ST419 2008/2009 Computational Statistics

Lecturer: Erik Baurdoux[1] B604
Assistant: Neil Bathia

Course Aims and Objective

This course teaches the fundamental computing skills required by practicing statisticians. We focus on analysis of data using a computer and simulation as a tool to improve understanding of statistical models. The software package R is taught throughout. On successful completion of the course you will be a competent R user with skills that adapt readily to other statistical software packages. A small subset of the things you will be able to do are listed below:

- generate detailed descriptive analyses of data sets (both numbers and graphics),
- fit statistical models and interpret the output from model fitting,
- design and perform simulation experiments to explore the properties of statistics,
- write your own R functions to simplify commonly performed tasks or to implement novel statistical methods.

Course Administration

Prerequisites

This is intended primarily for MSc Statistics, MSc Social Research Methods (Statistics) and MSc Operational Research students. You are required to have a good basic knowledge of statistics (to the level of ST203 Statistics for Management Sciences). The course makes extensive use of computers in teaching and assessment. Basic familiarity with machines running a Windows operating system is assumed.

[1]For this year's course, I have slightly modified the original course notes written by Dr. J. Penzer

Timetabling

You are required to attend four hours per week for this course.

Mondays 1400-1500	H102	Weeks 1-10	Lecture
Mondays 1500-1700	S169	Weeks 1-10	Computer Workshop
Tuesdays 1000-1100	H101	Weeks 1-9	Problem Class.

On Tuesday of week 10, we will be in H102 from 0900-1200 for students' presentations (see assessment below).

Assessment

This course is assessed by coursework (50%) and by a two hour examination during the summer term (50%). There are two pieces of coursework:

	Handed out	Due in	Marks
Group project	20/10/08	Written: 05/12/08	10%
		Presentation: 09/12/08	10%
Individual project	17/11/08	12/01/09	30%

Detailed instructions for each of the projects will be given. There will be two practice written tests.

Books

This course has detailed lecture notes. It should not be necessary to buy a book for this course.

Main texts (new R books are appearing all the time)

- Venable, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S, (Fourth Edition)*, Springer. (QA276.4 V44)
- Venables, W. N., Smith, D.M. and the R Core Development Team (2001) *An Introduction to R*, freely available from `http://cran.r-project.org/doc/manuals/R-intro.pdf`.

Course content

A rough timetable and outline of course contents are given below.

Week 1	**Data, Session Management and Simple Statistics** – R objects: vectors, data frames and model objects; assignment; descriptive analysis; managing objects; importing data; *t*-tests; simple regression; financial series.
Week 2	**More R Essentials and Graphics** – R objects: logical vectors, character sequences, lists; regular sequences; subset selection; descriptive plots: histogram, qq-plot, boxplot; interactive graphics; low level graphics.
Week 3	**Writing Functions in R** – R objects: functions, arrays, matrices; function syntax; flow control: conditioning, loops; R code from other sources; libraries. Group project allocated. Individual project instructions given.
Week 4	**Distributions and Simulation** – statistical concepts: distributions, moments, sample moments, properties of statistics; distributions in R: probability density, quantiles; pseudo-random number generation; interpretation of simulation results.
Week 5	Monday: Written test 1 and Group project work. Tuesday: Feedback on written test 1.
Week 6	**Linear Models I** – statistical concepts: multiple regression, model selection, diagnostics; model formulae; interpretation of linear model output.
Week 7	**Linear Models II** – statistical concepts: factors as explanatory variables, logistic regression; manipulating model objects. Individual project allocated.
Week 8	**Time Series Analysis** – statisical concepts: auto-correlation, ARMA models, GARCH models; R time series packages; graphics for exploratory analysis and diagnostics.
Week 9	Monday: Written test 2 and Individual project work. Tuesday: Feedback on written test 2.
Week 10	Monday: Revision. Tuesday: Student presentations (0900-1200, H102).

Teaching methods and materials

Lectures and problem classes

On Mondays we will give details of the application of R to new material. On Tuesdays we will go through the exercises from the previous day - you will be expected to have attempted to complete the exercises so that you can contribute to the class. Any common problems with the exercises or in the use of R will be dealt with during these sessions.

Computer practicals

The only way to learn a computer language is to actually use it. The computer practicals provide time when you can work on material from the lecture notes and exercises. Using a combination of the notes, the R help system and experimentation you should be able to work out most of the commands. The R commands in the text are there to illustrate certain points. You do not have to follow them rigidly. There is often more than one way to do something and you may be able to find a better solution than the one given. If you find something interesting or difficult, spend more time on it; if you get the idea straight away, move on. If you can't work something out by experimenting with it, ask me or Limin. **Please don't mindlessly type in commands from the notes without understanding what they do**.

Lecture notes

The course attempts to convey a large amount of information in a short space of time. Some of the material is of a technical nature and may not be covered explicitly in the lectures and classes. **You are expected to read the lecture notes thoroughly**. The syllabus is defined by the contents of the lecture notes with the exception of topics marked with a †. Additional reading is suggested at the end of each weeks notes. Some of the notational conventions adopted in the notes are described below.

† – additional non-examinable material for those who are interested.

* – material that you may want to skip first time around.

`typewriter font` – R related quantities and commands.

`italic typewriter font` – things to be replaced with with an appropriate value, identifier or expression.

italic font – a new term or idea.

If you miss a set of notes, additional copies are available from three sources:

- Public folder – to find copies of the notes in pdf format, open **Outlook** and go to `Public Folders` ⟶`All Public Folders` ⟶`Departments` ⟶`Statistics` ⟶`ST419` ⟶`Notes`
- Website – I will also put copies of the notes on my website at `http://stats.lse.ac.uk/baurdoux/CS.html`
- Moodle

The versions on the notes on the public folders and the website will be updated to include corrections. Despite my best efforts, **there will be mistakes in the notes**. If you spot something that looks wrong please let me know.

