GLM to the budworm data using a Binomial distribution since we have information on the proportion of worms dead and we are trying to relate that proportion to the sex of the worm and dosage. (The `trace` option is set to true so we can examine the number of iterations required in the model fitting process.) The results are printed below.

```
> budworm.lg <- glm(SF ~ sex/ldose, family = binomial,
     data = Budworms, trace = T)
Deviance = 5.016103 Iterations - 1
Deviance = 4.993734 Iterations - 2
Deviance = 4.993727 Iterations - 3
Deviance = 4.993727 Iterations - 4

> summary(budworm.lg, cor = F)
```

```
...
Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.39849  -0.32094  -0.07592   0.38220   1.10375

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -2.9935     0.5527  -5.416 6.09e-08 ***
sexM          0.1750     0.7783   0.225    0.822
sexF:ldose    0.9060     0.1671   5.422 5.89e-08 ***
sexM:ldose    1.2589     0.2121   5.937 2.91e-09 ***
```

Results indicate that dosage fitted separately for males and females is significant.

Displaying the Fit

We display the fit of the model using the following script, which shows the probability of death versus dosage plotted on the log scale. The `with` function temporarily attaches the Budworms dataset and allows us to extract the variables required to produce the plot. Males are shown in orange and females are shown in blue indicating that females have a lower probability of dying than males and this becomes more prominent as dosage increases. The plot is shown in Figure 58.

```
> with(Budworms,{
   plot(c(1,32), c(0,1), type = "n", xlab = "dose",
  log = "x", axes = F,ylab = "Pr(Death)")
   axis(1, at = 2^(0:5))
   axis(2)

   points(2^ldose[1:6], numdead[1:6]/20, pch = 4)
   points(2^ldose[7:12], numdead[7:12]/20, pch = 1)

   ld <- seq(0, 5, length = 100)
lines(2^ld, predict(budworm.lg, data.frame(ldose = ld,
  sex = factor(rep("M", length(ld)),
  levels =    levels(sex))),
  type = "response"), col = "orange", lwd = 2)
lines(2^ld, predict(budworm.lg, data.frame(ldose = ld,
  sex = factor(rep("F", length(ld)), levels = levels(sex))),
  type = "response"), lty = 2, col = "blue", lwd = 2)
})
```

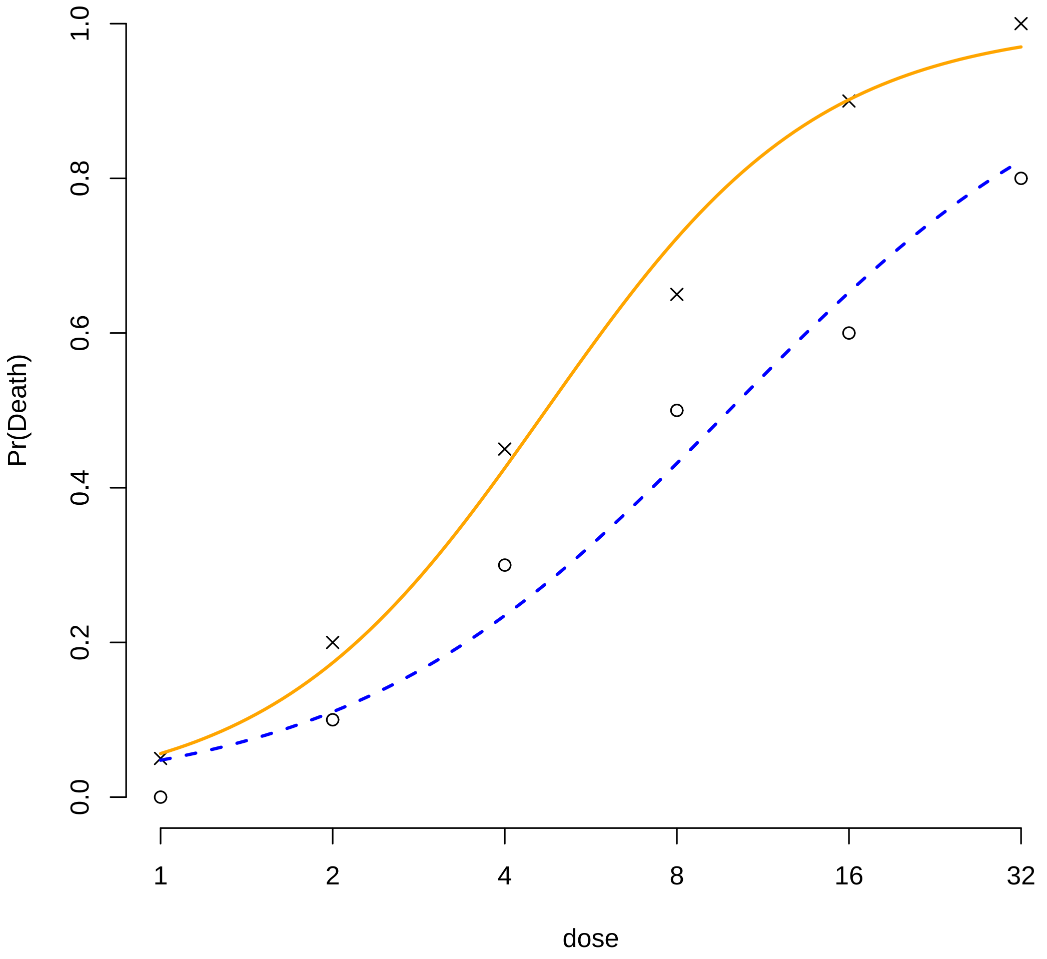


Figure 58: Plot of predictions for male (orange) and female (blue) worms.

Is Sex Significant?

This is a marginal term and so its meaning has to be interpreted carefully. Watch what happens if ldose is re-centred. In the previous analysis, we fitted separate lines for each sex and tested the hypothesis that the lines do not differ at zero log-dose. If we reparameterise the model to include the intercept at dose 8 we find that there is significant differences between sexes at dose 8 and the model fits reasonably well.

```
> budworm.lgA <- update(budworm.lg, . ˜ sex/I(ldose - 3))
Deviance = 5.016103 Iterations - 1
Deviance = 4.993734 Iterations - 2
Deviance = 4.993727 Iterations - 3
Deviance = 4.993727 Iterations - 4

> summary(budworm.lgA, cor = F)$coefficients
                    Estimate Std. Error   z value     Pr(>|z|)
(Intercept)       -0.2754324  0.2305173 -1.194845 2.321475e-01
sexM               1.2337258  0.3769761  3.272689 1.065295e-03
sexF:I(ldose - 3)  0.9060364  0.1671016  5.422068 5.891353e-08
sexM:I(ldose - 3)  1.2589494  0.2120655  5.936607 2.909816e-09
```

Checking for Curvature

We now check for curvature by adding in a squared term into the model and examining the fit using analysis of variance. Results suggest that there is no evidence of curvature in the data.

```
> anova(update(budworm.lgA, . ˜ . + sex/I((ldose - 3)ˆ2)),
test = "Chisq")
Deviance = 3.178919 Iterations - 1
Deviance = 3.171635 Iterations - 2
Deviance = 3.171634 Iterations - 3
Deviance = 3.171634 Iterations - 4
Deviance = 121.9229 Iterations - 1
Deviance = 118.7995 Iterations - 2
Deviance = 118.7986 Iterations - 3
Deviance = 118.7986 Iterations - 4
Deviance = 5.016103 Iterations - 1
Deviance = 4.993734 Iterations - 2
Deviance = 4.993727 Iterations - 3
Deviance = 4.993727 Iterations - 4
Analysis of Deviance Table

Model: binomial, link: logit

Response: SF

Terms added sequentially (first to last)

                     Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                                    11    124.876
sex                   1    6.077        10    118.799     0.014
sex:I(ldose - 3)      2  113.805         8      4.994 1.939e-25
sex:I((ldose - 3)^2)  2    1.822         6      3.172     0.402
```

Final Model: Parallelism

The final model we examine is one that checks for parallelism. If we reparameterise the model so that we fit separate parallel lines to each sex we find the following:

```
> budworm.lg0 <- glm(SF ~ sex + ldose - 1, family = binomial,
   Budworms, trace = T)
Deviance = 6.81165 Iterations - 1
Deviance = 6.757094 Iterations - 2
Deviance = 6.757064 Iterations - 3
Deviance = 6.757064 Iterations - 4

> anova(budworm.lg0, budworm.lgA, test = "Chisq")
Analysis of Deviance Table

Model 1: SF ~ sex + ldose - 1
Model 2: SF ~ sex + sex:I(ldose - 3)
  Resid. Df Resid. Dev Df Deviance P(>|Chi|)
1         9     6.7571
2         8     4.9937  1   1.7633    0.1842
```

Comparison with the previous model suggests that parallel lines is a reasonable assumption.

Effective Dosages

We now try and estimate for each sex, the dose where the probability of the moth dying is 50%. This is usually referred to as LD50 and it can be expressed as

$$x_p = (l(p) - \beta_0)/\beta_1$$

The MASS library has a function, `dose.p` for calculating x_p and its associated standard error. There is also a special print function for printing the results (see below for more details). If we use this function on the budworm fitted model, (where we fitted parallel lines), we find the dosages and standard errors for probabilities ranging between 0.25 and 0.75.

```
> dose.p
function(obj, cf = 1:2, p = 0.5)
{
  eta <- family(obj)$link(p)
  b <- coef(obj)[cf]
  x.p <- (eta - b[1])/b[2]
  names(x.p) <- paste("p = ", format(p), ":", sep = "")
  pd <-  - cbind(1, x.p)/b[2]
  SE <- sqrt(((pd %*% vcov(obj)[cf, cf]) * pd) %*% c(1, 1))
  res <- structure(x.p, SE = SE, p = p)
  oldClass(res) <- "glm.dose"
  res
}
> print.glm.dose <- function(x, ...){
    M <- cbind(x, attr(x, "SE"))
    dimnames(M) <- list(names(x), c("Dose", "SE"))
    x <- M
    NextMethod("print")

> dose.p(budworm.lg0, cf = c(1, 3), p = 1:3/4)
             Dose        SE
p = 0.25: 2.231265 0.2499089
p = 0.50: 3.263587 0.2297539
p = 0.75: 4.295910 0.2746874
```

Example: Low Birth Weight Data

The low birthweight data is described in Appendix I and contains information on low birth weight in infants born at a US hospital. A number of variables were collected that might explain the cause of the low birth weight.

We attach the `birthwt` dataset and examine some of the factors in the dataset.

```
> options(contrasts = c("contr.treatment", "contr.poly"))
> attach(birthwt)
> race <- factor(race, labels = c("white", "black", "other"))
> table(ptl)
   0  1 2 3
```

```
 159 24 5 1
> ptd <- factor(ptl > 0)
> table(ftv)
   0  1  2 3 4 6
 100 47 30 7 4 1
> ftv <- factor(ftv)
> levels(ftv)[ - (1:2)] <- "2+"
> table(ftv)
   0  1 2+
 100 47 42
> bwt <- data.frame(low = factor(low), age, lwt, race,
smoke = (smoke > 0), ptd, ht = (ht > 0), ui = (ui > 0), ftv)
> detach("birthwt")
> rm(race, ptd, ftv)
```

Initial Model

We consider as an initial model fitting a logistic regression to the low birth weight binary response and drop terms one at a time and test for significance. The resulting model is shown below.

```
> birthwt.glm <- glm(low ˜ ., family = binomial, data = bwt)
> dropterm(birthwt.glm, test = "Chisq")
Single term deletions
Model:
low ˜ age + lwt + race + smoke + ptd + ht + ui + ftv
       Df Deviance     AIC     LRT  Pr(Chi)
<none>     195.476 217.476
age     1  196.417 216.417   0.942 0.331796
lwt     1  200.949 220.949   5.474 0.019302 *
race    2  201.227 219.227   5.751 0.056380 .
```

```
smoke   1  198.674 218.674   3.198 0.073717 .
ptd     1  203.584 223.584   8.109 0.004406 **
ht      1  202.934 222.934   7.458 0.006314 **
ui      1  197.585 217.585   2.110 0.146342
ftv     2  196.834 214.834   1.358 0.507077
---
```

The previous model output suggests removing `fitv` and `age`. This is confirmed by successive deletion of these terms in the model. What happens, though, when we check for interactions between factors and curvatures in the numeric predictors? The next piece of code examines this issue.

```
> birthwt.step2 <- stepAIC(birthwt.glm,  ˜ .ˆ2 + I(scale(age)ˆ2) +
   I(scale(lwt)ˆ2), trace = F)
> birthwt.step2$anova
Stepwise Model Path
Analysis of Deviance Table
Initial Model:
low ˜ age + lwt + race + smoke + ptd + ht + ui + ftv
Final Model:
low ˜ age + lwt + smoke + ptd + ht + ui + ftv + age:ftv + smoke:ui
        Step Df  Deviance Resid. Df Resid. Dev      AIC
1                               178   195.4755 217.4755
2  + age:ftv  2 12.474896       176   183.0006 209.0006
3 + smoke:ui  1  3.056805       175   179.9438 207.9438
4     - race  2  3.129586       177   183.0734 207.0734
```

The model results in adding an interaction between `age` and `fitv` and removing race. Sometimes adding terms in one at a time and observing the coefficients may hint to confounding terms and/or interactions worthwhile exploring.

Final Model: Two Important Interactions

The final model we arrive at is a model that has main effect terms in age, weight of the mother, smoking status, hypertension, uterine irritability and number of physician visits as well as interactions between age and physician visits and smoking and uterine irritability.

```
> dropterm(birthwt.step2, test = "Chisq")
Single term deletions
Model:
low ˜ age + lwt + smoke + ptd + ht + ui + ftv + age:ftv + smoke:ui
         Df Deviance     AIC     LRT   Pr(Chi)
<none>       183.073 207.073
lwt       1  191.559 213.559   8.486 0.0035797 **
ptd       1  193.588 215.588  10.515 0.0011843 **
ht        1  191.211 213.211   8.137 0.0043361 **
age:ftv   2  199.003 219.003  15.930 0.0003475 ***
smoke:ui  1  186.986 208.986   3.913 0.0479224 *
---
```

We can check for linearity on age within number of physician visits. The idea is to fit separate spline terms within the levels of `ftv`, but keeping all other important terms (including interactions). It is important that spline terms be chosen with enough knots to allow non-linear behaviour to become apparent, but not so much that the fit becomes nearly indeterminate. This is how we do this.

We first create a grid of points ranging the full extent of the data.

```
> attach(bwt)
> BWT <- expand.grid(age=14:45, lwt = mean(lwt),
race = factor("white", levels = levels(race)),
smoke = c(T,F),
ptd = factor(c(T,F)),
ht = c(T,F),
ui = c(T,F),
ftv = levels(ftv))
> detach("bwt")
```

We then create a function that constructs natural splines with knots based on quantiles of the data. Here, the splines library is temporarily attached using the :: operator.

```
> nsAge <- function(x)
splines::ns(x, knots = quantile(bwt$age, 1:2/3),
Boundary.knots = range(bwt$age))
```

We then hard wire the knot placements using the `nsAge` function. This is referred to as *safe prediction*.

```
# Hard wiring the knot placements
> birthwt.glm2 <- glm(low ~ lwt + ptd + ht + smoke * ui +
     ftv/nsAge(age), binomial, bwt, trace = F)
```

If we are not careful about how to construct these knots we can have difficulties getting out the correct predictions (and not even know it). The sloppy version for constructing knots is shown below:

```
# Sloppy version
> birthwt.glm2 <- glm(low ~ lwt + ptd + ht + smoke * ui +
    ftv/splines::ns(age,df=3), binomial, bwt, trace = F)
```

We now predict using the `predict` function and overlay the predictions on a graph. Figures 59 and 60 show the results from hard wiring knots and calculating knots arbitrarily. Notice the differences in the predictions.

```
> prob <- predict(birthwt.glm2, BWT, type = "resp")
> xyplot(prob ~ age | ftv, BWT, type = "l",
subset = smoke == F & ptd == F & ht == F & ui == F,
    as.table = T, ylim = c(0, 1), ylab = "Pr(Low bwt)")
```

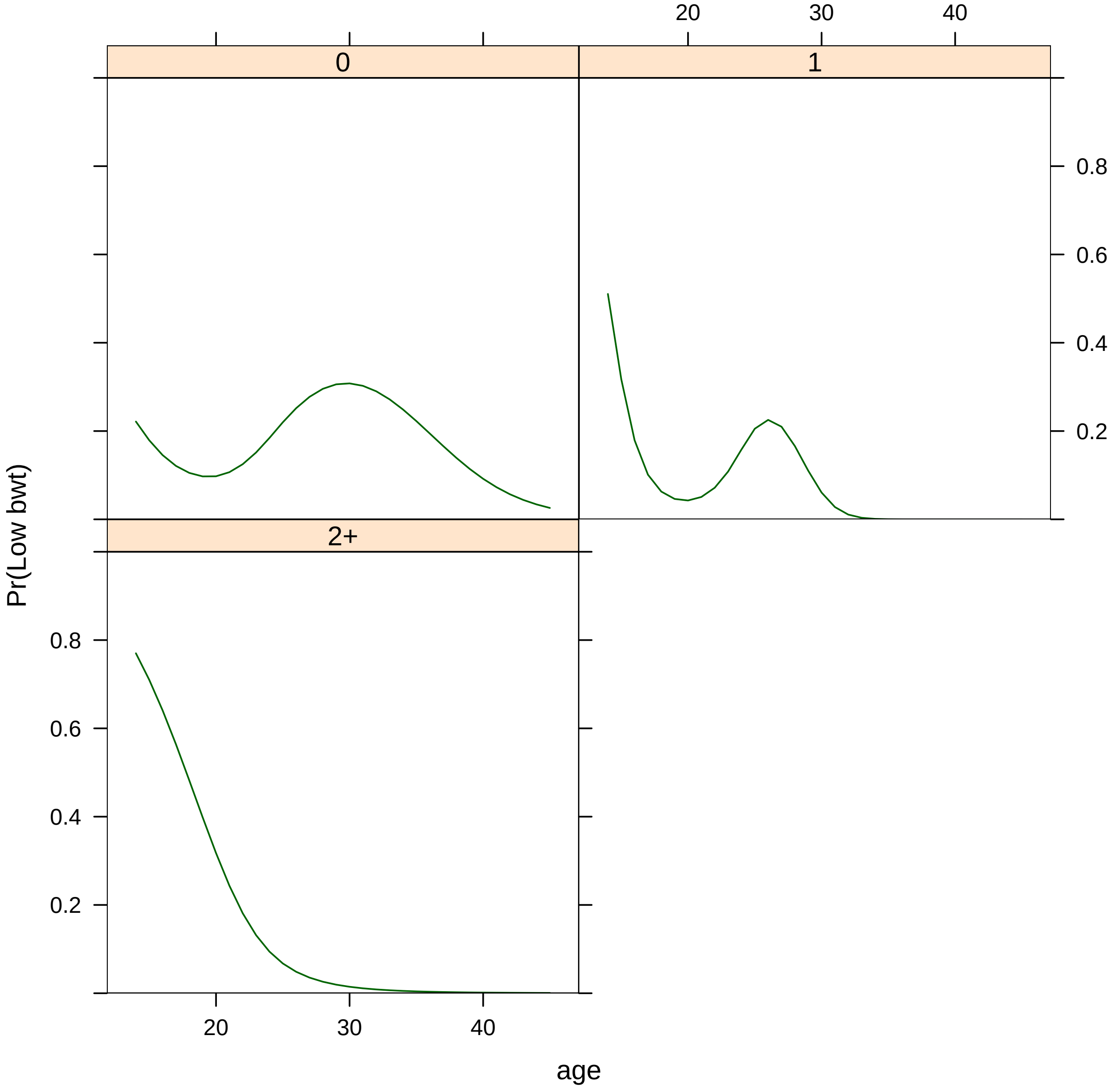


Figure 59: Hard Wiring Knot Placements

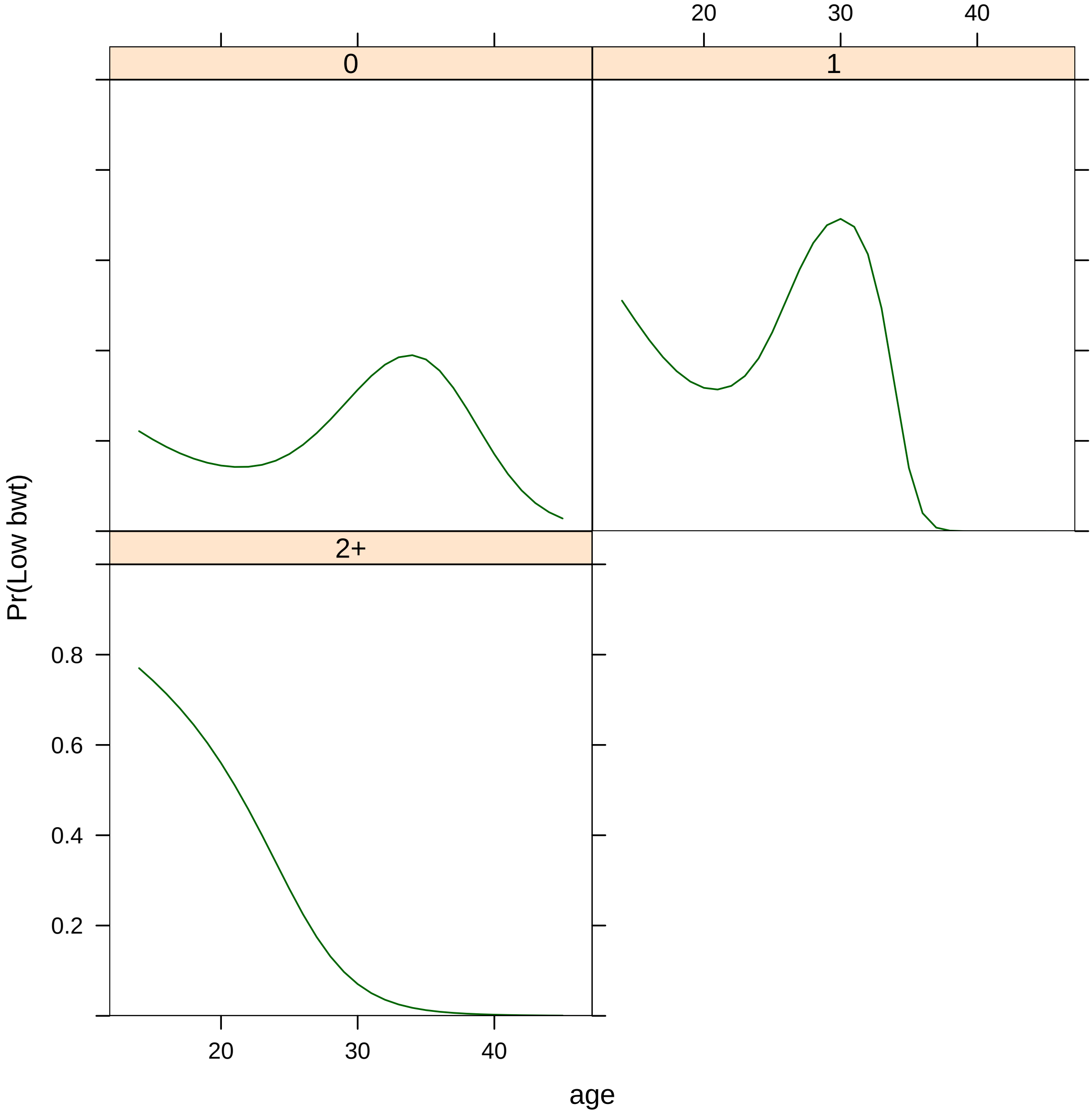


Figure 60: Sloppy Version

GLM Extensions

The Negative Binomial Distribution

The probability function of the negative binomial distribution is displayed below

$$\Pr(Y = y) = \frac{\Gamma(\theta + y)}{\Gamma(\theta) y!} \frac{\theta^\theta \mu^y}{(\theta + \mu)^{\theta + y}}, \quad y = 0, 1, 2, \ldots$$

with a mean-variance relationship expressed as

$$Var[Y] = \mu + \mu^2/\theta$$

Software for fitting negative binomial models are provided in the MASS library. The function `glm.nb` fits a negative binomial model to a set of data and it can also be fitted by optimisation functions, e.g. `optim`.

Genesis

We consider briefly, a Gamma mixture of Poissons (GLMM)

$$Y|G \sim \text{Po}(\mu G), \quad G \sim \gamma(\theta, \theta), \quad , E[G] = 1, \quad Var[G] = 1/\theta$$

and a compound Poisson

$$Y = X_1 + X_2 + \ldots + X_N, \quad N \sim \text{Po}, \quad X_i \sim \text{logarithmic}$$

We consider fitting these models to the Quine data as an example. We start with an initial Poisson fit however.

An Initial Poisson Fit

A Poisson model was fit to the Quine dataset, the results of which are displayed below. The summary of the fit indicates an excessively large deviance. Inspection of the mean variance relationship indicates that a negative binomial model may be more appropriate.

```
> quine.po1 <- glm(Days ˜ .ˆ4, poisson, quine, trace = T)
```

```
Deviance = 1371.787 Iterations - 1
Deviance = 1178.433 Iterations - 2
Deviance = 1173.905 Iterations - 3
Deviance = 1173.899 Iterations - 4
Deviance = 1173.899 Iterations - 5
> summary(quine.po1, cor = F)
Call:
glm(formula = Days ~ .^4, family = poisson, data = quine, trace = T)
(Dispersion Parameter for Poisson family taken to be 1 )

    Null Deviance: 2073.5 on 145 degrees of freedom
Residual Deviance: 1173.9 on 118 degrees of freedom
```

We consider the heuristic: $G \approx Y/\mu$ and use the fitted value from the Poisson fit as an estimate of μ. The fit of the model is shown below.

```
> t0 <- 1/var(quine$Days/fitted(quine.po1))
> t0
[1] 1.966012
> quine.nb1 <- glm.nb(Days ~ Eth * Lrn * Age * Sex,
data = quine, init.theta = t0, trace = 2)
Initial fit:
Deviance = 176.1053 Iterations - 1
Deviance = 169.9369 Iterations - 2
Deviance = 169.8431 Iterations - 3
Deviance = 169.8431 Iterations - 4
Deviance = 169.8431 Iterations - 5
Initial value for theta: 1.92836
Deviance = 167.4535 Iterations - 1
Theta( 1 ) = 1.92836 , 2(Ls - Lm) = 167.453
```

```
> quine.nb1$call$trace <- F  # turn off tracing
> dropterm(quine.nb1, test = "Chisq")
Single term deletions
Model:
Days ~ Eth * Lrn * Age * Sex
                Df      AIC      LRT   Pr(Chi)
         <none>      1095.3
Eth:Lrn:Age:Sex  2   1092.7      1.4    0.4956
```

The previous analysis indicated that the four-way interaction of terms was not required in the model. So we updated the model with this term removed. This provided the following results.

```
> quine.nb2 <- update(quine.nb1, . ~ . - Eth:Lrn:Age:Sex)
dropterm(quine.nb2, test = "Chisq", k = log(nrow(quine)))
Single term deletions
            Df      AIC      LRT   Pr(Chi)
     <none>     1170.30
Eth:Lrn:Age  2  1166.31     5.97   0.05045
Eth:Lrn:Sex  1  1167.91     2.60   0.10714
Eth:Age:Sex  3  1158.03     2.68   0.44348
Lrn:Age:Sex  2  1166.61     6.28   0.04330
```

Dropping the interaction between `Eth`, `Age` and `Sex` provides the following results

```
> quine.nb3 <- update(quine.nb2, . ~ . - Eth:Age:Sex)
dropterm(quine.nb3, test = "Chisq", k = log(nrow(quine)))
Single term deletions
...
            Df     AIC       LRT    Pr(Chi)
     <none>     1158.03
Eth:Lrn:Age  2  1153.83     5.77    0.05590
Eth:Lrn:Sex  1  1158.09     5.04    0.02479
```

```
Lrn:Age:Sex  2  1153.77     5.70    0.05779
```

We now consider also dropping the interaction between `Lrn`, `Age` and `Sex`. This produces the following output.

```
> quine.nb4 <- update(quine.nb3, . ~ . - Lrn:Age:Sex)
dropterm(quine.nb4, test = "Chisq", k = log(nrow(quine)))
Single term deletions
...
            Df      AIC      LRT   Pr(Chi)
     <none>     1153.77
    Age:Sex  3  1158.51    19.69 0.0001968
Eth:Lrn:Age  2  1148.12     4.32 0.1153202
Eth:Lrn:Sex  1  1154.27     5.49 0.0191463
```

We now drop the interaction between `Lrn`, `Age` and `Eth`.

```
> quine.nb5 <- update(quine.nb4, . ~ . - Lrn:Age:Eth)
> dropterm(quine.nb5, test = "Chisq", k = log(nrow(quine)))
Single term deletions
            Df      AIC      LRT   Pr(Chi)
     <none>     1148.12
    Eth:Age  3  1138.56     5.39 0.145324
    Lrn:Age  2  1141.94     3.79 0.150482
    Age:Sex  3  1154.31    21.14 0.000098
Eth:Lrn:Sex  1  1152.25     9.12 0.002535
```

We now drop the interaction between `Lrn` and `Age`.

```
> quine.nb6 <- update(quine.nb5, . ~ . - Lrn:Age)
> dropterm(quine.nb6, test = "Chisq", k = log(nrow(quine)))
Single term deletions
            Df      AIC      LRT   Pr(Chi)
     <none>     1141.94
    Eth:Age  3  1132.80     5.81 0.1214197
    Age:Sex  3  1145.43    18.44 0.0003569
Eth:Lrn:Sex  1  1145.40     8.44 0.0036727
```

Finally, we drop the interaction between Eth and Age to reveal the following summary table.

```
> quine.nb7 <- update(quine.nb6, . ~ . - Eth:Age)
        dropterm(quine.nb7, test = "Chisq",
        k = log(nrow(quine)))
Single term deletions
Model:
Days ~ Eth + Lrn + Age + Sex + Eth:Lrn + Eth:Sex + Lrn:Sex +
     Age:Sex + Eth:Lrn:Sex
            Df      AIC      LRT      Pr(Chi)
     <none>     1132.80
    Age:Sex  3  1136.46    18.62    0.0003277
Eth:Lrn:Sex  1  1140.23    12.42    0.0004244
> quine.check <- glm.nb(Days ~ Sex/(Age + Eth * Lrn), quine)
> deviance(quine.nb7)
> deviance(quine.check)
[1] 167.5558
[1] 167.5558
> range(fitted(quine.nb7) - fitted(quine.check))
[1] -1.484696e-06  5.938008e-07
```

Diagnostic Checks

Diagnostic checks of the fitted model are shown in Figure 61 and were produced using the following script.

```
> fv <- fitted(quine.nb7)
> rs <- resid(quine.nb7, type = "deviance")
> pv <- predict(quine.nb7)
> par(mfrow = c(1,2))
> plot(fv, rs, xlab = "fitted values",
ylab = "deviance residuals")
> abline(h = 0, lty = 4, lwd = 2, col = "orange")
> qqnorm(rs, ylab = "sorted deviance residuals")
> qqline(rs, col = "orange", lwd = 2, lty = 4)
> par(mfrow=c(1,1))
```

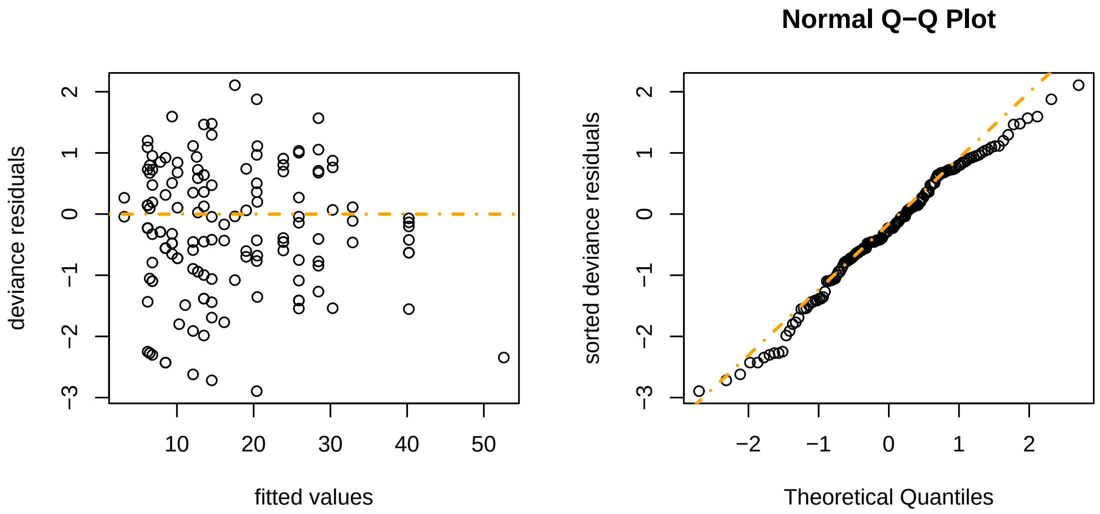


Figure 61: Diagnostic plots of the fitted model to the Quine dataset.

Some Notes on the model . . .

- We are led to the same model as with the transformed data.
- The big advantage we have with this analysis is that it is on the original scale, so predictions would be direct.
- Diagnostic analyses are still useful here, though they are less so with small count data
- Often the value for theta is not critical. One alternative to this is to fit the models with a fixed value for theta as ordinary glms.

Fixing Theta at a Constant Value

We now fix θ to be a constant value and examine the results. Fortunately we are led to the same model. This is a common occurrence if theta is a reasonable value to use.

```
> quine.glm1 <- glm(Days ~ Eth * Sex * Lrn * Age,
    negative.binomial(theta = t0), data = quine, trace = F)
> quine.step <- stepAIC(quine.glm1, k = log(nrow(quine)),
trace = F)
> dropterm(quine.step, test = "Chisq")
Single term deletions
Model:
Days ~ Eth + Sex + Lrn + Age + Eth:Sex + Eth:Lrn + Sex:Lrn +
    Sex:Age + Eth:Sex:Lrn
            Df  Deviance    AIC scaled dev.   Pr(Chi)
     <none>        195.99   1093.49
    Sex:Age  3    219.70   1108.48    20.99 0.0001056
Eth:Sex:Lrn  1    211.52   1105.24    13.75 0.0002086
```

Multinomial Models

Surrogate Poisson models offer a powerful way of analysing frequency data, even if the distribution is not Poisson. This is possible because the multinomial distribution can be viewed as a conditional distribution of independent Poisson variables, given their sum. In multiply classified frequency data, it is important to separate response and stimulus classifications (which may change according to viewpoint). With only one response classification, multinomial models may be fitted directly using `multinom`.

Example: Copenhagen Housing Data

The Copenhagen housing dataset is described in Appendix I. It contains information on three stimulus classifications: `Influence`, `Type` and `Contact`. There is one response classification: Satisfaction. The null model is `Influence*Type*Contact`, which corresponds to equal probabilities of 1/3 for each satisfaction class.

The simplest real model is `Influence*Type*Contact+Satisfaction`, which corresponds to a homogeneity model. More complex models are tested by their interactions with Satisfaction.

We begin with fitting a Poisson model based on the null hypothesis and then add to it a main effects term, `Satisfaction`, reflecting the *real* model.

```
> hous.glm0 <- glm(Freq ~ Infl*Type*Cont, poisson, housing)
> hous.glm1 <- update(hous.glm0, .~.+Sat)
> anova(hous.glm0, hous.glm1, test = "Chisq")
```

(Difference in deviance is 44.657 on 2 d.f.)

```
> addterm(hous.glm1, . ~ . + Sat * (Infl + Type + Cont),
test = "Chisq")
Single term additions
Model:
Freq ~ Infl + Type + Cont + Sat + Infl:Type + Infl:Cont +
   Type:Cont + Infl:Type:Cont
         Df  Deviance      AIC      LRT    Pr(Chi)
  <none>        217.46   610.43
Sat:Infl  4    111.08   512.05   106.37    0.00000
Sat:Type  6    156.79   561.76    60.67    0.00000
Sat:Cont  2    212.33   609.30     5.13    0.07708
```

It appears from the above analysis that all three terms are necessary, but no more.

```
> hous.glm2 <- update(hous.glm1, .~.+Sat*(Infl+Type+Cont))
```

To find a table of estimated probabilities we need to arrange the fitted values in a table (matrix) and normalize to have row sums equal to unity. How do we do this?

```
> attach(housing)
> levs <- lapply(housing[, -5], levels)
> dlev <- sapply(levs, length)


> ind <- do.call("cbind", lapply(housing[, -5],
    function(x) match(x, levels(x))))
> detach("housing")
```

(Note, The `do.call` function executes the `cbind` function to the `lapply` statement.).

```
> RF <- Pr <- array(0, dim = dlev, dimnames = levs)
> RF[ind] <- housing$Freq
> tots <- rep(apply(RF, 2:4, sum), each = 3)

> RF <- RF/as.vector(tots)
> RF

> Pr[ind] <- fitted(hous.glm2)
> Pr <- Pr/as.vector(tots)
> Pr
```

The printed results shown in Figure 62 can be cross-referenced against those produced from R.

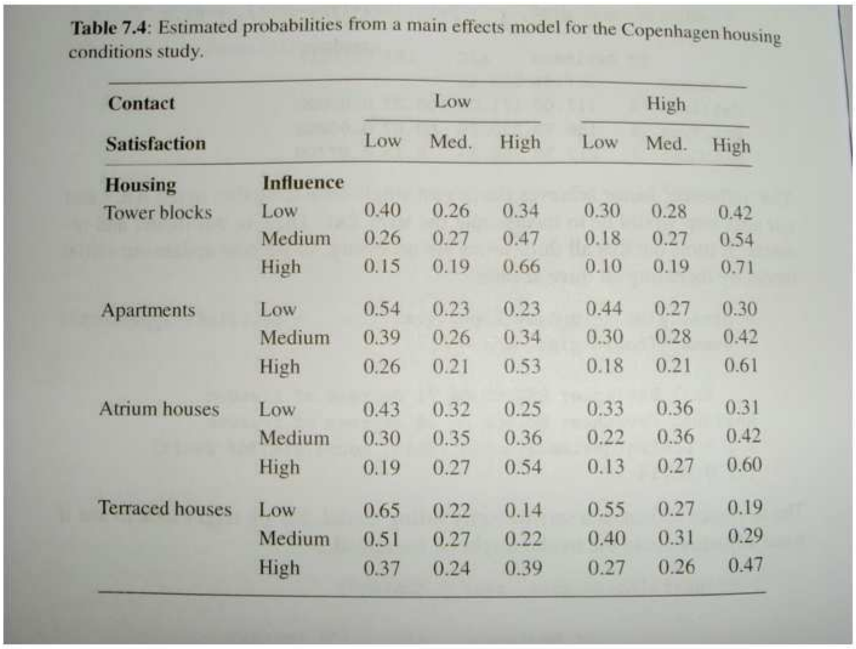


Table 7.4: Estimated probabilities from a main effects model for the Copenhagen housing conditions study.

Contact		Low			High		
Satisfaction		Low	Med.	High	Low	Med.	High
Housing	**Influence**						
Tower blocks	Low	0.40	0.26	0.34	0.30	0.28	0.42
	Medium	0.26	0.27	0.47	0.18	0.27	0.54
	High	0.15	0.19	0.66	0.10	0.19	0.71
Apartments	Low	0.54	0.23	0.23	0.44	0.27	0.30
	Medium	0.39	0.26	0.34	0.30	0.28	0.42
	High	0.26	0.21	0.53	0.18	0.21	0.61
Atrium houses	Low	0.43	0.32	0.25	0.33	0.36	0.31
	Medium	0.30	0.35	0.36	0.22	0.36	0.42
	High	0.19	0.27	0.54	0.13	0.27	0.60
Terraced houses	Low	0.65	0.22	0.14	0.55	0.27	0.19
	Medium	0.51	0.27	0.22	0.40	0.31	0.29
	High	0.37	0.24	0.39	0.27	0.26	0.47

Figure 62: Extract of the results as a check.

The function `multinom` is set up to take either a factor or a matrix with k columns as the response. In our case we have frequencies already supplied. These act as case weights. Fitted values from a multinomial fit are the matrix of probability estimates, with the columns corresponding to the response classes. Hence in our case they will occur three times over.

Fit a multinomial model and check that the fitted values agree with our surrogate Poisson estimates. Here is an example of how we might do this.

```
> require(nnet)
> hous.mult <- multinom(Sat ~ Infl + Type + Cont, data = housing,
   weights = Freq, trace = T)
# weights:  24 (14 variable)
initial  value 1846.767257
iter  10 value 1747.045232
final  value 1735.041933
converged
> round(fitted(hous.mult), 2)
    Low Medium High
 1 0.40   0.26 0.34
 2 0.40   0.26 0.34
 3 0.40   0.26 0.34
 4 0.26   0.27 0.47
71 0.27   0.26 0.47
72 0.27   0.26 0.47
> h1 <- t(fitted(hous.mult)[seq(3, 72, 3),  ])
> range(h1 - as.vector(Pr))
[1] -2.762006e-06  2.014685e-06
```

Proportional Odds Models

A parametrically economic version of the multinomial model is the proportional odds model, where the response classification is assumed ordered. The model may be specified as

$$\pi(x) = \Pr(Y \leq k|x), \quad \log(\frac{\pi(x)}{1-\pi(x)}) = \zeta_k - x^T \beta$$

Hence the cumulative probabilities conform to a logistic model, with parallelism in the logistic scale. The MASS library contains a function `polr` to fit such models. Here is an example.

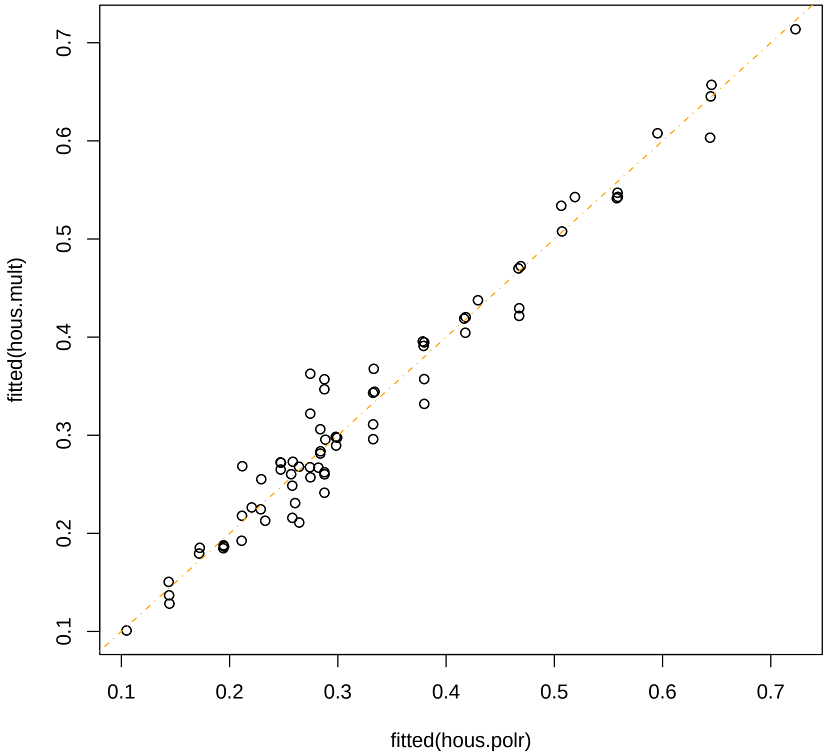


Figure 63:

```
> hous.polr <- polr(Sat ~ Infl+Type+Cont,
data = housing, weights = Freq)
> plot(fitted(hous.polr), fitted(hous.mult))
> abline(0, 1, col="orange", lty=4, lwd=1)
> hous.polr2 <- stepAIC(hous.polr, ~.^2,
k = log(24))
> hous.polr2$call$formula
Sat ~ Infl + Type + Cont + Infl:Type
```

With a more parsimonious model the automatic selection procedure uncovers a possible extra term.

A plot of fitted values from the two types of models is shown in Figure 63. The plot indicates similarities between the two fits.

Generalized Additive Models: An Introduction

Methodology

Overview

The generalized additive model assumes that the mean response is a sum of terms each depending on (usually) a single predictor:

$$Y = \alpha + \sum_{j=1}^{p} f_j(x_j) + \epsilon$$

If the f_js are linear terms, this is just like the regression models discussed in the previous sessions. If they are step functions then the main effect consists of a factor term. In general however, they may be smooth terms, with the degree of smoothness chosen by cross validation.

In some cases the additive terms may be known. They may consist of smoothing splines, local regression, splines with fixed degrees of freedom, harmonic terms or splines with known knots and boundary knot positions. Often it is useful to use a smoothing spline initially to determine the placement of knots and then follow up with fitting natural spline terms with fixed knot locations as natural splines have better computational properties.

Comparison with GLMs

Additive models are analogous to regression models. The generalized version of this is akin to the GLMs. GAMs may employ a link function to relate the linear predictor to the mean of the response, and they may have a non-normal distribution for example.

Fitting GAMs is the same process as fitting GLMs (but with one letter different in the function name). The fitting process is NOT maximum likelihood if there are any smoother terms present. A likelihood penalized by a roughness term is maximised, with the tuning constant chosen (usually) by cross-validation.

Inference for GAMs is difficult and somewhat contentious. Best regarded as an exploratory technique with standard models to follow. Some of the examples in the following sections will attempt to illustrate this.

Example: The Iowa Wheat Yield Data

The Iowa Wheat Yield data has already been analysed in previous sessions. The response consists of the yield of wheat in bushels/acre for the state of Iowa for the years 1930-1962. Potential predictors are `Year` (as surrogate), `Rain0`, `Rain1`, `Rain2`, `Rain3`, `Temp1`, `Temp2`, `Temp3` and `Temp4`. The problem focuses on building a predictor for yield from the predictors available. Note, with only 33 observations and 9 possible predictors some care has to be taken in choosing a model.