> **Asides**
> Interesting issues that arise from the topic under consideration are placed in boxes in the text. These may be hints on using R effectively, quick questions or suggestions for further work.

R Software and data

R software is freely available under the GNU General Public License. The R project homepage is `http://www.r-project.org/`. You can download the software to install on your own machine from `http://www.stats.bris.ac.uk/R/`. Data files associated with the course can be found in the `ST419` public folder under `Data`.

Communicating with me

By far the best way to communicate with me is via email. I will respond to sensible emails relating to:

- problems with material in the course,
- problems with exercises,
- mistakes in the notes.

The ST419 projects are unsupervised; I will not provide any direct assistance with projects.

Chapter 1

Data, Session Management and Simple Statistics

1.1 Key ideas

1.1.1 Objects

R works on *objects*. All objects have the following properties (referred to as *intrinsic attributes*):

- `mode`: tells us what kind of thing the object is – possible modes include `numeric`, `complex`, `logical`, `character` and `list`.
- `length`: is the number of components that make up the object.

At the simplest level, an object is a convenient way to store information. In statistics, we need to store observations of a variable of interest. This is done using a numeric vector. Note that there are no scalars in R; a number is just a numeric vector of length 1. Vectors are referred to as *atomic structures*; all of their components have the same mode.

If an object stores information, we need to name it so that we can refer to it later (and thus recover the information that it contains). The term used for the name of an object is *identifier*. An identifier is something that we choose. Identifiers can be chosen fairly freely in R. The points below are a few simple rules to bear in mind.

- In general any combination of letters, digits and the dot character can be used although it is obviously sensible to choose names that are reasonably descriptive.
- You cannot start an identifier with a digit or a dot so `moonbase3.sample` is acceptable but `3moons.samplebase` and `.sample3basemoon` are not.
- Identifiers are CASE SENSITIVE so `moon.sample` is different from `moon.Sample`. It is easy to get caught out by this.

- Some characters are already assigned values. These include `c`, `q`, `t`, `C`, `D`, `F`, `I` and `T`. Avoid using these as identifiers.

Typically we are interested in data sets that consist of several variables. In R, data sets are represented by an object known as a *data frame.* As with all objects, a data frame has the intrinsic attributes mode and length; data frames are of mode `list` and the length of a data frame is the number of variables that is contains. In common with many larger objects, a data frame has other attributes in addition to mode and length. The non-intrinsic attributes of a data frame are:

- `names`: these are the names of the variables that make up the data set,
- `row.names`: these are the names of the individuals on whom the observations are made,
- `class`: this attribute can be thought of as a detailed specification of the kind of thing the object is; in this case the class is `"data.frame"`.

The class attribute tells certain functions (*generic functions*) how to deal with the object. For example, objects of class `"data.frame"` are displayed on screen in a particular way.

1.1.2 Functions, arguments and return values

R works by calling functions. The notion of a function is a familiar one; a function takes a collection of inputs and maps them to a single output. In R, inputs and output are objects. The inputs are referred to as *arguments* and the output is referred to as the *return value.* Many useful functions are part of the standard R setup. Others are available via the inclusion of packages (see 3.2.2). One of the key advantages of R is the ease with which users can define their own functions (see 3.3). Functions are also objects; the mode of a function is `"function"` (sensibly enough). In computing functions have *side-effects.* For example, calling a function may cause a graph to be drawn. In many instances, it is the side-effect that is important rather than the return value.

1.1.3 Workspace and working directories

During an R session, a number of objects will be generated; for example we may generate vectors, data frames and functions. For the duration of the session, these objects are stored in an area of memory referred to as the *workspace.* If we want to save the objects for future use, we instruct R to write them to a file in our current *working directory* (directory is just another name for a folder). Note the distinction: things in memory are temporary (they will be lost when we log out); files are more permanent (they are stored on disk and the information they contain can be loaded into memory during our next session). Managing objects and files is an important part of using R effectively (see 1.2.7).

1.1.4 Two statistical ideas

In the initial (three week) period of the course, we will not be doing much statistics. However, we will use some simple statistical ideas to illustrate the use of R.

Hypothesis testing – a two-sided t-test example

Suppose that we have a random sample $y_1, \dots, y_n$ from a population that is normally distributed. We can view $y_1, \dots, y_n$ as being an instance of independent identically distributed random variables $Y_1, \dots, Y_n$, where $Y_i \sim N(\mu_Y, \sigma_Y^2)$ for $i = 1, \dots, n$. We would like to test

$$\begin{aligned} H_0 &: \quad \mu_Y = \mu_0, \\ H_1 &: \quad \mu_Y \neq \mu_0, \end{aligned}$$

where μ_0 is some real number. We set significance level α, that is, the probability of rejecting the null hypothesis when it is true is α. Our test statistic is

$$T = \frac{\bar{Y} - \mu_0}{S_Y/\sqrt{n}},$$

where

$$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i \quad \text{and} \quad S_Y^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \bar{Y})^2.$$

Note that $\bar{Y}$ and S_Y^2 are respectively unbiased estimators of μ_Y and σ_Y^2. Well established theory tells us, if H_0 is true, T follows a t-distribution on $n-1$ degrees of freedom. We evaluate the sample value of our statistic

$$t = \frac{\bar{y} - \mu_0}{s_y/\sqrt{n}}$$

where

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \quad \text{and} \quad s_y^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2.$$

This sample value is used to determine whether H_0 is rejected. Two (equivalent) approaches for the two-tailed case are described below.

- Critical value: c satisfies $P(|T| > c) = \alpha$. If $|t| > c$, we reject H_0.
- p-value: $p = P(|T| > |t|)$. If $p < \alpha$, we reject H_0.

For a given sample, these approaches lead to the same conclusion. Using p-values tends to be favoured; at any significance level larger than p, we reject H_0.

One tailed tests
The rules for drawing conclusions from a hypothesis test are easily adapted to one-tailed cases - try it.