Initial Linear Model

We consider fitting an initial linear model using all of the predictors and drop terms one at a time

```
> Iowa <- read.csv("Iowa.csv")
> iowa.lm1 <- lm(Yield ˜ ., Iowa)
> iowa.step <- stepAIC(iowa.lm1,
      scope = list(lower =  ˜ Year, upper =  ˜ .),
      k = log(nrow(Iowa)), trace = F)
> dropterm(iowa.step, test = "F", k = log(nrow(Iowa)),
sorted = T)
Single term deletions
Model:
Yield ˜ Year + Rain0 + Rain2 + Temp4
       Df Sum of Sq    RSS    AIC F Value     Pr(F)
<none>              1554.6  144.6
Temp4   1     188.0 1742.6  144.9     3.4   0.07641 .
Rain0   1     196.0 1750.6  145.0     3.5   0.07070 .
Rain2   1     240.2 1794.8  145.9     4.3   0.04680 *
Year    1    1796.2 3350.8  166.5    32.4 4.253e-06 ***
```

Even with the more stringent BIC penalty on model complexity, two of the terms found are only borderline significant in the conventional sense. This is a consequence of the small sample size. Nevertheless the terms found are tentatively realistic:

- `Year`: surrogate for crop improvements.

- `Rain0`: a measure of pre-season sowing conditions
- `Rain2`: rainfall during the critical growing month
- `Temp4`: climatic conditions during harvesting

We now investigate if strictly linear terms in these variables seems reasonable. We achieve this using additive models. These are described in the following section.

Additive Models

Two libraries are available for fitting additive models. The first is the `gam` library which was written by Trevor Hastie and follows closely to the S implementation. The second is a package called and `mgcv`. This library is an implementation by Simon Wood that provides more robust methods for estimating smoothing parameters. We will use both libraries in this session.

If we consider a non-parametric smoother in each term we obtain the following:

```
> require(gam)
> iowa.gam <- gam(Yield ~ s(Temp4) + s(Rain0) +
s(Rain2) + s(Year), data = Iowa)
> par(mfrow = c(2,2))
> plot(iowa.gam, se = T, ylim = c(-30, 30),resid = T)
```

Figure 64 displays a plot of the additive terms in the model. It can be important to keep the y-axes of these plots approximately the same to allow comparisons between terms.

A number of conclusions can be drawn from these additive plots.

- `Temp4`: There are two very hot years that appeared to have crop damage during harvest.
- `Rain0`: There appears to be a wide range where there is little difference in yield. However, very dry years may lead to a reduced yield and very wet years to an enhanced one perhaps.
- `Rain2`: One very dry growing month led to a reduced yield.
- `Year`: The strongest and most consistent predictor by far. Some evidence of a pause in new varieties during the war and immediately post-war period.

If we examine the summary of the GAM fit we see that `Year` is by far the best predictor, followed by `Rain0` and `Temp4`. Note the latter two are significant at the 1% level.

```
> summary(iowa.gam) # (result edited)
```

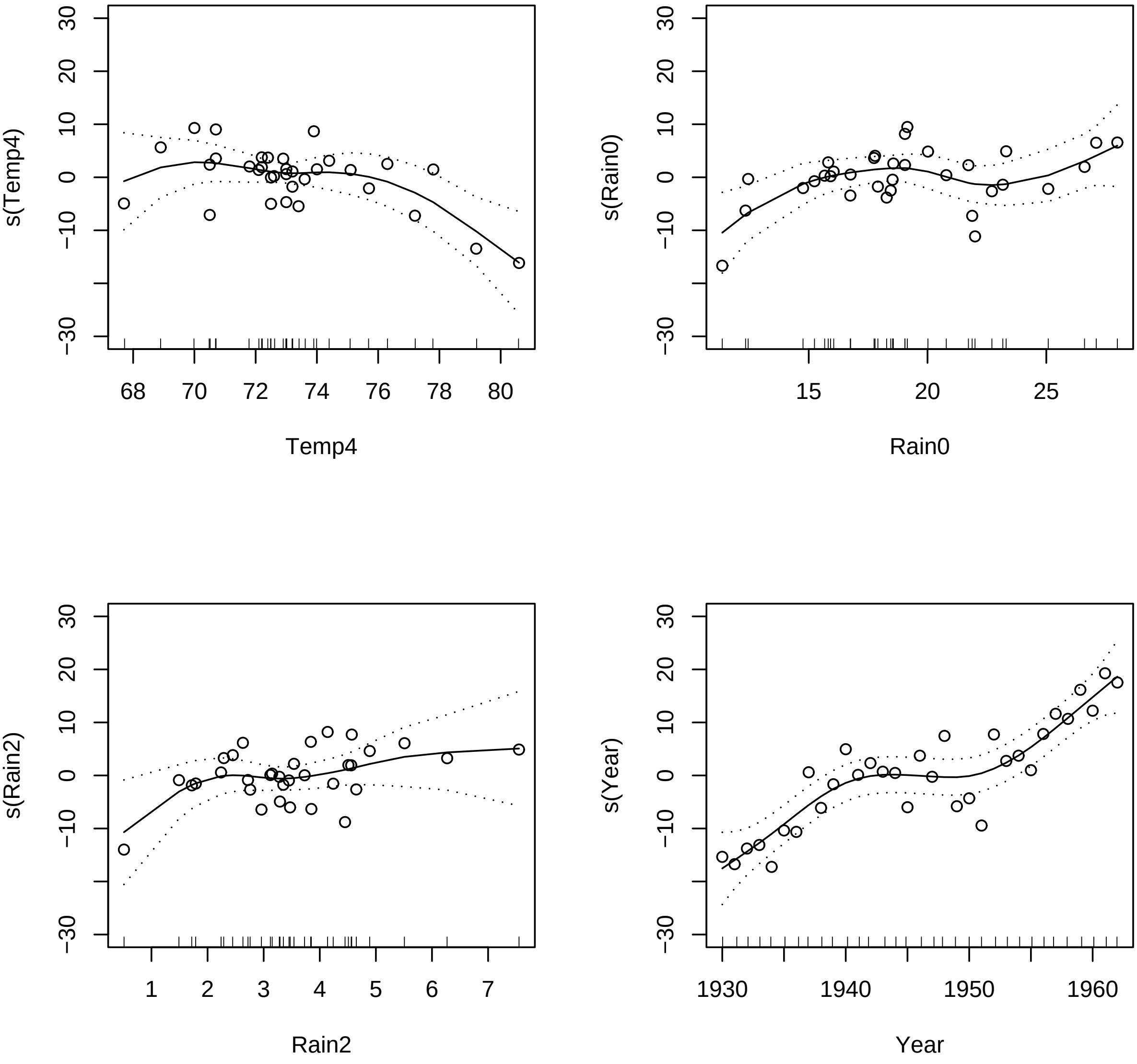


Figure 64: Plots of additive contributions

```
Call: gam(formula = Yield ~ s(Temp4) + s(Rain0) + s(Rain2) +
    s(Year), data = Iowa)
(Dispersion Parameter for Gaussian family taken to be 31.1175 )
Residual Deviance: 497.8868 on 16.002 degrees of freedom

Number of Local Scoring Iterations: 2
DF for Terms and F-values for Nonparametric Effects
            Df Npar Df   Npar F     Pr(F)
(Intercept)  1
   s(Temp4)  1       3 2.4709     0.09917
   s(Rain0)  1       3 3.0301     0.05993
   s(Rain2)  1       3 1.3746     0.28638
    s(Year)  1       3 3.6841     0.03437
```

Can we get to the same place with GLMs

An interesting exercise is to determine whether we can get to the same place with GLMs.

We fit a GLM to the Iowa Wheat Yield data using spline terms that are specified using natural spline functions. We could also consider fitting B-splines, which differ only in behaviour near the end points.

We need to specify the knot and boundary knot positions. This is recommended if prediction will be needed. Alternatively, if only exploratory analysis is being conducted, the degrees of freedom can be specified.

Each spline term is a collection of ordinary linear terms, but the coefficients have no simple meaning and the individual significance tests are meaningless. In this circumstance, splines are best regarded as a single composite term and retained or removed as a block. See the following for an implementation of the GLM with spline terms.

```
> iowa.ns <- lm(Yield ~ ns(Temp4, df=3) + ns(Rain0, df=3) +
   ns(Rain2, df = 3) + ns(Year, df=3), Iowa)
> tplot(iowa.ns, se = TRUE, rug = TRUE, partial = TRUE)
> dropterm(iowa.ns, test = "F", k = log(nrow(Iowa)))
Single term deletions
```

```
Model:
Yield ~ ns(Temp4, df = 3) + ns(Rain0, df = 3) +
    ns(Rain2, df = 3) + ns(Year, df = 3)
                  Df  Sum of Sq      RSS      AIC  F Value     Pr(F)
           <none>                 726.26   147.47
ns(Temp4, df = 3)  3     274.60  1000.86   147.56     2.52   0.08706
ns(Rain0, df = 3)  3     332.31  1058.57   149.41     3.05   0.05231
ns(Rain2, df = 3)  3      70.61   796.87   140.04     0.65   0.59327
 ns(Year, df = 3)  3    2022.93  2749.19   180.91    18.57   0.00001
```

Figure 65 presents a plot of each variable's contribution to the model. This plot was constructed using `tplot`.

The above results show a very similar pattern to the components fitted in the additive model. It is now clear that the term in `Rain2` is not useful and `Temp4` and `Rain0` terms will need to be re-assessed. The term in `Year` stands out as dominant with a clear pattern in the response curve and the partial residuals following it closely. Small data sets like this can be very misleading and therefore extreme caution is needed.

Example: Rock Data

A second example we investigate with GAMs is the Rock data, where the response consists of the permeability of rock samples taken from a petroleum reservoir. The potential predictors are `area`, `perimeter` and `shape`.

We focus on building a predictor for `log(perm)` using the available predictors. The following script sets out the model below. Make sure you have attached the `gam` library and not the `mgcv` library for this one.

```
> rock.lm <- lm(log(perm) ~ area + peri + shape,
   data = rock)
> summary(rock.lm)
Coefficients:
              Value  Std. Error   t value Pr(>|t|)
(Intercept)  5.3331      0.5487     9.720    0.000
       area  0.0005      0.0001     5.602    0.000
```

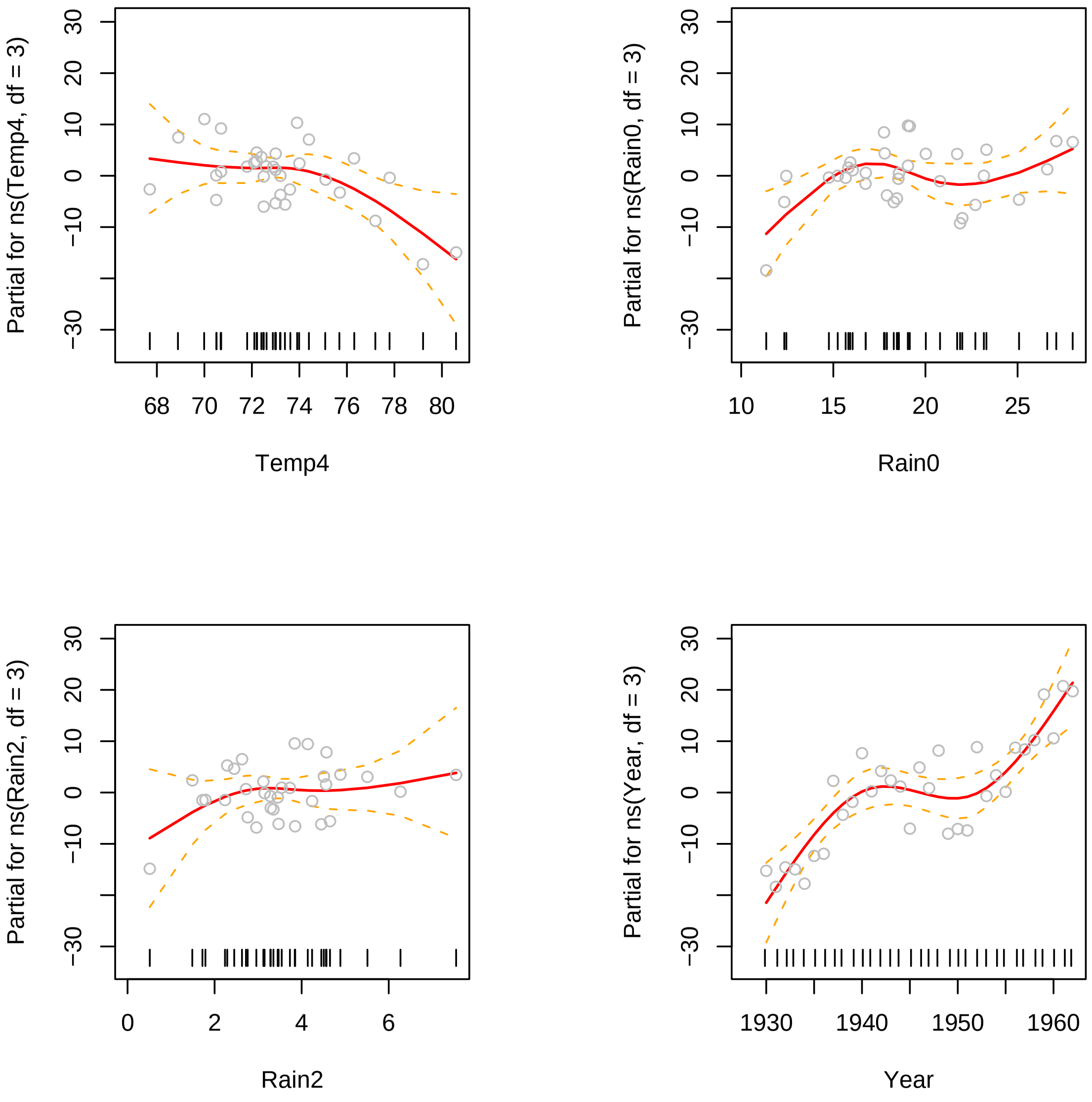


Figure 65: Plot of each variable's contribution to the GLM.

```
      peri -0.0015      0.0002    -8.623    0.000
     shape  1.7570      1.7560     1.000    0.323
```

The linear model suggests that both area and perimeter are strong predictors for the log of permeability. Fitting an additive model with smoothing splines fitted to each of the three terms produces the following.

```
> rock.gam <- gam(log(perm) ~ s(area) + s(peri) + s(shape),
     control = gam.control(maxit = 50,bf.maxit=50), data = rock)
> summary(rock.gam)
Call: gam(formula = log(perm) ~ s(area) + s(peri) + s(shape),
     data = rock, control = gam.control(maxit = 50,bf.maxit=50))
Residual Deviance: 26.0574 on 34.9981 degrees of freedom


Number of Local Scoring Iterations: 2
DF for Terms and F-values for Nonparametric Effects
            Df Npar Df   Npar F     Pr(F)
(Intercept)  1
    s(area)  1       3  0.34177    0.7953
    s(peri)  1       3  0.94085    0.4314
   s(shape)  1       3  1.43219    0.2499


> anova(rock.lm, rock.gam)
```

The analysis of variance shows little improvement in the fit of the model when spline terms are added. We now include linear terms for area and perimeter and a spline term for shape and compare the results against the previous two models.

```
> par(mfrow = c(2, 3), pty = "s")
> plot(rock.gam, se = TRUE)
> rock.gam1 <- gam(log(perm) ~ area + peri + s(shape), data = rock)
> plot(rock.gam1, se = TRUE)
> anova(rock.lm, rock.gam1, rock.gam)
```

The results once again suggest no improvement with the addition of the spline term on shape. The plots shown in Figure 66 also show that there is no advantage in fitting a spline term for any of the predictors but there are definite increase and decreasing trends for area and perimeter.

Although suggestive, the curve in shape is not particularly convincing. Choosing the degree of smoothness can be tricky. The `gam` function in Simon Woods (SWs) R implementation (mgcv library) offers a way around this. In this case, SWs `gam` function suggests essentially linear terms, at most, in all three variables. The script below produces a GAMs model based on this implementation and then produces plots of their contribution.

```
require(mgcv)

rock.gamSW <- gam(log(perm) ~ s(area) + s(peri) + s(shape),

   data = rock)

plot(rock.gamSW)
```

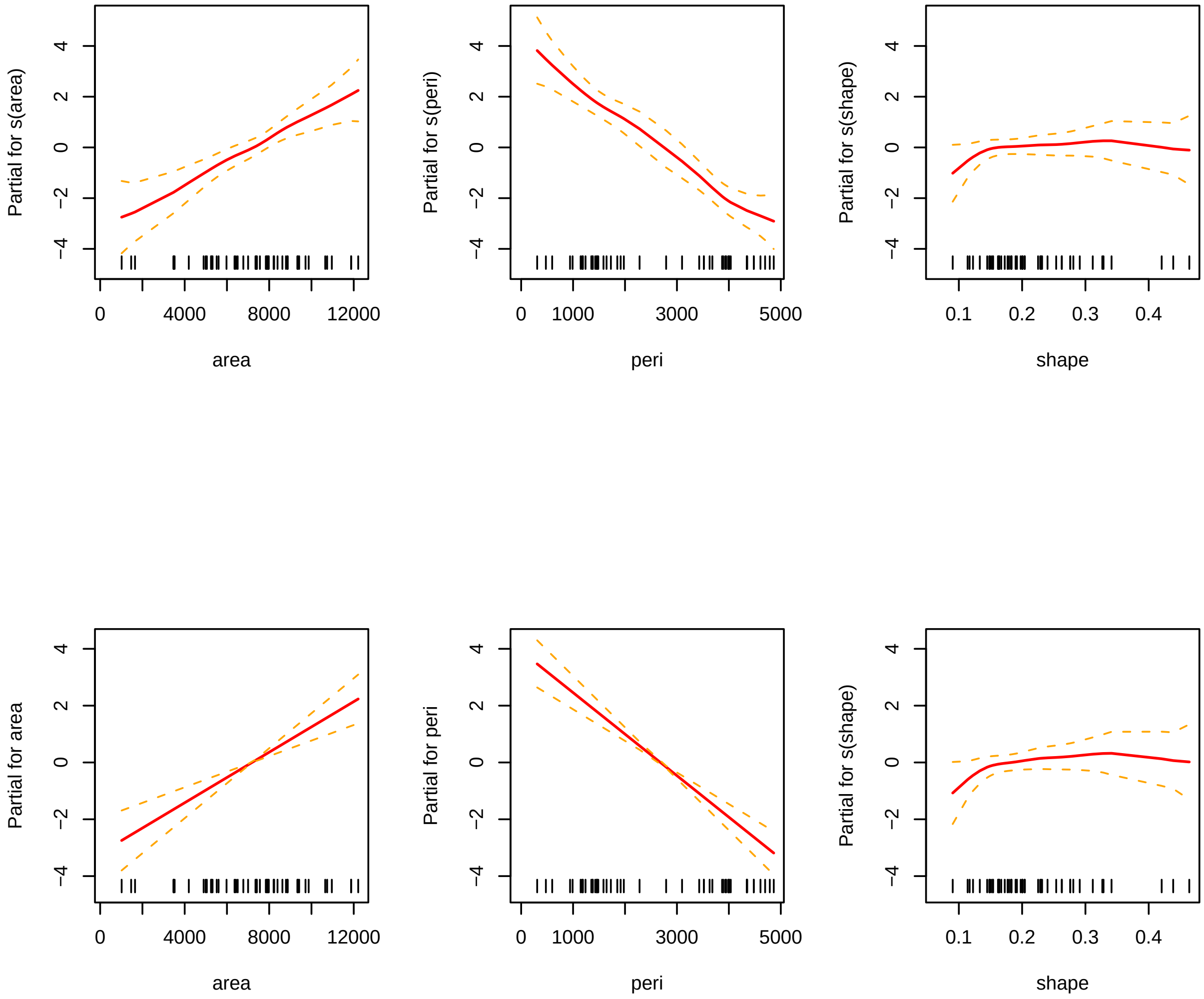


Figure 66: Partial residual plots showing the contribution of area, perimeter and shape to the fit of the model. The first three plots have spline terms fitted to all three terms, while the bottom three plots are the result of fitting linear terms for area and perimeter and a spline term for shape.

Advanced Graphics

Lattice Graphics

Lattice is a collection of functions roughly parallel to the *traditional* graphics functions. It implements the ideas of Bill Cleveland on data presentation given in *Visualising Data*. R *lattice graphics* mirrors the functionality of S-PLUSs Trellis graphics, but the low-level implementation is very different. Lattice graphics is based on Paul Murrels grid graphics implying that traditional and lattice graphics cannot (easily) be mixed.

Lattice graphics functions produce a graphics object. Printing this object is what produces the graphical output. After a lattice graphics object is printed, the user coordinate system is replaced by a (0, 1, 0, 1) system and interaction with lattice graphics is **not** possible.

A draft copy of a book entitled *R Graphics* is available on the web at the following web location: *http://www.stat.auckland.ac.nz/ paul/RGraphics/rgraphics.html*. It contains some nice examples (with scripts) for producing graphics and in particular, it provides some tips on how to work with Trellis graphics.

Example: Whiteside Data

We begin with the Whiteside data and produce some fairly simple trellis plots using the `xyplot` function. Figures 67 and 68 display the results.

```
> graphics.off()
> require(lattice)
> trellis.device()
> trellis.par.set(theme=col.whitebg())
> xyplot(Gas ˜ Temp, whiteside, groups = Insul,panel = panel.superpose)
> xyplot(Gas ˜ Temp | Insul, whiteside,
xlab = "External temperature",
ylab = "Gas consumption",
main = "Whiteside heating data", aspect = 0.6)
```

We can now add a least squares line using the following piece of code. This produces the plot shown in Figure 69.

```
> xyplot(Gas ˜ Temp | Insul, whiteside,
```

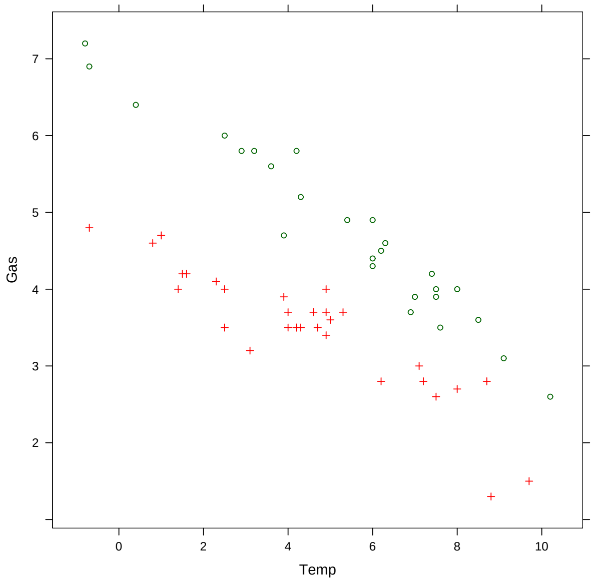


Figure 67: `xyplot` of the Whiteside data showing the relationships between gas and temperature for different levels of insulation.

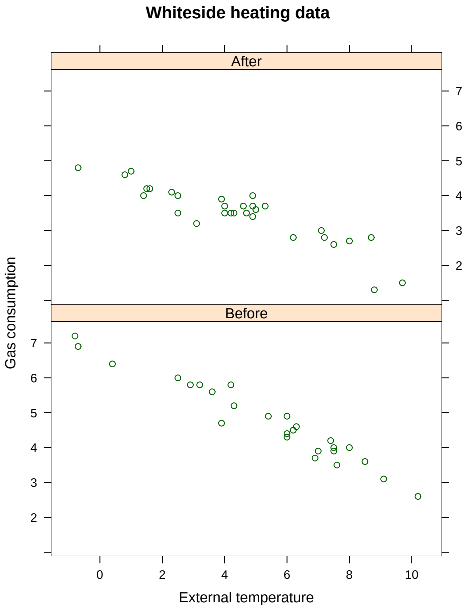


Figure 68: A more elegant plot showing gas consumption versus external temperature for different levels of insulation.

```
   xlab = "External temperature",
   ylab = "Gas consumption",
   main = "Whiteside heating data", aspect = 0.6,
   panel = function(x, y, ...) {
panel.xyplot(x, y, ...)
panel.lmline(x, y, ...,col="red")
    }, ylim = c(0, 7))
```

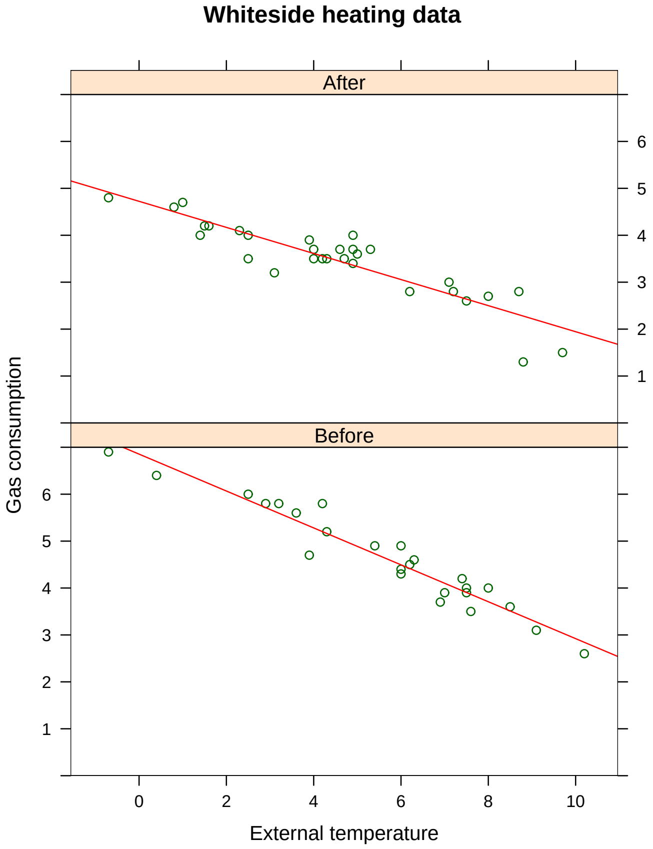


Figure 69: Adding a least squares line to the Whiteside data.

We may want to *de-link* the scales and let each plot have its own x and y axis. Figure 70 shows the result.

```
> xyplot(Gas ˜ Temp | Insul, whiteside,
   xlab = "External temperature",
   ylab = "Gas consumption",
   main = "Whiteside heating data",
   aspect = 0.6,
   panel = function(x, y, ...) {
     panel.xyplot(x, y, ...)
     panel.lmline(x, y, ...)
   },  prepanel = function(x, y, ...) {
   list(xlim = range(x), ylim = range(0, y),
      dx = NULL, dy = NULL)
    }, scales = list(relation = "free"))
```

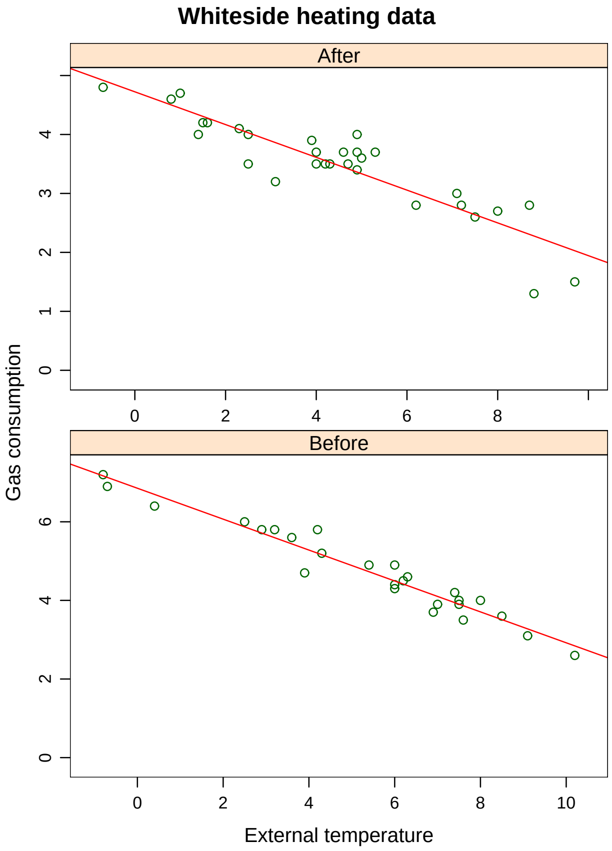


Figure 70: De-linking the scales to produce plots with their own axes.

Changing Trellis Parameters & Adding Keys

When a lattice device is opened, it has various colours, line-types and plotting symbols associated with it that any printing of lattice objects will use. You can change these choices but it gets messy!

You need to know how to access this scheme if you want to set a key saying what is what. The easiest way to achieve this is to keep an example on hand that works and re-read the help information with this example in mind.

The following example obtains information on symbols, (their size, colour, font etc). We print out the settings for symbols and display the settings using the `show.settings()` command. The result is displayed in Figure 71.

```
> sps <- trellis.par.get("superpose.symbol")
> sps
$alpha
[1] 1 1 1 1 1 1 1
$cex
[1] 0.7 0.7 0.7 0.7 0.7 0.7 0.7
$col
[1] "darkgreen" "red" "royalblue" "brown" "orange"
[6] "turquoise"  "orchid"
$font
[1] 1 1 1 1 1 1 1
$pch
[1]  1  3  6  0  5 16 17
> show.settings()
```

We now change a couple of these settings, display the results in Figure 72 and produce another version of the whiteside data. The result is displayed in Figure 73.

```
> sps$pch <- c(16,18,4,3,1,2,7)
> sps$col <- c("purple","pink","blue","orange",
"red","brown","black")
> trellis.par.set("superpose.symbol", sps)
```

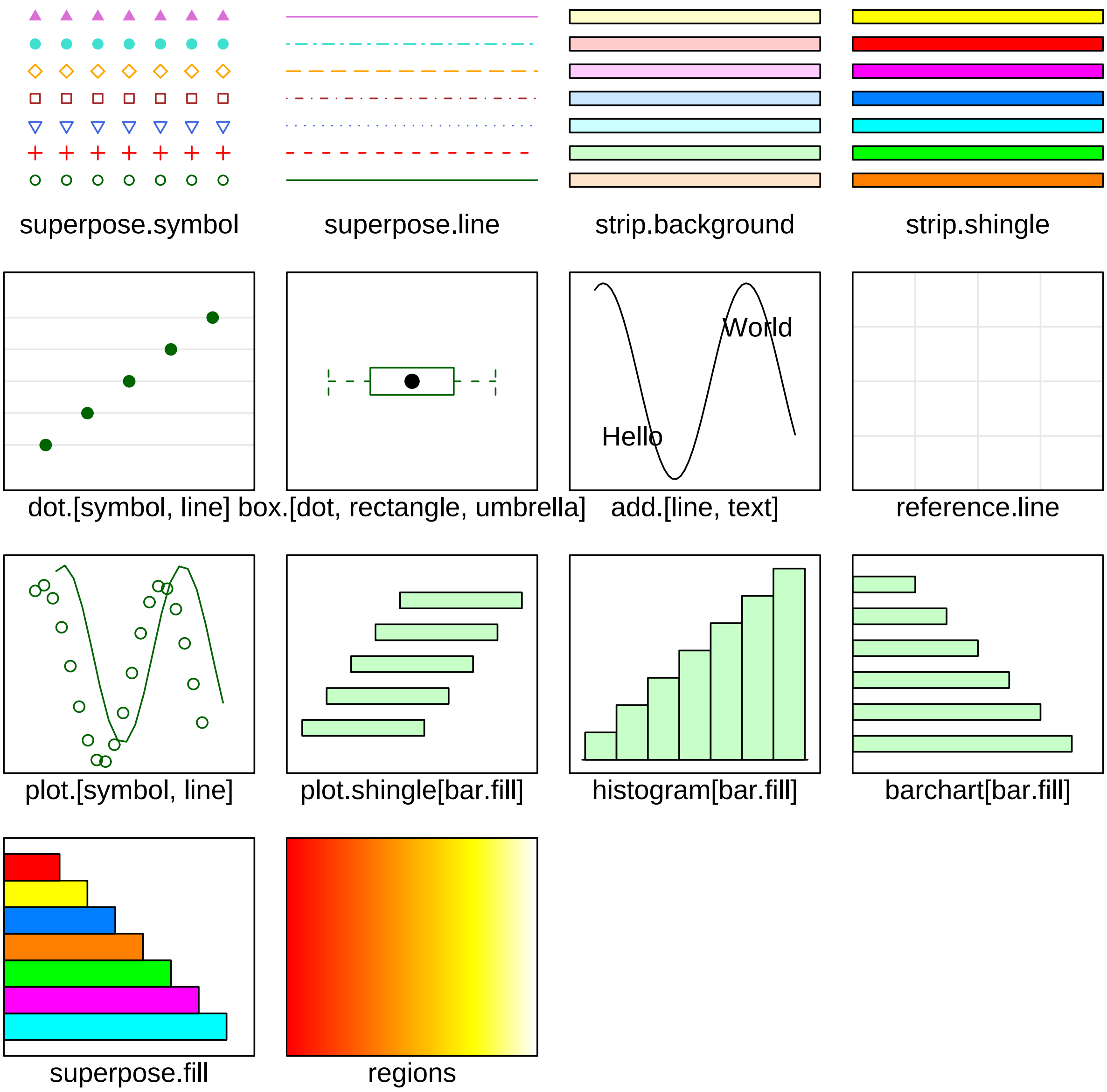


Figure 71: Current settings for trellis graphics.

```
> Rows(sps, 1:3)
$alpha
[1] 1 1 1
$cex
[1] 0.7 0.7 0.7
$col
[1] "purple" "pink" "blue"
$font
[1] 1 1 1
$pch
[1] 16 18 4
> show.settings()
> xyplot(Gas ~ Temp , whiteside,  groups=Insul,
xlab = "External temperature",
ylab = "Gas consumption",panel=panel.superpose,
main = "Whiteside heating data", aspect = 0.6,
ylim = c(0, 7))
```

Example: Stormer Viscometer Data

We illustrate some trellis graphics using the Stormer Viscometer data that is described in Appendix I. The dependent variable is `Time`, while the independent variables consist of `Viscosity` and `Weight`. The theoretical model that we are interested in fitting (and will be used elsewhere in the course is)

$$T = \frac{\beta V}{W - \theta} + \epsilon$$

Plotting `Time` versus `Viscosity` should give straight lines with slope depending on `Weight`.

```
> require(MASS)
> xyplot(Time ~ Viscosity, stormer, groups = Wt,
```

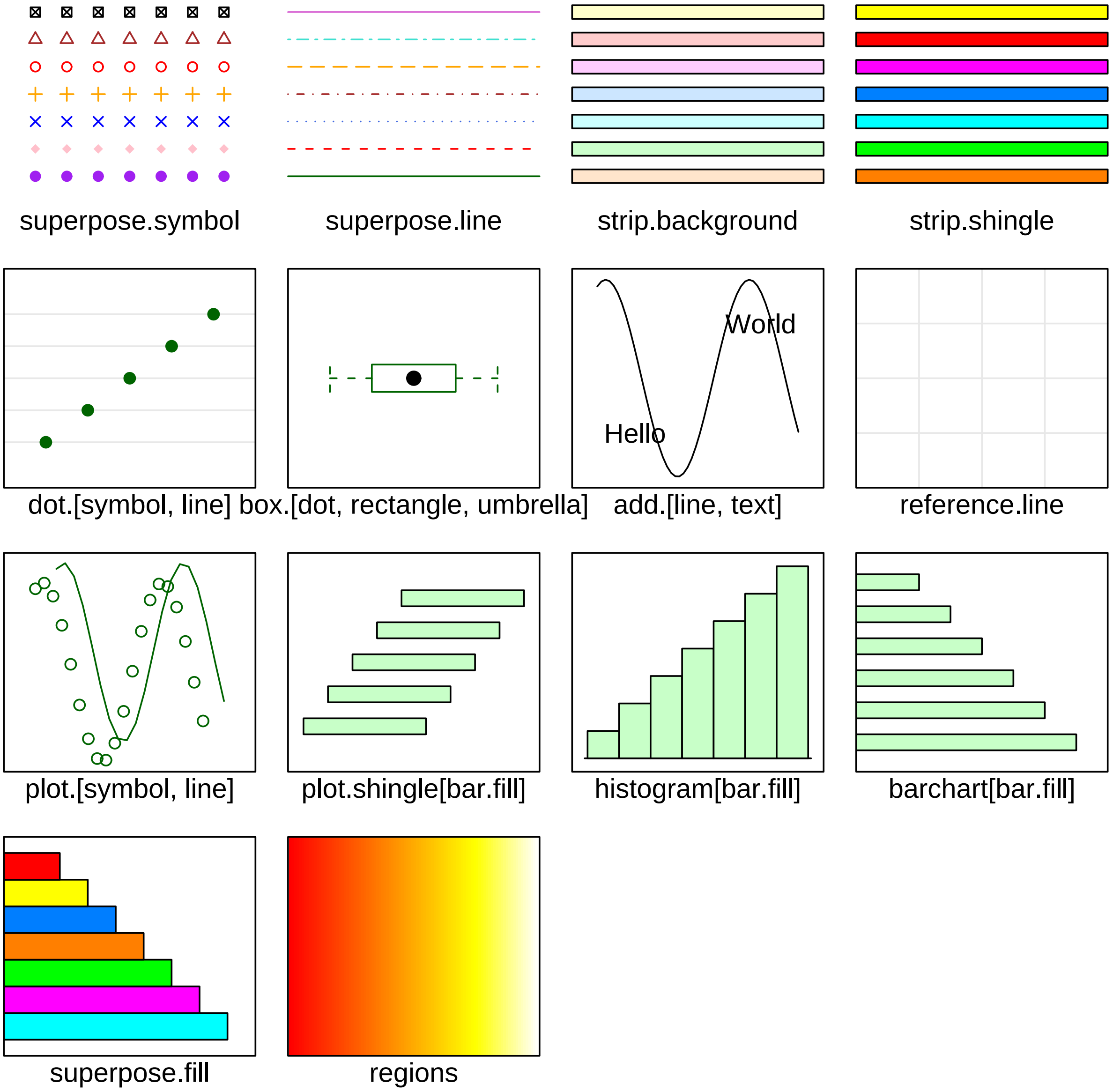


Figure 72: Trellis settings with the symbol information altered.

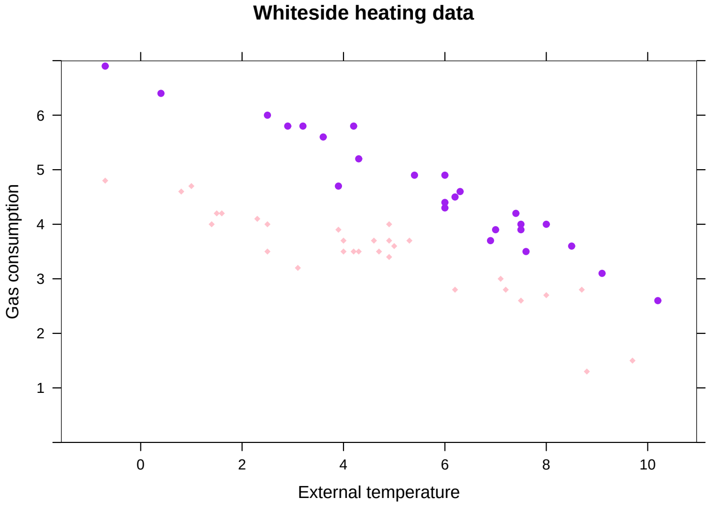


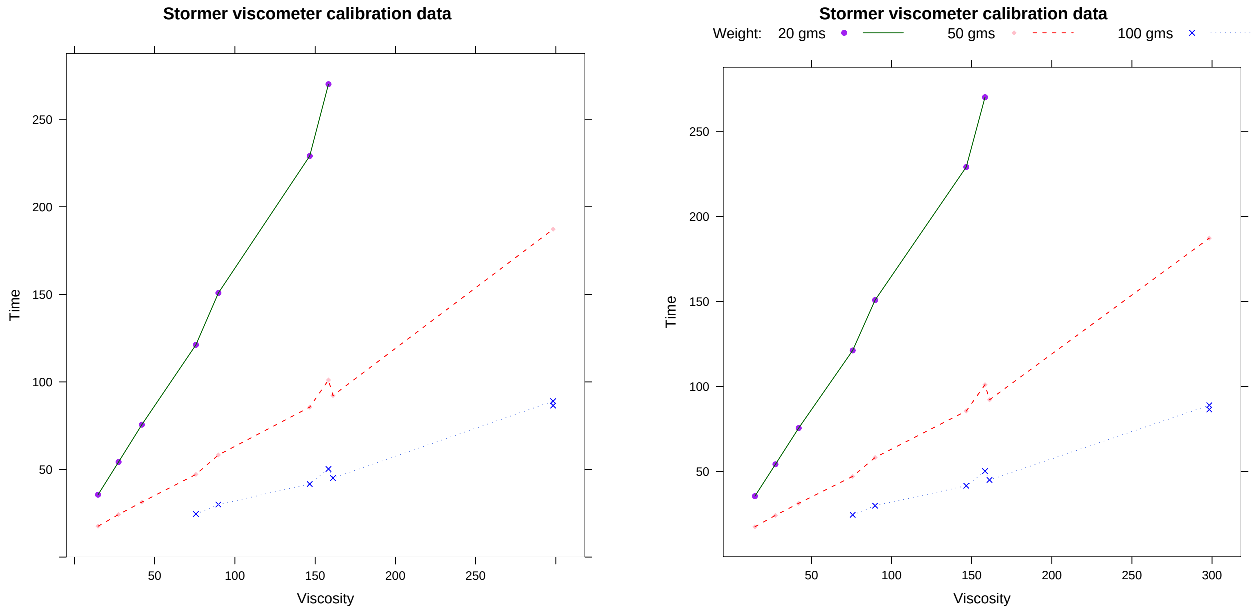
Figure 73: Scatterplot of the whiteside data under the new Trellis settings.

```
panel = panel.superpose, type = "b",
main = "Stormer viscometer calibration data")
```

Figure 74(a) displays the resulting plot for the three types of weight.

To make this plot a little more informative, we add a key or legend to the plot (Figure 74(b)). This key helps us identify the lines corresponding to each weight. Now we can conclude that for larger weights, it takes less time for an inner cylinder to perform a fixed number of revolutions than for smaller weights. For all three weights, the time taken appears to increase linearly with viscosity.

```
> xyplot(Time ˜ Viscosity, stormer,
groups = Wt, panel = panel.superpose,
type = "b",
main = "Stormer viscometer calibration data",
key =
    list(columns = 3,
text = list(paste(c("Weight:   ", "",
```



(a) (b)

Figure 74: Plot of time versus viscosity for three different types of weight (a) without a key and (b) with a key.

```
""),sort(unique(stormer$Wt)), "gms")),
points = Rows(sps, 1:3),
lines = Rows(sps, 1:3))
)
```

Adding Fitted Values

The following script is an illustration of how fitted values can be added to a plot. In the example, we generate some data x, which represents the explanatory variable. We then generate a response y, which represents a quadratic expression of the x's and add some error to the plot. We then produce a plot of the data with fitted values from fitting a spline overlayed. The result is displayed in Figure 75.

```
> dat <- data.frame(x = rep(1:20, 6),
     f = factor(rep(1:6, each = 20)))


> dat <- transform(dat,
y = 1 + (dat$x-10.5)ˆ2 + rnorm(120, sd = 10))
> fm <- lm(y ˜ f + poly(x, 2), dat)
> dat$fit <- fitted(fm)

> xyplot(y ˜ x | f, dat, subscripts = TRUE,
as.table = TRUE,
panel = function(x, y, subscripts, ...) {
panel.xyplot(x, y, ...)
llines(spline(x, dat$fit[subscripts]),
col="orchid")
})
```

Output Over Several Pages

Producing output over several pages is not very well handled in R. Probably the simplest way to do this is to set the graphics parameter:

```
> par(ask = TRUE)
```

and manually save each page as it appears. See the example below for an illustration. Note the use of the function `bringToTop()` to force the plotting window to appear on top of all other windows on the desktop.

```
>  with(dat,for(i in unique(f)){
       plot(x[f==i],y[f==i],xlab="x",ylab="y")
    lines(spline(x[f==i],fit[f==i]),col="red")
title(main=paste("Data group",i))
       bringToTop()
}
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
...
```

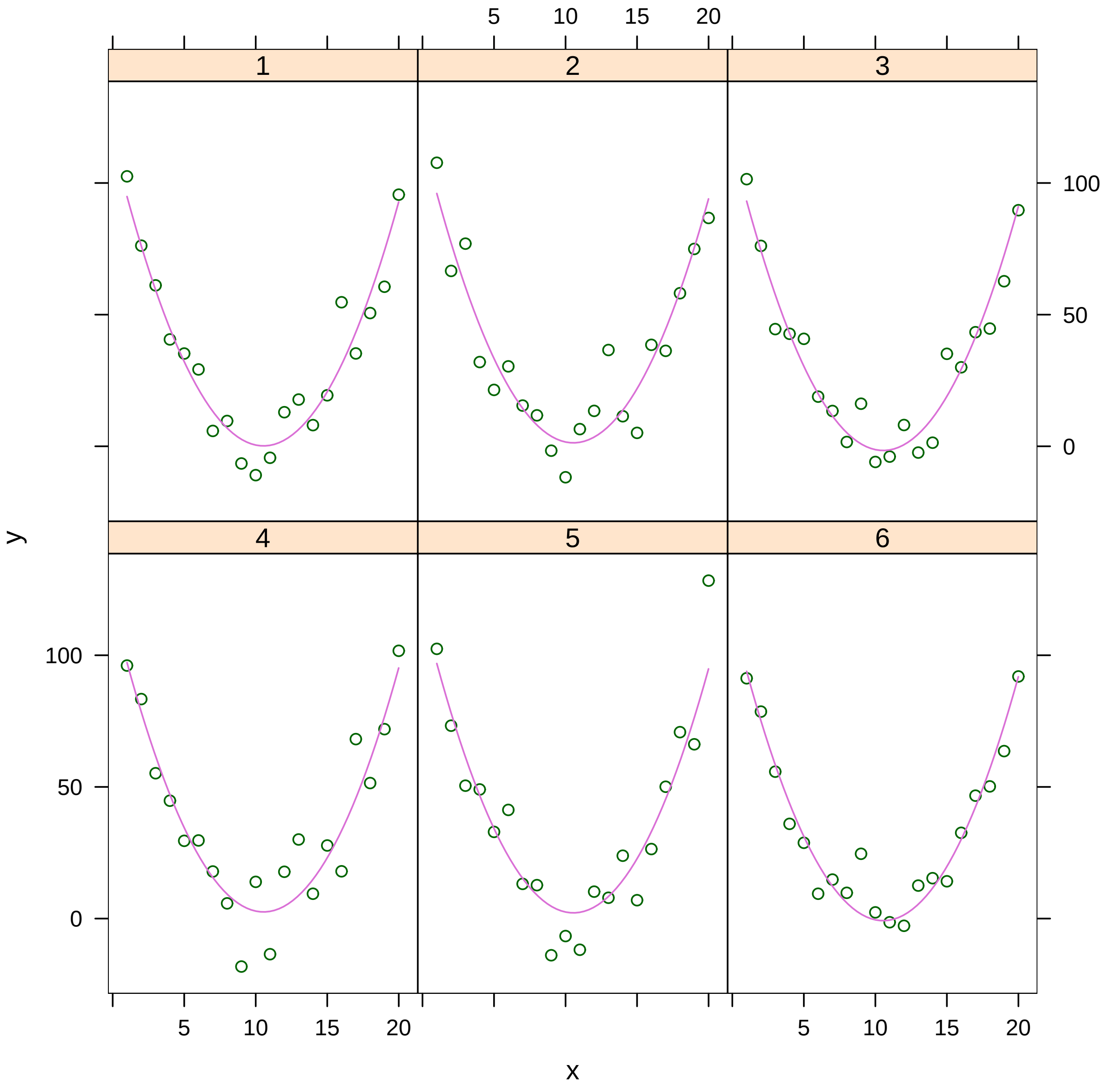


Figure 75: Plot of simulated data with fitted values from spline terms overlayed.

```
> par(ask = FALSE)
```

Unfortunately, the `ask=TRUE` parameter does not work for trellis functions. One way around this is to record the History of graphs displayed using the `History` option in the graphics window.

```
History -> Recording
```

Alternatively, we can send these graphs to file using the % option, which automatically generates files with different file names, increasing by one each time a new graph is plotted.

```
> trellis.device(win.metafile,
width = 7, filename = "myPlots%02d.wmf")
> xyplot(y ˜ x | f, dat, subscripts = TRUE,
as.table = TRUE,
panel = function(x, y, subscripts, ...) {
panel.xyplot(x, y, ...)
llines(spline(x, dat$fit[subscripts]),
col="orchid")
},
layout = c(2,1), aspect = "xy")
> dev.off()
```

Example: Volcanos in New Zealand

The third example in this session explores a topographic map of the Maunga Whau Volcano in New Zealand. The data for this example was digitized from a map by Ross Ihaka.