Simple regression

If we believe there is a linear relationship between a response variable, $y_1, \ldots, y_n$, and the values of a fixed explanatory factor, $x_1, \ldots, x_n$, we may model this using simple linear regression:

$$Y_i = \alpha + \beta x_i + \varepsilon_i, \quad \varepsilon_i \sim (0, \sigma_\varepsilon^2), \tag{1.1}$$

where α is the intercept parameter and β is the slope parameter. The notation $\varepsilon_i \sim (0, \sigma_\varepsilon^2)$ indicates that the errors, ε_i, are random variables with mean zero and variance σ_ε^2. The parameters α and β are estimated by least squares, that is, by finding $\hat{\alpha}$ and $\hat{\beta}$ that minimize $\sum_{i=1}^n (Y_i - \hat{Y}_i)^2$, where $\hat{Y}_i = \hat{\alpha} + \hat{\beta} x_i$ is the fitted value at i. It is easy to show (try it) that

$$\hat{\beta} = \frac{\sum_{i=1}^n (Y_i - \bar{Y})(x_i - \bar{x})}{\sum_{i=1}^n x_i^2 - n\bar{x}^2} \quad \text{and} \quad \hat{\alpha} = \bar{Y} - \hat{\beta}\bar{x}.$$

The residuals of the fitted modelled are defined as $E_i = Y_i - \hat{Y}_i$, for $i = 1, \ldots, n$. The sample residuals, $e_i = y_i - \hat{y}_i$, can be interpreted as the difference between what we observe and what the model predicts. Scaled versions of the sample residuals are used as diagnostics in regression. In practice, we often want to test whether the explanatory factor influences the response. The null hypothesis is that there is no linear relationship,

$$\begin{aligned} H_0 &: \quad \beta = 0, \\ H_1 &: \quad \beta \neq 0. \end{aligned}$$

This is tested using the statistic

$$\frac{\hat{\beta}}{S_\varepsilon / \sqrt{\sum_{i=1}^n x_i^2 - n\bar{x}^2}}$$

where

$$S_\varepsilon^2 = \frac{1}{n-2} \sum_{i=1}^n E_i^2$$

is an unbiased estimator for the error variance, σ_ε^2. We discuss the properties of these statistics in section 4.1.5.

1.1.5 Real life examples – financial time series

This is not a course in finance and I am no financial expert, however, financial data provide many of the examples we will consider. A few preliminaries are important. The frequency with which financial data are observed varies. We will work mostly with daily data, that is, observations made every working day. The presence of weekends and bank holidays means that the observations in daily data are not regularly spaced in time. Irregular spacing is a feature common to many financial series.

Suppose that we observe the price of an asset, $y_1, \ldots, y_n$. Our interest lies not in the actual price of the asset but in whether the price has increased or decreased relative to the previous price; in other words, we are interested in *returns*. There are many ways of constructing returns, three of which are given below; see Tsay (2002) for more details:

- simple gross return: $\frac{y_t}{y_{t-1}}$
- simple (net) return: $\frac{y_t - y_{t-1}}{y_{t-1}}$
- log return: $\log\left(\frac{y_t}{y_{t-1}}\right) = \log y_t - \log y_{t-1}$

A common assumption is that observed returns can be treated as instances of uncorrelated random variables. Uncorrelated returns are a consequence of market efficiency, that is, the assumption that the price of an stock reflects all of the information available. This is an approximation but, in many instances, a reasonably good one. We tend to focus on log returns for reasons that will become apparent later.

Financial professionals make interesting use of simple regression; the returns from a stock are regressed against the returns from an appropriate index. The resulting slope estimate is referred to as the *beta value* of the stock. The beta value is taken as a measure of the risk associated with the stock relative to market risk; if a beta value is greater than 1 the stock is considered more risky (and potentially more rewarding), if the beta value is less than 1 then the stock is less risky. We calculate beta values for real stocks and discuss problems associated with their interpretation in the exercise.

1.2 An introductory R session

1.2.1 Starting a new R session

Go to `Start` $\longrightarrow$`Programs` $\longrightarrow$`Statistics` $\longrightarrow$`R` to start the R software package.

1.2.2 Using R as a calculator

The simplest thing that R can do is evaluate arithmetic expressions.

```
> 1
[1] 1
> 1+4.23
[1] 5.23
> 1+1/2*9-3.14
[1] 2.36
# Note the order in which operations are performed
# in the final calculation
```

This rapidly becomes quite tedious.

Comments in R
R ignores anything after a `#` sign in a command. We will follow this convention. Anything after a `#` in a set of R commands is a comment.

1.2.3 Vectors and assignment

We can create vectors at the command prompt using the concatenation function `c(...)`.

`c(`*`object1`*`,`*`object2`*`,...)`

This function takes arguments of the same mode and returns a vector containing these values.

```
> c(1,2,3)
[1] 1 2 3
> c("Ali","Bet","Cat")
[1] "Ali" "Bet" "Cat"
```

In order to make use of vectors, we need identifiers for them (we do not want to have to write vectors from scratch every time we use them). This is done using the assignment operator `<-`.

`name` `<-` *`expression`*

`name` now refers to an object whose value is the result of evaluating *`expression`*.

```
> numbers <- c(1,2,3)
> people <- c("Ali","Bet","Cat")
> numbers
[1] 1 2 3
> people
[1] "Ali" "Bet" "Cat"
# Typing an object's identifier causes R
# to print the contents of the object
```

Simple arithmetic operations can be performed with vectors.

```
> c(1,2,3)+c(4,5,6)
[1] 5 7 9
> numbers + numbers
[1] 2 4 6
> numbers - c(8,7.5,-2)
[1] -7.0 -5.5 5.0
> c(1,2,4)*c(1,3,3)
[1] 1 6 12
> c(12,12,12)/numbers
[1] 12 6 4
```

Note in the above example that multiplication and division are done element by element.

Reusing commands
If you want to bring back a command which you have used earlier in the session, press the up arrow key ↑. This allows you to go back through the commands until you find the one you want. The commands reappear at the command line and can be edited and then run by pressing return.

The outcome of an arithmetic calculation can be given an identifier for later use.

```
> calc1 <- numbers + c(8,7.5,-2)
> calc2 <- calc1 * calc1
> calc1
[1] 9.0 9.5 1.0
> calc2
[1] 81.00 90.25 1.00
> calc1 <- calc1 + calc2
> calc1
[1] 90.00 99.75 2.00
> calc2
[1] 81.00 90.25 1.00
# Note:  in the final step we have updated the value of calc1
# by adding calc2 to the old value; calc1 changes but calc2 is unchanged
```

If we try to add together vectors of different lengths, R uses a recycling rule; the smaller vector is repeated until the dimensions match.

```
> small <- c(1,2)
> large <- c(0,0,0,0,0,0)
> large + small
[1] 1 2 1 2 1 2
```

If the dimension of the larger vector is not a multiple of the dimension of the smaller vector, a warning message will be given. The concatenation function can be used to concatenate vectors.

```
> c(small,large,small)
[1] 1 2 0 0 0 0 0 0 1 2
```

We have now created a number of objects. To ensure clarity in the following examples we need to remove all of the objects we have created.

```
> rm(list=objects())
```

The way that this command works is described in section 1.2.7.

We want to work with data sets. In general we have multiple observations for each variable. Vectors provide a convenient way to store observations.

Example - sheep weight

We have taken a random sample of the weight of 5 sheep in the UK. The weights (kg) are

84.5 72.6 75.7 94.8 71.3

We are going to put these values in a vector and illustrate some standard procedures.

```
> weight <- c(84.5, 72.6, 75.7, 94.8, 71.3)
> weight
[1] 84.5 72.6 75.7 94.8 71.3

> total <- sum(weight)
> numobs <- length(weight)
> meanweight <- total/numobs
> meanweight
[1] 79.78
# We have worked out the mean the hard way.  There is a quick way ...

> mean(weight)
[1] 79.78
```

1.2.4 Data frames

A data frame is an R object that can be thought of as representing a data set. A data frame consists of variables (columns vectors) of the same length with each row corresponding to an experimental unit. The general syntax for setting up a data frame is

`name <- data.frame(variable1, variable2, ...)`

Individual variables in a data frame are accessed using the `$` notation:

`name$variable`

Once a data frame has been created we can view and edit it in a spreadsheet format using the command `fix(...)` (or equivalently `data.entry(...)`). New variables can be added to an existing data frame by assignment.

Example - sheep again

Suppose that, for each of the sheep weighed in the example above, we also measure the height at the shoulder. The heights (cm) are

86.5 71.8 77.2 84.9 75.4.

We will set up another variable for height. We would also like to have a single structure in which the association between weight and height (that is, that they are two measurements of the same sheep) is made explicit. This is done by adding each variable to a dataframe. We will call the data frame `sheep` and view it using `fix(sheep)`.

```
> height <- c(86.5, 71.8, 77.2, 84.9, 75.4)
> sheep <- data.frame(weight, height)
> mean(sheep$height)
[1] 79.16
> fix(sheep)
# the spreadsheet window must be closed before we can continue
```

Suppose that a third variable consisting of measurements of the length of the sheep's backs becomes available. The values (in cm) are

130.4 100.2 109.4 140.6 101.4

We can include a new variable in the data frame using assignment. Suppose we choose the identifier `backlength` for this new variable:

```
> sheep$backlength <- c(130.4, 100.2, 109.4, 140.6, 101.4)
```

Look at the data in spreadsheet format to check what has happened.

1.2.5 Descriptive analysis

A set of descriptive statistics is produced by the function `summary(...)`. The argument can be an individual variable or a data frame. The output is a table. Most functions to generate descriptive statistics are reasonably obvious; a couple are given below with more in the glossary.

```
> summary(sheep$weight)
 Min.   1st Qu.  Median  Mean   3rd Qu.  Max.
 71.30  72.60    75.70   79.78  84.50    94.80
> summary(sheep)
 weight              height              backlength
 Min.     :  71.30   Min.     :  71.80   Min.        :  100.2
 1st Qu.  :  72.60   1st Qu.  :  75.40   1st Qu.     :  101.4
 Median   :  75.70   Median   :  77.20   Median      :  109.4
 Mean     :  79.78   Mean     :  79.16   Mean        :  116.4
 3rd Qu.  :  84.50   3rd Qu.  :  84.90   3rd Qu.     :  130.4
 Max.     :  94.80   Max.     :  86.50   Max.        :  140.6
> IQR(sheep$height)
[1] 9.5
> sd(sheep$backlength)
[1] 18.15269
```

1.2.6 The R help system

There are a number of different ways of getting help in R.

- If you have a query about a specific function then typing ? and then the functions name at the prompt will bring up the relevant help page.

  ```
  > ?mean
  > ?setwd
  > ?t.test
  ```

- If you problem is of a more general nature, then typing `help.start()` will open up a window which allows you to browse for the information you want. The search engine on this page is very helpful for finding context specific information.