We begin with an image plot of the volcano terrain using the `image` function in R, setting colours using a predefined colour system `terrain.colors`. A contour is overlayed to highlight the hotspots of the volcano. The resulting plot is shown in Figure 76.

```
> data(volcano)
> x <- 10*(1:nrow(volcano))
> y <- 10*(1:ncol(volcano))
> image(x, y, volcano,col = terrain.colors(100),
axes = FALSE)
> contour(x, y, volcano,
levels = seq(90, 200, by=5),add = TRUE,
col = "peru")
> axis(1, at = seq(100, 800, by = 100))
> axis(2, at = seq(100, 600, by = 100))
> box()
> title(main = "Maunga Whau Volcano",font.main = 4)
```

The `image` function is not part of the lattice library. However, we can achieve similar graphs using the trellis functions available in R. Here is how we might go about this.

```
> length(x)
[1] 87
> length(y)
[1] 61
> dim(volcano)
[1] 87 61
> vdat <- expand.grid(x = x, y = y)
> vdat$v <- as.vector(volcano)
> levelplot(v ~ x*y, vdat, contour=T, main="levelplot()")
> wireframe(v ~ x*y, vdat, drape=TRUE,
main="wireframe():default")
> wireframe(v ~ x*y, vdat, drape=TRUE,
col.regions =rainbow(100), main="wireframe():rainbow")
> wireframe(v ~ x*y,vdat,drape=TRUE, col.regions=topo.colors(100),
main="wireframe():topo.colors")
```

Four types of plots are produced using the above piece of code. The first shown in Figure 77(a) produces an image and an overlayed contour using default colours (heatmap) similar to what was produced by the `image` function. The second plot in Figure 77(b) produces a perspective mesh plot using default colours (heatmap) to show the ridges and gullies of the volcano. The third plot shown in Figure 77(c) is the same plot again but plotted with a different colour scheme (rainbow). The final plot shown in Figure 77(d)

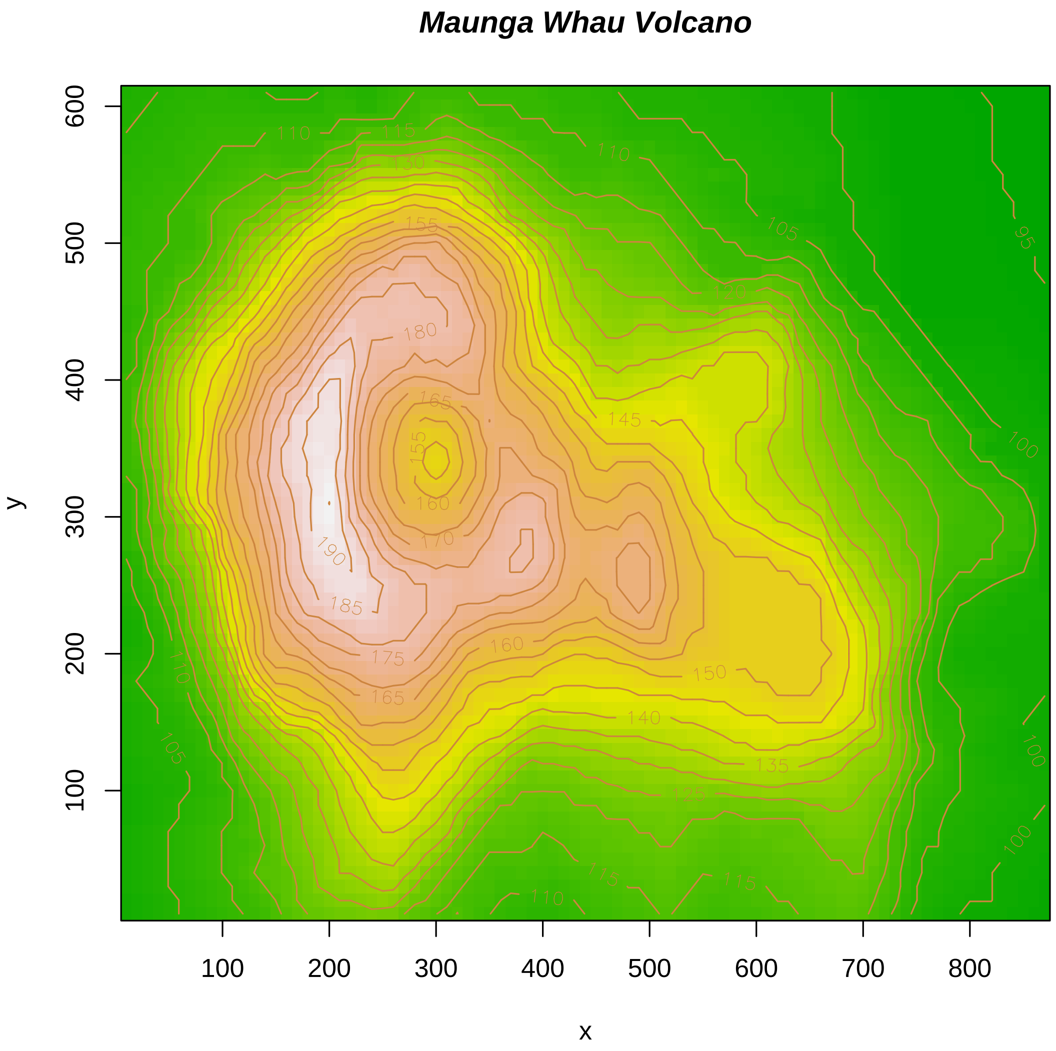


Figure 76: Image plot of the Maunga Whau Volcano in New Zealand. White areas show extreme values of the volcano where volcanic activity is likely to take place.

displays a perspective mesh plot with a topographic colour scheme.

A few notes on lattice graphics . . .

- Lattice graphics are constructed *all in one hit*. It is generally impossible to add to an existing trellis (lattice) graph.
- The functions build graphics objects and printing them produces graphics output. Printing must be done explicitly in loops or functions.
- Separate panels are preferred, with shared axes. This acts as a powerful data inspection device.
- Lattice provides a vast array of arguments and these are best assimilated as and when needed.
- All the action takes place inside the panel function. Using lattice graphics amounts to writing panel functions.

Colour Palettes

There are a range of color palettes predefined in R `rainbow, terrain.colors`, `heat.colors, topo.colors, cm.colors, gray`. We have already seen a couple of them in action.

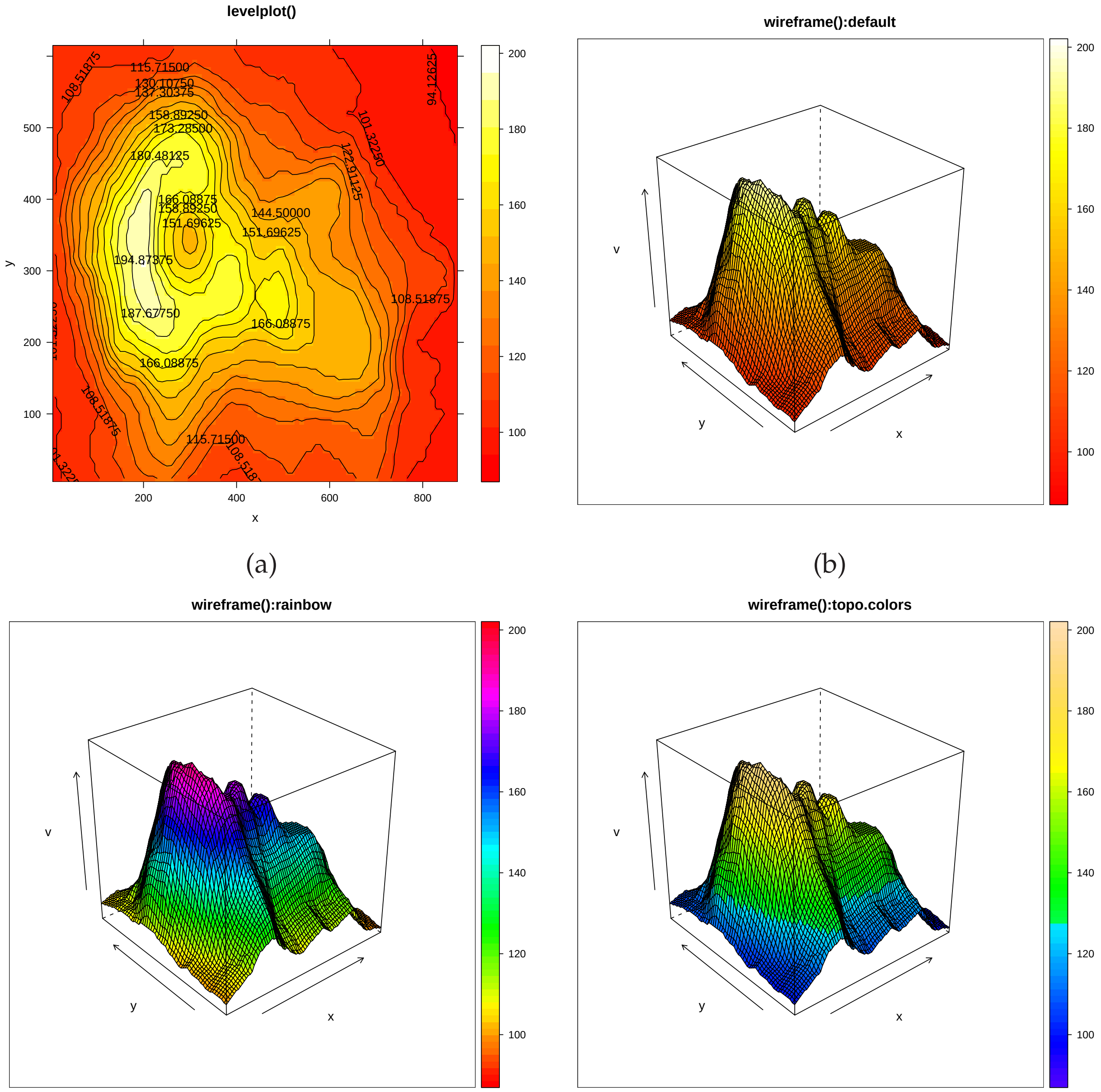
Run the `demo.pal()`, a function defined in the R-Help to view different colour palettes (Figure 78). For grey scale, the grey or gray functions can be used to allocate grey levels to a vector (Figure 79).

```
> barplot(rep(1,20),col=gray((0:20)/20),
main="Grey Scale Palette")
```

User Defined Colour Palettes

The pre-defined colour palettes are sufficient for most applications. However there are some occasions where you may want to define your own colour palette use the `gplots` package. Here is an example using the `volcano` dataset again.

```
> require(gplots)
> cols1 <- colorpanel(25,"green","yellow","red")
> fr <- cut(volcano,breaks=quantile(volcano,0:25/25),
include=T)
```



(a)

(b)

(c)

(d)

Figure 77: Plots showing the contours of a New Zealand volcano using an array of trellis functions

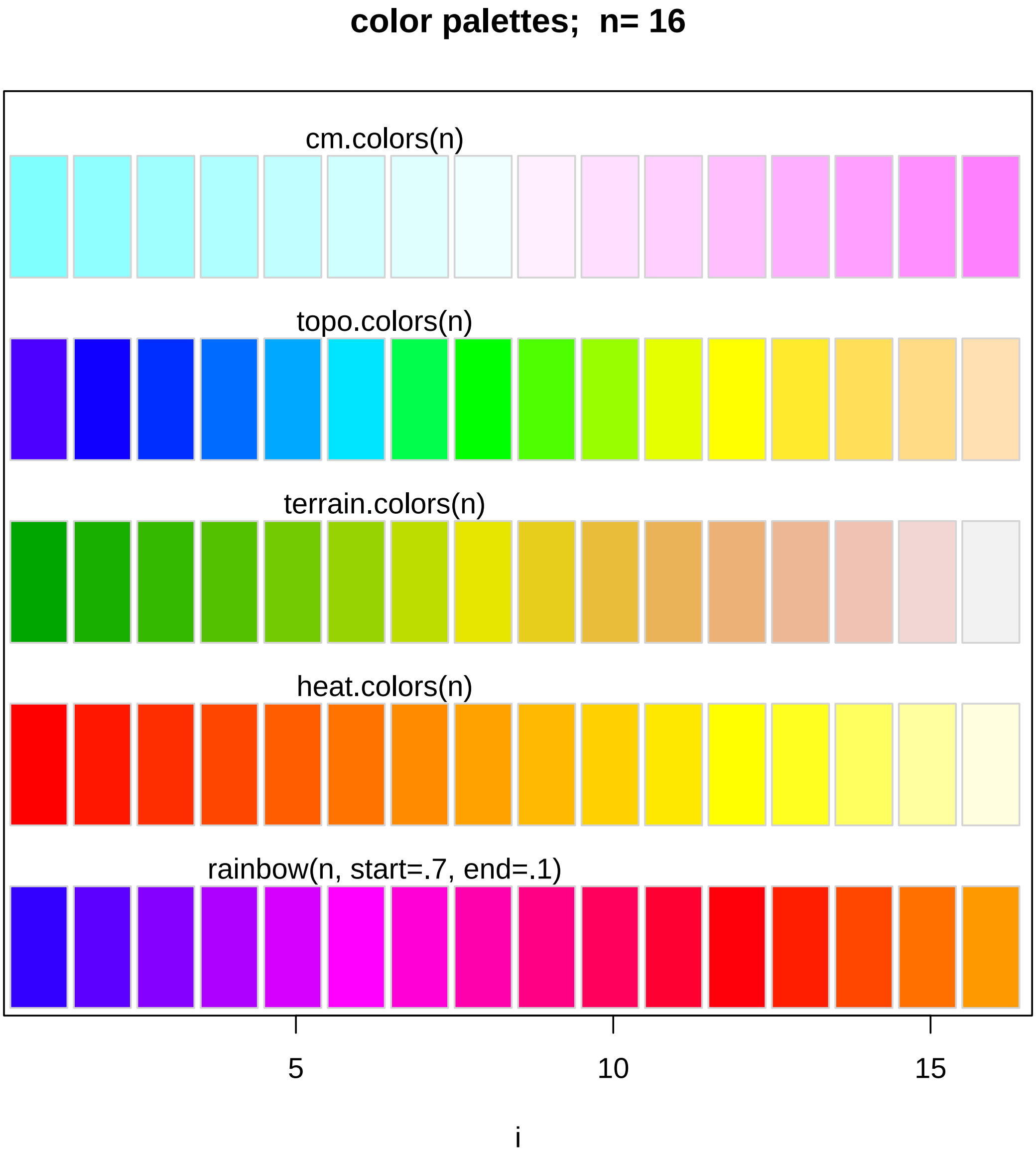


Figure 78: Colour palette produced using the `demo.pal()` function

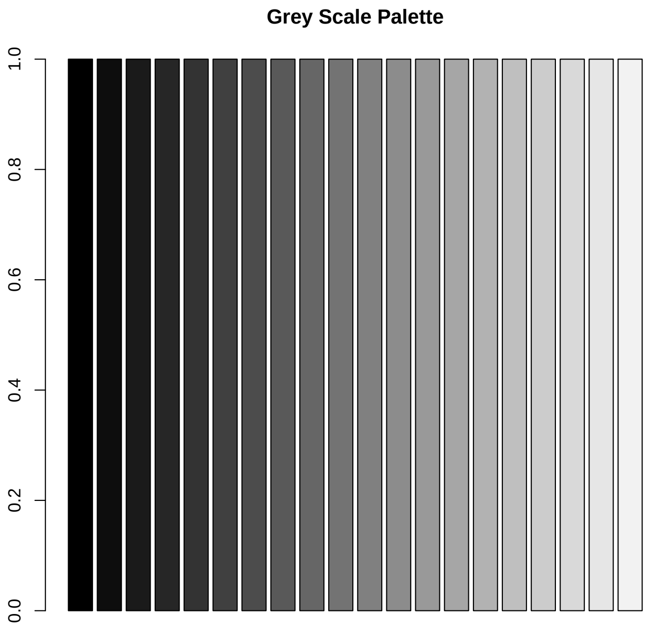
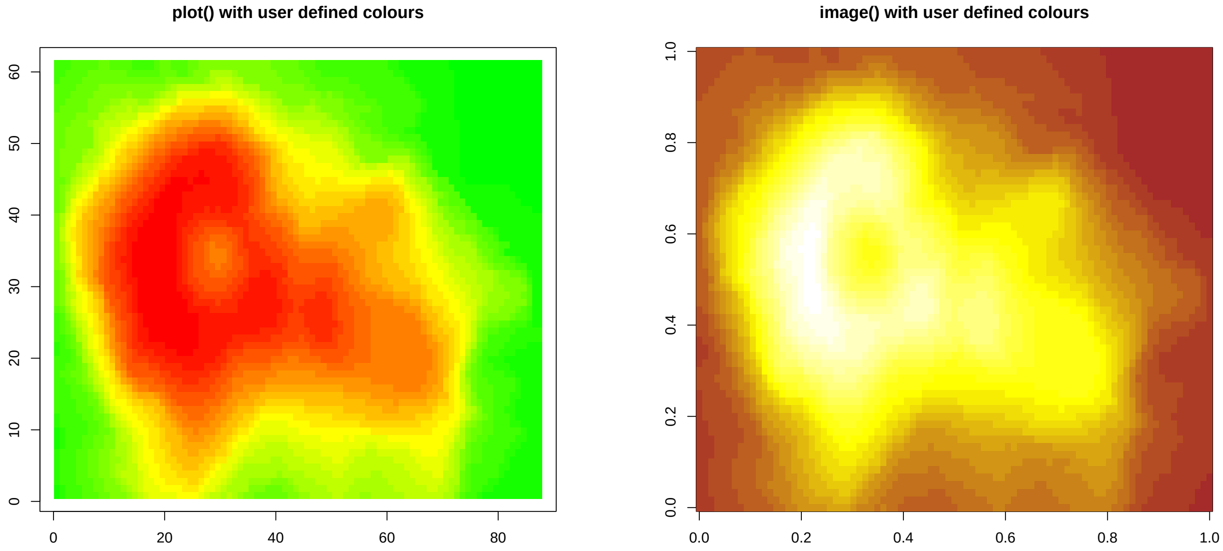


Figure 79: Grey scale palette.

```
> plot(row(volcano),col(volcano),pch=15,
col=cols1[fr],cex=1.5,xlab="",ylab="")       # plot()
> title(main="plot() with user defined colours")
> detach("package:gplots")
> cols2 <- gplots::colorpanel(25,"brown","yellow","white")
> image(volcano,col=cols2,
main="image() with user defined colours")    # image()
```

Figure 80 displays the resulting plots. Both plots are produced with user defined colours, however the plot in Figure 80(a) is produced using the plot command with a specified pixel size (`cex=1.5`) while the other in Figure 80(b) is produced using the `image` function. Both produce satisfactory plots.



(a) (b)

Figure 80: Image plots produced with (a) the `plot` function and (b) the `image` function for the volcano dataset using user defined palettes.

Mathematical Expressions

For many statistical and mathematical applications it is useful to place mathematical expressions on graphs. This can be achieved using `plotmath` which is part of the `grDevices` package.

Mathematical notation can be defined as an expression in any of the text-drawing functions (`text`, `mtext` or `axis`). Output is formatted in a similar way to TeX, so for TeX users, this transition is easy. For a complete overview of symbols see the help file on plotmath and run the demonstration.

`plotmath()`: an Example

Here is an example where mathematic expressions are required on the graph and in the title of the plot. We generate some data using the expression $y = \alpha^2 + \beta_2^2 x$. We generate x from a Normal distribution with a mean of zero and a standard deviation of 1. The resulting figure is shown in Figure 81.

```
# Generate Data
> x <- rnorm(5000)
> b <- 1
> y <- 1+b*x+rnorm(5000)
# Plot Data
> plot(x,y)
# Place mathematical expressions
> text(locator(1),
expression(paste(y,"=",alpha^2+beta[2]^2*x)))
> title(main=substitute(paste("Coefficients: ",alpha,"=",1,",",
        beta[2],"=",b),list(b=b)))
```

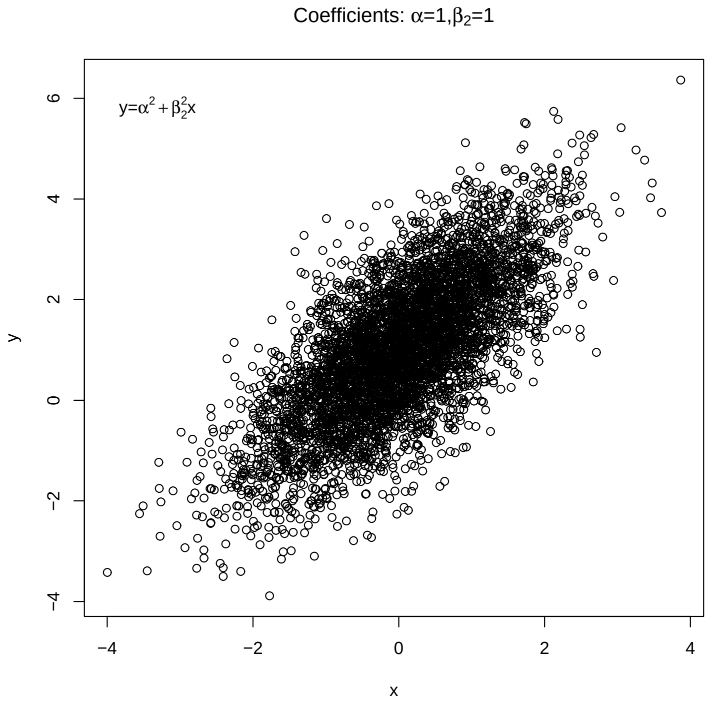


Figure 81: An example of putting mathematical expressions on a plot.

Maps, Coastlines and Co-ordinate Systems

Obtaining maps and coastlines for a study region can be obtained from Geographical Information Systems (GIS) such as ARC View but may require some manipulation. Often this can be quite tedious. An alternative and easy way to obtain coastlines that can be readily plotted in R is to extract a coastline from the *coastline extractor* website. This was developed by Rich Signell and is currently hosted by the National Geophysical Data Center in Boulder Colorado. Go to: *http://rimmer.ngdc.noaa.gov/mgg/coast/getcoast.html* for more information.

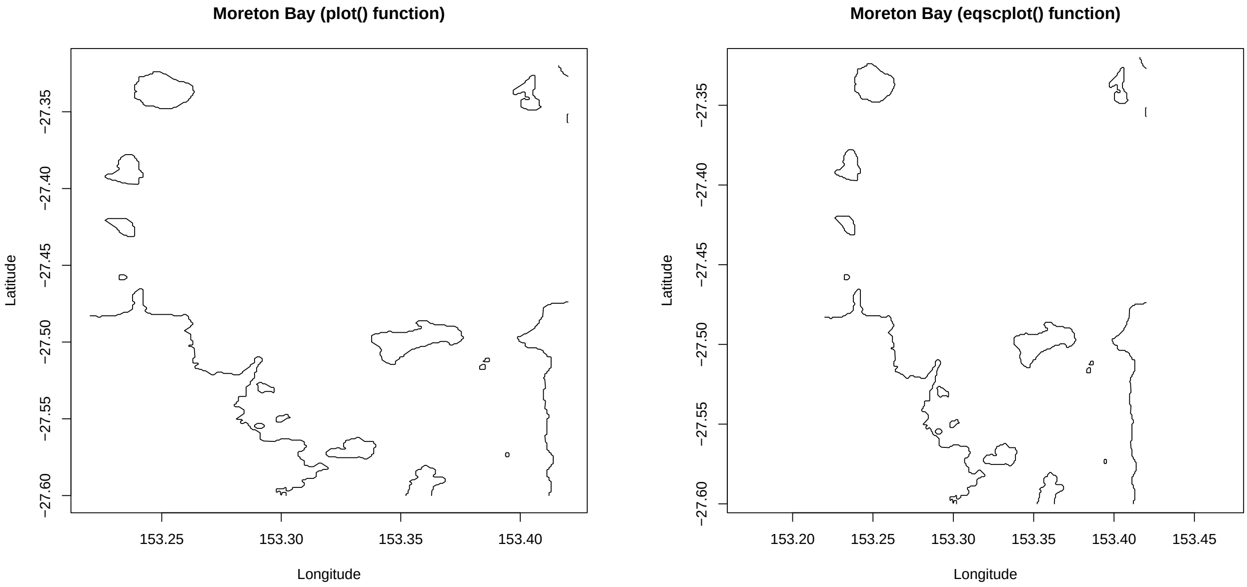
The website and data are free to use but need to be acknowledged in any publications or reports. The extractor program requires a range of latitudes and longitudes in your study region in order to extract the coastlines that you require. If no ranges are specified the entire set of coastlines for the world will be extracted and this is too big to use or manipulate. The website provides an option to save the data in *S-PLUS* format. This is suitable for input into R as it produces a text file with two columns: Longitudes and Latitudes that can be plotted once read in using the `read.table` command. **Try it when you have time.**

When dealing with maps, it is important to think about the plotting region. Furthermore, it is paramount to have a scaling of the axes which is geometrically accurate. This can be achieved using the `eqscplot` function from the MASS library. `eqscplot` ″behaves like the plot function but shrinks the scale on one axis until geometrical accuracy is attained″ (V&R, 2000).

The following example shows what happens when inappropriate scales are used for plotting coastlines. The data used were extracted from the *coastline extractor* website and represent islands in Moreton Bay in South East Queensland, Australia. (This is a study region for some coral and substrate work.)

```
# Read in Co-ordinates
> MB.cl <- read.table("MB_coastline.txt")
> names(MB.cl) <- c("Longitude","Latitude")
# Using plot()
> plot(MB.cl$Long,MB.cl$Lat,type="l",
  xlab="Longitude",ylab="Latitude")
> title(main="Moreton Bay (plot() function)")
# Using "eqscplot()"
> eqscplot(MB.cl$Long,MB.cl$Lat,type="l",
  xlab="Longitude",ylab="Latitude")
> title(main="Moreton Bay (eqscplot() function)")
```

Figure 82(a) shows a plot of the mainland and surrounding coastlines using the `plot` function, which stretches the coastline out of proportion. The plot in Figure 82(b) is the result of using the `eqscplot` function that plots the coastlines in proportion and shows no distortion.



(a) (b)

Figure 82: Plot of mainland and islands in Moreton Bay using data extracted from the *coastline extractor*. The first plot in (a) is produce without appropriate scaling applied to each axis. The second plot in (b) is the result of appropriate scaling using the `eqscplot` function.

Importing and Exporting

Getting Stuff In

There are a number of methods for reading data into R. Here is a summary:

- `scan()` offers a low-level reading facility
- `read.table()` can be used to read data frames from formatted text files
- `read.fwf()` can be used to read in data files that have a fixed width format
- `read.csv()` can be used to read data frames from comma separated variable files.
- When reading from excel files, the simplest method is to save each worksheet separately as a csv file and use `read.csv()` on each. (Wimpy!)
- A better way is to open a data connection to the excel file directly and use the odbc facilities

The easiest form of data to import into R is a text file, but this is only useful for small to medium problems. The primary function for reading in data in R is the `scan()` function. This is the underlying function for other more convenient functions such as `read.table()`, `read.fwf()` and `read.csv()`.

Each of these functions will be examined in more detail in the following sections.

The Low-Level Input Function: `scan()`

The simplest use of the `scan()` function is to read a vector of numbers:

```
> vec <- scan()
1: 22 35 1.7 2.5e+01 77
6:
Read 5 items
> vec
[1] 22.0 35.0  1.7 25.0 77.0
```

A blank line (two returns) signals the end of the input

Reading Characters with scan()

The scan() function can also be used to read in characters.

```
> chr <- scan(what = "", sep = "\n")
1: This is the first string
2: This is the second
3: and another
4: that’s all we need for now
5:
Read 4 items
> chr
[1] "This is the first string"
[2] "This is the second"
[3] "and another"
[4] "that’s all we need for now"
```

Mixed Characters and Numbers

Reading in a mixture of character data and numbers can also be achieved using the scan() function.

```
> lis <- scan(what = list(flag = "", x = 0, y = 0))
1: a 10 3.6
2: a 20 2.4
3: a 30 1.2
4: b 10 5.4
5: b 20 3.7
6: b 30 2.4
7:
Read 6 records
> dat <- as.data.frame(lis)
> dat
  flag  x   y
1    a 10 3.6
2    a 20 2.4
3    a 30 1.2
4    b 10 5.4
5    b 20 3.7
6    b 30 2.4
```

Importing Rectangular Grids: `read.table`

The most convenient way to read in rectangular grids, spreadsheets or text files is using the `read.table()` function. This function allows you to specify a header argument, separators, how to deal with missing values and unfilled or blank lines. This function is only suitable for small-medium sized datasets and is unsuitable for larger numerical matrices. The functions `read.csv` or `read.delim` could also be used to read in comma delimited rectangular files. Furthermore, `read.csv` is often a better choice for reading in comma delimited text files that have been exported from excel.

```
> samp1 <- read.csv("samp1.csv")
> samp1[1:3,]
      ID Name  Prob
1   1    a 0.812
2   2    b 0.982
3   3    c 0.725
```

Importing Fixed Width Formats: `read.fwf`

Fixed width formats is an uncommon format for many datasets. Most datasets are tab or comma delimited. For this type of format, a vector of widths needs to be specified along with the separator. The function `read.fwf` writes out a temporary tab-separated file and then makes a call to `read.table`. This function is useful only for small data files. Here is an example:

```
> dat.ff <- tempfile()
> cat(file=dat.ff,"12345678","abcdefgh",sep="\n")
> read.fwf(dat.ff,width=c(2,4,1,1))
  V1   V2 V3 V4
1 12 3456  7  8
2 ab cdef  g  h
> unlink(dat.ff)  # clean up afterwards
```

Editing Data

The functions `edit` and `fix` allow you to make changes to data files using a default editor. This is useful for small datasets and for obtaining a snapshot of your data without writing to file or printing to the screen.

The `fix` function allows you to make edits and then assigns the new edited version to the workspace

```
> fix(samp1)
```

The `edit` function invokes a text editor on the object, the result of which is a copy that can be assigned to a new object.

```
> samp1.new <- edit(samp1)
```

Importing Binary Files

Binary data written from another statistical package can be read into R (but really should be avoided). The R package `foreign` provides import facilities for: EpiInfo, Minitab, S-Plus, SAS, SPSS, Stat and Systat binary files. Here is a list of them.

- `read.epiinfo()` reads in EpiInfo text files
- `read.mtp()` imports Minitab worksheets
- `read.xport()` reads in SAS files in TRANSPORT format
- `read.S()` reads in binary objects produced by S-PLUS 3.x, 4.x or 2000 on (32-bit) Unix or Windows Data dumps from S-PLUS 5.x and 6.x using `dump(..., oldStyle=T)` can be read using the `data.restore` function
- `read.spss()` reads in files from SPSS created by the *save* and *export* commands
- `read.dta()` reads in binary Stata files

- `read.systat()` reads in rectangular files saved in Systat

Reading in Large Data Files

There are some limitations to the types of files that R can read in. Large data files can be a problem and here is the reason why. R stores objects and datasets in memory. So several copies of a dataset can be kept when executing a function. Data objects > 100Mb can cause R to run out of memory.

Database management systems may be more appropriate for extracting and summarising data. Open Database Connectivity (ODBC) allows a user to access data from a variety of database management systems.

There are several packages available that provide different levels of functionality: copying, data selection, querying and retrieval. Most however, relate to specific relational databases.

`RODBC` is a package that can deal with a few: Microsoft SQL Server, Access, MySQL (on Windows), Oracle and (also Excel, Dbase and text files). Many simultaneous connections are possible with `RODBC`.

The RODBC Package

As mentioned above, `RODBC` is a package that allows you to connect with a database and retrieve information. Important functions within RODBC include

- Establish connections to ODBC databases
  `odbcConnect, odbcConnectAccess, odbcConnectDbase, odbcConnectExcel`
- Listing of tables on an ODBC database
  `sqlTables`
- Reads a table on an ODBC database
  `sqlFetch`
- Queries an ODBC database and retrieves the results
  `sqlQuery`

Importing Excel Spreadsheets into R: An Example

The RODBC library allows you to import multiple sheets directly from an excel file (Note, sheet names cannot contain any spaces for this feature to work!). Here is an example.

```
> library(RODBC)

> con <- odbcConnectExcel("ExampleData.xls")  # open
          # the connection
> con
RODB Connection 1
Details:
   case=nochange
   DBQ=D:\Data\My Documents\Slides\R Course\ExampleData.xls
   DefaultDir=D:\Data\My Documents\Slides\R Course
   Driver={Microsoft Excel Driver (*.xls)}
   DriverId=790
   MaxBufferSize=2048
   PageTimeout=5
```

What tables (spreadsheets) are available? Using the `sqlTables` command, we find that there are two tables available, one called `samp1` and a second called `samp2`.

```
> sqlTables(con)
  TABLE_CAT
1 D:\\Data\\My Documents\\Slides\\R Course\\ExampleData
2 D:\\Data\\My Documents\\Slides\\R Course\\ExampleData

 TABLE_SCHEM TABLE_NAME   TABLE_TYPE REMARKS
1       <NA>     samp1$ SYSTEM TABLE    <NA>
2       <NA>     samp2$ SYSTEM TABLE    <NA>

> samp1 <- sqlFetch(con, "samp1")
> samp2 <- sqlFetch(con, "samp2")
> odbcCloseAll() # close the connection
```

Getting Stuff Out

Writing objects out is a much more easier task than reading files in. The function `cat` is the underlying function used to export data. The function can either write objects to file or alternatively, print the object to the screen. The `sink` function can be used to write data to file as well.

The `print` function prints the object and returns it invisibly.

The function `write`, writes a vector or matrix to file with a specified number of columns. For writing out matrices or data frames, `write.table` is recommended, however for very large matrices, `write.matrix` (MASS library) is the better option. The function `write.matrix` has the option of writing the matrix in blocks to cut down on the memory usage associated with writing large datasets to file.

The `cat` Function

The `cat` function is very useful for printing objects to the screen. For example,

```
> cat("Hello World\n")
Hello World
```

Alternatively, we could write text to file,

```
> cat("Hello World\n",file="output.txt")
```

Here is another example where we are writing text and output to the screen.

```
> pval <- 1-pchisq(2,1)
> pval
[1] 0.1572992
> cat("Test for Independence: p-value=",
round(pval,3),"\n")
Test for Independence: p-value= 0.157
```

The `sink` Function

The sink function can be used to send objects and text to a file. This is useful when you want to look at the contents of an object or function that may be too big to display on screen. Here is an example

```
> sink("output.txt")
> sample(1:100,100,replace=T)
> letters[1:10]
> sink()
```

Contents of *output.txt*

```
 [1]  29  19   8  35  82  29  35   3  61  19  45  58  49  71
[15] 38  31  82  64  64  16  19   8  26  73  33  40   3  35
[29] 25   9  13  16  52  40  63  59  24  22  78  16  34  50
[43]  5  54   2  22  67  96  33  52  50  72  56  43  23  81
[57] 68  95  56  15   6  42  89  82  92  10  79   4  99   8
[71] 57  29  72  87  98  24 100  82  38  59  69  40  45  40
[85] 56  18  45  76  29  76  35  74   8  76  41  69   8  29
[99]  9  27
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

The `write.table` Function

The function `write.table` can be used to write datasets to file. The function is more elaborate than `cat`, `print` and `sink`. We present two examples for writing data explicitly out to a file.

Example 1: Creating a connection before writing

```
> con <- file("myData.csv", "w+")
> write.table(myData, con, sep = ",")
> close(con)
```

Example 2: Writing data out explicitly

```
> write.table(myData, "myData.txt")
```

The `write.matrix` Function

For larger datasets, `write.matrix` may be more efficient in terms of how memory is handled. Data can be output in blocks using the `blocksize` argument as shown in the following script.

```
> require(MASS)
> write.matrix(myData,file="myData.csv",sep=",",
   blocksize=1000)
```

Getting Out Graphics

There are four ways to output graphics from R. Some are more straight forward than others. and it will depend on how you want to use and display the graphics. The following lists the choices available:

1. The `postscript` function will start a graphics device and produce an encapsulated postscript file containing the graphic
   - This is a flexible function that allows you to specify the size and orientation of the graphic.
   - These can be incorporated directly into LaTeX using the `includegraphics` command

   ```
   > postscript("graph.ps",paper="a4")
   > hist(rnorm(10000))
   > dev.off()
   windows
         2
   ```

   Other functions are available within R and can be used in a similar way: `windows, pdf, pictex, png, jpeg, bmp` and `xfig`

2. A simpler, less flexible version is achieved via the R Graphics window. A graphic can be saved by going

   ```
   File -> Save As
   ```

   Many options are listed: `metafile, postscript, pdf, png, bitmap` or `jpeg`

3. Graphics can be copied to the clipboard and pasted into Windows applications such as Word, Excel or Powerpoint. This is the most useful way to get graphics into presentations.

   ```
   File -> Copy to the clipboard -> as Metafile
   ```

4. Printing directly from R. This can be achieved using the print feature on the R Graphics Window.

Mixed Effects Models: An Introduction

Linear Mixed Effects Models

Example: Petroleum Data

The Petroleum dataset of N H Prater is described in Appendix 1. It contains information on ten samples of crude oil, which were partitioned and tested for yield at different *end points*. The samples themselves are classified by specific gravity (SG), vapour pressure (VP) and volatility as measured by the ASTM 10% point (V10).

We investigate this data for two specific reasons. First, we are interested in estimating the rate of rise in yield (Y) with end point (EP). Second, we are interested in explaining differences between samples in terms of external measures.

The plot shown in Figure 83 shows a scatterplot of yield versus end point broken up into panels by crude oil. Least square lines are overlayed on each plot to look for linearity.

```
> xyplot(Y ˜ EP | No, petrol,
panel = function(x, y, ...) {
panel.xyplot(x, y, ...)
panel.lmline(x, y, col = 3, lty = 4)
},
  as.table = T, aspect = "xy",
  xlab = "End point",
  ylab = "Yield (%)")
```

We can see quite evidently that straight lines fitted to the data is reasonable, but can the slopes be regarded as parallel? To investigate this question we fitted two types of models. The first fits separate slopes for crude oil, while the second just fits main effects terms. The analysis of variance table below suggests that parallel slopes cannot be discarded.

```
> petrol.lm1 <- aov(Y ˜ No/EP, petrol)
> petrol.lm2 <- aov(Y ˜ No + EP, petrol)
> anova(petrol.lm2, petrol.lm1)

Analysis of Variance Table
```

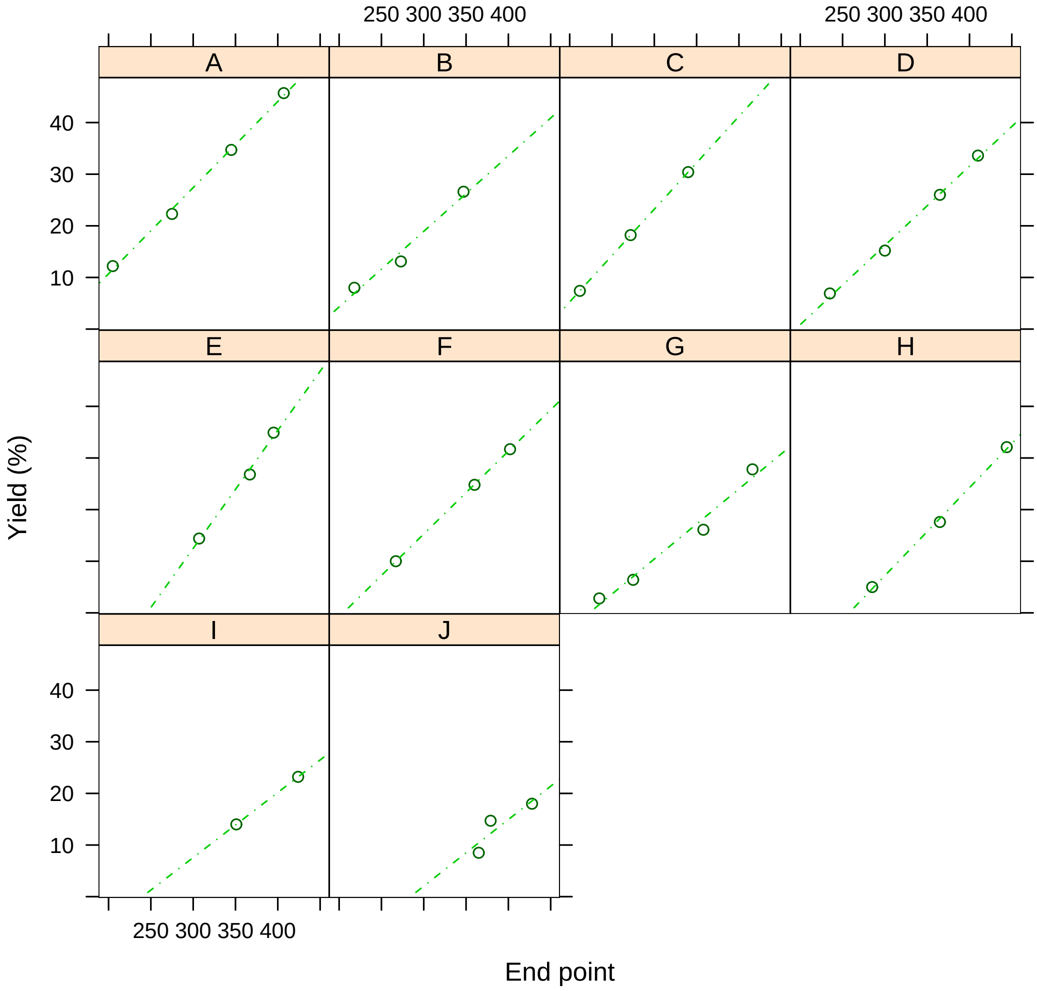


Figure 83: Scatterplot of yield versus end point broken up by crude oil.

```
Model 1: Y ~ No + EP
Model 2: Y ~ No/EP
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1     21 74.132
2     12 30.329  9    43.803 1.9257 0.1439
```

Can we explain differences in intercepts? To investigate this question we fit a multiple linear model to the data with crude oil removed and examine the fit against the model containing just crude oil and end point. The p-value is significant at the 5% level of significance and suggests that there is no real difference between intercepts.

```
> petrol.lm3 <- aov(Y ~ . - No, petrol)
> anova(petrol.lm3, petrol.lm2)

Analysis of Variance Table

Model 1: Y ~ (No + SG + VP + V10 + EP) - No
Model 2: Y ~ No + EP
  Res.Df     RSS Df Sum of Sq      F  Pr(>F)
1     27 134.804
2     21  74.132  6    60.672 2.8645 0.03368 *
```

We now attempt to look at the predictions from all three models: (1) Separate regression lines for each type of crude oil (green); (2) Separate intercept terms but a common slope (blue); (3) Common intercept term and common slope (red). Figure 84 presents the results from `xyplot` which compares the three models.