1.2.7 Session management and visibility

All of the objects created during an R session are stored in a *workspace* in memory. We can see the objects that are currently in the workspace by using the command `objects()`. Notice the `()`, these are vital for the command to work.

```
> objects()
[1] "height" "meanweight" "numobs" "sheep" "total"
[6] "weight"
```

The information in the variables `height` and `weight` is now *encapsulated* in the data frame `sheep`. We can tidy up our workspace by removing the `height` and `weight` variables (and various others that we are no longer interested in) using the `rm(...)` function. Do this and then check what is left.

```
> rm(height,weight,meanweight,numobs,total)
> objects()
[1] "sheep"
```

The `height` and `weight` variables are now only accessible via the `sheep` data frame.

```
> weight
Error:  Object "weight" not found
> sheep$weight
[1] 84.5 72.6 75.7 94.8 71.3
```

The advantage of this encapsulation of information is that we could now have another data frame, say `dog`, with `height` and `weight` variables without any ambiguity. However, the `$` notation can be a bit cumbersome. If we are going to be using the variables in the `sheep` data frame a lot, we can make them visible from the command line by using the `attach(...)` command. When we have finished using the data frame, it is good practice to use the `detach()` command (notice the empty `()` again) so the encapsulated variables are no longer visible.

```
> weight
Error:  Object "weight" not found
> attach(sheep)
> weight
```

```
[1] 84.5 72.6 75.7 94.8 71.3
> detach()
> weight
Error:  Object "weight" not found
```

We can save the current workspace to file at any time. It is a good idea to create a folder for each new piece of work that you start. We are going to create a folder called `ST419chapter1` for the work we are doing in this chapter. The correct R terminology for this folder is the working directory. You can set `ST419chapter1` to be the working directory whenever you want to work on objects created during work on this chapter (see §1.2.11).

- Create a folder

  In **My computer** use `File` ⟶`New` ⟶`Folder` to create a new folder in your `H:` space and give your new folder the name `ST419chapter1`.

- Set working directory

  ```
  > setwd("H:/ST419chapter1")
  ```

The command to save a workspace is `save.image(...)`. If we do not specify a file name the workspace will be saved in a file called `.Rdata`. It is usually a good idea to use something a little more informative than this default.

- Save current workspace

  ```
  > save.image("intro1.Rdata")
  ```

- List file in current working directory

  ```
  > dir()
  ```

The `dir()` command gives a listing of the files in the current working directory. You will see that your `ST419chapter1` directory now contains a file `intro1.Rdata`. This file contains the objects from our current session, that is, the `sheep` data frame. It is important to be clear about the distinction between objects (which are contained in a workspace, listed by `objects()`) and the workspace (which can be stored in a file, contained in a directory, listed by `dir()`).

1.2.8 Importing data

In practice, five observations would not provide a very good basis for inference about the entire UK sheep population. Usually we will want to import information from large (potentially very large) data sets that are in an electronic form. We will use data that are in a plain text format. The variables are in columns with the first row of the data giving the variable names. The file `sheep.dat` contains weight and height measurements from 100 randomly selected UK sheep. We are going to copy this file to our working directory `ST419chapter1`. The information is read into R using the `read.table(...)` function. This function returns a data frame.

- Get data files from public folders

  In **Outlook** go to `Public Folders` ⟶`All Public Folders` ⟶`Departments` ⟶`Statistics` ⟶`ST419` ⟶`Data` ⟶`Week1`. Select all the files using `Edit` ⟶`Select all`. The copy them to the clipboard using `Edit` ⟶`Copy`.

- Copy data files to working directory

  In **My computer** go to the `ST419chapter1` folder and use `Edit` ⟶`Paste` to put the data files into our working directory.

- Read data into R

  ```
  > sheep2 <- read.table("sheep.dat", header=TRUE)
  ```

Using `header = TRUE` gives us a data frame in which the variable names in the first row of the data file are used as identifiers for the columns of our data set. If we exclude the `header = TRUE`, the first line will be treated as a line of data.

1.2.9 A hypothesis test

Common wisdom states that the population mean sheep weight is 80kg. The data from 100 randomly selected sheep may be used to test this. We formulate a hypothesis test in which the null hypothesis is population mean, μ, of UK sheep is 80kg and the alternative is that the population mean takes a different value:

$$\begin{aligned} H_0 &: \mu = 80, \\ H_1 &: \mu \neq 80. \end{aligned}$$

We set significance level of 5%, that is $\alpha = 0.05$. Assuming that sheep weight is normally distributed with unknown variance, the appropriate test is a t-test (two-tailed). We can use the function `t.test(...)` to perform this test.

```
> attach(sheep2) # to make variables accessible
> t.test(weight, mu=80)

One Sample t-test

data:  weight
t = 2.1486, df = 99, p-value = 0.03411
alternative hypothesis:  true mean is not equal to 80
95 percent confidence interval:
  80.21048 85.29312
sample estimates:
mean of x
  82.7518
```

Notice the first argument of the t-test function is the variable that we want to test. The other arguments are optional. The argument `mu` is used to set the value of the mean that we would like to test (the default is zero). The output includes the sample value of our test statistic $t = 2.1486$ and the associated p-value 0.03411. For this example, $p < \alpha$ so we reject H_0 and conclude that there is evidence to suggest the mean weight of UK sheep is not 80kg. What conclusion would we have come to if the significance level had been 1%?

We can use the `alternative` argument to do one-tailed tests. For each of the following write down the hypotheses that are being tested and the conclusion of the test.

```
> t.test(weight, mu=80, alternative="greater")
> t.test(height, mu=66, alternative="less")
```

1.2.10 A linear model

The weight of sheep is of interest to farmers. However, weighing the sheep is time consuming and emotionally draining (the sheep do not like getting on the scales). Measuring sheep height is much easier. It would be enormously advantageous for the farmer to have a simple mechanism to approximate a sheep's weight from a measurement of its height. One obvious way to do this is to fit a simple linear regression model with height as the explanatory variable and weight as the response.

The plausibility of a linear model can be investigated with a simple scatter plot. The R command `plot(...)` is very versatile; here we use it in one of its simplest forms.

```
> plot(height,weight)
```

Notice that the x-axis variable is given first in this type of plot command. To fit linear models we use the R function `lm(...)`. Once again this is very flexible but is used here in a simple form to fit a simple linear regression of weight on height.

```
> lmres <- lm(weight ~ height)
```

You will notice two things

- The strange argument `weight ~ height`: this is a *model formula.* The ~ means "described by". So the command here is asking for a linear model in which weight is described by height. We will deal with the model formula notation in much greater detail in section 5.2.
- Nothing happens: no output from our model is printed to the screen. This is because R works by putting all of the information into the object returned by the `lm(...)` function. This is known as a *model object.* In this instance we are storing the information in a model object called `lmres` which we can then interrogate using *extractor functions.*

The simplest way to extract information is just to type the identifier of a model object (in this case we have chosen the identifier `lmres`). We can also use the `summary(...)` function to provide more detailed information or `abline(...)` to generate a fitted line plot. For each of the commands below make a note of the output.

```
> lmres
> summary(lmres)
> abline(lmres)
```

From the output of these commands write down the slope and intercept estimates. Does height influence weight? What weight would you predict for a sheep with height 56cm?