```
> tmp <- update(petrol.lm2, .~.-1)
> a0 <- predict(tmp, type="terms")[, "No"]
> b0 <- coef(tmp)["EP"]

> a <- with(petrol,cbind(1, SG, VP, V10)) %*% coef(petrol.lm3)[1:4]
> b <- coef(petrol.lm3)["EP"]
> xyplot(Y ~ EP | No, petrol, subscripts = T,
panel = function(x, y,  subscripts, ...) {
panel.xyplot(x, y, ...)
panel.lmline(x, y, col = "green", lty = 1)
panel.abline(a0[subscripts][1], b0, col = "blue",
lty = 2)
panel.abline(a[subscripts][1], b, col = "red",
lty = 3)
}, as.table = T, aspect = "xy", xlab = "End point",
ylab = "Yield (%)",
key=list(lines= list(lty = 1:3,
col = c("green","blue","red")),columns=3,
       text=list(c("separate","parallel","explained"))))
```

Adding a Random Term

The external (sample-specific) variables explain much of the differences between the intercepts, but there is still significant variation between samples unexplained. One natural way to model this is to assume that, in addition to the systematic variation between intercepts explained by regression on the external variables, there is a random term as well. We assume this term to be normal with zero mean. Its variance measures the extent of this additional variation. For more information about fitting random effect terms in the model see Pinheiro & Bates (2000).

Fitting random effect terms in R can be achieved using the `nlme` library and more specific, the `lme` function as shown below.

```
> require(nlme)
> petrol.lme <- lme(Y ~ SG + VP + V10 + EP, data = petrol,
    random = ~1 | No)


> summary(petrol.lme)


Linear mixed-effects model fit by REML
```

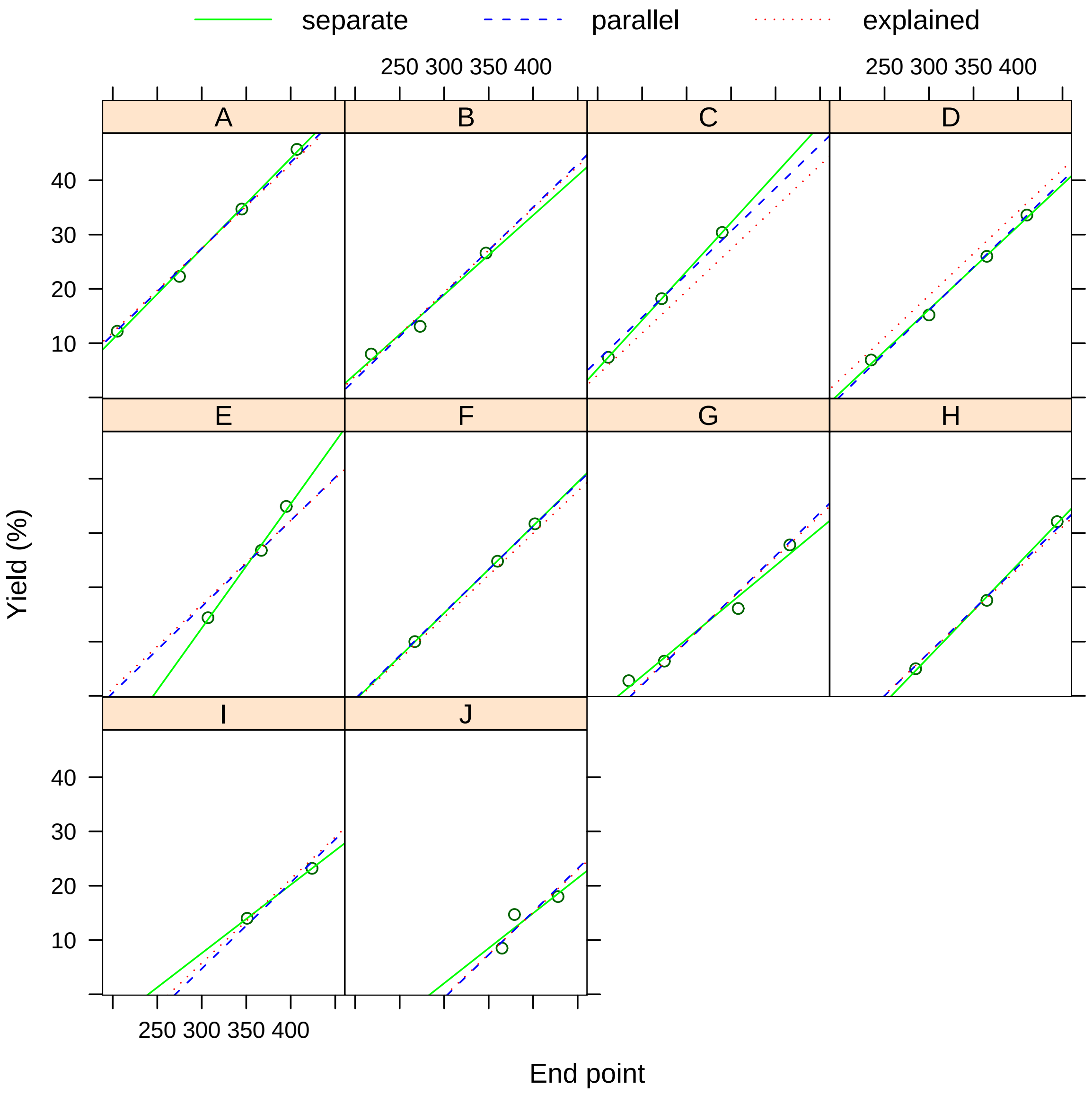


Figure 84: Comparison of three models using `xyplot`

```
Random effects:
 Formula:  ~ 1 | No
        (Intercept) Residual
StdDev:    1.445028  1.872146

Fixed effects: Y ~ SG + VP + V10 + EP
                Value Std.Error DF   t-value p-value
(Intercept) -6.134690  14.55592 21  -0.42146  0.6777
         SG  0.219397   0.14696  6   1.49295  0.1861
         VP  0.545861   0.52059  6   1.04855  0.3348
        V10 -0.154244   0.03997  6  -3.85929  0.0084
         EP  0.157177   0.00559 21  28.12841  0.0000
```

This model fits random intercept terms for crude oil and fixed terms for the other explanatory variables in the model. By default, the model is fitted using restricted maximum likelihood (REML). The output summarises the variation between crude oil samples ($\sigma_C^2 = 2.088$) and other variation not explained by the model ($\sigma^2 = 3.505$). Fixed effect terms are summarised in a table beneath the random effect terms in the usual way. Only endpoint and volatility are important predictors in the model.

If we compare these results with the previous model shown below, we notice that both SG and VP have changed considerably in terms of their level of significance. In the previous fixed effects model, SG was a significant predictor but in the mixed effects model, it is no longer significant. This suggests that most of the variability that was being explained by SG was captured in the random effect term in the model. This indicates that there are differences between crude oil samples.

```
> round(summary.lm(petrol.lm3)$coef, 5)
               Value Std. Error   t value Pr(>|t|)
(Intercept) -6.82077   10.12315  -0.67378  0.50618
         SG  0.22725    0.09994   2.27390  0.03114
         VP  0.55373    0.36975   1.49756  0.14585
        V10 -0.14954    0.02923  -5.11597  0.00002
         EP  0.15465    0.00645  23.99221  0.00000
```

```
Fixed effects: Y ~ SG + VP + V10 + EP
                Value Std.Error DF   t-value p-value
(Intercept) -6.134690  14.55592 21  -0.42146  0.6777
         SG  0.219397   0.14696  6   1.49295  0.1861
         VP  0.545861   0.52059  6   1.04855  0.3348
        V10 -0.154244   0.03997  6  -3.85929  0.0084
         EP  0.157177   0.00559 21  28.12841  0.0000
```

Adding Random Slopes

The deviation from parallel regressions was not significant, but still somewhat suspicious. We might consider making both the intercept and the slope have a random component. This is also achieved using the random argument in the `lme` model as shown below.

```
> petrol.lme2 <- lme(Y ~ SG + VP + V10 + EP, data = petrol,
   random = ~ 1 + EP | No)

> summary(petrol.lme2)

> summary(petrol.lme2)

Linear mixed-effects model fit by REML

Random effects:
 Formula: ~1 + EP | No
 Structure: General positive-definite, Log-Cholesky parametrization
            StdDev      Corr
(Intercept) 1.728521061 (Intr)
EP          0.003301084 -0.537
Residual    1.853930438

Fixed effects: Y ~ SG + VP + V10 + EP
```

```
                 Value Std.Error DF   t-value p-value
(Intercept) -6.230273 14.633344 21 -0.425759  0.6746
SG           0.219106  0.147953  6  1.480914  0.1891
VP           0.548149  0.523157  6  1.047772  0.3351
V10         -0.153937  0.040216  6 -3.827752  0.0087
EP           0.157249  0.005667 21 27.749578  0.0000
```

The results suggest substantial variation between crude oil samples as indicated before ($\sigma_C^2 = 2.99$) and only slight variation attributed to slopes. Other unexplained variation is estimated to be around 3.4. Once again, only V10 and EP are significant in the fixed effects summary.

Generalized Linear Mixed Effects Models

Generalized linear mixed effects models (GLMMs) are an extension to linear mixed effects models that allow for non-normal error structures. The glmmPQL function within the MASS library fits GLMMs. We discuss these models in the context of a marine example.

Example: Great Barrier Reef

The recovery of benthos after trawling on the Great Barrier Reef is an important research topic in Australia. To investigate the impact of trawling on benthos a longitudinal study was devised consisting of six control plots and six impact plots. These were visited two times prior to treatment and four times after treatment. Inspection of the trawled area was conducted by benthic sled.

The total number of animals of each species was counted for each transect and the mean trawl intensity in the transect was recorded along with the total swept area of the sled video camera, topography and other variables.

The Model

The model considered is a generalized linear mixed effects model for the total count Y_{pmd} for a control or pre-experiment treatment plot using a Poisson distribution as follows:

$$Y_{pmd}|C_m, E_{mp} \sim \mathrm{Po}(\exp(\tau_d + C_m + E_{mp})), \quad C_m \sim \mathrm{N}(0, \sigma_c^2), \quad E_{mp} \sim \mathrm{N}(0, \sigma_E^2)$$

In the above equation, Y_{pmd} represents the total count taken at plot p, month m and depth d; τ_d is the treatment at depth d; C_m represents the cruise conducted in month m, E_{mp} represents an error term.

The random terms in the model are important as they allow for uncontrolled influences due to the particular cruise and plot. After adjusting for this variability it is envisaged that the true effects due to trawling and recovery can be more easily disentangled.

For a post-experiment treatment plot the distribution is similar, but the Poisson mean has additional spline terms in mean trawl intensity ($n_s(t)$) and a factor term to allow for recovery (γ_m).

$$Y_{pmd}|C_m, E_{mp} \sim \mathrm{Po}(\exp(\tau_d + n_s(t) + \gamma_m + C_m + E_{mp}))$$

Some Functions and Data

Before fitting the model, some manipulation of the data is required. Furthermore, functions for setting knots in spline terms for the post-experiment treatment plot are required. These are set out below.

```
> Species <- scan("SpeciesNames.csv", what = "")
> Benthos <- read.csv("Benthos.csv")
> match(Species, names(Benthos))
[1] 13 16 20 21 22 23 27 29 30 31 34 37 39 40 42 44 45 48
```

These first few lines identify rows in which species names from the `Species` data file match those in the `Benthos` data file.

The `bin` function shown below, converts a factor into a matrix of treatment contrasts. Try running it on the `Time` variable in the `Benthos` dataset.

```
> bin <- function(fac, all = F) {
#
# binary matrix for a factor
#
  fac <- as.factor(fac)
  X <- outer(fac, levels(fac), "==") + 0
  dimnames(X) <- list(NULL,paste("B",1:ncol(X),sep=""))
  if(all) X else X[, -1]
}
```

The `nsi` function sets up knots for variables based on quantiles of the data. It temporarily attaches the `splines` library to construct natural splines of the data. The `guard` function sets values less than -5 to missing. This is used in the plotting routines that follow.

```
> nsi <- function(x, k = 3, Intensity = Benthos$Intensity){
#
# natural spline in intensity
#
knots <- as.vector(quantile(unique(Intensity),0:k/k))
splines::ns(x, knots = knots[2:k],
    Boundary.knots = knots[c(1, k + 1)])
}

> guard <- function(x) ifelse(x < -5, NA, x)
```

Setting up the Prediction Data

The following piece of code massages the data into a format for fitting the model and obtaining predictions. It creates transformations of the data and joins them with the existing data to form a new prediction dataset, `newData`.

```
> vals <- with(Benthos, tapply(Intensity, Impact, mean))
> msa <- with(Benthos, mean(SweptArea))

> newData <- Benthos[, c("Cruise", "Plot", "Months",
"Treatment", "Time", "Impact", "Topography", "Unity")]
> newData <- transform(newData,
           Months = as.numeric(as.character(Months)),
           Intensity = vals[factor(Impact)],
           SweptArea = rep(msa, nrow(Benthos)))
> newData <- with(newData, newData[order(Months), ])
```

The Key Species

To get a listing of the key species in the dataset, the `print` function can be used. These are names of some columns in the Benthos data frame. We need to fit the master model for each and produce plots of the fixed and random effects

```
> print(Species, quote = F)
 [1] Alcyonacea                Annella.reticulata
 [3] Ctenocella.pectinata      Cymbastela.coralliophila
 [5] Dichotella.divergens      Echinogorgia.sp.
 [7] Hypodistoma.deeratum      Ianthella.flabelliformis
 [9] Junceella.fragilis        Junceella.juncea
[11] Nephtheidae               Porifera
[13] Sarcophyton.sp.           Scleractinia
[15] Solenocaulon.sp.          Subergorgia.suberosa
[17] Turbinaria.frondens       Xestospongia.testudinaria
```

The Trick: Fitting Multiple Similar Models

In order to fit multiple similar models we need to set up a function. The following piece of code defines the GLMM that was described above. The `Quote()` function is required because it tells R to treat everything between the brackets as text. If the `Quote()` function is not used, R will try to evaluate the right hand side and we do not want to do this yet.

```
> fitCommand <- Quote({
 fm <- glmmPQL(Species ~ Topography +
I(Impact * cbind(nsi(Intensity), bin(Time))) +
offset(log(SweptArea)),
random = ~1|Cruise/Plot,
family = poisson, data = Benthos, verbose = T)
})
```

Putting the Trick to Work

To put this trick to work we need to construct a *manual* loop and we start with defining `spno` to equal 1.

```
> spno <- 1

### --- start of (manual) main loop

> mname <- paste("GLMM", Species[spno], sep=".")
> sname <- Species[spno]
> thisFitCommand <- do.call("substitute", list(fitCommand,
list(fm = as.name(mname), Species = as.name(sname))))

> eval(thisFitCommand)
> print(spno <- spno + 1)
[1] 2
### --- end of main loop
> objects(pat = "GLMM.*")
[1] "GLMM.Alcyonacea"
```

The `mname` and `sname` objects get the model name, which will store the model results and species name respectively. The `do.call` function executes a function call from the name of the function and a list of arguments passed to it. The function being executed here is the `substitute` function. This function substitutes the values of `fm` and `Species` into the `fitCommand` function. The `eval` function simply evaluates the function with those substitutions in place. We then move on to the next species and repeat for all available species. Try running this for a few species and examine results. Here, we have only run it for one species.

This is basically a clever way of repeating analyses for different species without having to type in the model each time.

Plots

We now examine some plots of the fitted model and predictions using data from one of the species, *Alcyonacea*.

The first few lines form predictions at different levels of grouping. When `level=0`, the predictions are derived at the population level and do not include random effect terms. By default, predictions are constructed at the highest grouping level and are based on fixed and random effects. Note, the offset term is also added to the predictions.

```
> pfm <- predict(GLMM.Alcyonacea, newData) +
log(newData$SweptArea)
> pfm0 <- predict(GLMM.Alcyonacea, newData, level=0) +
log(newData$SweptArea)
> graphics.off()
```

The plot of predictions for fixed and random effects and fixed effects only is shown in

Figures 85 and Figures 86 respectively. This was constructed using the following piece of code.

```
> trellis.device()
> trellis.par.set(col.whitebg())
> xyplot(guard(pfm) ~ Months|Treatment*Topography, newData,
        groups = Plot, panel = panel.superpose, type="b",
main = "Alcyonacea", ylab = "log(mean)",
sub = "fixed and random effects")


> trellis.device()
> trellis.par.set(col.whitebg())
> xyplot(exp(pfm0) ~ Months|Treatment*Topography, newData,
        groups = Plot, panel = panel.superpose, type="b",
main = "Alcyonacea", ylab = "mean",
sub = "fixed effects only")
```

The results indicate some variation between cruises as well as between shallow and deep plots (Figure 85). After adjusting for the variation between cruises and plots within cruises, not much can be explained by the fixed effects (Figure 86. There does however, indicate a sharp decline in the mean number of species in the impacted site after approximately 10 months. This is particularly evident in deep waters but soon recovers at about 20 months.

Variance Components

A closer look at the variance components shows some variation explained by cruise and plot within cruise, however, only slight. Most of the variation is unexplained, indicating that large proportion of variation is not being adequately explained by this model.

```
> summary(GLMM.Alcyonacea, corr = F)
Linear mixed-effects model fit by maximum likelihood
 Data: Benthos
      AIC      BIC    logLik
  339.7326 371.9182 -157.8663
```

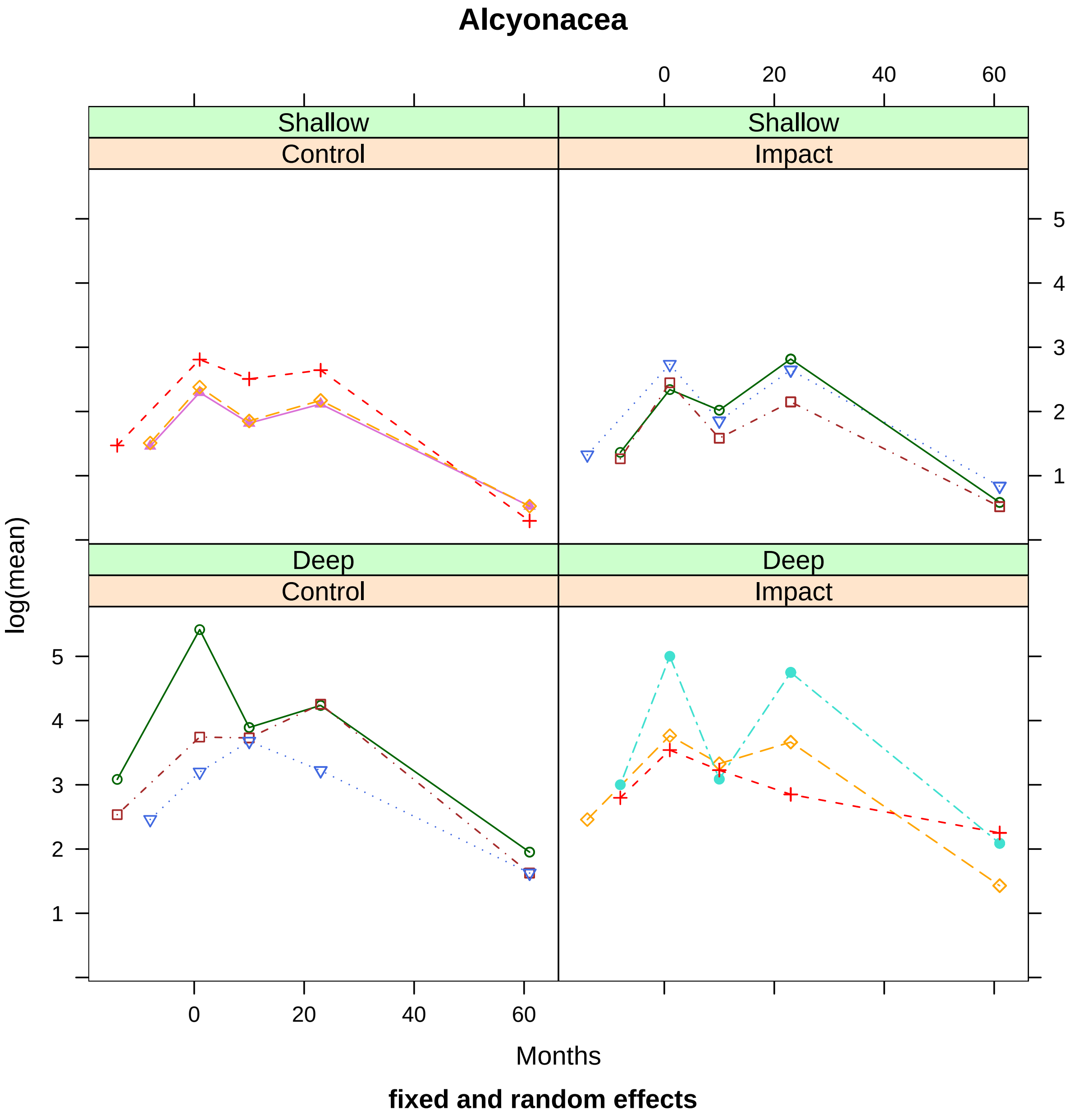


Figure 85: Plot of fixed and random effects

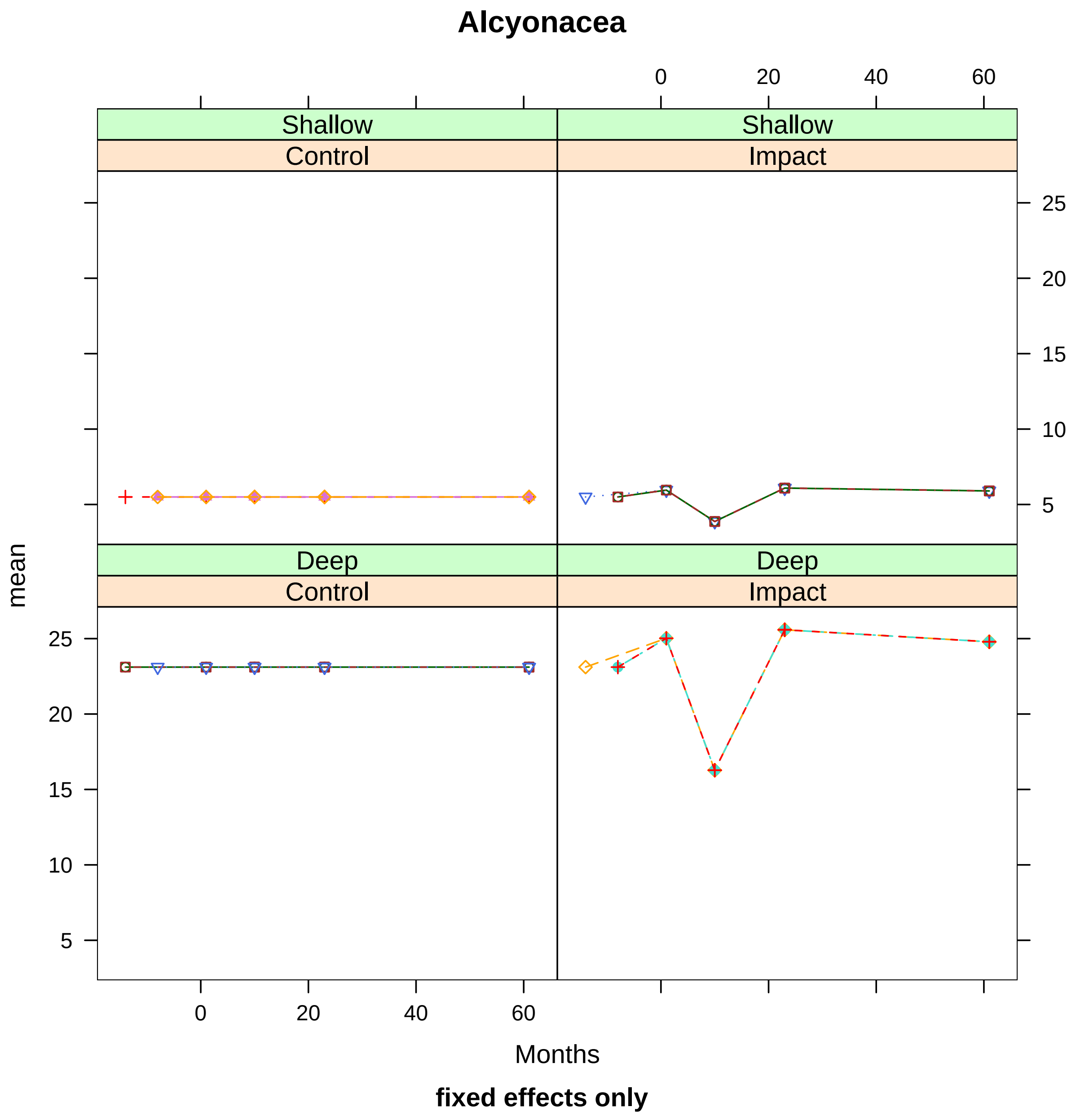


Figure 86: Plot of fixed effects only

```
Random effects:
 Formula: ~1 | Cruise
        (Intercept)
StdDev:   0.7587749

 Formula: ~1 | Plot %in% Cruise
        (Intercept) Residual
StdDev:    0.591846 2.495568

Variance function:
 Structure: fixed weights
 Formula: ~invwt
Fixed effects: Alcyonacea ~ Topography + I(Impact *
    cbind(nsi(Intensity), bin(Time))) + offset(log(SweptArea))
```

For this study, the model produces a result where the components of variance are smaller than the underlying Poisson component. This type of model is a useful way of isolating various parts of the model to reveal (perhaps speculatively) the fixed pattern. It is also a good way of handling overdispersion.

There is a close connection between GLMMs and Negative Binomial Models but this is somewhat controversial. The inference process is still a matter of debate.

Example: Muscle Data

The muscle dataset can be viewed using a scatterplot of log(length) versus concentration of calcium chloride broken down by heart muscle strip (Figure 87). The dataset is described in Appendix I.

```
> xyplot(log(Length) ~ Conc | Strip, muscle, as.table = T,
    xlab = "CaCl concentration")
```

Initial Fit: Fixed Effects Only

One model that we may consider fitting is a fixed effects model of the form

$$log(l_{ij}) = \alpha_j + \beta_j \rho^{c_{ij}} + \epsilon$$

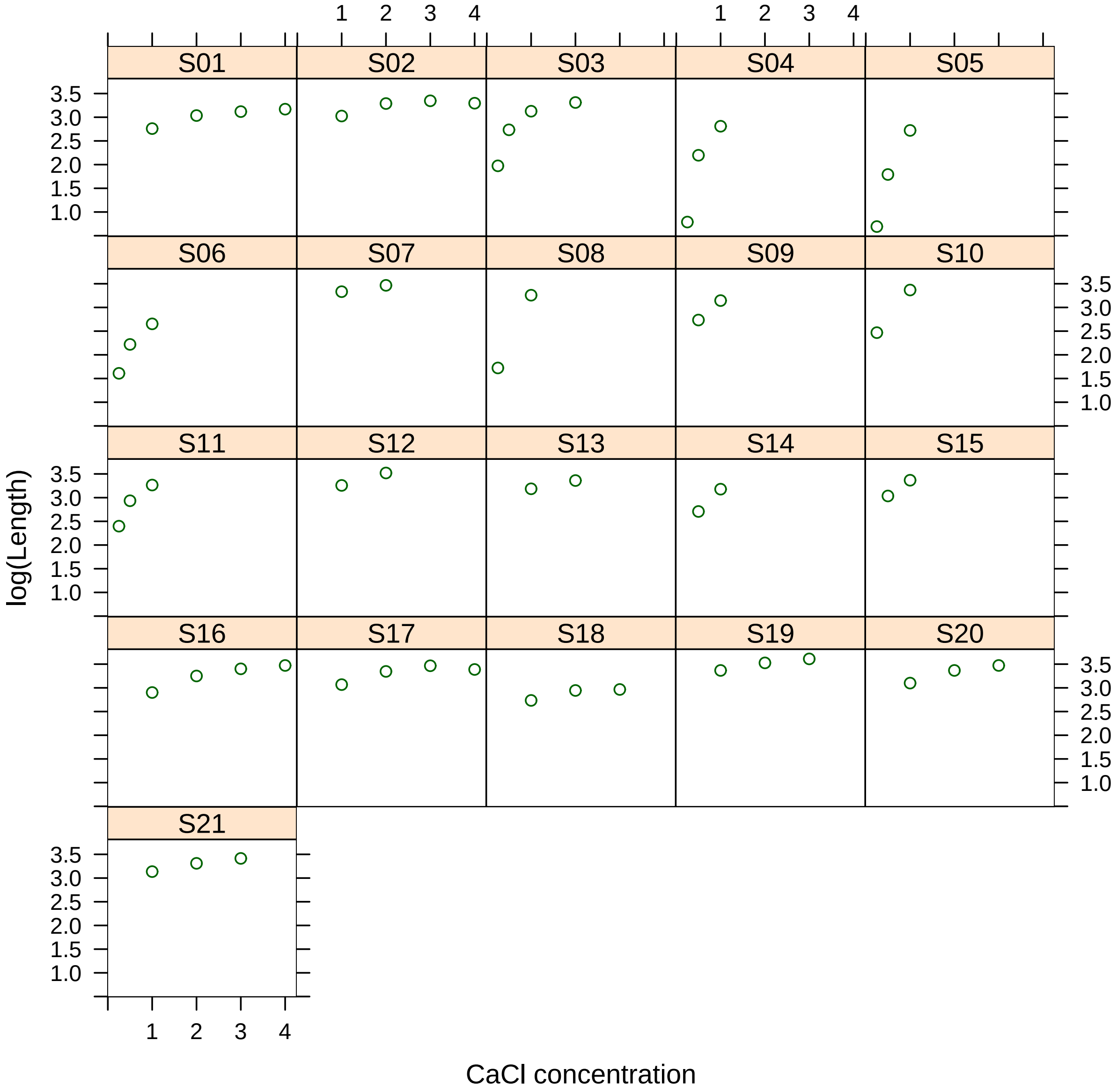


Figure 87: Plot of log(length) versus concentration of calcium chloride broken down by heart muscle.

where l_{ij} is the length of the strip and c_{ij} is the concentration for observation i and strip j. If we fit this as a fixed effects model, we need to estimate 43 parameters but we only have 61 data points. Therefore, special care needs to be taken with fitting the model.

We use the `plinear` algorithm in the non-linear least squares function to simplify the model specification and to make the fitting process more robust. In this type of model we only need to specify the non-linear parameters and then re-fit the model in a standard way to simplify predictions. Here is how we do this.

```
> X <- model.matrix(˜Strip - 1, muscle)


> muscle.nls1 <- nls(log(Length) ˜
cbind(X, X*rhoˆConc), muscle,
start = c(rho = 0.1), algorithm = "plinear", trace = T)
......
> b <- as.vector(coef(muscle.nls1))


> init <- list(rho = b[1],alpha = b[2:22],
beta = b[23:43])
> muscle.nls2 <- nls(log(Length) ˜ alpha[Strip] +
     beta[Strip]*rhoˆConc, muscle, start = init, trace = T)
......
```

Prediction and Presentation of Fit

We present the results from this model fit in Figure 88 using the following piece of code.

```
> Muscle <- expand.grid(Conc = seq(0.25, 4, len=20),
Strip = levels(muscle$Strip), Length = 0)
> Muscle <- rbind(muscle, Muscle)
> Muscle <- with(Muscle, Muscle[order(Strip, Conc), ]
> Muscle$fit <- predict(muscle.nls2, Muscle)


> xyplot(fit ˜ Conc|Strip, Muscle, type = "l", subscripts = T,
    prepanel = function(x, y, subscripts)
```

```
 list(xlim = range(x),
ylim = range(y, Muscle$fit[subscripts]),
dx = NULL, dy=NULL),
    panel = function(x, y, subscripts, ...) {
    panel.xyplot(x, y, ...)
    panel.points(x, log(Muscle$Length[subscripts]))
    },
    as.table = T, ylab = "log(Length)",
 xlab = "CaCl concentration")
```

The plot indicates that there is an increase in length (on the log scale) as the concentration increases up to 1.5 for most strips. However, the impact levels off at a length of approximately 3 (on the log scale).

Fitting a Mixed Effects Model

A mixed effects model may be more sensible for this application as we can fit random terms for the α's and β's within each strip, therefore reducing the number of parameters that need to be estimated.

The following piece of code fits the non-linear mixed effects model and plots the results, which are shown in Figure 89. Most of the variation is exhibited by the non-linear term in the model, β. This is confirmed in the summary of the fit of the model shown below.

```
> ival <- sapply(init, mean)


> muscle.nlme <- nlme(log(Length) ~ alpha + beta*rho^Conc,
muscle,fixed = rho+alpha+beta ~ 1,
random = alpha + beta ~ 1|Strip, start = ival)


> Muscle$RandomFit <- predict(muscle.nlme, Muscle)


> xyplot(RandomFit ~ Conc|Strip, Muscle, type = "l",
subscripts = T,
  xlim = with(Muscle,range(Conc)),
  ylim = with(Muscle,range(RandomFit,fit,log(muscle$Length))),
```

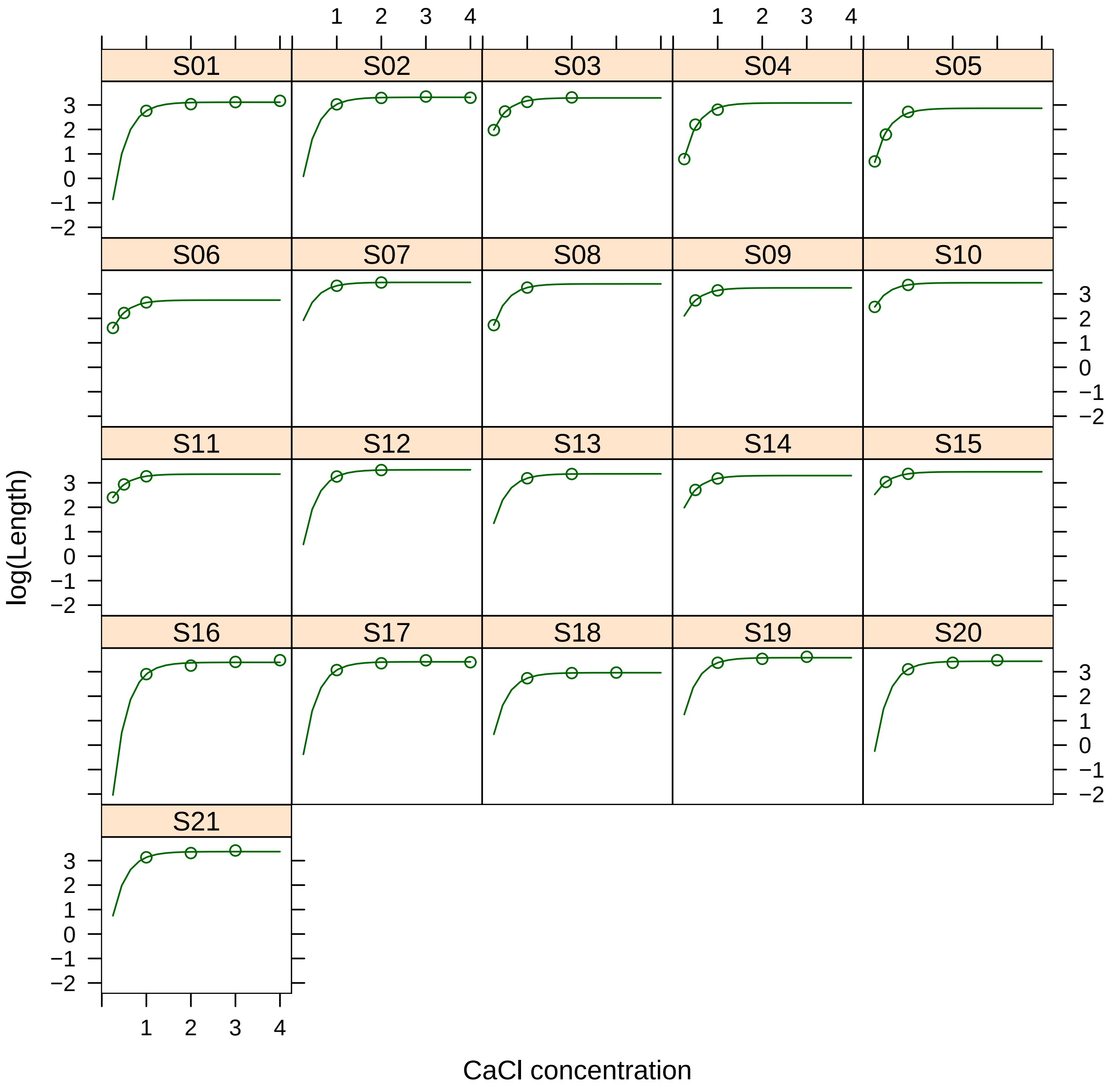


Figure 88: Results from the non-linear least squares fit.

```
  par.strip.text = list(lines=1,cex=0.7),
  panel = function(x, y, subscripts, ...) {
  panel.xyplot(x, y, ...)
  panel.lines(x, Muscle$fit[subscripts], col=2,lty=2)
  panel.points(x, log(Muscle$Length[subscripts]))
  }, as.table = T, ylab = "log(Length)",
xlab = "CaCl concentration")
```

```
> summary(muscle.nlme)
Nonlinear mixed-effects model fit by maximum likelihood
  Model: log(Length) ~ alpha + beta * rho^Conc
 ...
Random effects:
 Formula: list(alpha ~ 1, beta ~ 1)
...
         StdDev     Corr
alpha    0.16991613 alpha
beta     0.88268574 0.35
Residual 0.07692169

Fixed effects: rho + alpha + beta ~ 1
          Value  Std.Error DF  t-value p-value
rho    0.094083 0.01349832 37  6.96996       0
alpha  3.363546 0.04486974 37 74.96247       0
beta  -2.817441 0.29503865 37 -9.54940       0
```

Some final comments are noted below:

- Non-linear regression offers a way of integrating empirical and process models. If different kinds of non-linear regressions are to be repeatedly done then some investment in `selfStart` models pays off.
- The `plinear` algorithm offers an effective way of using the simplicity of linear parameters.

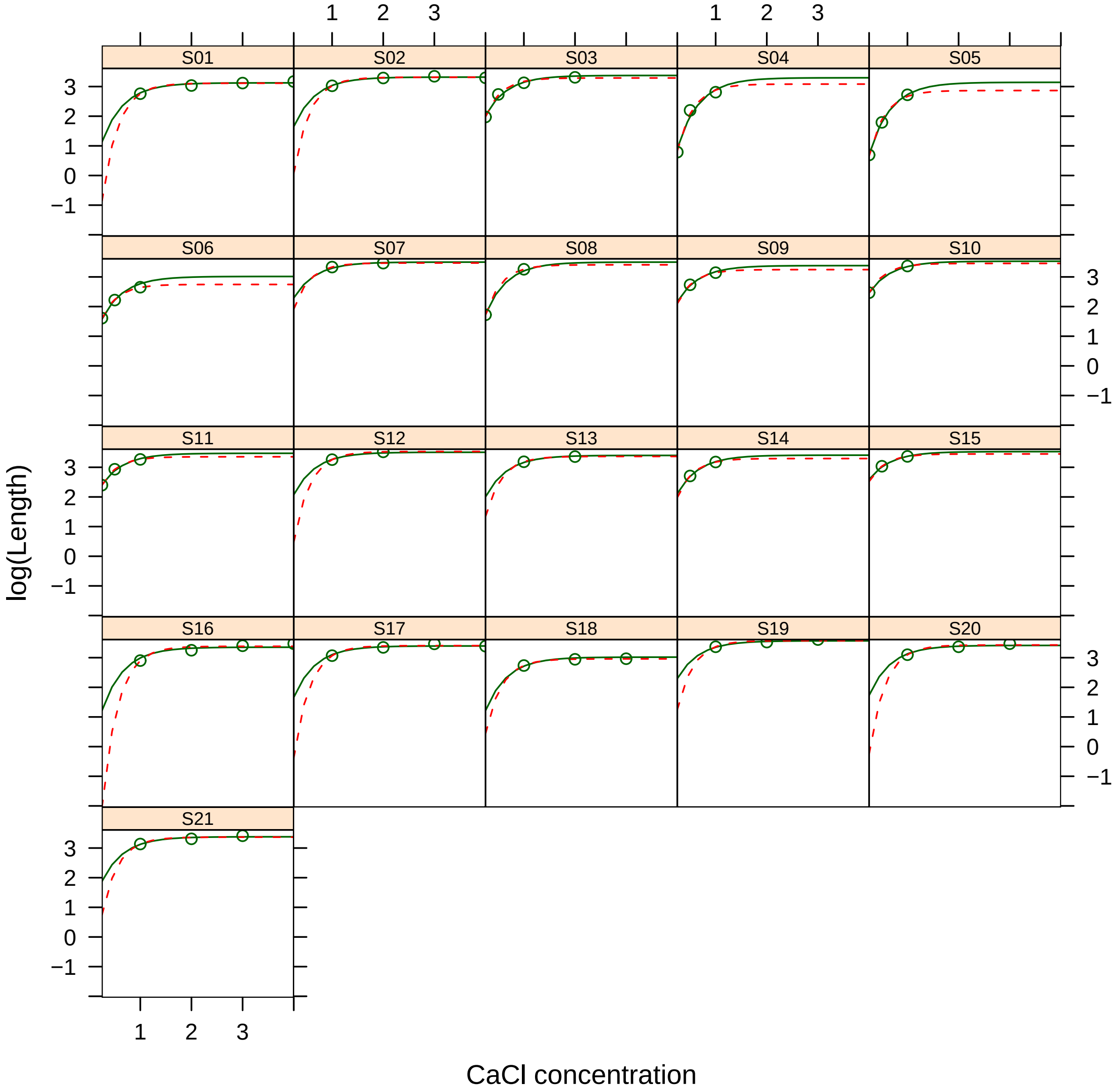


Figure 89: Results of non-linear mixed effects model for the Muscle dataset

- Random effects can be very tricky, but they offer a way of borrowing strength from one group to another, in order to gain a more holistic representation of the process generating the data.

Programming

Techniques for programming in R are described extensively by Venables & Ripley (2000) in the S Programming book. We will touch on a few aspects in this session.

The S programming book covers a number of topics, from the S language to developing classes and incorporating programs. The chapters are summarised below.

Introductory Example

We illustrate some of the programming features in R through the following introductory example.

Example: How many balls of diameter 2 can you fit inside a box of side 2?

To answer this question we need to know how to calculate the volume of a sphere and the volume of the cube.

The volume of a sphere of side r is

$$\frac{\pi^{d/2} r^d}{\Gamma(d/2+1)}$$

where d is the number of dimensions and $r = 1$ (for this example). The volume of the cube is 2^d. The proportion (p) of the cube occupied by the sphere is

$$\frac{\pi^{d/2}}{\Gamma(d/2+1)d^2}$$

Therefore the maximum number (N) is

$$\frac{\Gamma(d/2+1)2^d}{\pi^{d/2}}$$

Figure 90 graphically portrays the problem for the scenario when $d = 3$ and $r = 1$ and indicates that nearly 2 balls can fit in a cube of size 2.

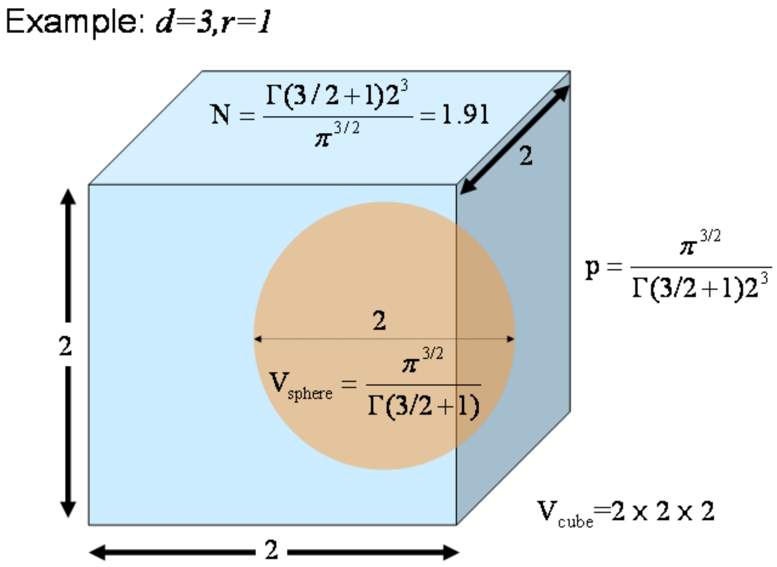

Example: d=3,r=1
$N = \frac{\Gamma(3/2+1)2^3}{\pi^{3/2}} = 1.91$
2
$p = \frac{\pi^{3/2}}{\Gamma(3/2+1)2^3}$
2
2
$V_{sphere} = \frac{\pi^{3/2}}{\Gamma(3/2+1)}$
V_{cube}=2 x 2 x 2
2

Figure 90: Illustration of the problem when $d = 3$ and $r = 1$.

To calculate the relative volume we can write a function in R as follows:

```
> rvball <- function(d)
exp(d/2 * log(pi) - d * log(2) - lgamma(d/2 + 1))
```

This function takes on one argument, d and evaluates the volume for this value. The function is also vectorized so we can check several cases at once:

```
> structure(floor(1/rvball(1:10)), names = 1:10)
 1 2 3 4 5  6  7  8   9  10
 1 1 1 3 6 12 27 63 155 401
```

Consider now how we might go about checking this relative volume formula by simulation.

1. Sample n points at random in the origin-centred cube.
2. Calculate how many of them are closer to the origin than 1 in the sense of Euclidean distance, say k, Return k and n, but can we make it behave like the ratio k/n when that would be convenient?

We allow for large numbers of trials but conduct them in blocks so as not to use too much memory all at once. The object will carry information about itself, including how it was generated. The object has a class so we can provide methods for key generic functions that allow results to be conveniently handled.

```
> mcrvball <-
function(d, N = 10000, blocksize = 10000) {
     n2 <- inside <- 0
     while(n2 < N) {
     n1 <- n2
     n2 <- min(n2 + blocksize, N)
     No <- n2 - n1
     samp <- matrix(runif(No * d, -1, 1), No, d)
     inside <- inside +
       sum((samp * samp) %*% rep(1, d) < 1)
     }
     res <- list(dimensions = d, inside = inside,
     total = N, call = match.call())
     oldClass(res) <- "mcrvball"
     res
}
```

The above function takes on the arguments d, N and $blocksize$. It begins by first initialising $n2$ and `inside` to be zero. It then runs a number of scripts within a `while` loop until the condition $n2 < N$ is not satisfied.

To understand what this function is doing, try inserting a `browser()`. This will stop R where you put the browser and allow you to inspect objects created in the function in more detail.

The following piece of code writes a print method for ”mcrvball” objects. When you issue the command `print(obj)`, the `print.mcrvball` function will be executed. It provides a neat way of viewing the dimensions, the estimate of the proportion of balls and the actual proportion of the cube occupied for $d = 2$.

```
> print.mcrvball <- function(x, ...) {
cat("Dim.:", x$d,
"Estimated:", signif(x$inside/x$total, 4),
"Actual:", signif(rvball(x$dim), 4), "\n")
invisible(x)
}
```

The `Ops` class allows for arithmetic operations to be performed and calculated. It is a function called within other functions. It is not called directly. The `NextMethod` function is usually used within methods to call the next method. It does this by looking down the list of classes to see whether there is a method for the current generic. For this example, the `Ops` function calculates the ratio between the number of samples inside and the total number. This is what is printed in the output.

```
> Ops.mcrvball <- function(e1, e2) {
if(!is.null(class(e1)) && class(e1) == "mcrvball")
e1 <- e1$inside/e1$total
if(!missing(e2) && !is.null(class(e2)) &&
class(e2) == "mcrvball")
e2 <- e2$inside/e2$total
NextMethod()
}
```

We try out these functions in a test run for a variety of dimensions as follows and compare with actual values. We notice that the Monte Carlo estimates are not too different to the Actual estimate.

```
> for(i in 4:10) print(mcrvball(i, 1000000))
Dim.: 4 Estimated: 0.3081 Actual: 0.3084
Dim.: 5 Estimated: 0.1647 Actual: 0.1645
Dim.: 6 Estimated: 0.08089 Actual: 0.08075
Dim.: 7 Estimated: 0.03673 Actual: 0.03691
Dim.: 8 Estimated: 0.01588 Actual: 0.01585
Dim.: 9 Estimated: 0.006375 Actual: 0.006442
Dim.: 10 Estimated: 0.002428 Actual: 0.00249

> X <- matrix(NA, 7, 2)
> X[,2] <- floor(1/rvball(4:10))

> for(i in 4:10)
 X[i-3,1] <- floor(1/mcrvball(i, 100000))
> dimnames(X) <- list(4:10,
c("Monte Carlo", "Actual"))
> X
   Monte Carlo Actual
 4           3      3
 5           6      6
 6          12     12
 7          27     27
 8          64     63
 9         151    155
10         375    401
```

Are the Monte Carlo results likely to be biased downwards?

The Call Component and Updating

If an object has a call component, `update()` may be used to create a new object by making simple modifications to the call. Here it is not very useful, but it does work:

```
> p1 <- mcrvball(10)
> floor(1/p1)
[1] 625
> p2 <- update(p1, N = 200000)
> floor(1/p2)
[1] 416
```

Special methods for lm and glm objects allow convenient forms for changes to formulas, such as the use of the period.

Combining Two Estimates

We combine two relative frequencies by adding numerators and denominators. The most convenient way to do this is with a binary operator.

```
> "%+%" <- function(e1, e2) UseMethod("%+%")


> "%+%.mcrvball" <- function(e1, e2) {
if(e1$dimensions != e2$dimensions)
stop("ball dimensions differ!")
res <- list(dimensions = e1$dimensions,
inside = e1$inside + e2$inside,
total = e1$total + e2$total,
call = e1$call)
oldClass(res) <- "mcrvball"
res
}
> p1 %+% p2
Dim.: 10 Estimated: 0.002362 Actual: 0.00249
> floor(1/(p1 %+% p2))
[1] 423
```

Now, for efficiency and convenience, we collect results as we go

```
> p0 <- p1
> for(i in 1:10) p0 <- p0 %+% print(update(p1))
Dim.: 10 Estimated: 0.0023 Actual: 0.00249
Dim.: 10 Estimated: 0.0026 Actual: 0.00249

Dim.: 10 Estimated: 0.0018 Actual: 0.00249
Dim.: 10 Estimated: 0.0031 Actual: 0.00249
> p0
Dim.: 10 Estimated: 0.002345 Actual: 0.00249
> floor(1/p0)
[1] 426
> unlist(sapply(p0, as.character))
 dimensions inside    total      call1 call2
       "10"  "258" "110000" "mcrvball"   "10"
```

Some Lessons

We summarise some of the main points from this example.

1. Vectorization. (`rvball`)
2. Taking the whole object view of the problem. (`mcrvball`)
3. Object orientation: put all the information you are likely to need into the object and give it a class. (`mcrvball`)
4. Methods for existing generics. (`print.mcrvball`)
5. Group generic functions. (`Ops.mcrvball`)
6. Binary operators and new generic functions. (%+%)

Some Under-used Array and Other Facilities

Some functions that are sometimes overlooked when it comes to dealing with arrays are listed below.

1. `cumsum, cummax, cummin, cumprod`
2. `pmax, pmin`
3. The empty index, especially in replacements.
4. `row, col` and `slice.index`
5. `outer`
6. `aperm`
7. `diag` and `diag<-`
8. A matrix as an index.

To illustrate some of these functions, we write a function to construct a tri-diagonal matrix with constant values in each diagonal position. The function appears below and it takes on arguments r and v. The `outer` function creates a 3×4 matrix with the elements in each row and column being cumulatively added. The diagonal elements and those to the left and right of the diagonal are then replaced with the values set for v.

```
> tridiag <- function(r, v) {
     cind <- as.vector(outer(-1:1, 1:r, "+"))
     rind <- rep(1:r, each = 3)
     mind <- cbind(rind, cind)
     mind <- mind[ -c(1, 3*r), ]
     X <- matrix(0, r, r)
     X[mind] <- rep(v, r)[ -c(1, 3*r)]
     X
    }
> tridiag(4, c(1,2,1))
     [,1] [,2] [,3] [,4]
[1,]    2    1    0    0
[2,]    1    2    1    0
[3,]    0    1    2    1
[4,]    0    0    1    2
```

A more direct calculation can be achieved using subsetting as shown below:

```
> X <- matrix(0, 5, 5)
> X[row(X) == col(X)] <- 2
> X[abs(row(X) - col(X)) == 1] <- 1
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    1    0    0    0
[2,]    1    2    1    0    0
[3,]    0    1    2    1    0
[4,]    0    0    1    2    1
[5,]    0    0    0    1    2
```

Some Little-used Debugging Functions and Support Systems

Some useful debugging functions are listed in the following table. They are presented in increasing order of investment and payoff:

1. `cat` or `print` statements, strategically placed.
2. `trace`, `untrace` and `tprint`
3. `browser`
4. `inspect`
5. emacs with ESS

I find that the most useful debugging function is `browser()`. To quit from the browser you can type a capital Q at the prompt. Typing `c` at each browser pause will cause R to advance to the next set of calculations.

Compiled Code and Packages

Compiling code and packages is a useful feature in R. The add-on packages available from the CRAN website have all been generated in this way. Having this feature in R allows you to perform some complex and computational calculations outside of R and bring in the results for printing and graphically displaying the information.

We present a very simple example to show how a package can be developed. You will need a couple of other packages though. These will be described in detail later.

The example we use is one which can be computed within R itself. It is a function to calculate the convolution of two numerical sequences

$$a = \{a_i\}, b = \{b_j\}.(a * b)_k = \sum_{i+j=k} a_i b_j$$

We first consider a C function to do the crude arithmetic and how to link it to R. Then we consider four possible R functions to perform the computation. Finally we look at system timings for a relatively large computation.

The C function in VR_convolve.c

The C function for producing this convolution is shown below:

```
void VR_convolve(double *a, long *na,
 double *b, long *nb,
 double *ab)
{
  int i, j, nab = *na + *nb - 1;

  for(i = 0; i < nab; i++)
    ab[i] = 0.0;
  for(i = 0; i < *na; i++)
    for(j = 0; j < *nb; j++)
      ab[i + j] += a[i] * b[j];
}
```

Note that for this to work, all arguments to the function must be pointers.

Before making the DLL, some care is required to ensure that the arguments supplied to the function have the correct storage mode. Creating the dll is relatively easy provided you have the MinGW compiler installed. You can easily download this package from the Internet. It is Free.

Ensure that `R.exe` and `gcc` are both visible on the %PATH% (Windows) environment variable (if using Windows). The compiler, `gcc` can be obtained from the tools zip file from Bates website. (See *http://www.murdoch-sutherland.com/Rtools/*).

In either a DOS or cygwin window (if you also have cygwin installed), type in the following:

```
$ gcc O2 c VR_convolve.c
$ dllwrap o VR_convolve.dll def VR_convolve.def
     VR_convolve.o
```

Several files (either Fortran or C or both) can be compiled at once. See S Programming (p241-) for more details on how the compilation of files to produce DLL's can be performed.

Loading the Shared Library

Once the DLL has been created, we need to load the shared library. This is done in the function we propose to use. The function `is.loaded` checks to see if the DLL has been loaded. If not, then it is loaded using `dyn.load`. Storage modes for each parameter are defined. This is important so R knows how to interact correctly with the C or Fortran code.

```
> convolve3 <- function(a, b) {
  if(!is.loaded(symbol.C("VR_convolve")))
    dyn.load("VR_convolve.dll")
  storage.mode(a) <- "double"
  storage.mode(b) <- "double"

  .C("VR_convolve",
     a,
     length(a),
     b, length(b),
     ab = double(length(a) + length(b) - 1))$ab
}
```

A few explanations about loading in code . . .

- .C() takes the name of the entry point as a character string as its first argument
- subsequent arguments correspond to those of the C function itself, with correct storage mode ensured
- Subsequent arguments may be named
- The result is a list with names as supplied to the arguments. The results of the C function must be reflected as changes in the supplied arguments
- In our case only one result was needed, so this is the only one we named (`ab`)

- `is.loaded()` may be used to check if an entry point is visible
- `symbol.C()` and `symbol.For()` can be used to ensure portability across platforms

Three Pure R Code Contenders

We now examine three pure R code contenders and evaluate their performance with the C function written. The first uses two for loops. The second uses one for loop, while the third uses an apply function.

```
> convolve0 <- function(a, b) {
  ab <- rep(0, length(a) + length(b) - 1)
  for(i in 1:length(a))
    for(j in 1:length(b))
      ab[i+j-1] <- ab[i+j-1] + a[i]*b[j]
  ab
}

> convolve1 <- function(a, b) {
  ab <- rep(0, length(a) + length(b) - 1)
  ind <- 1:length(a)
  for(j in 1:length(b)) {
    ab[ind] <- ab[ind] + a*b[j]
    ind <- ind + 1
  }
  ab
}

> convolve2 <- function(a, b)
  tapply(outer(a, b), outer(seq(along = a), seq(along = b),
   "+"), sum)
```

We evaluate each function by running the function on some sample data, recorded the time and compare estimates. The first three values reported from `system.time` represents the user CPU, system CPU, and elapsed or real time. The results show increasing improvement as we move from `convolve0` to `convolve3`.

```
> a <- rep(1/1000, 1000)
> b <- dbinom(0:1000, 1000, 0.5)
> system.time(ab0 <- convolve0(a, b))
[1] 12.65  0.09 13.27    NA    NA
> system.time(ab1 <- convolve1(a, b))
[1] 0.27 0.02 0.29   NA   NA
> system.time(ab2 <- convolve2(a, b))
[1] 1.63 0.03 1.73   NA   NA
> system.time(ab3 <- convolve3(a, b))
[1]  0  0  0 NA NA
> range(abs(ab0-ab1)+abs(ab1-ab2)+abs(ab2-ab3))
[1] 0.000000e+00 2.602085e-18
```

Some comments regarding this comparison . . .

- The slick version is NOT the fastest, probably because of memory use
- Both the conventional R versions are much faster than the translated Fortran first version
- Much bigger problems are needed before the compiled code version registers on the timing
- This first invocation will, in fact, be the slowest as it needs to load the dll before it does anything.
- A good strategy is to reserve C for the heavy computational part of the calculation but do everything else in pure R.
- The `.Call` interface allows you to manipulate R objects directly from within C code. This requires much more specialised coding using C macros that are rather sketchily documented.

Making and Maintaining Extensions

For more information about making and maintaining extensions, read the section in *Writing R Extensions* (at least once).

Read the `readme.packages` file and consult the web portals for the tools. Make sure they are visible within your %PATH%. This is a very specialised area, but the payoff is large when you have it under control.

The function `package.skeleton()` establishes a good part of the work needing to be done, but then you have to do it.

Building packages does the compilation as part of the process.

`install.packages()` and `update.packages()` may be used to get packages from CRAN (or .zip files) and update packages from CRAN from within R itself. They require administrator capabilities within Windows (usually).

Neural Networks: An Introduction

Methodology

Neural networks can be considered a type of nonlinear regression that takes a set of inputs (explanatory variables), transforms and weights these within a set of hidden units and hidden layers to produce a set of outputs or predictions (that are also transformed). Figure 91 is an example of a feed forward neural network consisting of five inputs, a hidden layer that contains three units and an output layer that contains three outputs.

The model can be expressed as follows:

$$y_k = \phi_0(\alpha_k + \sum_{i \to k} w_{ik}x_i + \sum_{j \to k} w_{jk}\phi_j(\alpha_j + \sum_{i \to j} w_{ij}x_i)$$

where ϕ_o is the transformation on the hidden layers to produce the output, α_k represent the biases associated with each observation k, w_{ik} are weights assigned to variable x_i and ϕ_h is the transformation applied to the hidden units in the hidden layer.

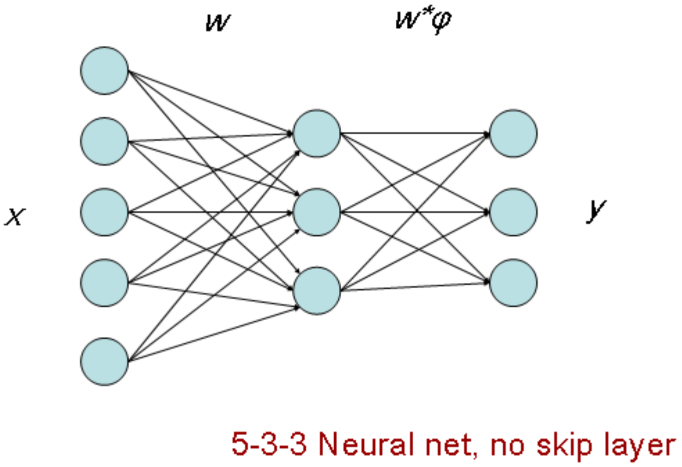


Figure 91: An example of a neural network.

Regression Function

The expression shown above can be viewed as regression equations:

$$y_k = f_k(w; x)$$

with internal layer transformations that are typically logistic

$$\phi_h(z) = \frac{\exp(z)}{1 + \exp(z)}$$

Output transformations that are most often used are either linear, logistic or consist of a threshold. Estimation is either through ordinary least squares, logistic or using log-linear modelling estimation (softmax). For example, the fitting function for least squares is

$$E = \sum_p ||t^p - y^p||^2$$

where t^p is the target output and y^p is the output for the p-th example pattern.

Penalized Estimation

To ensure that the function f is smooth, we can restrict the estimates to have a limited number of spline knots. An alternative approach is to use *regularization* where the fitting criterion becomes

$$E + \lambda C(f)$$

where $C(f)$ is a penalty function and λ is the degree of *roughness*. In neural networks, the penalty function is typically the sum of squares of the weights w_{ij}, known as weight decay. Having this penalty term in the fitting process helps with the optimization process and tries to avoid overfitting. Choices of lambda vary depending on the type of fitting function, E chosen. For an entropy fit (deviance), good choices for λ lie between 0.01 and 0.1. For least squares, choices for λ lie between $\lambda \approx 10^{-4} - 10^{-2}$.

Special Cases

The neural network shown in Figure 91 is a typical model for most problems. However, situations may arise where special types of neural network models are warranted. These are outlined below.

- Linear Regression (the hard way)
  A model with no internal nodes, no penalties and a linear output function (Multivariate)
- Logistic Regression (the hard way)
  A model with no internal nodes, no penalties, a logistic output function and one output node
- Logistic Regression (again)
  A model with no skip layer, one internal node and a Kullback-Liebler objective function with no penalties
- Multiple Logistic, Multinomial Regression
  A model with no internal nodes, a softmax objective function and no penalties

Linear Regression the Hard Way: Check

The following sample code is an example of how to perform a linear regression on the `Cars93` dataset using neural networks.

We first fit the model using the `nnet` function. To fit a linear model we need to specify a model with no internal nodes (`size=0`), no penalties (`decay=0`) and a linear output function (`linout=T`).

```
> require(nnet)
> tst <- nnet(1000/MPG.city ~ Weight+Origin+Type,
  Cars93, linout = T, size = 0,
decay = 0, rang = 0, skip = T, trace = T)
# weights:  8
initial  value 213926.871110
iter  10 value 1560.625585
final  value 1461.849465
> coef(tst)
      b->o       i1->o       i2->o       i3->o
6.01509864  0.01337820 -1.20290333 -0.65095242
     i4->o       i5->o       i6->o       i7->o
0.78448665 -1.48859419  2.96276360  2.38761012
```

We now check the coefficients produced from the neural network with those estimated using linear regression and the `lm` model.

```
> tst1 <- lm(1000/MPG.city ~ Weight + Origin+Type, Cars93)
> coef(tst1)
  (Intercept)      Weight   Originnon-USA     TypeLarge
   6.01509685  0.01337820     -1.20290356   -0.65095077

  TypeMidsize   TypeSmall  TypeSporty       TypeVan
   0.78448750 -1.48859321      2.96276400    2.38761020

> range(coef(tst) - coef(tst1))
[1] -1.646830e-06  1.793509e-06
```

We see that the results are very similar. The difference between coefficients range between -1.6×10^{-6} and 1.79×10^{-6}.

Logistic Regression is a Special Case

We now examine logistic regression models fit as a neural network using the birth weight data. Once again we specify no weight decay (`decay=0`) and no input layers (`size=0`). By default, logistic output units are specified.

```
> tst <- nnet(low ~ ptd + ftv/age, data = bwt, skip = T, size = 0,
        decay = 0, rang = 0, trace = T)
# weights:  7
initial  value 131.004817
iter  10 value 101.713567
final  value 101.676271
converged
> tst1 <- glm(low ~ ptd + ftv/age, binomial, bwt)
> range(coef(tst) - coef(tst1))
[1] -0.0001691730  0.0003218216
```

These are compared with estimates produced using a generalized linear model and once again the differences between coefficients are negligible.

We now set up a training and test set and begin with a parametric model using the birth weight data again. Predictions from the GLM are then tabulated with the actual data to produce a confusion matrix. We see below that 28 observations are misclassified in this model.

```
> sb1 <- sample(1:nrow(bwt), 100)
> bwt.train <- bwt[sb1,  ]
> bwt.test <- bwt[ - sb1,  ]
> bm.train <- update(tst1, data = bwt.train)
> bm.tst <- predict(bm.train, bwt.test, type = "resp")
> bm.class <- round(bm.tst)
> table(bm.class, bwt.test$low)
   0  1
0 53 16
1 12  8
```

Now Consider a Tree Model

Now we consider a tree model. (These will be explained in the following two sections in more detail.) The `tree` function is used here to fit a classification tree.

```
> require(tree)
> tm.train <- tree(low ~ race + smoke + age + ptd + ht + ui + ftv,
      bwt.train)
> plot(cv.tree(tm.train, FUN = prune.misclass))
> tm.train <- prune.tree(tm.train, best = 6)
> tm.class <- predict(tm.train, bwt.test,
type = "class")
> table(tm.class, bwt.test$low)
   0  1
0 43 12
1 22 12
```

The optimal model is one that misclassifies 34 observations, a substantially higher proportion of cases. One option that might improve misclassification rates is setting up a loss function. (This will be described in the next session.)

Some Initial Explorations of a Neural Network

Neural networks may prove to be a better way of modelling this data but it is not clear what degree of non-linearity is warranted. Here is one that we tried. It consists of a network with one hidden layer and three units and a weight decay of 0.001.

```
> X0 <- model.matrix( ~ race + smoke + age + ptd + ht + ui + ftv,
       bwt.train)[, -1]
> std <- function(x) (x - min(x))/(max(x) - min(x))
> X <- apply(X0, 2, std)
> nm.train <- nnet(low ~ X, data = bwt.train, size = 3, skip = T,
       decay = 0.001, trace = T, maxit = 1000)
# weights:  43
initial  value 71.792542
iter  10 value 55.167514
iter  20 value 50.647657

...
iter 210 value 19.234755
iter 220 value 19.234745
final  value 19.234743
converged
```

Test Data

We then obtain predictions from this model and compare it with the actual classifications.

```
> X <- model.matrix( ~ race + smoke + age + ptd + ht + ui + ftv,
      bwt.test)[, -1]
> for(j in 1:ncol(X))
  X[, j] <- (X[, j]  min(X0[, j]))/
(max(X0[, j]) - min(X0[, j]))
> nm.tst <- predict(nm.train, newdata = bwt.test, type = "raw")
> nm.class <- round(nm.tst)
> table(nm.class, bwt.test$low)
   0  1
```

```
0 46 15
1 19  9
```

The results indicate that

- The tree model does about as well as the parametric model and is constructed more-or-less blind.
- The neural network model is by far the most difficult to fit and performs (apparently) slightly worse than the other models.
- Cross-validation should perhaps be used to come to some decision on the amount of smoothing warranted in the NN model (See Venables & Ripley (2002)).
- Neural networks may be more useful with much larger data sets and more complex prediction situations (maybe)

Tree-based Models I

RPART offers two libraries for constructing Classification and Regression Tree (CART) models. The first is an S-PLUS/R native library called `tree`, which implements the traditional S-PLUS/R tree methodology described in Chambers & Hastie (1992). The second is the `rpart` library developed by Beth Atkinson and Terry Therneau of the Mayo Clinic, Rochester, New York, which implements methodology closer to the traditional CART version of trees due to Breiman et al. (1984).

Both have their advantages and disadvantages. However, we mostly favour the RPART version in this course. Although we focus on this library, all examples can be done using the tree library.

Decision Tree Methodology

Decision trees have been applied to a number of interesting problems in the medical, environmental and financial areas. They have been recognised as a useful modelling tool among non-statisticians, particularly in the medical area, as they produce a model that is very easy to interpret.

Decision trees have a number of interesting features, which are summarised below.

- Decision trees are non-parametric and therefore do not require Normality assumptions of the data.
- Decision trees can handle data of different types: continuous, categorical, ordinal, binary. Transformations of the data are therefore not required.
- Trees can be useful for detecting important variables, interactions and identifying outliers. This can be useful at the exploratory stage of modelling.

  Figure 92 displays a diagram of a decision tree (not produced from R) that highlights an interaction between `age` and `sex`. The interaction is suggested because of multiple splitting that occurs using these variables. If decision trees were being used as an exploratory tool, this model would suggest incorporating an interaction term between `sex` and `age` in a model.

  Figure 93 shows a mock tree with an outlier present on the `age` variable. This example shows a fairly obvious outlier but an important one none-the-less.
- Decision trees cope with missing data by identifying surrogate splits in the modelling process. Surrogate splits are splits highly associated with the primary split (which will be explained below). They can be used in situations when data cannot be collected using the primary split variable.
- Decision trees are used for analysis of a wide range of applications

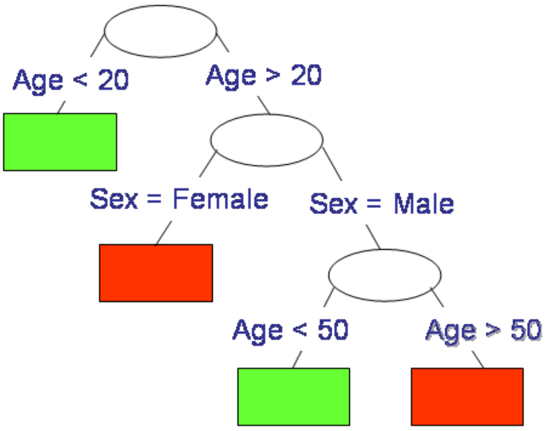


Figure 92: Illustration of Interactions

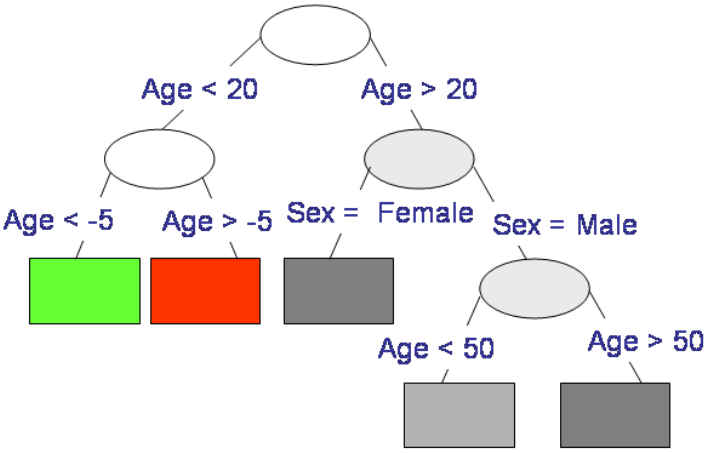


Figure 93: Illustration of Outliers

The methodology can be summarised into three main steps. These are described below.

1. Splitting Criterion

   Splits are formed on subsets of the data in a *greedy* fashion. To begin with, a variable and split location is defined (usually based on a particular criterion: Gini (classification), sums of squares (regression)) from the entire dataset. The data is partitioned into two groups based on this split and the process is repeated on the two subgroups. (Hence the reason why the algorithm is sometimes referred to as *greedy*). Splitting continues until a large tree is constructed where only a small number of observations of the same class reside in each terminal node.

2. Pruning Procedure

   Pruning commences once a large tree has been grown. It involves successively snipping back splits of the tree into smaller trees using a cost complexity measure (which will be described later). This involves computing an error measure, usually calculated using cross-validation.

3. Tree Selection

   Tree selection is typically based on cross-validation and/or the use of a test dataset for larger applications. The tree yielding the lowest cross-validated error rate is examined. In addition, trees that are smaller in size but comparable in accuracy (corresponding to the *1 SE rule*) are also investigated.

Tree Definitions

Figure 94 summarises the key terms used in decision tree modelling. The node which is to be split is referred to as the *parent node*. The *branches* emanating from the parent node define the *split* which is a logical statement comprised of one of the variables in the dataset and a value indicating the split location. A split forms two new nodes, a left child node and a right child node. Each is assigned a prediction, whether a classification or a mean (for regression problems).

Split Criterion

The split criterion for a decision tree model is different for different types of responses. The two main types of response are categorical, where we might be interested in producing a classification or continuous, where interest lies in producing a mean prediction.

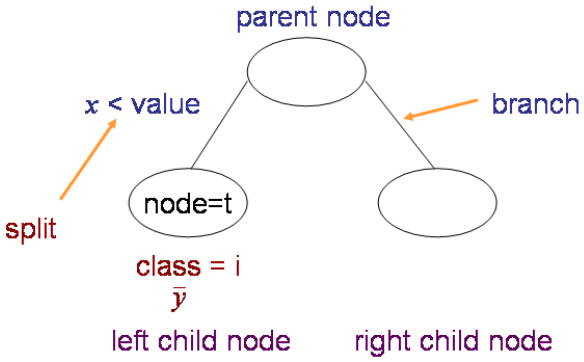


Figure 94: Key terms in decision tree modelling

For classification problems, *Gini* is the most common split criterion used. It is defined as

$$i(t) = \sum_{i \neq j} p(i|t)p(j|t)$$

where $p(i|t)$ is the probability that item in node t is of class i and $i(t)$ is the impurity measure or misclassification error.

For a two-class problem, this expression reduces to $2p(1|t)p(2|t)$.

Costs can be incorporated into the impurity measure to reflect weightings of different groups. For example, in a medical study, we are more interested in the accurate classification of sick or dying patients more than the well patients. Often though, the number of sick patients in medical studies is limited and this compromises prediction for the sick cases somewhat. To avoid this problem, we weight the sick cases higher than the well cases and we do this using costs. Sometimes this is referred to as setting up a loss matrix. When we incorporate unequal costs into a classification tree model, we alter $p(i|t)$ and $p(j|t)$. This is described in more detail in Breiman et al. (1984).

For regression problems, the impurity measure consists of the residual sums of squares, similar to what we examine in the output from a linear regression model. The expression for the impurity measure, $i(t)$ is

$$i(t) = \sum \{y(i|t)\bar{y}(t)\}^2$$

where $\bar{y}(t)$ is the mean of observations in node t and $y(i|t)$ represents observation i in node t.

Split Formation

Figure 95 illustrates the actual split formation from the parent node (P) into left (L) and right (R) child nodes. In determining the best split for a node, the impurity measure is evaluated for the parent node ($i(t)_P$), left child node ($i(t)_L$) and right child node ($i(t)_R$). The change in impurity is then determined through the expression

$$\Delta = i(t)_P - [p_L i(t)_L + p_R i(t)_R]$$

where p_L and p_R represent the proportions of the data sent to the left and right respectively. The split producing the greatest change in impurity is therefore selected.

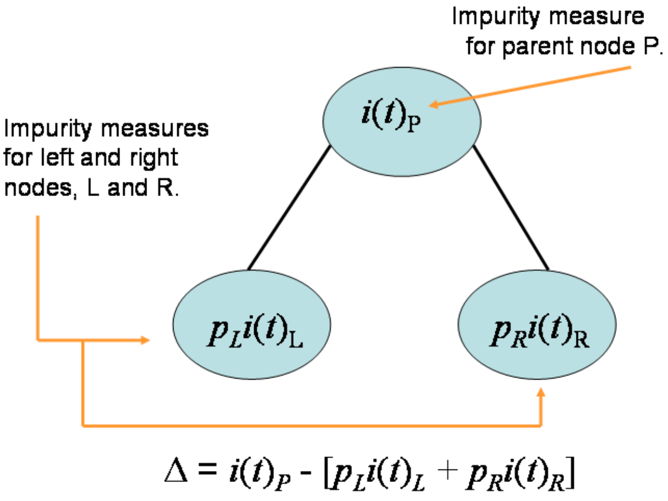


Figure 95: Illustration of split formation

Recursive Partitioning And Regression Trees (RPART)

The RPART software is going to be the focus in this section. Documentation for RPART can be downloaded in pdf form from the CRAN website. The help is usually sufficient to get you started.

RPART functions for fitting and producing decision trees can be broken up into two distinct types: modelling functions and plotting functions. Those functions responsible for fitting the decision tree include

- `rpart`: fitting the actual model
- `rpart.control`: tuning parameters that feed into `rpart`
- `summary.rpart`: summarises the fitted model
- `snip.rpart`: interactive pruning feature
- `prune.rpart`: pruning

Plotting functions responsible for producing the output as a nice graphical figure include `plot.rpart, text.rpart` and `post.rpart`. The latter function is responsible for producing postscript versions of the fitted models.

RPART Features

RPART offers models for different types of problems. We have just been focusing on fitting classification trees and regression trees. However, RPART offers methodologies for survival and count data. Descriptions of these models are summarised in the technical report by Therneau & Atkinson (1997).

Fitting the Model

The function `rpart()` is used to fit the decision tree. It is comprised of the following arguments that you can extract from the help file.

formula	`y` ∼ `x1+x2+x3+`...
data	data where the above variables can be found
na.action	Default: omits observations with missing data in y
method	`class` or `anova`
parms	priors and loss matrix (class only)
control	parameters that control the model fitting

The model is controlled by the function `rpart.control()`. Arguments to this function are outlined below.

minsplit	minimum no. of observations in a node before a split occurs
cp	complexity parameter
usesurrogate	How to use surrogates for missing data:

0 = display only
1 = use surrogates to split variables with missing data.
If all are missing, no split is created.
2 = Breimans surrogate rule.

Breiman's surrogate rule is described in his book and uses surrogates to split variables with missing data. However if all surrogates have missing data, the observation is directed to the node with the largest proportion of observations.

A Classification Example: Fisher's Iris Data

Fisher's Iris dataset is described in detail in Appendix I and has been used throughout other sections of these lecture notes. The data consists of data recorded for three species of Iris: Setosa, Versicolor and Virginica. Four pieces of information were recorded for 150 observations. These included sepal length and width and Petal length and width, both of which are recorded in centimetres. The aim is to classify the three species of iris based on petal and sepal measurements using a classification tree approach.

Fitting the Model

Before fitting the model, we first have to attach the `rpart` library. This can be done using the `require` function. We set the seed value to ensure that we all get the same results as sometimes we may get different trees through the cross-validation process. To produce a really large tree, we make the complexity parameter (`cp`) really small.

```
> require(rpart)
> set.seed(123456)
> iris.rp <- rpart(Species ˜ ., method="class", data=iris,
   control=rpart.control(minsplit=4,cp=0.000001))
> summary(iris.rp)
Call:
...
       CP nsplit rel error xerror       xstd
1 0.50000      0      1.00   1.21 0.04836666
2 0.44000      1      0.50   0.76 0.06123180
```

```
3 0.02000      2      0.06   0.10 0.03055050
4 0.01000      3      0.04   0.11 0.03192700
5 0.00001      5      0.02   0.10 0.03055050


Node number 1: 150 observations,    complexity param=0.5
predicted class=setosa      expected loss=0.6666667
 class counts:    50    50    50
 probabilities: 0.333 0.333 0.333
 left son=2 (50 obs) right son=3 (100 obs)
 Primary splits:
  Petal.Length < 2.45 to the left, improve=50.00000, (0 missing)
  Petal.Width  < 0.8  to the left, improve=50.00000, (0 missing)
Surrogate splits:
  Petal.Width  < 0.8  to the left, agree=1.000, adj=1.00, (0 split)
  Sepal.Length < 5.45 to the left, agree=0.920, adj=0.76, (0 split)
```

The summary of the model provides us with some interesting information. The complexity table shown at the top of the summary, provides information about all of the trees considered for the final model. It lists their complexity parameter, the number of splits, the resubstitution error rate, cross-validated error rate and the associated standard error. These latter three statistics will be described shortly.

The node information is described in sections beneath the complexity table. It provides information about the predicted class for that node, the number of observations, primary and competing splits and surrogates splits that can be used in the event of missing data. The primary split chosen for that node is the first one displayed. The splits beneath it are competing splits for that node. This information is useful when you have large datasets with many variables, some of which may be highly correlated.

The tree may be plotted using the following commands

```
> plot(iris.rp)
> text(iris.rp)
```

to produce a tree like the one shown in Figure 96. Predicted classes are displayed beneath each terminal node of the tree. Important splits that provide the best discrimination between classes are those that have the longest branches. For larger, more complicated trees it is sometimes useful to specify `uniform=T` as an argument to `plot`.

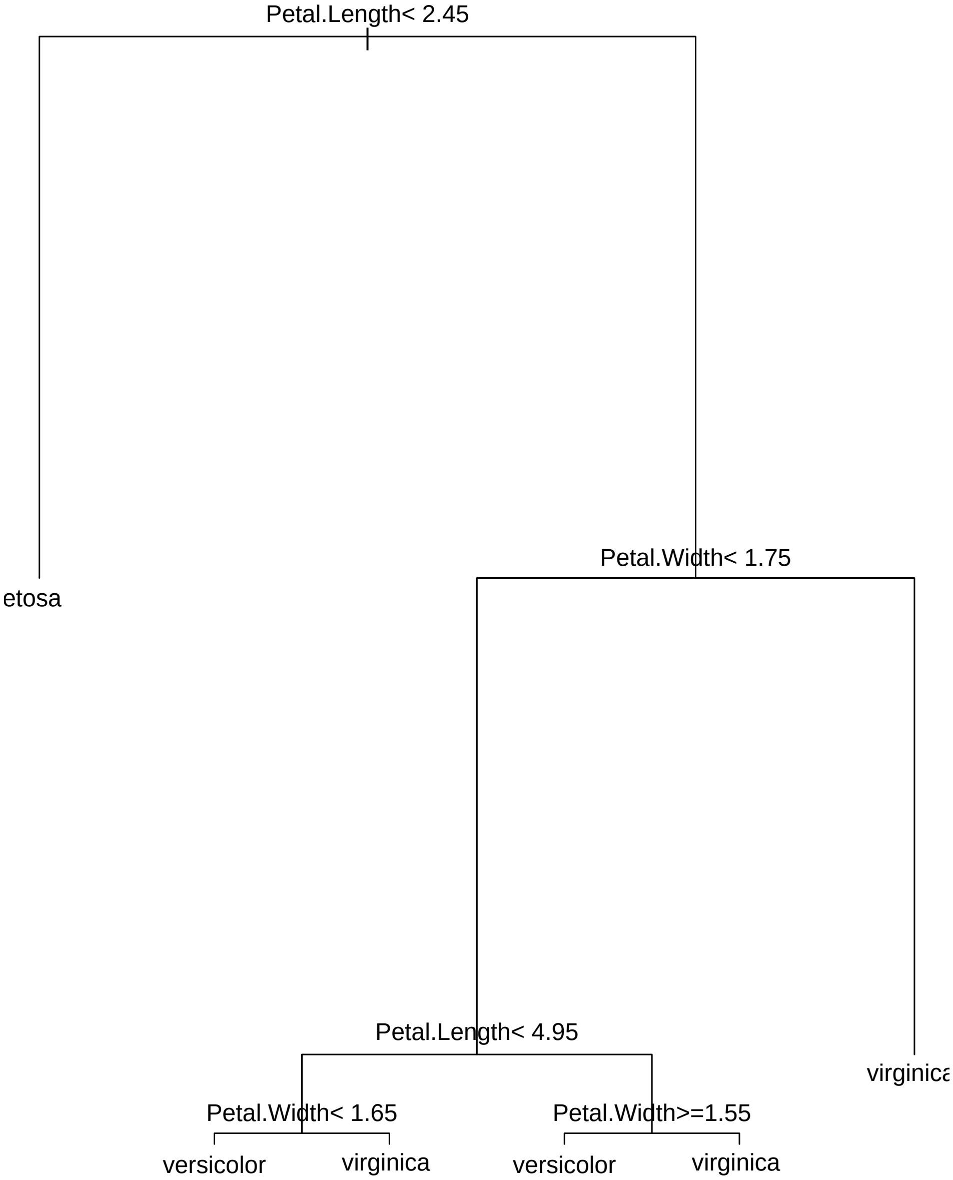


Figure 96: Classification tree produced on Fisher's Iris data (unpruned)

Recursive Partitioning Up Close

To illustrate the recursive partitioning process, we can produce a scatterplot of the primary split variables and overlay the plot with the locations of the splits. Figure 97 is the result of running the following script in R. The plot is produced using Trellis graphics.

```
require(lattice)
trellis.par.set(col.whitebg())
xyplot(Petal.Width~Petal.Length,iris,groups=Species,pch=16,
    col=c("red","green","blue"),
    panel=function(x,y,groups,...){
     panel.superpose(x,y,groups,...)
     panel.abline(v=2.45,lty=2)
     panel.segments(2.45,1.75,max(x)*2,1.75,lty=2)
     panel.segments(4.95,min(y)*-2,4.95,1.75,lty=2)
     panel.segments(2.45,1.65,4.95,1.65,lty=2)
     panel.segments(4.95,1.55,max(x)*2,1.55,lty=2)
   },
  key=list(columns=3,col=c("red","green","blue"),
    text=list(c("setosa","versicolor","virginica"))))
```

The plot shows the three species of iris, Setosa (red), Versicolor (green) and Virginica (blue) with the splits overlayed to illustrate how the data was partitioned. The first split on petal length ideally partitions setosa from the remaining two species. However the remaining two species are a little more difficult to partition based on petal length and petal width. Notice how we could have also used petal width to partition setosa from versicolor and virginica. In the summary output, petal width was listed as a surrogate with an agreement rating of 1, indicating that it would produce the same partition.

The Complexity Table

```
> printcp(iris.rp)
Classification tree:
rpart(formula = Species ~ Sepal.Length + Sepal.Width + Petal.Length +
```

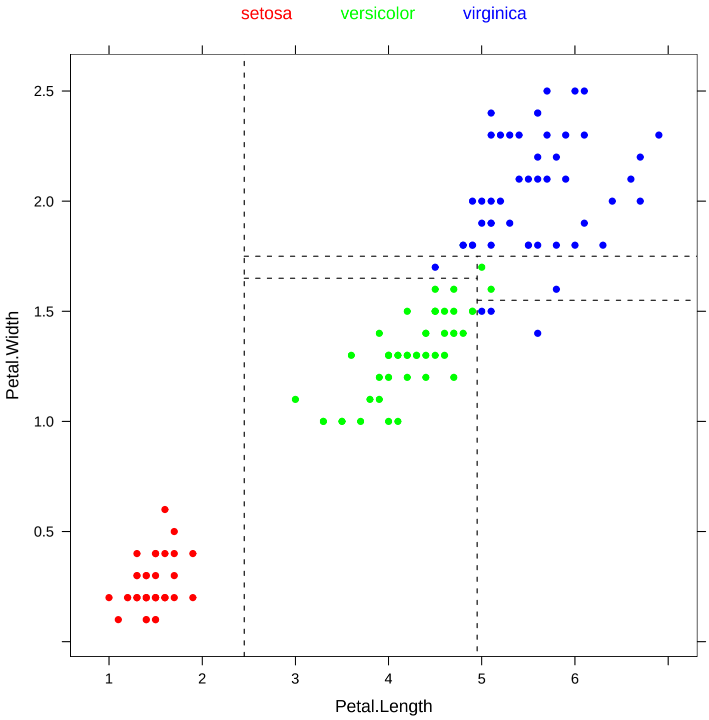


Figure 97:

```
    Petal.Width, data = iris, method = "class",
    control = rpart.control(minsplit = 4, cp = 1e-05))

Variables actually used in tree construction:
[1] Petal.Length Petal.Width
Root node error: 100/150 = 0.66667
n= 150
       CP nsplit rel error xerror     xstd
1 0.50000      0      1.00   1.21 0.048367
2 0.44000      1      0.50   0.76 0.061232
3 0.02000      2      0.06   0.10 0.030551
4 0.01000      3      0.04   0.11 0.031927
5 0.00001      5      0.02   0.10 0.030551
```

The printcp command prints out the complexity table that we saw in the summary of the tree. In this table, the tree yielding the lowest cross-validated error rate (xerror) is tree number 3. The tree yielding the minimum resubstitution error rate is tree number 5. The largest tree will always yield the lowest resubstitution error rate. Why? This will be explained later.

For larger trees, looking at a printout of the complexity table can be quite frightening. Sometimes a plot can help and this can be achieved using the plotcp function. Figure 98 produces the result and indicates that the best tree is the tree with 3 terminal nodes corresponding to a complexity value of 0.094.