1.2.11 Quitting and returning to a saved workspace

Before ending a session it is advisable to remove any unwanted objects and then save the workspace in the current working directory.

```
> save.image("intro2.Rdata")
> quit()
```

On quitting you will be given the option of saving the current workspace. If you say `Yes` the current workspace will be saved to the default file `.Rdata` (we have already saved the workspace, there is no need to save it again).

In order to return to the workspace we have just saved. Restart R, set the appropriate working directory and then use the `load(...)` command to bring back the saved objects.

```
> setwd("H:/ST419chapter1")
> load("intro2.Rdata")
> objects()
# you should have all the objects from the previous session
```

1.2.12 Properties of objects†

We have already used the function `length(...)` to get the length of a vector. This can be applied to other objects.

```
> length(sheep2)
[1] 2
> length(plot)
[1] 1
```

The length of a data frame is the number of variables it contains. The length of a function can be evaluated but is a bit meaningless. The function `mode(...)` gives us the mode of an object, while `attributes(...)` lists the non-intrinsic attributes.

```
> mode(sheep2)
[1] "list"
> attributes(sheep2)
$names
```

```
[1] "weight" "height"

$class
[1] "data.frame"

$row.names
[1] "1" "2" "3" # ...  etc to "100"

> mode(plot)
[1] "function"
> attributes(plot)
NULL

> mode(lmres)
[1] "list"
> attributes(lmres)
$names
[1] "coefficients" "residuals" "effects" "rank"
[5] "fitted.values" "assign" "qr" "df.residual"
[9] "xlevels" "call" "terms" "model"

$class
[1] "lm"
```

Notice that, although both are of mode `"list"`, data frames are of class `"data.frame"` while linear model objects are of class `"lm"`. This influences the way they appear on screen when you type the identifier.

```
> sheep2
> lmres
```

A data frame is printed out in full while very little output is given for a linear model object. We can force R to treat both of these objects as lists by using the `unclass(...)` function. Try the following scroll back through the screen to see all the output.

```
> unclass(sheep2)
> unclass(lmres)
```

Notice that the final part of the output from `> unclass(lmres)` looks like the output from `sheep2`. This is because `lmres` contains an attribute (`$model`) that is a data frame made up of the variables that we put into our model. We can find out the class of something using the `class(...)` function.

```
> class(lmres$model)
[1] "data.frame"
```

This is not quite the end of the story. The mode is a convenient device that disguises a slightly messier reality. R really works on the *type* of an object. In some instances type is more specific than mode, in others the two properties are identical. We can find out the type of something using the function `typeof(...)`. The way in which it is stored is given by `storage.mode(...)`.

```
> typeof(sheep2)
[1] "list"
> typeof(sheep2$weight)
[1] "double"
> z <- 1 + 1i
> typeof(z)
[1] "complex"
> typeof(plot)
[1] "closure"
> storage.mode(plot)
[1] "function"
```

1.3 Glossary for chapter 1

- **Basic arithmetic operators** `()` brackets in arithmetic calculations
  `^` raise to the power of
  `/` division
  `*` multiplication
  `+` addition
  `-` subtraction
- **Assignment operator** `<-` assigns the value of the expression on the right to the identifier on the left.
- **Concatenation function** `c(...)` argument is a set of values; returns a vector containing those values.
- **Simple statistical functions** `sum(...)`
  `mean(...)`
  `median(...)`
  `range(...)`
  `sd(...)` standard deviation
  `mad(...)` mean absolute deviation
  `IQR(...)` inter-quartile range
  `min(...)` minimum
  `max(...)` maximum
- **Help** `?` ... will provide information on function usage and arguments
  `help.start()` opens html help files

- **Session management** `objects()` lists objects in workspace
  `ls()` does the same as `objects()`
  `rm(...)` removes the objects listed from workspace
  `rm(list = objects())` removes all objects from workspace
  `getwd()` returns the current working directory
  `setwd(...)` set the working directory
  `dir()` list files in current directory
  `save.image(...)` saves workspace to specified file
  `quit()` ends R session
  `load(...)` loads a saved workspace from file
- **Load data from text file** `read.table(...)` argument is a file in the working directory. Return value is a data frame.
- **Editing data** `fix(...)` or `data.entry(...)` brings up spreadsheet like environment for editing data.
- **Scatterplot** `plot(...)` is a versatile plotting command, draws a scatterplot when two arguments are given.
- **Linear model fitting** `lm(...)` can fit a variety of linear models, argument is a model formula.
- **Basic inference** `t.test(...)` Student's t-test
  `var.test(...)` F test to compare two variances
  `chisq.test(...)` Pearson's χ^2 test for count data
  `cor.test(...)` test for association
- **Object properties** `length(...)`
  `mode(...)`
  `typeof(...)`
  `class(...)`
  `attributes(...)` returns all non-intrinsic attributes
- **Object contents** `summary(...)` information on an object; output depends on class
  `str(...)` compact details of object structure
  `dput(...)` text listing of object

1.4 Exercise

The data file `IntelNASDAQ.dat` is a plain text file containing the daily closing prices for Intel stock and the NASDAQ index from 3rd January 2005 to 9th September 2005. The top row contains the variable names `Intel`, `NASDAQ`, `IntelB1` and `NASDAQB1`.