```
> plotcp(iris.rp)
```

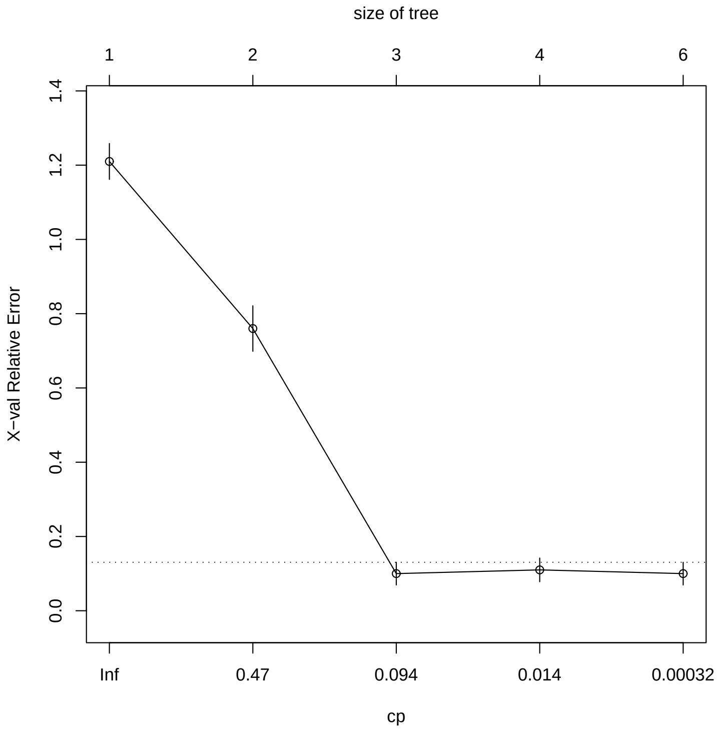


Figure 98: Graphical representation of the complexity table.

Pruning Trees

Once a large tree has been grown, it is important to prune the tree back to avoid overfitting. This is important for two reasons

- to ensure that the tree is small enough to avoid putting random variation into predictions

- to ensure that the tree is large enough to avoid putting systematic biases into predictions

Trees are pruned using a cost complexity measure which is defined as follows

$$R_\alpha = R + \alpha \times T$$

In the above expression, T represents the number of splits/terminal nodes in the tree, R represents the *tree risk* and α represents the complexity parameter, which is a penalty term that controls the size of the tree. The estimate of *tree risk* differs depending on the type of tree produced. For a classification tree, *tree risk* refers to the misclassification error, while for a regression tree, *tree risk* corresponds to the residual sum of squares.

Illustration of Pruning

Figure 99 illustrates the pruning process using a tree with five splits. At each split, we can produce an estimate for the resubstitution rate, R. This decreases as we move further down the tree. For the terminal node at the bottom of the tree (orange), α is 0. This increases as we move towards the root node (cyan). Calculation of R_α is via the above formula. For the top node, R_α is 1 and becomes smaller as the tree gets bigger.

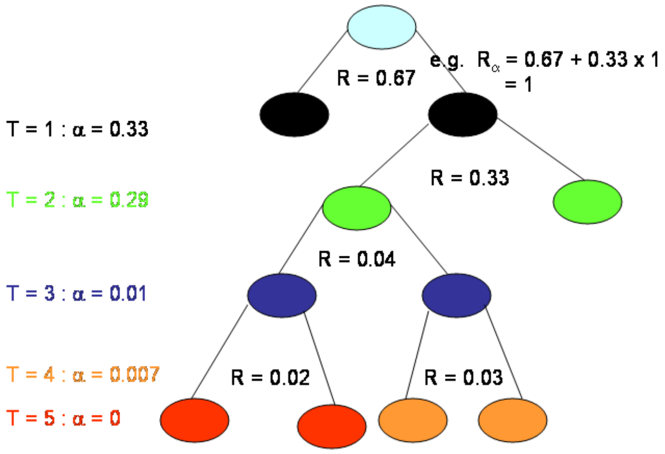
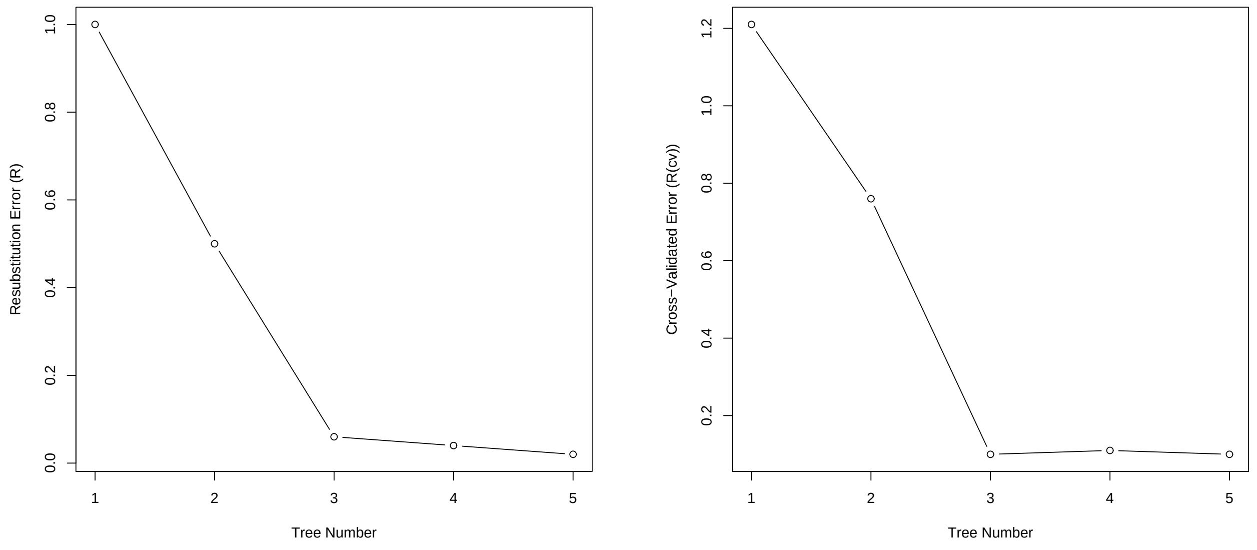


Figure 99: Illustration of the pruning process.

Figure 100(a) is the result of plotting the resubstitution rates versus tree size and shows this downward trend in error as tree size increases. The following script produces this plot.

```
# Plot the resubstitution error
> with(iris.rp,plot(cptable[,3],xlab="Tree Number",
    ylab="Resubstitution Error (R)",type="b"))
```



(a) (b)

Figure 100: Plot of (a) the resubstitution error rate and (b) the cross-validated error rate for trees produced on the iris dataset.

Although this is a measure of error, the resubstitution error rate is not all that useful in helping to decide on an optimal tree. Of course we could always choose the largest tree because this has the lowest resubstitution error rate but this tree would be biased. Furthermore, if we obtained a validation dataset, it is anticipated that classification using this tree would be poor.

Tree Selection

Instead of selecting a tree based on the resubstitution error rate, 10-fold cross-validation is used to obtain a cross-validated error rate, from which, the optimal tree is selected.

To recap, 10-fold cross-validation involves creating 10 random subsets of the original data, setting one portion aside as a test set, constructing a tree for the remaining 9 portions and evaluating the tree using the test portion. This is repeated for all portions and an estimate of the error, R_α is evaluated. Adding up the error across the 10 portions represents the cross-validated error rate, R_α^{CV}.

If we now plot the error rates computed via cross-validation using the following script

```
> with(iris.rp,plot(cptable[,4],xlab="Tree Number",
     ylab="Cross-Validated Error (R(cv))",type="b"))
```

we obtain a plot of the tree error like that shown in Figure 100(b). The minimum now occurs somewhere near the tree with three terminal nodes.

Selecting the Right Sized Tree

There are a number of ways to select the *right sized* tree. The first is to simply choose the tree with the lowest cross-validated error rate: $min(R_{\alpha}^{CV})$. The second, uses the *1 standard error* (SE) rule that is outlined in Breiman et al. (1984). This represents a tree, smaller in size but within one standard error to the tree yielding the minimum cross-validated error rate: $minR_{\alpha}^{CV} + SE$.

There is no hard and fast rule for tree selection. These are merely a guide.

Illustration of the 1 SE rule

The 1 SE rule is illustrated using the following piece of code, producing the plot in Figure 101.

```
 plotcp(iris.rp)
 with(iris.rp, {
   lines(cptable[,2]+1,cptable[,3],type="b",col="red")
  legend(locator(1),c("Resub. Error","CV Error","min(CV Error)+1SE"),
    lty=c(1,1,2),col=c("red","black","black"),bty="n")
  })
```

The script produces a plot of the resubstitution error (red) and the cross-validated error rate (black), with a dotted line drawn at the minimum plus one standard error. The dotted line helps us to choose the tree within one standard error of the tree yielding the minimum. We do this by seeing if a smaller sized tree has a cross-validated estimate below the dotted line. In this case, tree selection is straight forward and the tree yielding the lowest cross-validated error rate is also the 1SE tree.

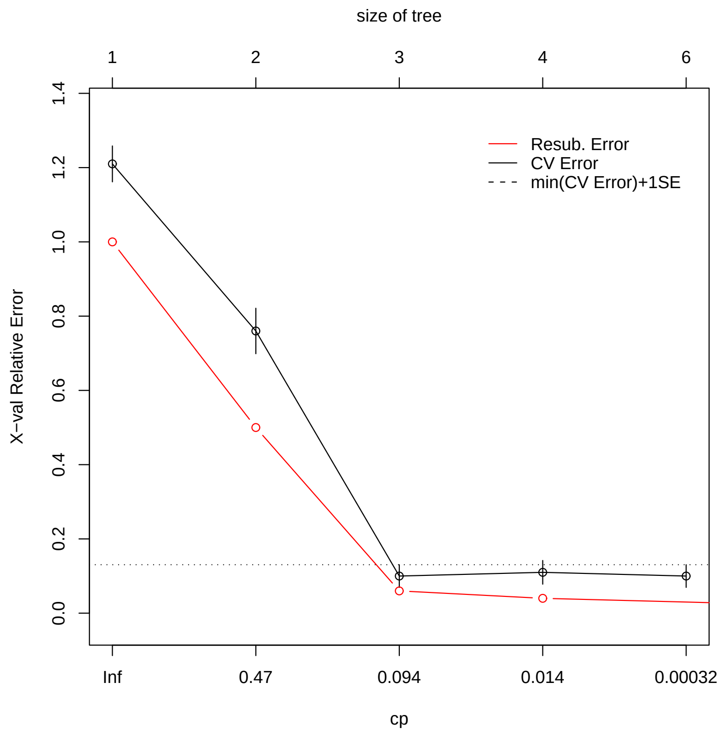


Figure 101: Illustration of the one standard error rule.

Pruning the Tree

We can use the information in the complexity table or alternatively the complexity plot (Figure 101) to prune the tree. To select the tree with three splits, we choose a complexity value greater than 0.094 but less than 0.47, which represents the tree with two splits. The tree can then be pruned using the `prune` function and associated summary tables and plots can be produced.

```
> iris.prune <- prune(iris.rp,cp=0.1)
> summary(iris.prune)
Call:
...
  n= 150

    CP nsplit rel error xerror       xstd
1 0.50      0      1.00   1.21 0.04836666
2 0.44      1      0.50   0.76 0.06123180
```

```
3 0.10      2      0.06   0.10 0.03055050

Node number 1: 150 observations,    complexity param=0.5
  predicted class=setosa      expected loss=0.6666667
    class counts:    50    50    50
   probabilities: 0.333 0.333 0.333
  left son=2 (50 obs) right son=3 (100 obs)
  Primary splits:
      Petal.Length < 2.45 to the left,  improve=50.00000, (0 missing)
      Petal.Width  < 0.8  to the left,  improve=50.00000, (0 missing)
Surrogate splits:
      Petal.Width  < 0.8  to the left,  agree=1.000, adj=1.00, (0 split)
      Sepal.Length < 5.45 to the left,  agree=0.920, adj=0.76, (0 split)

Node number 2: 50 observations
  predicted class=setosa      expected loss=0
    class counts:    50     0     0
   probabilities: 1.000 0.000 0.000

Node number 3: 100 observations,    complexity param=0.44
  predicted class=versicolor  expected loss=0.5
    class counts:     0    50    50
   probabilities: 0.000 0.500 0.500
  left son=6 (54 obs) right son=7 (46 obs)
  Primary splits:
      Petal.Width  < 1.75 to the left,  improve=38.969400, (0 missing)
      Petal.Length < 4.75 to the left,  improve=37.353540, (0 missing)
Surrogate splits:
      Petal.Length < 4.75 to the left,  agree=0.91, adj=0.804, (0 split)
      Sepal.Length < 6.15 to the left,  agree=0.73, adj=0.413, (0 split)

Node number 6: 54 observations
  predicted class=versicolor  expected loss=0.09259259
    class counts:     0    49     5
   probabilities: 0.000 0.907 0.093

Node number 7: 46 observations
  predicted class=virginica   expected loss=0.02173913
    class counts:     0     1    45
   probabilities: 0.000 0.022 0.978

> plot(iris.prune)
> text(iris.prune)
```

A partitioned plot can also be produced from the pruned model using the following piece of code.

```
require(lattice)
trellis.par.set(theme=col.whitebg())
xyplot(Petal.Width~Petal.Length,iris,groups=Species,pch=16,
  col=c("red","green","blue"),main="Partitioned Plot",
    panel=function(x,y,groups,...){
     panel.superpose(x,y,groups,...)
     panel.abline(v=2.45,lty=2)
     panel.segments(2.45,1.75,max(x),1.75,lty=2)
   },
  key=list(columns=3,col=c("red","green","blue"),
    text=list(c("setosa","versicolor","virginica"))))
```

The resulting plot is shown in Figure 103 and shows two distinct partitions of the data into the three species. Using this tree it is evident that there is some misclassification between versicolor and virginica.

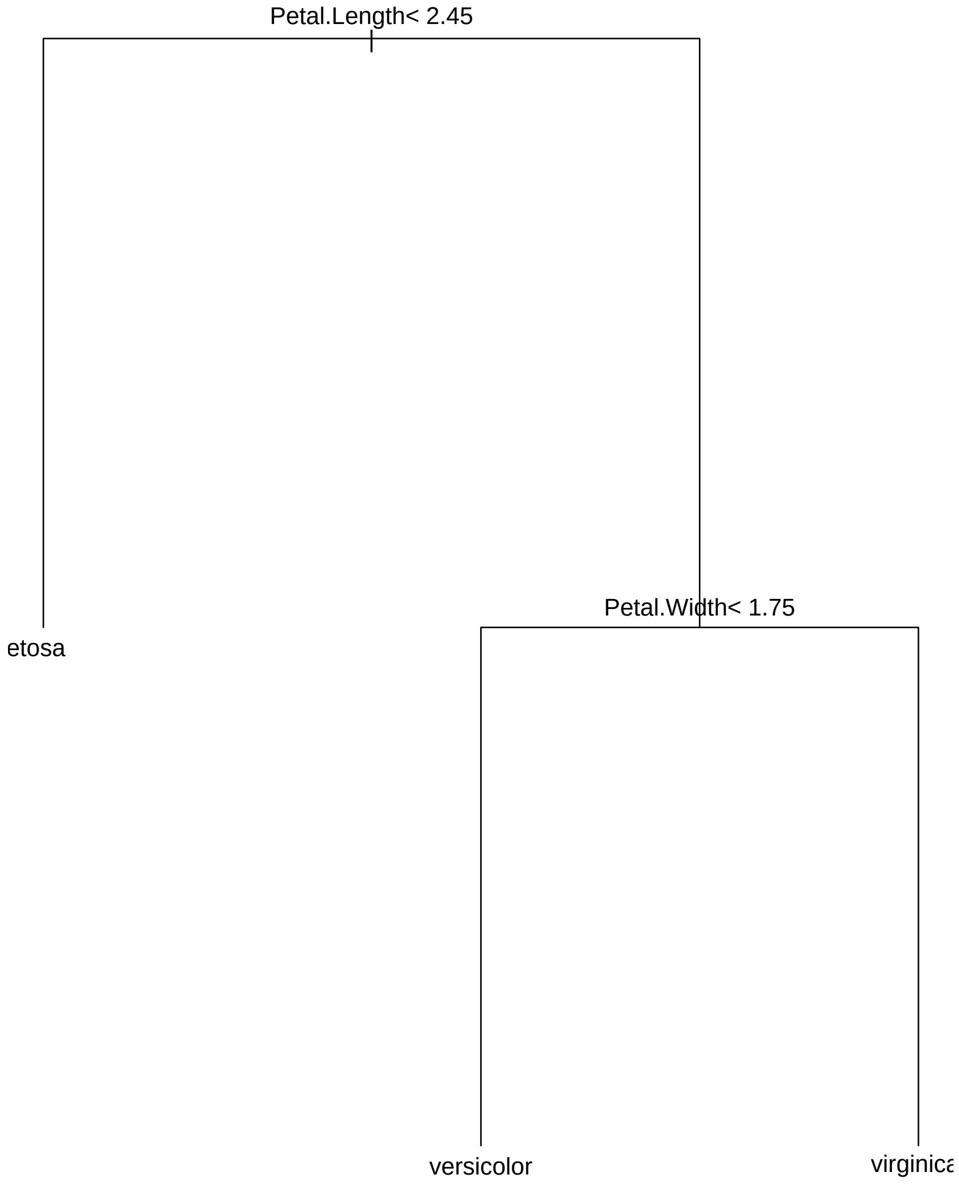


Figure 102: Pruned tree.

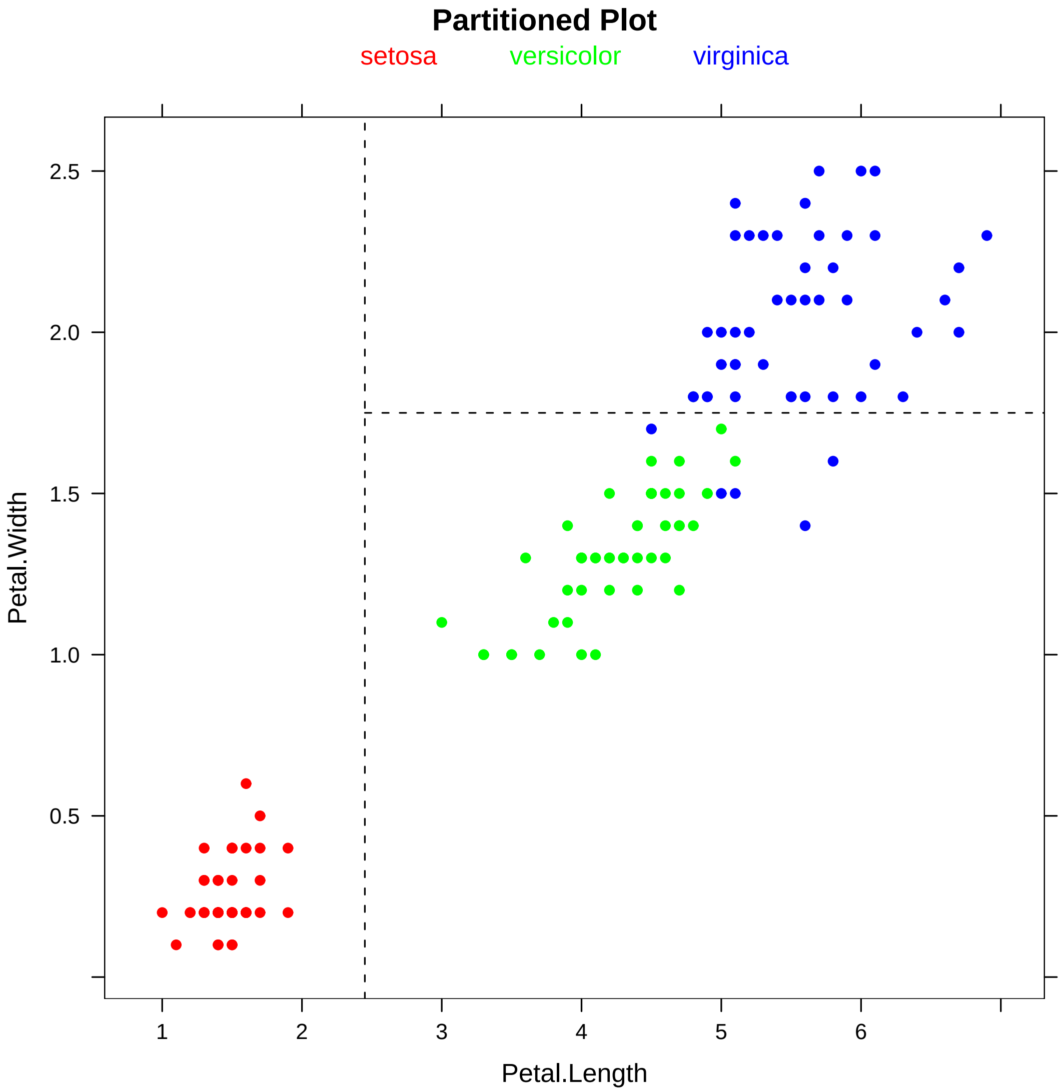


Figure 103: Partitioned plot of the pruned model

Predictions

Predictions can be made using the `predict` function. For example,

```
> iris.pred <- predict(iris.prune,type="class")
```

will produce predicted classes for the iris data based on the pruned model. To assess the performance of the model, we can produce a confusion matrix as follows

```
> table(iris.pred,iris$Species)
iris.pred    setosa versicolor virginica
  setosa     50       0          0
  versicolor  0      49          5
  virginica   0       1         45
```

The results indicate that the model misclassifies six species. Refer to the previous plot to see the misclassifications. The real test of course comes from a training/test sample.

Laboratory Exercise: Try this as an exercise.

Plotting Options

There are a number of ways to produce a plot of a decision tree using the RPART software. Here are four versions to choose from.

- Version A (Figure 104) is the default and produces simple square shaped splits with predicted classifications at terminal nodes.

  ```
  > plot.rpart(iris.prune,main="Version A")
  > text.rpart(iris.prune)
  ```

- Version B (Figure 105) provides a *prettier* version with angled splits and classification breakdowns at terminal nodes.

  ```
  > plot(iris.prune,uniform=T,branch=0.1,
  main="Version B")
  > text(iris.prune,pretty=1,use.n=T)
  ```

- Version C (Figure 106) provides a more fancy version with elipses for intermediate nodes and rectangles for terminal nodes. (N.B. Earlier versions of RPART blocked out the elipses and rectangles to make the node information more legible.)

```
> plot(iris.prune,uniform=T,branch=0.1,
margin=0.1,main="Version C")
> text(iris.rp2,pretty=1,all=T,use.n=T,fancy=T)
```

- The postscript version sends the previous plot to file

```
> post.rpart(iris.prune,filename="iris.ps",
main="Postscript Version")
```

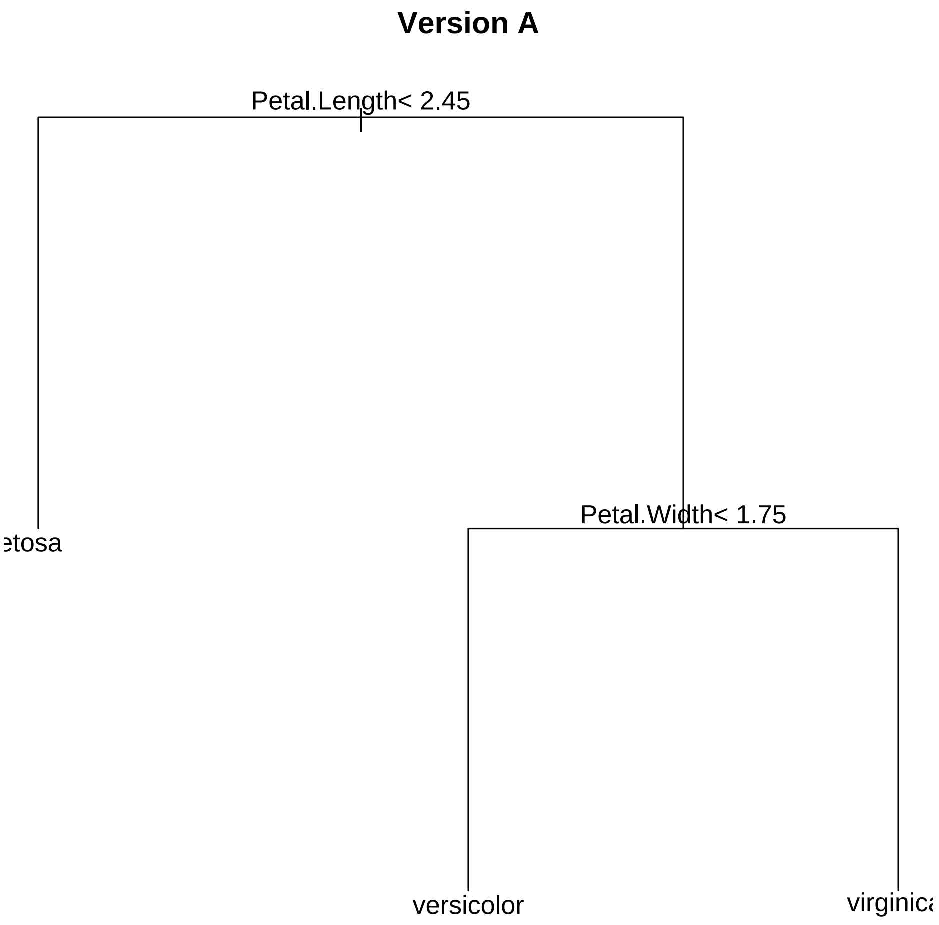


Figure 104: Default plot.

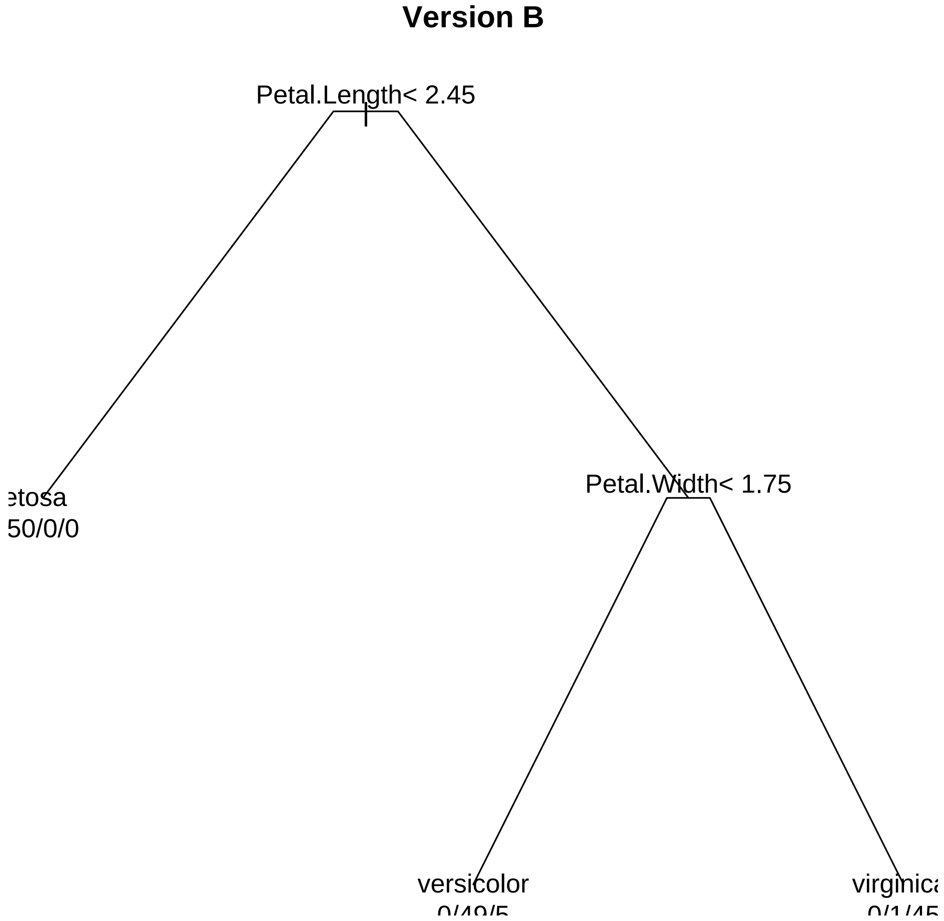


Figure 105: Prettier version with angled splits.

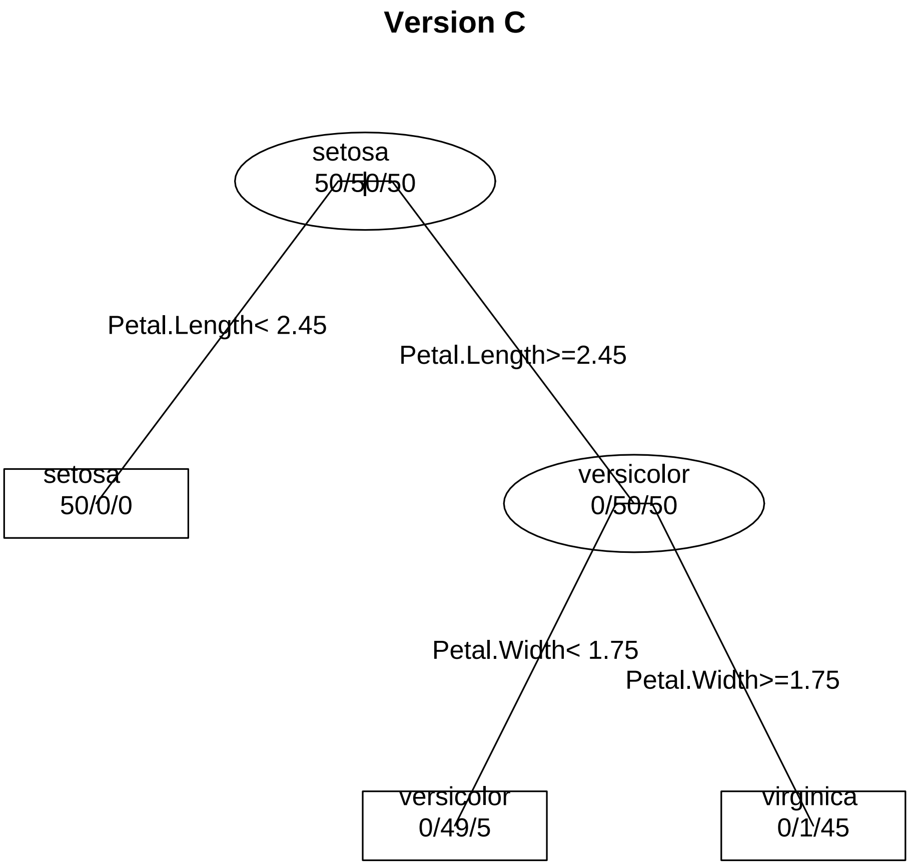


Figure 106: Fancy version. Note that in earlier versions of R, the elipses and boxes had white backgrounds.

Tree-based Models II

Regression Trees

Example: Boston Housing Data

The Boston Housing dataset is described in Appendix I and contains information about owner occupied homes in Boston. For more information regarding this dataset please see the paper by Harrison & Rubinfeld (1978).

The Boston Housing dataset will be used to create a regression tree to predict the median value of owner occupied homes in Boston using a variety of explanatory variables listed in the table below.

Covariates

`crim`	per capita crime rate by town
`zn`	proportion of residential land zoned for lots over 25,000 sq.ft.
`indus`	proportion of non-retail business acres per town
`chas`	Charles River dummy variable (= 2 if tract bounds river;1 otherwise)
`nox`	nitric oxides concentration (parts per 10 million)
`rm`	average number of rooms per dwelling
`age`	proportion of owner-occupied units built prior to 1940
`dis`	weighted distances to five Boston employment centres
`rad`	index of accessibility to radial highways
`tax`	full-value property-tax rate per $10,000
`ptratio`	pupil-teacher ratio by town
`black`	1000(Bk - 0.63)$\hat{2}$ where Bk is the proportion of blacks by town
`lstat`	% lower status of the population

Specification of the regression tree model can be performed using the following R code

```
> require(MASS)

> require(rpart)

> ?Boston

> boston.rp <- rpart(medv˜.,method="anova",data=Boston,
      control=rpart.control(cp=0.0001))
> summary(boston.rp)
```

The summary of the fit is shown below. Note that in comparison to the classification tree, many more trees are produced. The largest tree with 39 terminal nodes has the smalled cross-validated error rate. However this would be too large and biased to predict from.

The primary split for the first split of the tree is on the average number of rooms per dwelling. If a house has fewer than seven rooms (split=6.941) then that observation moves off to the left side of the tree otherwise it moves off to the right side of the tree. A competing split for that tree is the lower status of the population. This variable is also a surrogate for the `rm`. So if we did not know the number of rooms for a house we could use lower population status to perform that split instead.

```
Call:
rpart(formula = medv ˜ ., data = Boston, method = "anova",
  control = rpart.control(cp = 1e-04))
  n= 506
             CP nsplit rel error    xerror       xstd
1  0.4527442007      0 1.0000000 1.0027628 0.08298070
2  0.1711724363      1 0.5472558 0.6153397 0.05783459
3  0.0716578409      2 0.3760834 0.4343630 0.04807590
4  0.0361642808      3 0.3044255 0.3401563 0.04301558
...
37 0.0002274363     39 0.1170596 0.2502034 0.03772380
38 0.0001955788     40 0.1168322 0.2486663 0.03723079
39 0.0001000000     41 0.1166366 0.2483847 0.03722298

Node number 1: 506 observations,    complexity param=0.4527442
mean=22.53281, MSE=84.41956
left son=2 (430 obs) right son=3 (76 obs)
Primary splits:
 rm      < 6.941    to the left,  improve=0.4527442, (0 missing)
 lstat   < 9.725    to the right, improve=0.4423650, (0 missing)
 indus   < 6.66     to the right, improve=0.2594613, (0 missing)
Surrogate splits:
 lstat   < 4.83     to the right, agree=0.891, adj=0.276, (0 split)
 ptratio < 14.55    to the right, agree=0.875, adj=0.171, (0 split)
...
```

If we look at the complexity table in more detail we see that the tree corresponding to the lowest error rate is a tree with 27 terminal nodes. However, if we look higher up on the table, we see that there is a tree smaller in size but within one standard error of the minimum ($0.2786 < 0.2482 + 0.03755$) . This is a tree that has 12 terminal nodes.

```
           CP nsplit rel error xerror    xstd
 1  0.4527442      0    1.0000 1.0028 0.08298
 2  0.1711724      1    0.5473 0.6153 0.05783
 3  0.0716578      2    0.3761 0.4344 0.04808
 4  0.0361643      3    0.3044 0.3402 0.04302
 5  0.0333692      4    0.2683 0.3498 0.04594
 6  0.0266130      5    0.2349 0.3460 0.04537
 7  0.0158512      6    0.2083 0.3148 0.04314
 8  0.0082454      7    0.1924 0.2953 0.04396
 9  0.0072654      8    0.1842 0.2885 0.04195
10  0.0069311      9    0.1769 0.2869 0.04115
11  0.0061263     10    0.1700 0.2879 0.04124
12  0.0048053     11    0.1639 0.2786 0.04128 <- 1 SE
13  0.0045609     12    0.1591 0.2706 0.04081
14  0.0039410     13    0.1545 0.2708 0.04082
15  0.0033161     14    0.1506 0.2647 0.04067
16  0.0031206     15    0.1472 0.2608 0.03991
17  0.0022459     16    0.1441 0.2561 0.03977
18  0.0022354     18    0.1396 0.2541 0.03892
19  0.0021721     19    0.1374 0.2539 0.03892
20  0.0019336     20    0.1352 0.2532 0.03769
```

```
21  0.0017169     21    0.1333 0.2514 0.03768
22  0.0014440     22    0.1316 0.2495 0.03766
23  0.0014098     23    0.1301 0.2487 0.03758
24  0.0013635     24    0.1287 0.2490 0.03757
25  0.0012778     25    0.1273 0.2491 0.03758
26  0.0012474     26    0.1261 0.2487 0.03758
27  0.0011373     28    0.1236 0.2482 0.03755 <- Minimum
28  0.0009633     29    0.1224 0.2484 0.03753
29  0.0008487     30    0.1215 0.2485 0.03755
30  0.0007099     31    0.1206 0.2516 0.03779
31  0.0005879     32    0.1199 0.2505 0.03773
32  0.0005115     33    0.1193 0.2502 0.03772
33  0.0003794     34    0.1188 0.2499 0.03772
34  0.0003719     35    0.1184 0.2497 0.03773
35  0.0003439     36    0.1181 0.2501 0.03773
36  0.0003311     37    0.1177 0.2500 0.03773
37  0.0002274     39    0.1171 0.2502 0.03772
38  0.0001956     40    0.1168 0.2487 0.03723
39  0.0001000     41    0.1166 0.2484 0.03722
```

An alternative way to select the tree is through a plot of the cross-validated error versus the complexity parameter or tree size. Figure 107 summarises the information in the complexity table and highlights the optimal tree using a dotted line.

Both the complexity table and the complexity plot reveal a regression tree with 12 splits. Pruning the tree can be achieved by selecting a complexity value that is greater than that produced for the optimal tree but less than the complexity value for the next tree above it. For this example, we require a tree that has a complexity parameter between 0.0048 and 0.0061. Figure 108 displays the resulting pruned model.

```
> boston.prune <- prune(boston.rp,cp=0.005)
> plot(boston.prune)
> text(boston.prune)
```

The model shown in Figure 108 shows an important first split on theaverage number of

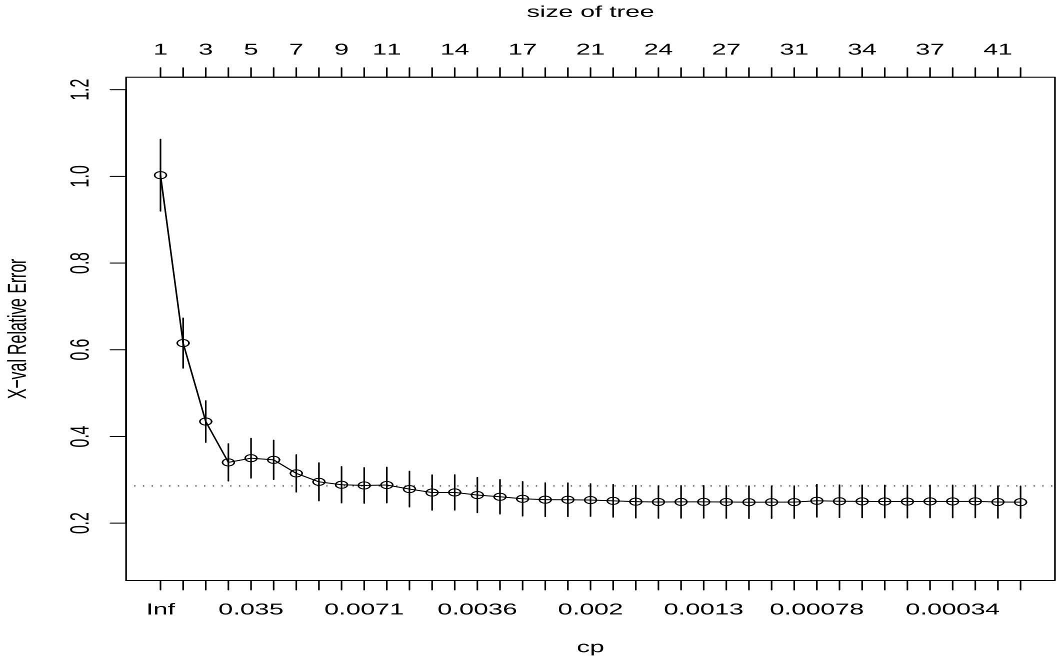


Figure 107: Plot of cross-validated errors produced for all trees versus the complexity value.

rooms per dwelling. The second split off to the left on `lstat` also seems to be quite an important split in terms of the models ability to partition the data to reduce the residual sums of squares.

The more expensive houses appear to be those that have a higher number of rooms (on average), while the cheap houses have a smaller number of rooms (<6.941 on average) and have a low status in the population with a high crime rate.

Interactive pruning allows you to interact with the tree and snip off splits of the tree using the mouse. For larger trees, this feature works best when you set the `uniform=T` option in the plot. The following script shows how you can interact with a plot.

```
> plot(boston.prune,uniform=T,branch=0.1)
> text(boston.prune,pretty=1,use.n=T)
> boston.prune.int <-  snip.rpart(boston.prune)
```

Clicking once on the node reveals information about that node. Clicking a second time removes that split. Figure 109 produces a tree with 9 terminal nodes resulting from this exercise of interactive pruning.

Predictions

We now examine the predictions from both models. We do this using the `predict` func-

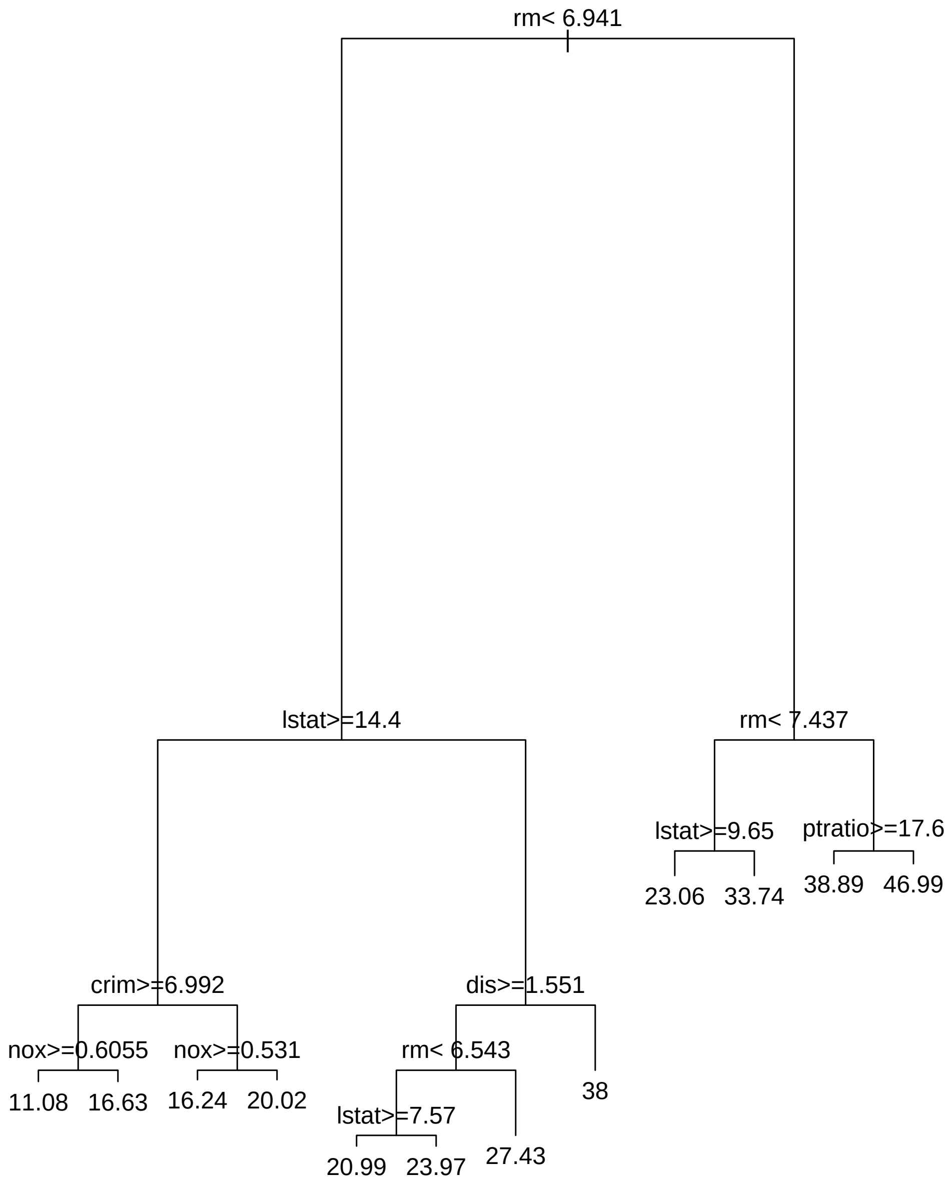


Figure 108: Pruned model.

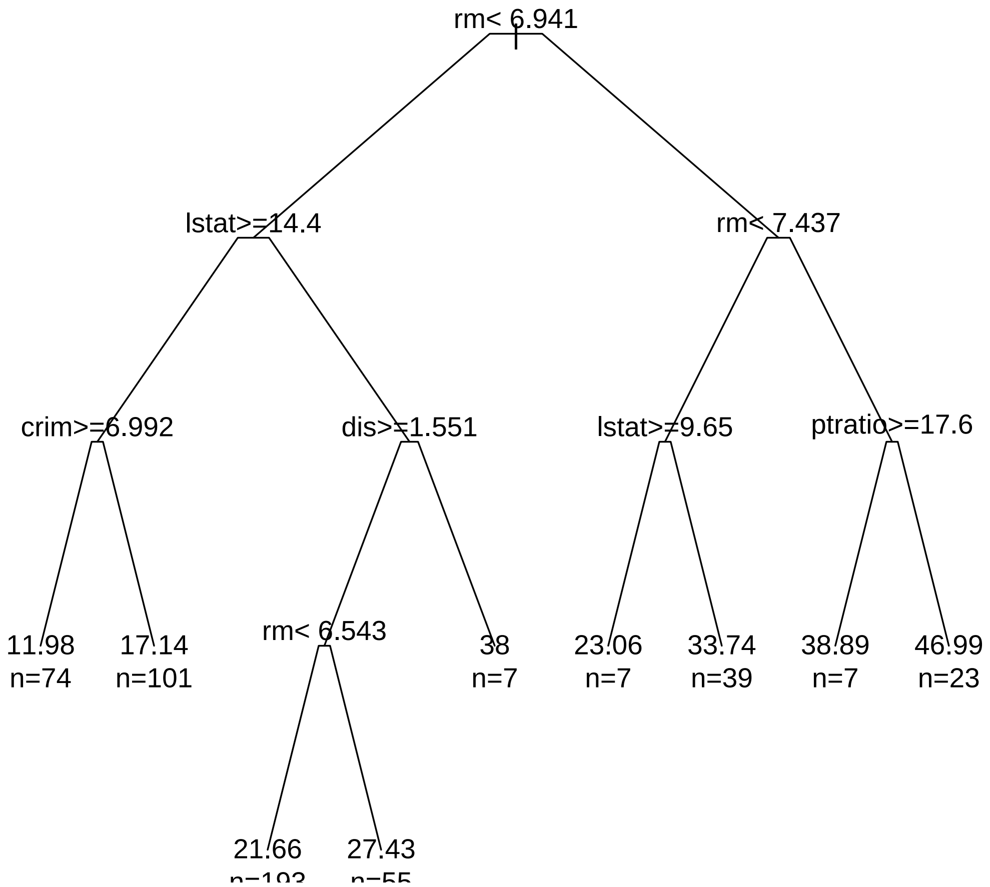


Figure 109: Pruned tree resulting from interactive pruning

tion as follows:

```
# Model 1: Pruning based on the 1SE rule
> boston.pred1 <- predict(boston.prune)


# Model 2: Result from interactive pruning
> boston.pred2 <- predict(boston.prune.int)
```

To assess the performance of each model, we compute the correlation matrix of predictions with the actual response.

```
> boston.mat.pred <- cbind(Boston$medv,
boston.pred1,boston.pred2)
> boston.mat.pred <- data.frame(boston.mat.pred)
> names(boston.mat.pred) <- c("medv","pred.m1","pred.m2")
> cor(boston.mat.pred)
           medv   pred.m1   pred.m2
medv    1.0000000 0.9144071 0.9032262
pred.m1 0.9144071 1.0000000 0.9877725
pred.m2 0.9032262 0.9877725 1.0000000
```

The correlations indicate that the predictions between both models are highly correlated with the response. Model 1 predictions are slightly better than model 2 predictions.

The predictions can be plotting using the following script. The `eqscplot` function is used to set up a square plotting region with axes that are in proportion to one another.

```
> par(mfrow=c(1,2),pty="s")
> with(boston.mat.pred,{
  eqscplot(pred.m1,medv,
    xlim=range(pred.m1,pred.m2),ylab="Actual",
    xlab="Predicted",main="Model 1")
  abline(0,1,col="blue",lty=5)
  eqscplot(pred.m2,medv,
    xlim=range(pred.m1,pred.m2),ylab="Actual",
    xlab="Predicted",main="Model 2")
  abline(0,1,col="blue",lty=5)
  par(mfrow=c(1,1))
 })
```

The plots in Figure 110 indicate that both models do reasonably well in predicting the median value of house prices in Boston. However, model 1 is a slight improvement over model 2.

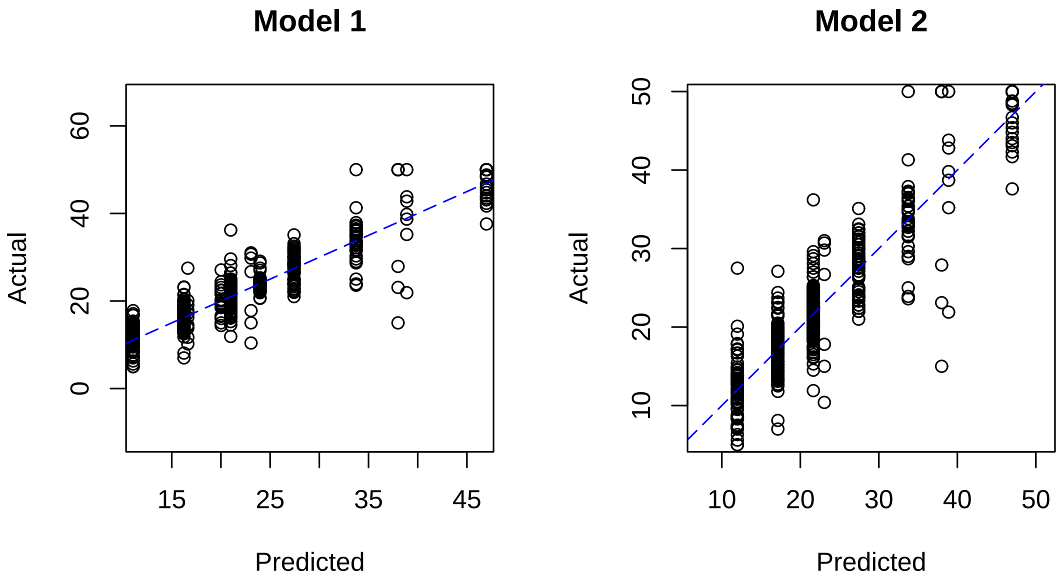


Figure 110: Actual versus predicted values for models 1 and 2.

Omitting a Variable

What happens to the tree when we omit the average number of rooms per dwelling (`rm`)? The following script updates the decision tree and removes this variable from the set of explanatory variables that the algorithm has to choose from.

```
> boston.rp.omitRM <- update(boston.rp,~.-rm)
> summary(boston.rp.omitRM)
Call:
             CP nsplit rel error    xerror       xstd
1  0.4423649998      0 1.0000000 1.0024048 0.08301982
2  0.1528339955      1 0.5576350 0.6023236 0.04816705
3  0.0627501370      2 0.4048010 0.4405161 0.04012065

Node number 1: 506 observations,    complexity param=0.442365
mean=22.53281, MSE=84.41956
left son=2 (294 obs) right son=3 (212 obs)
Primary splits:
      lstat   < 9.725    to the right, improve=0.4423650, (0 missing)
      indus   < 6.66     to the right, improve=0.2594613, (0 missing)
Surrogate splits:
      indus < 7.625    to the right, agree=0.822, adj=0.575, (0 split)
      nox   < 0.519    to the right, agree=0.802, adj=0.528, (0 split)
```

The result is that the primary split is now on `lstat` with surrogate splits on `indus` and `nox`. If we look closely at the original model below, we see that the competing split at node 1 is `lstat` at 9.725.

```
Node number 1: 506 observations,    complexity param=0.4527442
mean=22.53281, MSE=84.41956
left son=2 (430 obs) right son=3 (76 obs)
Primary splits:
      rm      < 6.941    to the left,  improve=0.4527442, (0 missing)
      lstat   < 9.725    to the right, improve=0.4423650, (0 missing)
      indus   < 6.66     to the right, improve=0.2594613, (0 missing)
```

```
Surrogate splits:
      lstat   < 4.83     to the right, agree=0.891, adj=0.276, (0 split)
      ptratio < 14.55    to the right, agree=0.875, adj=0.171, (0 split)
```

In the absence of `rm` the new model uses the competing split of the previous model to make the first split.

```
Node number 1: 506 observations,    complexity param=0.442365
mean=22.53281, MSE=84.41956
left son=2 (294 obs) right son=3 (212 obs)
Primary splits:
      lstat   < 9.725    to the right, improve=0.4423650, (0 missing)
      indus   < 6.66     to the right, improve=0.2594613, (0 missing)
Surrogate splits:
      indus < 7.625    to the right, agree=0.822, adj=0.575, (0 split)
      nox   < 0.519    to the right, agree=0.802, adj=0.528, (0 split)
```

Examining Performance through a Test/Training Set

To obtain a realistic estimate of model performance, we randomly divide the dataset into a training and test set and use the training dataset to fit the model and the test dataset to validate the model. Here is how we do this:

```
> set.seed(1234)
> n <- nrow(Boston)
# Sample 80% of the data
> boston.samp <- sample(n, round(n*0.8))

> bostonTrain <- Boston[boston.samp,]
> bostonTest <- Boston[-boston.samp,]

> testPred <- function(fit, data = bostonTest) {
#
# mean squared error for the performance of a
# predictor on the test data.
#
testVals <- data[,"medv"]
predVals <- predict(fit, data[, ])
sqrt(sum((testVals - predVals)ˆ2)/nrow(data))
}
# MSE for previous model (Resubstitution Rate)
> testPred(boston.prune,Boston)
[1] 3.719268
```

We compute the mean squared error for the previous model, where the entire dataset was used to fit the model and also validate it. The mean squared error estimate is 3.719268. This can be considered a resubstitution error rate.

Fitting the model again, but to the training dataset only and examining the complexity table reveals that the best model (based on the 1SE rule) is a tree with seven terminal nodes (see Figure 111.

```
> bostonTrain.rp <- rpart(medv˜.,data=bostonTrain,
method="anova",cp=0.0001)
> plotcp(bostonTrain.rp)
> abline(v=7,lty=2,col="red")  # 1 SE rule

# pruning the tree
> bostonTrain.prune <- prune(bostonTrain.rp,cp=0.01)
# plotting the tree
> plot(bostonTrain.prune)
> text(bostonTrain.prune)
# Computing the MSE
# Training dataset
> testPred(bostonTrain.prune,bostonTrain)
[1] 4.059407
# Test dataset
```

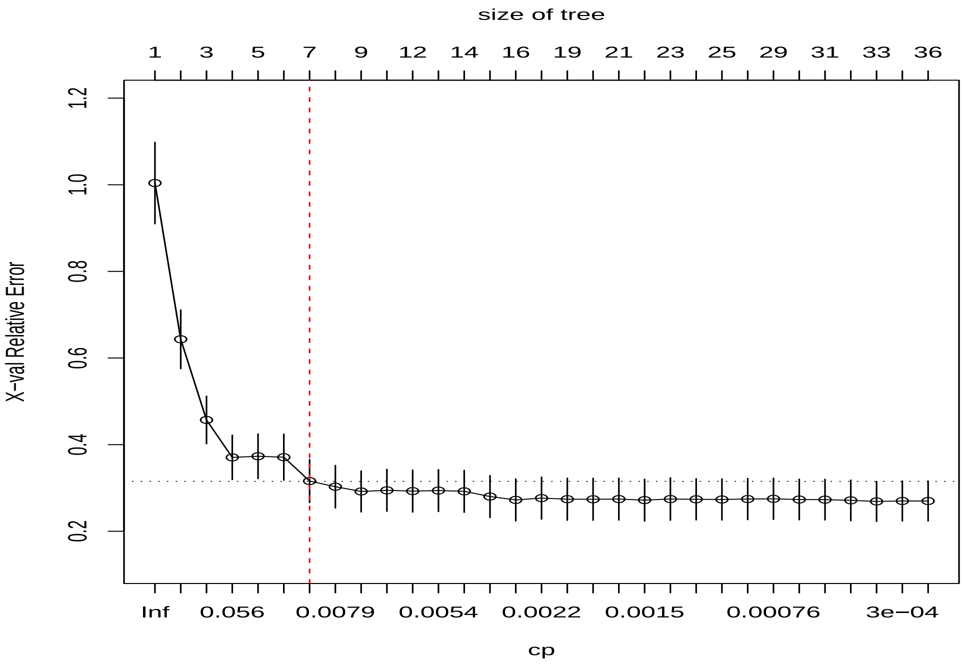


Figure 111: Complexity plot on decision tree fitted to the training data

```
> testPred(bostonTrain.prune,bostonTest)
[1] 4.782395
```

Pruning the tree and computing the mean squared error rates for the training and test set respectively produce values of 4.059407 and 4.782395.

The prediction performance of the models can be examined through plots of the actual versus predicted values as shown below in Figure 113.

```
> bostonTest.pred <- predict(bostonTrain.prune,bostonTest)
> with(bostonTest,{
  cr <- range(bostonTest.pred,medv)
  eqscplot(bostonTest.pred,medv,xlim=cr,ylim=cr,
        ylab="Actual",xlab="Predicted",main="Test Dataset")
  abline(0,1,col="blue",lty=5)
})
```

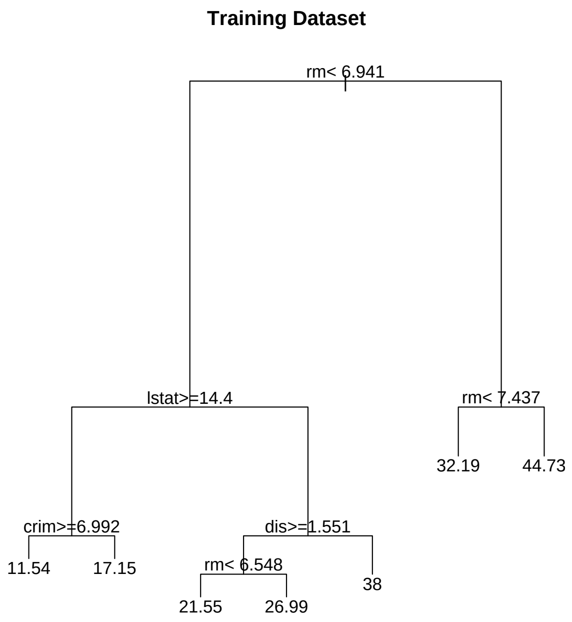


Figure 112: Pruned model produced using the training dataset

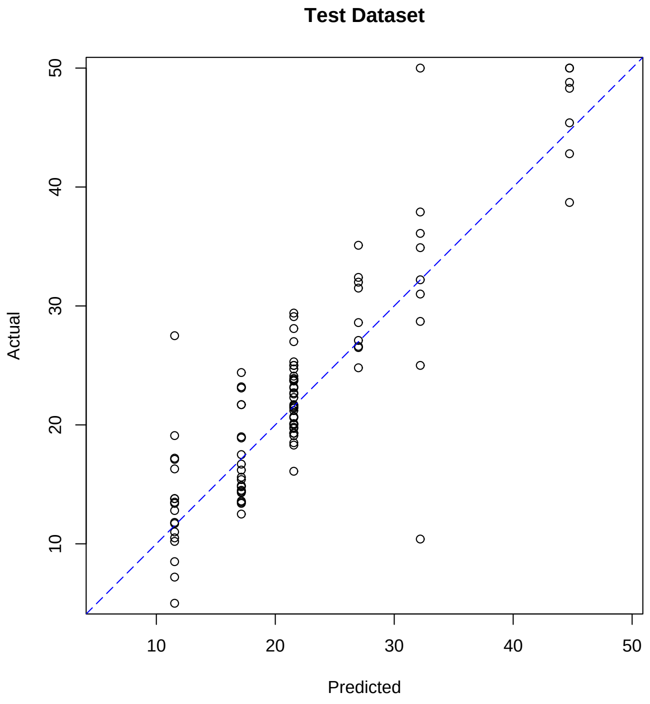


Figure 113: Plot of actual versus predicted values.

Advanced Topics

Throughout this session we have highlighted the useful features of decision trees and their ability to handle different data types and produce models that are readily interpretable. A major pitfall of decision trees is their instability. That is, if we were to introduce additional data or omit some data from the original dataset, the decision tree is likely to produce a tree with different splits.

Methods for overcoming tree instability that have been a hot topic over the past few years are aggregating methods, one of which is Bootstrap Aggregation, otherwise known as Bagging.

Bootstrap Aggregation (or Bagging)

Bagging is a technique for considering how different the result might have been if the algorithm were a little less greedy. Bootstrap training samples of the data are used to construct a forest of trees. Predictions from each tree are averaged (regression trees) or if classification trees are produced, a majority vote is used. The number of trees grown in the forest is still a matter of some debate. Random Forests develops this idea much further.

Pros/Cons of Bootstrap Aggregation Methods

The primary advantage of Bagging and most model aggregation methods is their ability to eliminate weaknesses concerned with model instability. However, the result is not a single tree, but a forest of trees with predictions that are averaged across models. This can make interpretation difficult.

An alternative approach, which overcomes model instability but focuses more on the interpretation of models is MART (Multiple Additive Regression Trees). Components of the model are additive and it is this feature of MART which makes it easier to see individual contributions to the model in a graphical way.

Some Bagging Functions

The following piece of script produces a bagged sample and fits a decision tree. By default 200 random bootstrapped samples are used by the algorithm. The `predict.bagRpart` function produces predictions on the bagged samples.

```
> bsample <- function(dataFrame) # bootstrap sampling
     dataFrame[sample(nrow(dataFrame), rep = T),  ]

> simpleBagging <- function(object,
 data = eval(object$call$data), nBags = 200, ...) {
```

```
 bagsFull <- list()
 for(j in 1:nBags)
 bagsFull[[j]] <- update(object,
 data = bsample(data))
     oldClass(bagsFull) <- "bagRpart"
     bagsFull
}

> predict.bagRpart <- function(object, newdata, ...)
 apply(sapply(object, predict, newdata = newdata), 1, mean)
```

Execute and Compare Results

We perform bagging on the Boston training dataset, form bagged predictions and compare with the *unbagged* models. The script for producing these comparisons is shown below.

```
> boston.bag <- simpleBagging(bostonTrain.rp)
> testPred(boston.bag)  # bit better!
[1] 3.369215
# Forming Predictions
> boston.bag.pred <- predict(boston.bag, bostonTest)
# Forming Correlation matrix to assess performance
> boston.mat.pred <- cbind(bostonTest$medv,
bostonTest.pred,boston.bag.pred)
> boston.mat.pred <- data.frame(boston.mat.pred)
> names(boston.mat.pred) <- c("medv","pred.m1",
"pred.bag")

> cor(boston.mat.pred)
         medv   pred.m1  pred.bag
medv     1.0000000 0.8737471 0.9432429
```

```
pred.m1  0.8737471 1.0000000 0.9468350
pred.bag 0.9432429 0.9468350 1.0000000
```

As shown above, the mean squared error esimate is an improvement on previous models. The correlation matrix also shows a vast improvment on predictions. Plotting these predictions, shows that the bagged predictons are much more closer to the 0-1 line comared to predictions made for the training/test set and the resubstitution rate. Figure 114 displays these results.

```
# Plotting
> with(bostonTest,{
  par(mfrow = c(2,2), pty = "s")
  frame()
  eqscplot(boston.bag.pred,medv,ylab="Actual",
xlab="Predicted",main="Bagging")
  abline(0, 1, lty = 4, col = "blue")
  eqscplot(bostonTest.pred, medv,  ylab="Actual",xlab="Predicted",
    main="Train/Test")
  abline(0, 1, lty = 4, col = "blue")
  with(Boston,
    eqscplot(boston.pred1,medv,ylab="Actual",xlab="Predicted",
       main="Resubstitution"))
  abline(0, 1, lty = 4, col = "blue")
  par(mfrow=c(1,1))
})
```

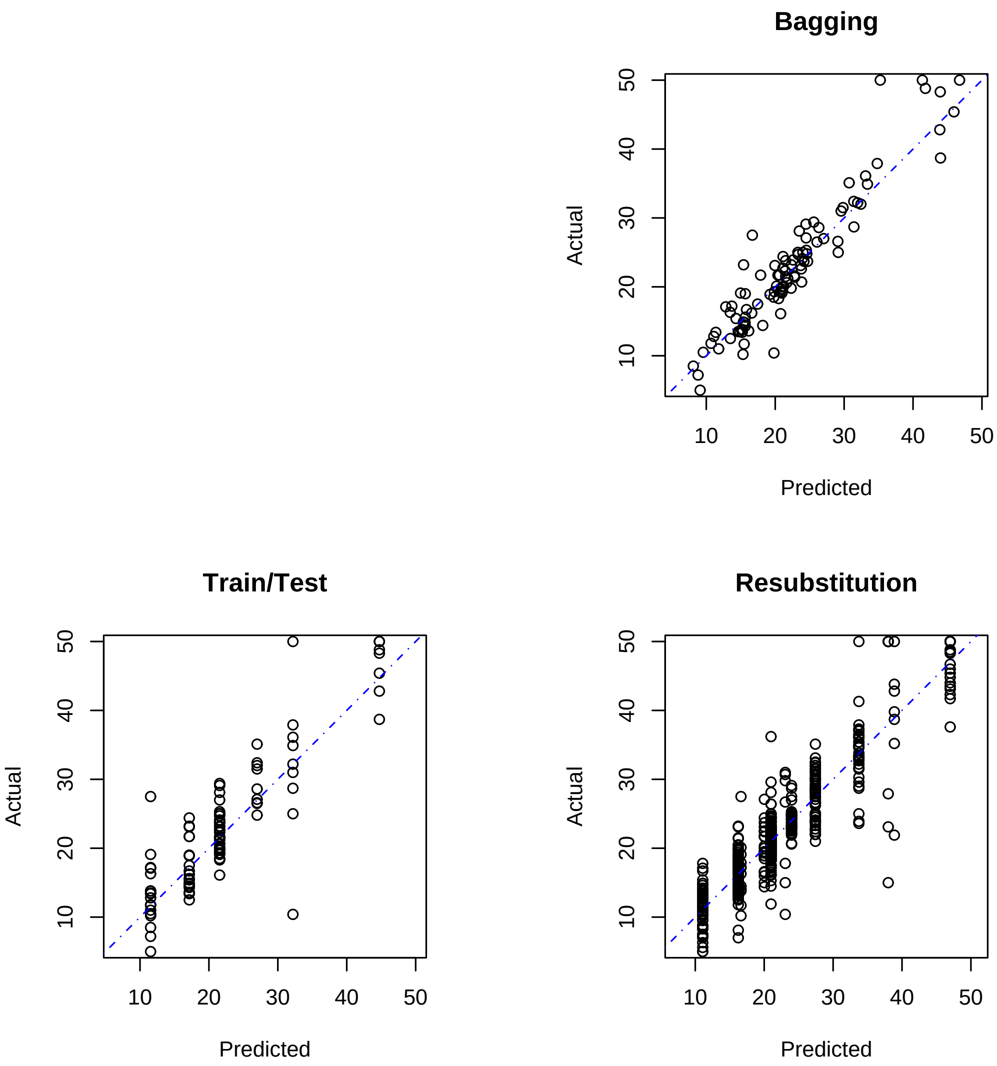


Figure 114: Comparison of predictions for three models: (1) Bagged decision tree, (2) Training/Test and (3) Resubstitution estimate.

APPENDIX I

Datasets

These descriptions are an adaptation of the information provided in the R Help system

Absenteeism from School in NSW

Description

Information about the absenteeism rates from schools in rural New South Wales, Australia. Classifications were made by culture, age, sex and learner status.

Variables

Name	Description	Mode
`Eth`	Ethnic background: aboriginal (A), not (N)	factor
`Sex`	Sex: female (F), male(M)	factor
`Age`	Age group: primary (F0), form 1 (F1), form 2 (F2), form 3 (F3)	factor
`Lrn`	Learner status: average (AL), slow (SL)	factor
`Days`	Days absent from school in the year	numeric

R Object

`quine`

Location and Source

This dataset is part of the `MASS` package and is described in more detail in Venables & Ripley (2002). For more information about this dataset see Quine (1978).

Cars93 Dataset

Description

The Cars93 dataset contains information on the sales of cars in the United States of America in 1993. Cars selected for this study were selected at random from two US consumer reports.

Variables

Name	Description	Mode
`Manufacturer`	Manufacturer of vehicle	character
`Model`	Model of vehicle	character
`Type`	Type: Small, sporty, compact, midsize, large or van	factor
`Min.Price`	Minimum price (in $1000)	numeric
`Price`	Midrange price	numeric
`Max.Price`	Maximum price (in $1000)	numeric
`MPG.city`	Miles per gallon by EPA rating (city driving)	numeric
`MPG.highway`	Miles per gallon by EPA rating (city driving)	numeric
`AirBags`	Airbags standard: none, driver only, driver and passenger	factor
`DriveTrain`	Drive train type: rear wheel, front wheel or 4WD	factor
`Cylinders`	Number of cylinders	numeric
`EngineSize`	Engine size in litres	numeric
`Horsepower`	Maximum horsepower	numeric
`RPM`	Revolutions per minute at maximum horsepower	numeric
`Rev.per.mile`	Engine revolutions per mile	numeric
`Man.trans.avail`	Manual transmission: yes or no	factor
`Fuel.tank.capacity`	Fuel tank capacity in US gallons	numeric
`Passengers`	Number of passengers	numeric
`Length`	Length of the vehicle in inches	numeric
`Wheelbase`	Wheelbase in inches	numeric
`Width`	Width of the vehicle in inches	numeric
`Turn.circle`	U-turn space in feet	numeric

Name	Description	Mode
`Rear.seat.room`	Rear seat room in inches	numeric
`Luggage.room`	Luggage capacity in cubic feet	numeric
`Weight`	Weight of the vehicle in pounds	numeric
`Origin`	Of non-USA or USA company origins	factor
`Make`	Manufacturer and Model of vehicle	character

R Object

`Cars93`

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Lock (1993) or visit the following web site: *http://www.amstat.org/publications/jse/vin1/datasets.lock.html*.

Car Road Tests Data

Description

Data extracted from the 1974 *Motor Trend* US magazine. It contains data on fuel consumption, automobile design and performance for 32 vehicles.

Variables

Name	Description	Mode
`mpg`	Miles per US gallon	numeric
`cyl`	Number of cylinders	numeric
`disp`	Displacement in cubic inches	numeric
`hp`	Gross horsepower	numeric
`drat`	Rear axle ratio	numeric
`wt`	Weight in pounds per 1000	numeric
`qsec`	Quarter mile time	numeric
`vs`	V or S	factor
`am`	Transmission: automatic (0), manual (1)	factor
`gear`	Number of forward gears	numeric
`carb`	Number of caruretors	numeric

R Object

```
mtcars
```

Location and Source

This dataset is part of the `datasets` package. For more information about this dataset see Henderson & Velleman (1981).

Copenhagen Housing Conditions Study

Description

Survey of Copenhagen housing conditions.

Variables

Name	Description	Mode
`Sat`	Satisfaction of householders with their present housing circumstances: Low, Medium, High	factor
`Infl`	Perceived degree of influence householders have on their propery: Low, Medium, High	numeric
`Type`	Type of rental accommodation: Tower, Atrium, Appartment, Terrace	factor
`Cont`	Contact residents are afforded with other residents: Low, High	factor
`Freq`	Number of residents in each class	numeric

R Object

`housing`

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Madsen (1976).

Fisher's Iris Data

Description

The iris dataset contains measurements in centimeters of the sepal length and width and petal length and width for 50 flowers from each of three species of iris: *setosa*, *versicolor* and *virginica*.

R provides two versions of the iris dataset

- `iris`: dataframe with 150 rows and 5 columns, described below
- `iris3`: 3-dimensional array of 50× 4× 3, where the first dimension represents the case number within the species subsample, the second gives the measurements for each species and the third provides information on species.

Variables

Name	Description	Mode
`Sepal.Length`	Sepal length	numeric
`Sepal.Width`	Sepal width	numeric
`Petal.Length`	Petal length	numeric
`Petal.Width`	Petal width	numeric
`Species`	Species names	factor

R Object

`iris, iris3`

Location and Source

This dataset is part of the `datasets` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Anderson (1935) and Fisher (1936).

Iowa Wheat Yield Data

Description

The Iowa dataset is a toy example that summarises the yield of wheat (bushels per acre) for the state of Iowa between 1930-1962. In addition to yield, year, rainfall and temperature were recorded as the main predictors of yield.

Variables

Name	Description	Mode
`Year`	Year of harvest	numeric
`Rain0`	Pre-season rainfall	numeric
`Temp1`	Mean temperature for growing month 1	numeric
`Rain1`	Rainfall for growing month 1	numeric
`Temp2`	Mean temperature for growing month 2	numeric
`Rain2`	Rainfall for growing month 2	numeric
`Temp3`	Mean temperature for growing month 3	numeric
`Rain3`	Rainfall for growing month 3	numeric
`Temp4`	Mean temperature for harvest month	numeric
`Yield`	Yield in bushels per acre	numeric

R Object

`iowa.xls, iowa.csv`

Location and Source

This dataset is provided as an excel spread sheet or comma delimited text file. For more information about this dataset see Draper & Smith (1981).

Janka Hardness Data

Description

Janka Hardness is an importance rating of Australian hardwood timbers. The test itself measures the force required to imbed a steel ball into a piece of wood and therefore provides a good indication to how the timber will withstand denting and wear.

Janka hardness is strongly related to the density of the timber and can usually be modelled using a polynomial relationship. The dataset consists of density and hardness measurements from 36 Australian Eucalypt hardwoods.

Variables

Name	Description	Mode
`Density`	Density measurements	numeric
`Hardness`	Janka hardness	numeric

R Object

`janka.xls, janka.csv`

Location and Source

This dataset is part of the `SemiPar` package and is described in detail in Williams (1959).

Lung Disease Dataset

Description

The lung disease dataset describes monthly deaths from bronchitis, emphysema and asthma in the United Kingdom from 1974 to 1979. There are three datasets available for analysis:

- `ldeaths`: monthly deaths reported for both sexes
- `mdeaths`: monthly deaths reported for males only
- `fdeaths`: monthly deaths reported for females only

Variables

Time series

R Object

```
ldeaths, mdeaths, fdeaths
```

Location and Source

This dataset is part of the `datasets` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Diggle (1990).

Moreton Bay Data

Description

Coastline of islands in Moreton Bay in South-East Queensland.

Variables

Matrix of values corresponding to latitudes and longitudes

R Object

`MB_coastline.txt`

Location and Source

This dataset was produced using coastline extractor, a web based (free) service developed by the National Geophysical Data Center in Boulder Colorado USA. For more information see: *http://www.ngdc.noaa.gov/mgg/shorelines/shorelines.html*

Muscle Contraction in Rat Hearts

Description

Experiment to assess the influence of calcium chloride on the contraction of the heart muscle of 21 rats.

Variables

Name	Description	Mode
`Strip`	heart muscle strip (S01-S21)	factor
`Conc`	Concentration of calcium chloride solution	numeric
`Length`	Change in length	numeric

R Object

`muscle`

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Linder et al. (1964).

Petroleum Rock Samples

Description

Dataset contains information on 48 rock samples taken from a petroleum reservoir, where twelve core samples were sampled by four cross-sections.

Variables

Name	Description	Mode
`area`	Area (in pixels) or pores space	numeric
`peri`	Perimeter (in pixels)	numeric
`shape`	Perimeter per square root area	numeric
`perm`	Permeability in milli-Darcies	factor

R Object

`rock`

Location and Source

This dataset is part of the `datasets` package and is described in detail in Venables & Ripley (2002).

Petrol Refinery Data

Description

Prater's petrol refinery data contains information about the petroleum refining process. This dataset was originally used by Prater (1956) to build an estimation equation for yield.

Variables

Name	Description	Mode
`No`	Crude oil sample identification label (A-J)	factor
`SG`	Specific Gravity	numeric
`VP`	Vapour pressure (psi)	numeric
`V10`	Volatility of crude oil	numeric
`EP`	Desired volatility of gasoline	numeric
`Y`	Yield as a percentage of crude	numeric

R Object

`petrol`

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Prater (1956).

Recovery of Benthos on the GBR

Description

Study investigates the recovery of benthos after trawling on the Great Barrier Reef (GBR). Six control plots and six impact plots were visited two times prior to treatment and four times post treatment. The total number of animals of each species was counted for each transect using a benthic sled.

Variables

Benthos

Name	Description	Mode
`Intensity`	Mean trawl intensity	numeric
`SweptArea`	Total swept area of the sled video camera	numeric
`Cruise`	Boat identification	factor
`Plot`	Site identification	character
`Months`	Survey Month: -14, -8, 1, 10, 23, 61	factor
`Treatment`	Treatment: Control, Impact	factor
`Time`	Survey Month: pre-treatment (0), post-treatment (1, 10, 23, 61)	factor
`Impact`	Indicator for surveys performed pre and post treatment	factor
`Topography`	Topography: Shallow, Deep	factor
`Species`	Species counts (1 column for each species)	numeric

R Object

`Benthos.csv, SpeciesNames.csv`

Location and Source

The data for this study is contained in an excel spreadsheet. Detailed information about the study design and data collected is contained in a CSIRO report. See Pitcher et al. (2004).

Stormer Viscometer Data

Description

The stormer viscometer dataset contains measurements on the viscosity of a fluid, which is achieved by measuring the time taken for an inner cylinder in the viscometer to perform a fixed number of revolutions in response to some weight.

Variables

Name	Description	Mode
`Viscosity`	Viscosity of fluid	numeric
`Wt`	Actuating weight	numeric
`Time`	Time taken	numeric

R Object

`stormer`

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Williams (1959).

US State Facts and Figures

Description

Statistics provided for 50 states of the United States of America

Variables

Name	Description	Mode
`Population`	Population estimate taken from 1 July 1975	numeric
`Income`	Per capita 1974 income	numeric
`Illiteracy`	Percent of population illiterate	numeric
`Life Exp`	life expectancy in years	numeric
`Murder`	murder rate per 100,000	numeric
`HS Grad`	percent high school graduates	numeric
`Frost`	mean number of days with minimum temperature below freezing	numeric
`Area`	land area in square miles	numeric

R Object

```
state.x77
```

Location and Source

This dataset is part of the `datasets` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Becker et al. (1988).

Volcano Data

Description

The volcano dataset provides topographic information on a 10m × 10m grid for Maunga Whau (Mt Eden), one of 50 volcanos in the Auckland volcanic field.

Variables

Matrix consisting of 87 rows (grid lines running east to west) and 61 columns (grid lines running south to north).

R Object

`volcano`

Location and Source

This dataset is part of the `datasets` package (which is automatically loaded in an R session). It has been digitized from a topographic map by Ross Ihaka.

Whiteside's Data

Description

The Whiteside dataset evolved from a private study conducted by Mr Derek Whiteside of the United Kingdom Building Research Station. Mr Whiteside was interested in the effect of insulation on gas consumption at his place of residence in south-east England. He recorded these measurements over two heating seasons: (1) 26 weeks before and (2) 30 weeks after cavity-wall insulation was installed.

Variables

Name	Description	Mode
`Insul`	*Before* or *After* insulation	factor
`Temp`	Average outside temperature in degrees Celsius	numeric
`Gas`	Weekly gas consumption in 1000s of cubic feet	numeric

R Object

```
whiteside
```

Location and Source

This dataset is part of the `MASS` package and is described in detail in Venables & Ripley (2002). For more information about this dataset see Hand et al. (1993).

APPENDIX II

Laboratory Exercises

These exercises are an adaptation of Bill Venable's Data Analysis and Graphics Course for S-PLUS.

Lab 1: R - An Introductory Session

The following session is intended to introduce you to some features of R, some of which were highlighted in the first session of the workshop.

Starting the R Session

`Create Directory`	Create a new directory on the desktop. Call it Session1.
`Right Click R Icon`	Create a shortcut by right clicking on the R icon. Rename the R session to `Session1`. Right click the R Icon and select Properties. Alter the Start in directory to reflect the new directory that you set up on the desktop. Click on OK once it has been entered.
`Double Click R Icon`	Start R by double clicking on the R icon that you have just created.
`Edit GUI Preferences`	Go to the Edit Menu and select GUI Preferences. Select SDI for Single window display. Click on OK.

Setting up the Graphics Device

`> require(lattice)`	Attach the trellis library
`> ?  trellis.par.set(col.whitebg())`	Get help on trellis parameter settings
`> show.settings()`	- Show current trellis settings
`> trellis.par.set(col.whitebg())`	Change the color of the background
`> windows()`	Standard graphics device

Simple Model

We will start off with a little artificial simple linear regression example, just to get you used to assignments, calculations and plotting.

The data we generate will be of the form $y = 1 + 2x + \text{errors}$ and then try to estimate the intercept and slope, which we know are 1 and 2 respectively and see how well we do.

The errors will be made to *fan out* as you go along the line. Such *heteroscedasticity* is a common feature of many real data sets.

```
> x <- 1:50
```

Make $x = (1, 2, \ldots, 49, 50)$.

```
> w <- 1 + x/2
```

We will use w to make the errors *fan out* along the line.

```
> rdata <- data.frame(x=x,
 y=1+2*x+rnorm(x)*w)
> rdata
```

Make a *data frame* of two columns, x and y, and look at it.

```
> fm <- lm(y∼x,data=rdata)
> summary(fm)
```

Fit a simple linear regression of y on x and look at the analysis.

```
> attach(rdata)
```

Make the columns in the data frame visible as variables.

```
> search()
> objects(2)
```

Look at the *search list*. This is the database from which R finds its objects. Look at those in position 2.

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

Standard point plot.

```
> abline(1,2,lty=3)
```

The true regression line: (intercept: $a = 1$, slope: $b = 2$, line type: 3)

```
> abline(coef(fm),col="blue")
```

Mark in the estimated regression line

```
> segments(x,fitted(fm),x,y,
 lty=4,col="red")
```

Join the points vertically to the fitted line, thus showing the errors or *residuals*. We will look more closely at the residuals below. At any time you can make a hard copy of the graphics by clicking on the `File` menu and selecting `Print`.

```
> plot(fitted(fm),resid(fm),
 xlab="Fitted values",
 ylab="Residuals",main=
 "Residuals vs Fitted")
> abline(h=0,lty=4)
```

A standard regression diagnostic plot to check for unequal variance. Can you see it?

```
> qqnorm(resid(fm),main=
 "Normal Scores Plot")
> qqline(resid(fm))
```

A Normal scores plot to check for skewness, kurtosis and outliers in the residuals. (Not very useful here since these properties are bound up with unequal variance.

```
> detach("rdata")
```

Remove data frame from the search list.

```
> rm(fm,x,rdata)
```

Clean up.

Cars93 Dataset

```
> require(MASS)
```

Now we look briefly at some actual data.

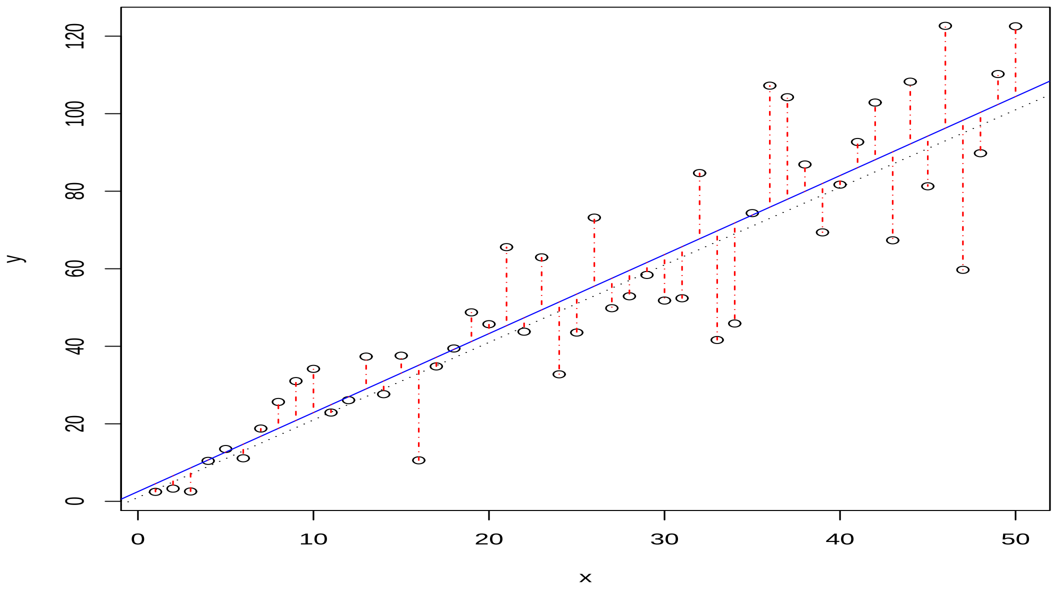


Figure 115: A simulated heteroscedastic regression.

```
> ?Cars93
```

The `Cars93` data frame contains information about 93 makes of car (or van) on sale in the USA in 1993. For each make there are 26 variables recorded ranging from `Price` (of the basic model) through to `Weight`. The help document gives more information.

```
> with(Cars93,plot(Type,
 Price,
ylab="Price (in $1,000)")
```

`Type` is a factor, not a quantitative variable. In this case a plot gives a boxplot by default.

```
> attach(Cars93)
> Tf <- table(Type)
> Tf
> Cf <- table(Cylinders)
> Cf
```

How many cars are there of each type? With each number of cylinders?

```
> TC <- table(Type,Cylinders)
> TC
```

A two-way table. What types of car were the two with five cylinders?

```
> Make[Cylinders=="5"]
```

Which makes are they? (Note *two* equal signs.

```
> rbind(cbind(TC,Tf),
 c(Cf,sum(Cf)))
```

Put the marginal totals on the table.

```
> plot(Weight,MPG.city)
```

City MPG drops off with increasing weight, but

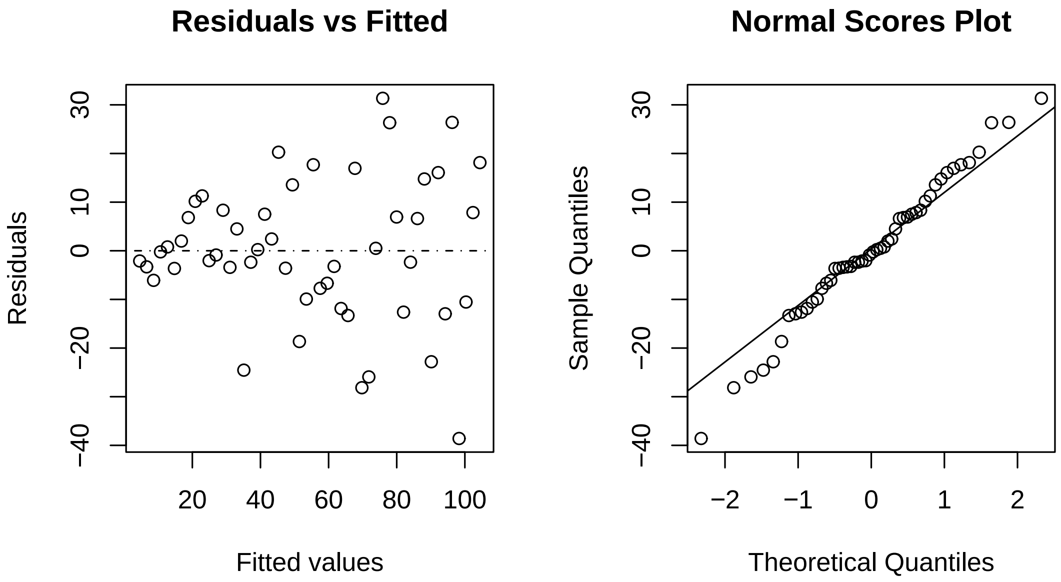


Figure 116: Heteroscedastic regression: diagnostic plots

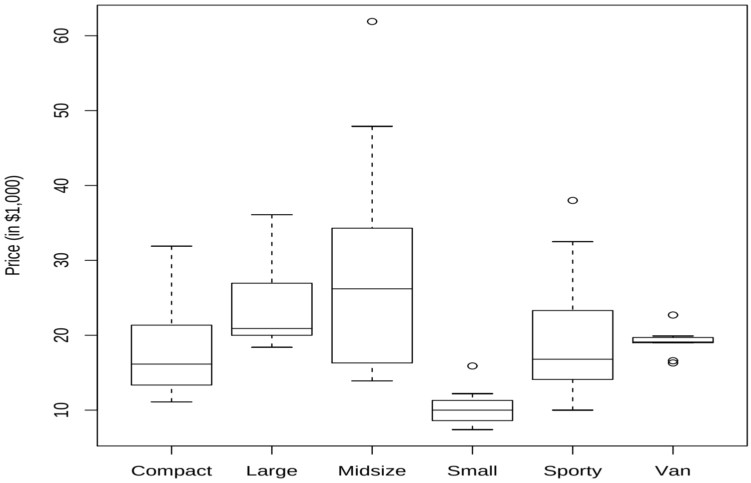


Figure 117: Boxplots of price on type of car

not in a straight line, which would be simpler. Try *gallons per mile* instead.

```
> Cars93T <- transform(Cars93,
 GPM=1/MPG.city)
> with(Cars93T,
 plot(Weight,GPM,pch=3))
> fm <- lm(GPM Weight,Cars93T)
> abline(fm,lty=4,col="blue")
```

The regression looks much more linear. Note the use of the `transform` and `with` functions. Fit a straight line model and add it to the curve. There is a lot of deviation, but the model seems appropriate.

```
> plot(fitted(fm),
 resid(fm))
> abline(h=0,lty=2)
```

Large positive residuals indicate good fuel economy (after allowing for weight), whereas large negative residuals indicate poor fuel economy.

```
> identify(fitted(fm),
 resid(fm),Make)
```

With the mouse, position the cursor in the plot window. It will change shape. Place it next to one of the more extreme residuals and click with the *left* mouse button. Repeat for as many as you like. When you have identified enough, stop the process by clicking on the `stop` button in the top left hand corner of the plotting region.

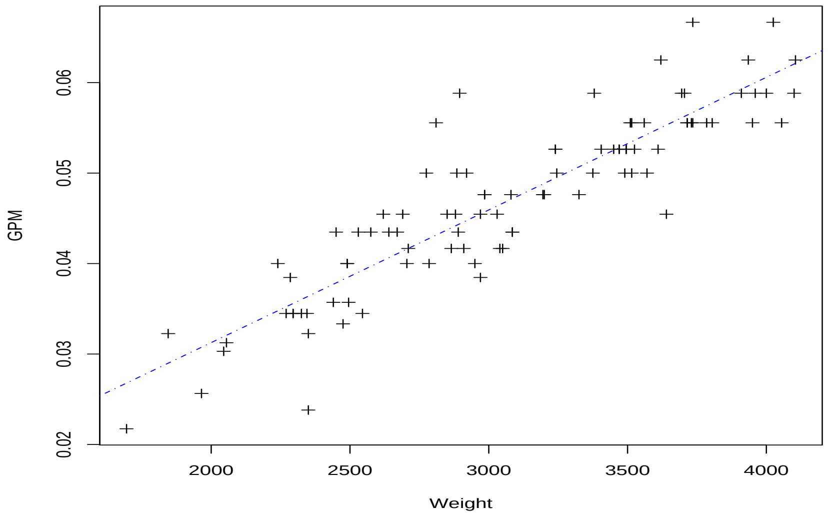


Figure 118: Gallons per mile vs car weight

Graphics

We now look at some more graphical facilities: contour and 3-dimensional perspective plots.

```
> x <- seq(-pi,pi,len=50)
> y <- x
```

x is a vector of 50 equally spaced values in $-\pi \leq x \leq \pi$. y is the same.

```
> f <- outer(x,y,
 function(x,y)
 cos(y)/(1+x2̂))
```

After this command, f is a square matrix with rows and columns indexed by x and y respectively, of values of the function $\cos(y)/(1+x^2)$.

```
> par(pty="s")
```

region to *square*.

```
> contour(x,y,f)
> contour(x,y,f,
 nint=15,add=T)
```

Make a contour map of f and add in more lines for more detail.

```
> fa <- (f-t(f))/2
```

`fa` is the *asymmetric* part of f. `t()` is transpose).

```
> contour(x,y,fa,
 nint=15)
```

Make a contour and

```
> persp(x,y,f)
> persp(x,y,fa)
> image(x,y,f)
```

Make some pretty perspective and high density image plots, of which, you can get hard copies if you wish.

```
> image(x,y,fa)
> objects()
> rm(x,y,f,fa)
```

and clean up before moving on.

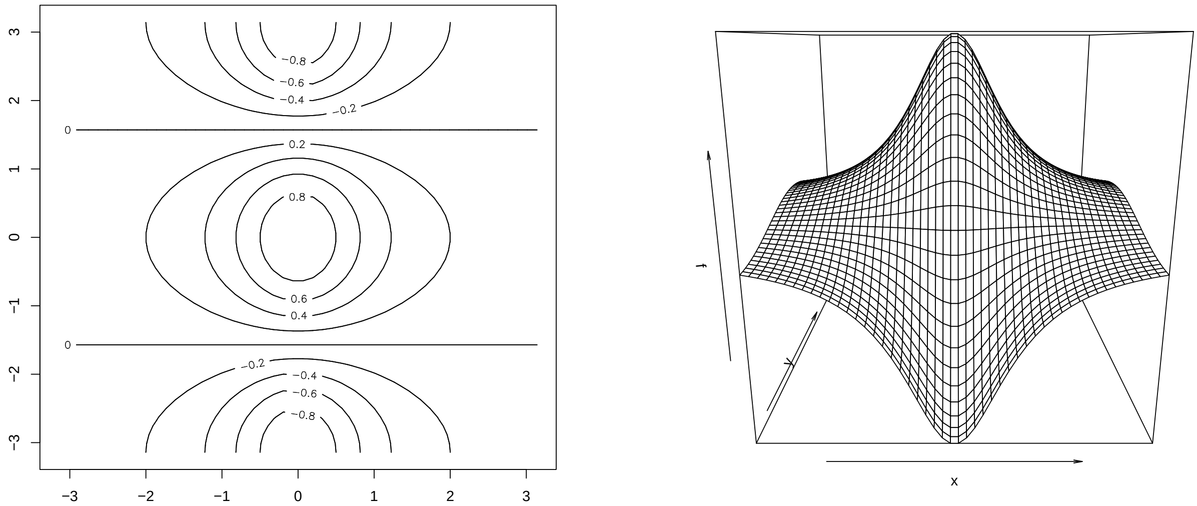


Figure 119: Contour and perspective mesh plot of f

Complex Arithmetic

```
> th <- seq(-pi,pi,len=200)
> z <- exp(1i*th)
```

R can do complex arithmetic also. `1i` is used for the complex number i.