1. Import data: Create a data frame (with a suitable identifier) containing values from the file `IntelNASDAQ.dat`.

2. Descriptive analysis: Using the `plot(...)` command with single argument will generate a time series plot of the data (data values on the y-axis, time on the x-axis). Using the argument `type = "l"` (for lines) will improve the appearance of the plot (more on this in the next chapter).
   (a) Generate a time series plot and summary statistics, means and standard deviations for `Intel` and `NASDAQ` variables.
   (b) Is the mean value useful in this context?
   (c) Can we perform meaningful comparisons of the variability of `Intel` and `NASDAQ`?
3. Generate log returns: The variables `IntelB1` and `NASDAQB1` contain back-shifted values of the original series, that is, y_{t-1} for $t = 2, \dots, n$.
   (a) Look at the data using the spreadsheet format available in R to make sure you understand how `IntelB1` and `NASDAQB1` are derived from `Intel` and `NASDAQ`. [Note: `NA` (not available) denotes a missing value, more on these in section 3.2.5.]
   (b) Using an appropriate arithmetic combination of the original and the back-shifted values, generate log return series for Intel and NASDAQ.
   (c) Put the log return variables you have just generated into the data frame and remove them from the workspace.
   (d) Attach the data frame.
4. Descriptive analysis of log returns: We often want R to ignore missing values in a variable. This is done using the argument `na.rm = TRUE` – you will need this to calculate the mean below.
   (a) Generate time series plots and summary statistics for the log return series.
   (b) How could the sign of the mean of the series be interpreted?
5. A hypothesis test: We suspect that Intel stock has not done very well. We can test the null hypothesis that the mean return is zero against the alternative that the mean return is negative ($H_0 : \mu = 0$ versus $H_1 : \mu < 0$.)
   (a) Choose a significance level and perform the appropriate hypothesis test.
   (b) Interpret the output of the hypothesis test and write down your conclusions.
   (c) Comment on the validity of the test.
6. Generate beta value: We would like to determine how risky investment in Intel is relative to an investment tracking the NASDAQ100. One possible approach is to evaluate the beta value for Intel.
   (a) Plot Intel log returns against NASDAQ log returns. Does a linear relationship seem plausible?
   (b) Perform simple linear regression of Intel log returns on NASDAQ log returns. Write down and interpret the beta value from this fit.
   (c) Why might we doubt the usefulness of a beta value calculated in this naive way?

(d) † Using the function `lm(...)`, R automatically generates output associated with testing $H_0 : \beta = 0$ versus $H_1 : \beta \neq 0$ for our simple regression model. What is really of interest is to test $H_0 : \beta = 1$ versus $H_1 : \beta \neq 1$. Evaluate the appropriate statistic for this hypothesis test.

7. † Find financial series: `http://finance.yahoo.com` is a good starting point. Find a reasonably long series (four years of daily data should do) and import it into R. Plot the original series, log returns and the squares of the log returns. What do you notice about large values in the squares of the log returns plot?

1.5 Reading

1.5.1 Directly related reading

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition, Springer. [Chapter 1 and objects and data frames sections of chapter 2.]
- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 1, sections 2.1, 2.2, 6.3 and 7.1.]

1.5.2 Supplementary reading†

- Newbold, P (1995) *Statistics for Business and Economics*, 4th edition, Prentice-Hall. [A new edition became available in 2003. One of many books with a good elementary introduction to hypothesis testing and regression (chapter 9 and chapter 12 in the 4th edition).]
- Tsay, R (2002) *Analysis of Financial Time Series*, Wiley. [For those who want to know more about financial series.]
- Murakami, H (1982) *A Wild Sheep Chase* (English translation A. Birnbaum (1989)), Vintage. [Fiction – no relevance other than a possible influence on the choice of animal in the examples.]

Chapter 2

More R Essentials and Graphics

In this chapter we look at accessing part of an object. We have seen some examples of this already; in section 1.2.4 we use the `$` operator to select a named variable (vector) from a data frame. We may also be interested in selecting part of a vector, for example, those values lying in a particular range (trimming). We introduce regular sequence and logical vectors and show how they can be used in selection. The second theme of this chapter is graphics. Pictorial representations of both data and the results of statistical analysis are of profound importance. Graphics are one of the key tools which statistician use to communicate with each other and with the rest of the world (the people who pay our wages). R has powerful graphical capabilities. In this chapter we introduce some of the basics and provide pointers to more sophisticated techniques.

2.1 Key Ideas

2.1.1 Logic and logical operators

Logic is fundamental to the operation of a computer. We use logic as a mechanism for selection and as a tool in writing functions (see section 3.3). There are many equivalence ways of thinking about logical operations; some alternatives are given below.

AND, OR and truth tables

Logical statements are a part of natural language, for example, "sheep eat grass", "the Queen is dead" or "I am Spartacus". A logical statement is either true or false (we will not consider anything other than two-valued logic). In computing, logical statements can be formed by comparing two expressions. One possible comparison is for equality; for example, `(`*`expr1`*` == `*`expr2`*`)` is a logical statement. Note the double equality, `==`, to denote comparison; in R a single equality, `=`, means assignment (equivalent to `<-`) which is not what we want here. As with natural language, a logical statement in computing is either `TRUE` or `FALSE`. Logical statements are combined using the operators `AND` and `OR` (these are denoted `&` and `|` in R – more details in section 2.2.6). Combining a true statement with a true statement results in a true statement (the same holds for false statements). However, the result of combining a true statement with a false statement depends on the operator used. The truth tables below demonstrate.

AND	FALSE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	TRUE

OR	FALSE	TRUE
FALSE	FALSE	TRUE
TRUE	TRUE	TRUE

Table 2.1: Truth tables for `AND` and `OR` operators

The negation operator `NOT` (`!` in R) is applied to a single statement (unary operator). Negation turns a true statement in to a false one and vice versa. Suppose that `A` and `B` represent logical statements. De Morgan's law then tells us that

`NOT(A AND B) = NOT(A) OR NOT(B)` and `NOT(A OR B) = NOT(A) AND NOT(B).`

One consequence of de Morgan's laws is that the `AND` operation can be represented in terms of `OR` and `NOT` since `(A AND B) = NOT(NOT(A) OR NOT(B))`; similarly, `(A OR B) = NOT(NOT(A) AND NOT(B))`.