```
> par(pty="s")
> plot(z,type="l")
```

Plotting complex arguments means plot imaginary versus real parts. This should be a circle.

```
> w <- rnorm(1000)+
 rnorm(1000)*1i
> w <- ifelse(abs(w)>1,
 1/w,w)
```

Suppose we want to sample points within the unit disc. One method would be to take complex numbers with random standard normal real and imaginary parts and map any points outside the disc onto their reciprocal.

```
> plot(w,xlim=c(-1,1),
 ylim=c(-1,1),pch=4,
 xlab="",ylab="",
 axes=F)
> lines(z)
```

All points are inside the unit disc but the distribution is not uniform.

```
> w <- sqrt(runif(1000))*
 exp(2*pi*runif(1000)*li)
```

The second method uses the uniform distribution. The points should now look more evenly spaced over the disc.

```
> plot(w,xlim=c(-1,1),
 ylim=c(-1,1),pch=1,
 xlab="",ylab="",
 axes=F)
> lines(z)
```

You are not expected to know why this works by the way,but working out why it does so is a little problem for the mathematically adept.

```
> rm(th,w,z)
```

Clean up again

```
> q()
```

Quit the R session

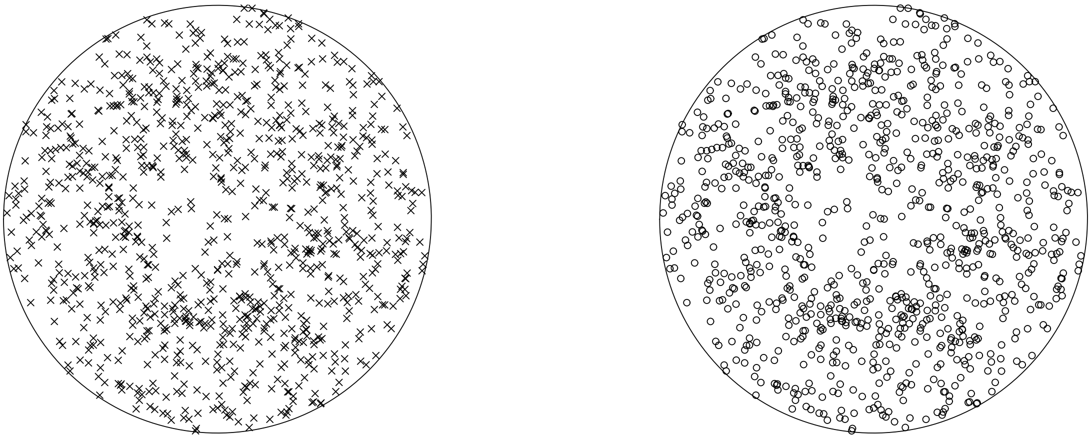


Figure 120: Two samplings of the unit disc

Lab 2: Understanding R Objects

In this exercise you will be expected to do some of the calculations for yourself. Use the help facilities as much as you need. After the pace of the introductory session this one may seem a little slow. All of the datasets are located in the MASS library so do not forget to attach it. Incidentally, if you want to locate a dataset use the `find` command.

Animal Brain and Body Sizes

Read the help file for the `Animals` data frame, which gives average brain and body weights for certain species of land animals. Attach the data frame at level 2 of the search list and work through the following.

Try using the **Tinn-R** editor to write and run scripts produced in this session.

1. Make a *dotchart* of `log(body)` sizes.
   ```
   > dotchart(log(body),row.names(Animals),xlab="log(body)")
   ```
   The second argument to `dotchart()` gives the names to appear on the left. With data frames, the *row names* become the *names* of each variable.

   Notice that the information from the chart is somewhat unclear with the animals in no special order. One way to improve this is to arrange them in sorted order.

   ```
   > s <- sort.list(body)
   > dotchart(log(body[s]),row.names(Animals[s,]),xlab="log(body)")
   ```

   The dotchart should now be much more comprehensible.

2. Produce a similar dotchart of log(brain) sizes arranging the animals in sorted order by *body* size. What interesting features do you notice, if any?

Detach the data frame and cleanup temporaries before proceeding to the next question.

H O Holck's Cat Data

Examine the data frame `cats` through its help file.

1. Attach the data frame at position 2 and look at the levels of the factor `Sex`.

   ```
   > sex <- levels(Sex)
   > sex
   ```

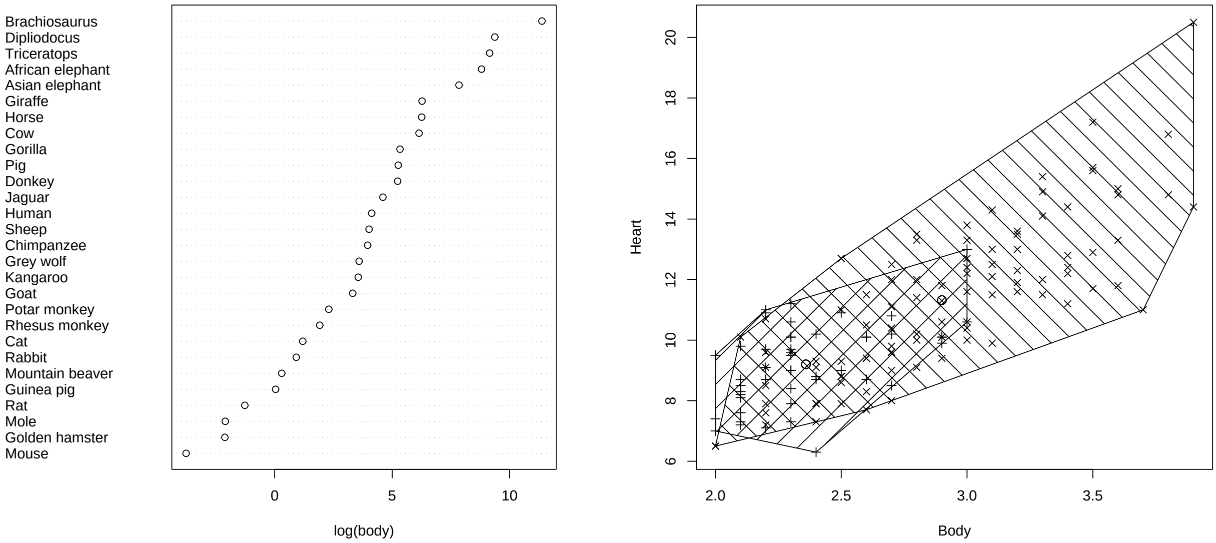


Figure 121: Two anatomical data displays

Note: lower case here

2. Plot heart weight against body weight, marking in the two samples with a different character. One way to do this is using a simple loop:

```
> plot(Bwt,Hst,type="n",xlab="Body",ylab="Heart") # axes only
> for(i in 1:2)
      points(Bwt[Sex==sex[i]],Hwt[Sex==sex[i]],pch=2+i)
```

(Loops will be discussed later in the course.)

Another way to do something similar would be to use `text()` in place of the last two commands:

```
> text(Bwt,Hwt,c("*","+")[Sex])
```

but this way does not allow the more precise plotting symbols to be used. Try both.

3. Find the means of each sample and print them. A direct way to do this is as follows:

```
> BwtF <- mean(Bwt[Sex=="F"]); ...
```

With only two groups this is probably the simplest. With more than two groups though, it becomes exceedingly tedious. A better way is to use the `tapply()` function, (whose semantics you should look up).

```
> Bmeans <- tapply(Bwt,Sex,mean)
> Hmeans <- tapply(Hwt,Sex,mean)
```

4. Look up the `symbols()` command and use it to add $\frac{1}{10}$inch circles to your plot at the two sample mean points.

   ```
   > symbols(Bmeans,Hmeans,circles=c(1,1),inches=0.05,add=T)
   ```

5. **[Optional]** Try to put polygons on the diagram showing the convex hull of each sex.

   Here is a fairly slick way to do it, but it could be done in a loop as well:

   ```
   > hF <- chull(xF <- as.matrix(cats[Sex=="F",-1]))
   > hM <- chull(xM <- as.matrix(cats[Sex=="M",-1]))
   > polygon(xF[hF,],dens=5,angle=45)
   > polygon(xM[hM,],dens=5,angle=135)
   ```

Combining Two Data Frames with Some Common Rows

The data frame `mammals` contains brain and body weight data for 62 species of land mammals, including some of the entries also contained in the `Animals` data frame. Our problem here is to put the two data frames together, removing duplicate entries.

Since the variable names of the two data frames are the same, and the row names are identical where they refer to the same animal, we can discover where the duplicates are by looking at the two `row.names` vectors together. There is a useful function for checking whether an entry in a vector has already occurred earlier:

```
> nam <- c(row.names(Animals),row.names(mammals))
> dup <- duplicated(nam)
```

We now put the data frames together and remove duplicate entries. The function `rbind()` may be used to *bind* data frames together *by rows*, that is, stack them on top of each other, aligning variables with the same name.

```
> AnimalsALL <- rbind(Animals,mammals)[!dup,]
> rm(dup,nam)
```

Note the empty second subscript position. This subsets elements by row.

The Tuggeranong House Data

1. Attach the Tuggeranong house price data frame and look at the variables

   N.B. The Tuggeranong house data is not part of the MASS dataset. The data is provided as a comma delimited text file and can be read in using the `read.csv` function which will be discussed in a later session.

   ```
   > house <- read.csv("houses.csv")
   > attach(house)
   > house
   ```

2. Make a factor from the `Rooms` variable directly (3 classes) and from the `Age` variable by cutting the range into three equal parts. Rename the `Cent.Heat` factor for convenience.

   ```
   > rms <- factor(Rooms)
   > age <- factor(cut(Age,3))
   > cht <- CentralHeating
   ```

   Note, instead of explicitly creating these variables and storing them temporarily in the workspace, we could have used the `transform` function to create a new data frame with extra columns corresponding to these objects.

   ```
   > houseT <- transform(house,rms=factor(Rooms),
     age=factor(cut(Age,3)),cht=CentralHeating)
   ```

3. Find two-way frequency tables for each pair of factors. Although the table is of only a small number of frequencies, does their appear to be any two way association?

4. Use Pearson χ^2 statistic to test for association in any of the three tables you choose.

   [The statistic is defined as follows:

   $$\chi^2 = \sum_i \sum_j \frac{(F_{ij} - E_{ij})^2}{E_{ij}}$$

   If the two classifying factors are A and B, then F_{ij} is the frequency of class (A_i,B_j) and E_{ij} is the estimated expected frequency. The F_{ij}'s and E_{ij}'s may be calculated simply as follows:

   ```
   > Fij <- table(A,B)
   > Eij <- outer(table(A),table(B))/sum(Fij)
   ```

Look up the `outer` function in the help documents.]

5. Using the `help` facility look up the functions `chisq.test` and `fisher.test`.

   Use the first to check your answer from the previous question and the latter to check how well the χ^2 test approximates the Fisher exact test in this instance.

6. Detach the house data and clean up.

The Anorexia Data

The `anorexia` data frame gives information on three samples of young female anorexia patients undergoing treatment. There are three groups, namely *Control*, *Cognitive Behavioural treatment* and *Family treatment*, as given by the factor `Treat`. For each patient the pre-study weight and post-study weight, in pounds, are given as variables `Prewt` and `Postwt` respectively.

1. Select the control patients and test the hypothesis of no change in weight over the study period. Compare the results of a paired t-test and a two-sample t-test:

```
> attach(anorexia[anorexia$Treat=="Cont",])
> t.test(Prewt,Postwt,paired=T)
> t.test(Prewt,Postwt)
> detach()
```

2. Use a coplot to compare pre-weight with post-weight for the three treatment groups. Inside the panels, plot the points as well as the least squares line.

```
> attach(anorexia)
> panel.fn <- function(x,y,...){
      points(x,y,pch=3)
      abline(lm(y~x),col="blue")
  }
> coplot(Postwt ~ Prewt | Treat,panel=panel.fn)
```

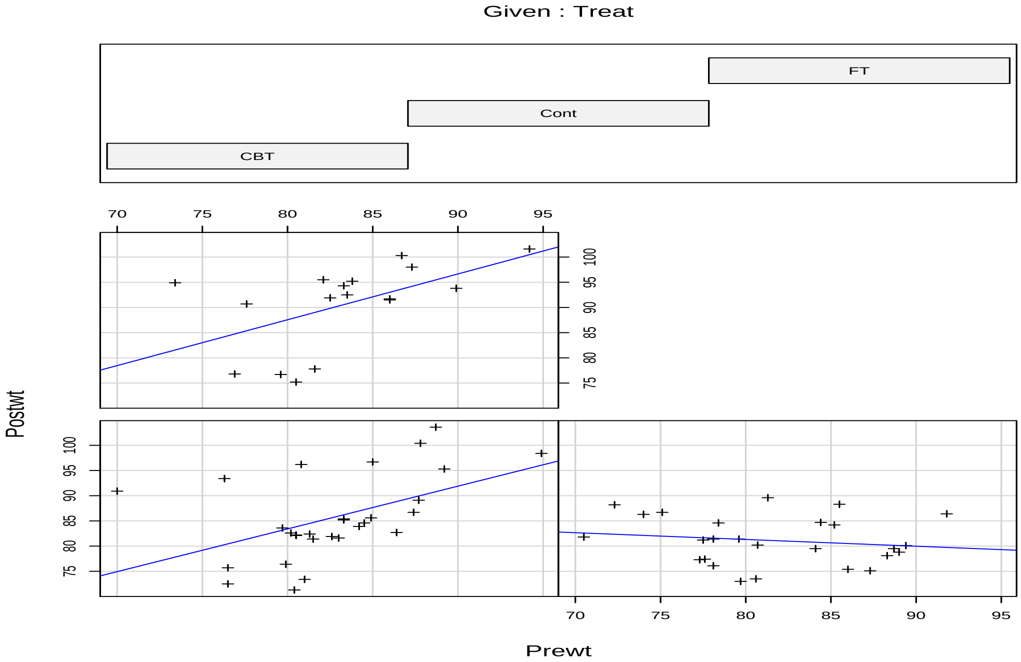


Figure 122: Scatterplot of anorexia data conditioned by treatment. Each panel is overlayed with a least squares line.

Lab 3: Elementary Graphics

Scatterplots and Related Issues

1. Attach the `AnimalsALL` data frame that you made at the end of the last laboratory session. (The `Animals` data frame will do if you do not get that far). Also open a small graphics window.

   ```
   > attach(AnimalsALL}
   > windows()
   ```

2. Plot `brain` against `body`, firstly using ordinary axis scales and then using logarithmic scales in both directions, and compare the results. Put in explicit axis labels:

   ```
   > plot(body,brain, xlab="Average body weight (kg)",
          ylab="Average brain weight (gm)")
   > plot(body,brain, xlab="Average body weight (kg)",
          ylab="Average brain weight (gm)",log="xy")
   ```

3. Using `identify` discover which animals lie above or below the notional regression line of the log-transformed values.

   ```
   > identify(body,brain,row.names(AnimalsALL))
   ```

4. Now plot both variables log-transformed and include on the plot both the ordinary least squares and the robust *least trimmed squares* regression lines, using a different line type for the second.

   ```
   > plot(log(body),log(brain),pch=4)
   > abline(lsfit(log(body),log(brain)),lty=1)
   > abline(ltsreg(log(body),log(brain)),lty=2)
   ```

   Why are the two so different do you think?

5. Next we will insert a legend to identify ordinary least squares (OLS) and least trimmed squares (LTS) regression lines by line type. The legend box should be placed in an otherwise empty part of the plot and a convenient way of doing this interactively is to use the `locator` function in conjunction with the mouse.

   ```
   > legend(locator(1),legend=c("OLS line","LTS line"),lty=1:2)
   ```

At this point you should place the mouse in the plot window, locate the point where you want the top left hand corner of the legend box to appear and click with the left button.

Student Survey Data

The data frame `survey` contains the results of a survey of 237 first-year Statistics students at Adelaide University.

1. Attach the data frame for later use, and have a look at the help file for this data frame. For a graphical summary of all the variables, type

   ```
   > plot(survey}
   ```

   Note that this produces a dotchart for factor variables and a normal scores plot for the numeric variables.

2. One component of this data frame, `Exer` is a factor object containing the responses to a question asking how often the students exercised. Produce a barplot of these responses using `plot()`.

   The `table()` function when applied to a factor object returns the frequencies of each level of the factor. Use this to create a pie chart of the responses with the `pie()` function. Do you like this better than the bar plot? Which is more informative? Which gives a better picture of exercise habits of students? The `pie()` function takes an argument `names=` which can be used to put labels on each pie slice. Redraw the pie chart with labels.

   (**Hint:** use the `levels()` function to generate the labels.)

   You could also add a legend to identify the slices using the `locator()` function to position it. The function call to do this will be something like

   ```
   > legend(locator(1), legend = ..., fill=1:3)
   ```

3. You might like to try the same things with the `Smoke` variable, which records responses to the question *How often do you smoke?* Note that `table()` and `levels()` ignore missing values. If you wish to include non-respondents in your chart use `summary()` to generate the values and `names()` on the summary object to generate the labels.

4. Is there a relationship between pulse rate and exercising? Create boxplots of `Pulse` for each level of `Exer` with the command

   ```
   > plot(Exer,Pulse}
   ```

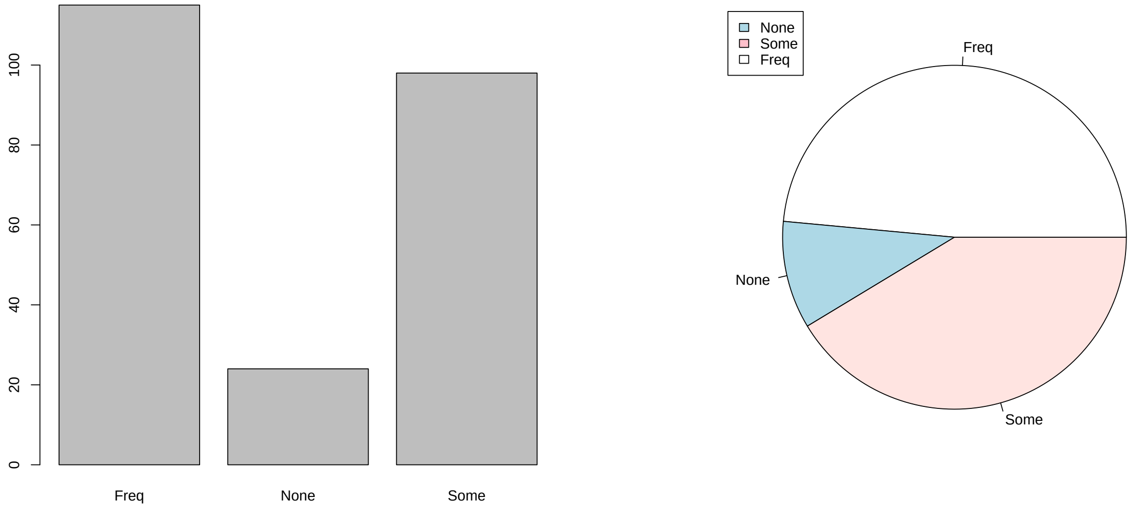


Figure 123: Two representations of the `Exer` frequencies

Does there appear to be a relationship? You may wish to test this formally with the `aov` command. What other relationships can you find in these data?

The Swiss Banknote Data

In an effort to detect counterfeits, the dimensions of 100 known fake Swiss banknotes and 100 supposedly legal notes were measured. The length of the note and its diagonal were measured along with the length of each side.

This dataset is located in the `alr3` library. To access a copy, you must first attach the library and then load the dataset.

```
> require(alr3)
> data(banknote)
```

1. Attach the data frame and plot the variables against each other in a scatterplot matrix:

   ```
   > attach(banknote)
   > pairs(banknote)
   ```

   This is not very informative since legitimate and fake notes are not separated. To separate them we need to do something more specific within each panel with a panel function.

```
> pairs(banknote[,-1],panel=
    function(x,y,fake){
      xy <- cbind(x,y)
      points(xy[fake==0,],pch=15)
      points(xy[fake==1,],pch=0)
    }, fake=Y)
```

Note that if the panel function has more arguments than x and y, these may be supplied as named arguments to `pairs`, usually at the end.

Which two variables seem best to discriminate the counterfeit and legal notes?

2. Generate a co-plot of these two variables, given `Y`.

```
> coplot(<var1> ~ <var2> | Y,data=banknote)
```

Does your co-plot suggest that some of the *legal* notes may in fact be undetected.

Lab 4: Manipulating Data

Birth Dates

Repeat the birth date example discussed in the lecture for your own birth date.

The Cloud Data

In this exercise we do some polynomial regression calculations from scratch using raw matrix manipulation and verify the results using R tools. (A side effect is to reinforce the value of learning to use those tools effectively!)

1. The cloud data is not part of the MASS library but is available as text file and can be read in using `read.table`. After reading in the data, look at the data.

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

2. Attach the data and for ease of typing make two new variables, x and y equal to the I_s% and cloud point vectors respectively. Let n be the number of observations, for future reference.

```
> attach(cloud)
> x <- Ispc
> y <- Cloudpt
> n <- length(x)
```

3. Mean correct the x vector and construct a model matrix for a cubic regression of y on x, (where the powers are of the mean corrected x-vector). Include the constant term, of course.

```
> xmc <- x-mean(x)
> X <- cbind(1,xmc,xmc^2,xmc^3)
> X
```

4. The regression coefficients are $b = (X'X)^{-1}X'y$. First calculate the matrices. Here are two alternative ways of calculating $X'X$.

```
> XX <- t(X) %*% X    # the literalist way
> XX <- crossprod(X)  # more efficient alternative
```

The $X'y$ matrix is, similarly,

```
> Xy <- crossprod(X,y)  # 2 argument form
```

Now calculate the regression coefficients in `b` using `solve()`.

5. Calculate
   - the fitted values, $f = Xb$,
   - the residuals, $r = y - f$,
   - the residual mean square $s^2 = (\sum_i r_i^2)/(n-4)$ and
   - the variance matrix of the regression coefficients, $s^2(X'X)^{-1}$, which you call, say `Vb`.
6. The square roots of the diagonal entries of `Vb` give the standard errors, and the ratio of the regression coefficients to these give the t-statistics.

   ```
   > se <- sqrt(diag(Vb))
   > ts <- b/se
   ```

7. Finally, arrange the results in a *comprehensible* form and print them out:

   ```
   > R <- cbind(b,se,ts,1-pt(ts,n-3))
   > rnames <- c("Constant","Linear","Quadratic","Cubic")
   > cnames <- c("Coef","Std. Err.","t-stat","Tail prob.")
   > dimnames(R) <- list(rnames,cnames)
   > R
   ```

8. Check the calculations by the standard tools, for example

   ```
   > fm <- lm(y~1+xmc+I(xmc^2)+I(xmc^3))
   > summary(fm)  # check the table
   > b-coef(fm)   # check for numerical discrepancies
   ```

9. Detach data frame and clean up

The Longley Data

This is a famous data set used in a benchmark study of least squares software accuracy. It comes as a system dataset and `?longley` will give more information. Our purpose is to study this data, possibly seeing why it is so commonly used to check out least squares calculations. The data set is too small to study effectively by graphical methods, so we will stick to calculations.

1. Find the means and variances of each variable. Do this any way you like, but compare your answer with the results of

   ```
   > mns <- apply(longley,2,mean)
   > mns
   > vrs <- apply(longley,2,var)
   > vrs
   ```

2. Compare the results of the two commands

   ```
   > apply(longley,2,var)
   > var(longley)
   ```

   What is going on?

3. Find the correlation matrix between all variables. (`cor()`)

4. Find the singular value decomposition of the mean corrected X-matrix and print out the singular values. (`svd()`)

   The simplest way to calculate and subtract the sample mean from each column of a matrix is to use the scale function:

   ```
   > Xc <- scale(longley[,-7],scale=F)
   ```

   What happens if we leave off the `scale=F` argument? Investigate. Note that `scale` works either with matrices or data frames.

   Another way is to use the model fitting functions:

   ```
   > Xc <- resid(lm(longley.x~1))
   ```

   A different way, that keeps the mean vector as well, is as follows

   ```
   > xb <- apply(longley.x,2,mean) # calculate the 6 means
   > dm <- dim(longley.x)
   > Xc <- longley.x - matrix(xb,dm[1],dm[2],byrow=T) # correct
   ```

Convince yourself why all three should work and verify that they lead to the same result.

The ratio of the largest to the smallest singular value is called the *condition number* of the matrix. Calculate it and print it out.

Lab 5: Classical Linear Models

H O Holck's cats data, revisited

As noted earlier, the data frame `cats` gives the sex heart and body weights of 144 hapless domestic cats used in animal experiments. We will explore in a simple way the relationship between heart and body weight and check for differences between sexes.

1. Use a coplot to show the relationship of heart weight to body weight separately for the two sexes. Include the least squares line as a guide.

2. Fit separate regressions of heart weight on body weight for each sex. Also fit parallel regression lines and a single regression line for the two sexes. Test the models and comment.

```
> cats.m0 <- aov(Hwt~Sex/Bwt,data=cats) # separate
> cats.m1 <- aov(Hwt~Sex+Bwt,data=cats) # parallel
> cats.m2 <- aov(Hwt~Bwt,data=cats)     # identical
> anova(cats.m2,cats.m1,cats.m0)
```

3. Consider an alternative model that postulates that heart weight is proportional to body weight. This leads fairly naturally to a model of the form

$$log(\text{Hwt}) = \beta_0 + \beta_1 \log \text{Bwt} + \epsilon$$

with a natural null hypothesis of $\beta_1 = 1$. Explore various models with the log transformed data and comment.

Cars Dataset

Read the details of the data frame `Cars93`.

Build a regression equation for predicting the miles per gallon in city travel using the other variables (except `MPG.highway`, of course).

Check your final suggested equation with suitable diagnostic plots. If there appears to be variance heterogeneity (which is suspected) repeat the construction using either a log-transformed or reciprocal response variable.

Notice that the reciprocal transformation leads to an easily appreciated quantity, the *gallons per mile* used in city travel.

The Painters Data

Read the description of the `painters` data either from the notes or using `?painters` within R.

1. To see how the schools differ on each of the characteristics, plot the data frame using the method appropriate to a design:

   ```
   > plot.design(painters
   ```

   This will give a series of four plots that should be understandable.

2. Perform a single classification analysis of variance on each variable on school.

3. **Optional** Find the matrix of residuals for the four variables and hence the *within school* variance and correlation matrices.

Lab 6: Non-Linear Regression

The Stormer Viscometer Data

1. Work through the material covered in the lecture notes for fitting a non-linear regression model to the stormer dataset.
2. As a graphical challenge, display the fitted regression surface and the fitted points on a perspective diagram. Try also a contour diagram and an image plot.

The Steam Data

The steam data (data frame `steam`, see the online help documents) has (at least) two sensible nonlinear regression models, depending on whether the error is taken as additive of multiplicative on the original scale. For an additive error model it becomes

$$P = \alpha \exp\{\frac{\beta t}{\gamma + t}\} + E$$

and for a multiplicative error model:

$$\log P = \log \alpha + \{\frac{\beta t}{\gamma + t}\} + E^*$$

1. Devise a suitably ingenious method for finding initial values for the parameters. For example, plot `log(Press)` against `Temp/(g+Temp)` for some value of `g`, say 1. Then adjust `g` by either doubling or halving until the plot seems to be an approximately straight line.

   (This leads to an initial value somewhere near $\hat{\gamma}_0 = 200$.)
2. Fit both models and compare
   (a) The parameter estimates, and
   (b) The fitted values and simple pointwise confidence regions for them

   on the original scale. Comment

Lab 7& 8: Generalized Linear Models and GAMs

Snail Mortality Data

The snail mortality dataset consists of and experiment conducted on a group of 20 snails, which were held for periods of 1, 2, 3 or 4 weeks in carefully controlled conditions of temperature and relative humidity. Two species of snail were examined: A and B.

1. Read the help file on the snails data and attach the data frame.
2. Coerce the species variable into a factor and create a suitable response variable and call it Y.

   ```
   > snails$Species <- as.factor(Species)
   > snails$Y <- cbind(Deaths,20-Deaths)
   ```

   Optional: Try using the `transform` function to create new variables for `Species` and `Y`.
3. Fit separate linear models for the two species.

   ```
   > fm <- glm(Y~Species/(Temp+Rel.Hum+Exposure),
      family=binomial, data=snails, trace=T)
   ```
4. Now determine whether the response surfaces for the two species should be considered parallel.

   ```
   > fm0 <- update(fm,
       ~ Species+Temp+Rel.Hum+Exposure)
   > anova(fm0,fm,test="Chisq")
   ```
5. Produce some diagnostic plots and assess the fit of the model.

The Janka Data

1. Fit a locally weighted regression to the Janka data and compare it with the fitted curve from the model fitted when last you used the Janka data. (Remember, the janka data is located in a `.csv` file.)

   ```
   > fma <- gam(Hardness ~ lo(Density),data=janka)
   ```

2. Experiment with the `span=` and `degree=` parameters of the `lo()` function to see how (if anything much) it affects the characteristics of the fitted curve in this case.
3. Compare this fitted curve with a *local regression model* obtained by

```
> fmb <- loess(Hardness ~ Density,data=janka)
```

The Birth Weight Data

Look at the `birthwt` data frame. Build up a model (as done in the lectures) for estimating the probability of low birth weight in terms of the other predictions (excluding actual birth weight, of course).

See how the picture changes if *low* birth weight is defined differently, that is at some other truncation level. At the moment, `low` is an indicator of `birthwt` for values less than 2.5kg.

Lab 9: Advanced Graphics

Graphics Examples

Work through the material covered in the lecture notes. Make sure you use the Tinn-R editor for submitting scripts. Define your own colour palettes and view the results.

The Akima Data

1. The `akima` dataset is located in the `akima` library. To access the data

   ```
   > require(akima)
   > data(akima)
   ```

   Read about the akima data which we will be using for 3D graphs.
2. Interpolate the data using the `interp` function. Get help on the function so you understand its usage.
3. Produce a contour, image and perspective mesh plot of the data. Investigate setting up your own colour scheme.

Heights of New York Choral Society Singers

1. The `singer` dataset is located in the `lattice` package. Load this package and read the help supplied for this dataset.
2. Look up the `histogram` function in the help section and see what parameters can be specified. Produce a histogram of heights broken down by voice type using the `histogram` lattice function.
3. Add to the previous plot a density
4. Investigate the *Normality* of the data for each voice type by producing a Normal scores plot. Use the `qqmath` function to create this plot.

Lab 10: Mixed Effects Models

The Rail Dataset

The `Rail` dataset is part of the `nlme` library. It contains information on travel time for a certain type of wave resulting from longitudinal stress of rails on tracks. The data consists of a factor, `Rail` that gives the number of the rail that the measurement was taken and the travel time, `travel` for the head-waves in the rail.

1. Attach the `nlme` library and read the help provided for the Rail dataset.
2. Fit a linear model to the `Rail` data of the form

   $$y_{ij} = \beta + \epsilon_{ij}$$

   and look at the results. This is a simple mean model.
3. Produce some diagnostic plots. A useful plot to produce is a boxplot of residuals by rail number and comment on the plot. Is there any variability between rails?
4. Now fit a fixed effects model of the form

   $$y_{ij} = \beta_i + \epsilon_{ij}$$

   This model fits a separate fixed effects term to each rail. (Do not fit the intercept). Examine the results and pay particular attention to the residual standard error.
5. Produce a residual plot for this new model and compare the results.
6. Although this model accounts for the effects due to rails, it models the specific sample of rails and not the population of rails. This is where a random effects model may prove useful.

   A random effects model of the form

   $$y_{ij} = \bar{\beta} + (\beta_i - \bar{\beta}) + \epsilon_{ij}$$

   treats the rail effects as random variations around the population mean. Try fitting this model using `lme` and examine the results:

   ```
   > fitRail.lme <- lme(travel~1,data=Rail,random=~1|Rail)
   > summary(fitRail.lme)
   ```

What does the fitted model tell us? Produce plots of the residuals from the fitted model using the `plot` function. Is this model any better? More appropriate?

The Pixel Dataset

The `Pixel` dataset is another dataset in the `nlme` library that contains information on the pixel intensities of CT scans of dogs over time. Read the help section for more information about this dataset.

1. Using the functions in the lattice (trellis) library, produce an exploratory plot that shows the panel intensity versus the time (in days) of the `Pixel` data, for each dog and for each side.
2. It seems plausible to fit some random terms in the model to capture the variability between dogs and sides within dogs (there is some variation apparent between sides but not much).

   Fit a mixed effects model using `lme` with a quadratic term for `Day` in the fixed effects part of the model and random terms for `Dog` and `Side` within `Dog`.

   ```
   > pixel.lme <- lme(pixel~day + I(day^2),data=Pixel,
      random=list(Dog=~day,Side=~1))
   > intervals(pixel.lme)
   ```

   Comment on the results.
3. Produce a plot of the predicted values for each dog/side combination.

   ```
   > plot(augPred(pixel.lme))
   ```

   Comment on the plot.

Lab 11: Programming

Elementary Programming Examples

The trick with programming is to start with the simplest prototype version of the function, test is as you go, and gradually incorporate the more general and flexible features you need. A second tip is to write simple functions to do single, special tasks reliably, and gradually build the final function from these building blocks.

These two principles are sometimes called *bottom up design* and *modular design* respectively.

1. Your task is to write a function `T.test` that will perform a two sample t-test, similar to the job done by the system function `t.test`.

   (a) The formula for the statistic is

   $$t = \frac{\bar{y}_1 - \bar{y}_2}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

   where n_1 and n_2 are the two samples sizes, $\bar{y}_1$ and $\bar{y}_2$ are the two sample means, and s is the square root of the pooled estimate of variance, s^2, given by

   $$s^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{(n_1 - 1) + (n_2 - 1)}$$

   where s_1^2 and s_2^2 are the two sample variances. Suppose the function initially has the header

   ```
   T.test <- function(y1,y2){ ...
   }
   ```

   i. First find n_1 and n_2.

   ii. Next calculate the numerator of t

   iii. Then calculate the pooled variance and hence s

   iv. Finally, calculate the value of t and return that as the value of the function

   (b) Suppose the test is two-tailed. The significance probability is then defined as $1 - 2\Pr(|T| > t)$ where $T \sim t_{n_1+n_2-2}$. In R this is calculated as

   ```
   2*(1-pt(abs(t),n1+n2-2))
   ```

Amend your function so that it calculates this quantity and returns as its value a list with two components: the t-statistic and the significance probability.

(c) Include a preliminary test of equality of variances. The significance probability for this test may be found either by the calculation (which is easy enough) or as `var.test(y1,y2)$p.value`. Perform the test early in the problem and issue a warning if the significance probability is less than 0.05.

(d) Sometimes it may be useful to have a graphical representation of the samples as well as the numerical result. Allow the user optionally to request boxplots with an additional logical parameter `boxplots`:

```
T.test <- function(y1,y2,boxplots=F){
   ...
   if(boxplots) {
      f <- factor(rep(paste(‘‘Sample’’,1:2),c(n1,n2(((
      y <- c(y1,y2)
      plot(f,y)
   }
   ...
}
```

(e) Finally, it might be useful to allow the user to specify the two samples as a two-level factor and vector rather than as two separate sample vectors. Modify the function to allow this alternative call sequence:

```
T.test <- function(y1,y2,boxplots=F){
   if(is.factor(y1)) { # y1=factor, y2=all y’s
      l <- levels(y1)
      tm <- y2[y1==l[1]]
      y2 <- y2[y1==l[1]]
      y1 <- tm
    }
    ... # proceed as before
}
```

(f) Check for all likely error conditions and issue either warnings or stops to aid the user.

(g) As a last check, test the program on the `Cars93` data, comparing the Price levels for the two car origins (omitting large cars).

2. The Box and Cox power family of transformations was covered in one of the lectures. It is defined as

$$f(y, \lambda) = \begin{cases} (y^\lambda - 1)/\lambda & \text{if } \lambda \neq 0 \\ \log y & \text{if } \lambda = 0 \end{cases}$$

Write a function to calculate such a transformation. The function should be of two arguments, `y`: a vector of data values and `lambda`: a scalar specifying the exponent. Stop with an error message if any `y` value is negative or zero.

(a) For the Janka hardness data, consider a second degree polynomial regression of `y=Hardness` on `x=Density`. Notice how with untransformed y the residuals show a distinct pattern of increasing variance with the mean. Consider alternative regressions with a Box and Cox power transformed y using powers $\lambda = 1, \frac{1}{2}, \frac{1}{4}$ and 0. Which model would you prefer if it were important to operate in a y scale where the variance were uniform?

(b) With the `Cars93` data we noticed that the reciprocal of `MPG.city` was in some sense simpler to use as a response than the variable in the original scale. This is effectively a Box and Cox power transform with $\lambda = -1$. Consider other values for λ, say $\lambda = 0, -\frac{1}{2}, -1$ and $-\frac{3}{2}$ and recommend a value accordingly. Use the regression variable `Weight` and the factors `Type` and `Cylinders` as the predictors, if necessary.

Round Robin Tournaments

Write a function of one integer variable to generate a round robin tournament draw. One possible algorithm is as follows:

1. Begin with an arbitrary pairing of teams. If there is an odd number of teams increase them by a *team 0* representing a bye.
2. Write the original pairings as two columns and cycle the teams in the way shown in the diagram: If there are n teams the tournament draw is complete after $n-1$ cycles.
3. Return the result as an $n/2 \times 2 \times (n - 1$ array of integers.
4. Give the array a class, say `robin` and write a print method that will print the result in a convenient way.
5. The functions `round.robin()` and `print.round.robin` provide a solution

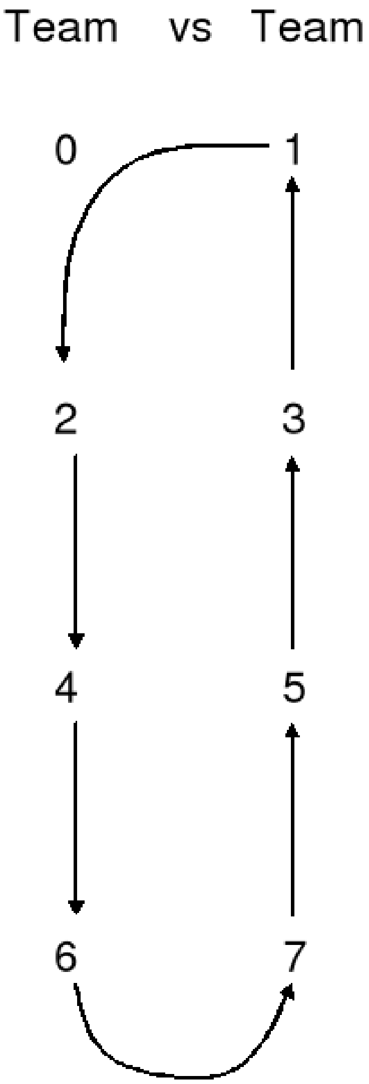


Figure 124: Cycle method for round robin tournament draws

Lab 12: Neural Networks

The Rock Data

1. The rock data is part of the `datasets` package. Call up help on this dataset.
2. Attach the rock dataset to the current session and divide the area and perimeter variables by 10000. Form a new dataset that has the transformed area, perimeter and the shape variable included. Call this dataset `rock.x`.
3. Fit a neural network using the `nnet` function. Remember to attach the neural network library.

```
> rock.nn <- nnet(rock.x,log(perm), size=3, decay=1e-3,
    linout=T,skip=T,maxit=100)
> summary(rock.nn)
```

4. Investigate the fit of the model

The Crab Data

We now consider fitting a neural network to a classification problem: classifying the sex of crabs. Get up the help on the `crabs` dataset and read about it.

1. Let's start by fitting a generalized linear model to the data. We first need to transform the explanatory variables `FL`, `RW`, `CL`, `CW`, and `BD` on to the log scale.

```
> dcrabs <- log(crabs[,4:7])
> dcrabs <- cbind(dcrabs,sex=crabs$sex)
```

2. Now fit a logistic regression model to the data and look at the results.
3. Produce some predictions using the `predict` function with `type="response"`.
4. **Optional**: Try to perform a cross-validation study to assess the performance of the model. Hint: Try using the `subset` argument in the glm function.
5. Now fit a neural network to the crabs dataset and compare the results.

```
> cr.nn <- nnet(dcrabs,crabs$sex=="M",size=2,decay=1e-3,
     skip=T,entropy=T,maxit=500)
```

6. **Optional**: Repeat the cross-validation study but base it around the neural network. How do the two models compare?

Lab 13& 14: Classification and Regression Trees

The Crab Data Revisited

In the previous laboratory session, the crab data was used to fit a logistic regression using the `glm` function and a neural network.

Try fitting a classification tree to the crab data and compare with previous models.

The Cuckoo Data

The `cuckoo` data shows the lengths and breadths of various cuckoo eggs of the European cuckoo, *Cuculus canoris* classified by the species of bird in the nest of which the egg was found.

1. The data is located in an Excel spreadsheet. Attach the `RODBC` library and read in the dataset.
2. Reduce the data frame to a smaller one got by excluding those eggs for which the host species was unknown or for which the numbers of eggs found were less than 5.
3. Working with the reduced data frame, develop a classification tree using the `rpart` package for predicting the host species from the egg dimensions, to the extent to which this seems to be possible.
4. Try and amend the code provided in the lecture for partitioning the tree for this data.
5. Use the tree model to predict the host species for the unnamed host eggs of the full data frame.

The Student Survey Data

The survey data comes from a student survey done to produce *live* data for the first year statistics class in the University of Adelaide, 1992. The variables recorded included `Sex`, `Pulse`, `Exer` (an exercise level rating), `Smoke`, `Height` and `Age`.

There are missing values in the data frame, so be sure to use appropriate actions to accommodate them.

1. Fit a naive linear model for `Pulse` in terms of the other variables listed above.

2. Construct a regression tree for `Pulse` in terms of the other variables listed. Compare the results with that of the linear model. Comment.

3. Prune the tree to what you consider to be a sensible level and compare the residuals with those of the linear model, for example, using histograms.

4. Cross-validation with this data often suggests using a tree with just one node! That is, accurate prediction of pulse rates from the other variables is most reliably done using the simple average. However, cross validation is itself a stochastic technique. Try it and see what is suggested.

5. **Optional**: Try bagging out on this tree and see what improvements are made with regards to prediction.

Bibliography

Anderson, E. (1935), 'The irises of the Gaspe Peninsula', *Bulletin of the American Iris Society* **59**, 2–5.

Becker, R. A., Chambers, J. M. & Wilks, A. R. (1988), *The New S language*, Wadsworth and Brooks.

Breiman, L., Friedman, J. H., Olshen, R. A. & Stone, C. J. (1984), *Classification and Regression Trees*, Wadsworth.

Chambers, J. M. & Hastie, T. J., eds (1992), *Statistical Models in S*, Chapman and Hall, New York.

Diggle, P. J. (1990), *Time Series: A Biostatistical Introduction*, Oxford.

Draper, N. R. & Smith, H., eds (1981), *Applied Regression Analysis, Second Edition*, Wiley, New York.

Fisher, R. A. (1936), 'The use of multiple measurements in taxonomic problems', *Annals of Eugenics* **7**, 179–188.

Hand, D. J., Daly, F., McConwa, K., Lunn, D. & Ostrowski, E., eds (1993), *A Handbook of Small Datasets*, Chapman and Hall.

Harrison, D. & Rubinfeld, D. L. (1978), 'Hedonic prices and the demand for clean air', *Journal of Environmental Economics and Management* **5**, 81–102.

Henderson & Velleman (1981), 'Building multiple regression models interactively', *Biometrics* **37**, 391–411.

Linder, A., Chakravarti, I. M. & Vuagnat, P. (1964), Fitting asymptotic regression curves with different asymptotes, *in* C. R. Rao, ed., 'Contributions to Statistics', Oxford, pp. 221–228.

Lock, R. H. (1993), '1993 new car data', *Journal of Statistics Education* **1**, 1–.

Madsen, M. (1976), 'Statistical analysis of multiple contingency tables: Two examples', *Scandinavian Journal of Statistics* **3**, 97–106.

Pinheiro, J. C. & Bates, D. M. (2000), *Mixed-Effects Models in S and S-PLUS*, Springer, New York.

Pitcher, C. R., Austin, M., Burridge, C. Y., Bustamante, R. H., Cheers, S. J., Ellis, N., Jones, P. N., Koutsoukos, A. G., Moeseneder, C. H., Smith, G. P., Venables, W. & Wassenberg, T. J. (2004), Recovery of seabed habitat from the impact of prawn trawling in the far northern section of the Great Barrier Reef Marine Park: CSIRO Marine Research final report to GBRMPA, Technical report, CSIRO Marine Research and CSIRO Mathematical and Information Sciences.

Prater, N. H. (1956), 'Estimate gasoline yields from crudes', *Petroleum Refiner* **35**, 236–238.

Quine, S. (1978), 'The analysis of unbalanced cross classifications (with discussion)', *Journal of the Royal Statistical Society, Series A* **141**, 195–223.

Signell, R. (2005), *Coastline Extractor*, U.S. Geological Survey, Geologic Division, Coastal and Marine Program, http://rimmer.ngdc.noaa.gov/mgg/coast/getcoast.html, Boulder Colorado. Hosted by the National Geophysical Data Center.

Therneau, T. & Atkinson, E. (1997), An introduction to recursive partitioning using the RPART routines, Technical report, Mayo Foundation, Rochester.

Venables, W. N. & Ripley, B. R. (2000), *S Programming*, Springer, New York.

Venables, W. N. & Ripley, B. R. (2002), *Modern Applied Statistics with S, 4th Edition*, Springer-Verlag, New York.

Williams, E. J., ed. (1959), *Regression Analysis*, Wiley.