Alternative views of logical operations †

The operators `AND` and `OR` are analogous to intersection and union in set theory with `TRUE` and `FALSE` corresponding to inclusion or exclusion from a set. To illustrate, suppose that Y and Z are sets. Then $x \in Y$ but $x \notin Z$ implies $x \in Y \cup Z$ but $x \notin Y \cap Z$. This leads to a tables with the same structure as the truth tables 2.1.1.

$\cap$	$x \notin Y$	$x \in Y$
$x \notin Z$	$x \notin Y \cap Z$	$x \notin Y \cap Z$
$x \in Z$	$x \notin Y \cap Z$	$x \in Y \cap Z$

$\cup$	$x \notin Y$	$x \in Y$
$x \notin Z$	$x \notin Y \cup Z$	$x \in Y \cup Z$
$x \in Z$	$x \in Y \cup Z$	$x \in Y \cup Z$

Table 2.2: Inclusion tables for $\cap$ and $\cup$ operators

The complement can be interpreted as negation; Y^c is the set of all elements that are not in Y. De Morgan's laws now take on a familiar form

$$(Y \cap Z)^c = Y^c \cup Z^c \text{ and } (Y \cup Z)^c = Y^c \cap Z^c.$$

We consider two-valued logic. Another system that works on just two values is modular arithmetic where the modulus is 2 (referred to as mod 2 arithmetic). The two values are 0 and 1 (representing `FALSE` and TRUE). The results of a mod 2 arithmetic expression is the remainder when the value calculated in the usual way is divided by 2. The logical operation `AND` is equivalent to mod 2 multiplication and `NOT` is represented by mod 2 addition of 1.

mod 2 de Morgan
Use one of de Morgan's laws to derive and expression for `OR` using mod 2 arithmetic

2.1.2 Graphics in statistics

Graphics form an important part of any descriptive analysis of a data set; a histogram provides a visual impression of the distribution of the data and comparison with specific probability distributions, such as the normal, are possible using a quantile-quantile (qq) plot. The distribution of several variables can be compared using parallel boxplots and relationships investigated using scatter plots. Some plots are specific to a type of data; for example, in time series analysis, time series plots and correlograms (plots that are indicative of serial correlation) are commonly used. Graphical methods also play a role in model building and the analysis of output from a fitting process, in particular, diagnostic plots are used to determine whether a model is an adequate representation of the data.

2.2 R Essentials

2.2.1 A reminder on getting started

At the start of each chapter's practical work we will go through the same process; create a directory for the work, copy the data into this directory, start R and change the working directory to the directory we have just created.

- Create a folder

  In **My computer** use `File` ⟶`New` ⟶`Folder` to create a new folder in your `H:` space and give your new folder the name `ST419chapter2`.

- Get data files from public folders

  In Outlook go to `Public Folders` ⟶`All Public Folders` ⟶`Departments` ⟶`Statistics` ⟶`ST419` ⟶`Data` ⟶`Week 2`. Select all the files using `Edit` ⟶`Select all`. The copy them to the clipboard using `Edit` ⟶`Copy`.

- Copy data files to working directory

  In Windows Explorer go to the `ST419chapter2` and use `Edit` ⟶`Paste` to put the data files into this directory.

- Start R

  Go to `Start` ⟶`Programs` ⟶`Statistics` ⟶`R` to start the R software package.

- Set working directory

  ```
  > setwd("H:/ST419chapter2")
  ```

For the initial part of the work we are going to use two data sets relating to marks. The first is a toy data set with marks (percentage) for six students in three MSc courses. These data are in a file called `marks.dat`. The second is a larger data set with marks (out of 40) for three difficult exams for a second year undergraduate group (`marks2.dat`). The candidate numbers have been changed to protect the innocent.

```
> mkmsc <- read.table("marks.dat", header=TRUE)
> mk2nd <- read.table("marks2.dat", header=TRUE)
> mkmsc
> attributes(mk2nd)
```

The identifiers for the data frames are `mkmsc` for the MSc course marks and `mk2nd` for the second year exam marks. The variable in `mkmsc` are `courseA`, `courseB` and `courseC`; for `mk2nd` we have `exam1`, `exam2` and `exam3`.

2.2.2 Named arguments and default values

In mathematics, the order in which arguments are passed to a function determines what we do with them; for example, if $f(u, v) = u^2 + 3v$ then $f(v, u) = v^2 + 3u$, $f(a, b) = a^2 + 3b$ and so on. We are so accustomed to this idea that it seems trivial. R functions can operate using the same convention; for example, `plot(exam1,exam2)` will plot `exam2` against `exam1` (that is, `exam1` on the x-axis and `exam2` on the y-axis) while `plot(exam2,exam1)` will plot `exam1` against `exam2`. The position of the argument in the call tells R what to do with it.

```
> attach(mk2nd)
> plot(exam1, exam2)
> plot(exam2, exam1)
```

An alternative, that is available in R and many other languages, is to use named arguments; for example, the arguments of the plot command are `x` and `y`. If we name arguments, this takes precedence over the ordering so `plot(y = exam2, x = exam1)` has exactly the same effect as `plot(x = exam1, y = exam2)` (which is also the same as `plot(exam1,exam2)`).

```
> plot(y=exam2, x=exam1)
> plot(x=exam1, y=exam2)
```

In the previous example, there was little benefit (other than clarity) in the use of named arguments. The real use of named arguments is when some arguments have default values. We have encountered this already in the `t.test(...)` function of section 1.2.9. Often a combination of position and naming arguments is used. We will illustrate with three of the arguments of the `t.test(...)` function, namely `x` the data to test (the first argument), `mu` the population mean in the null hypothesis and `alternative` which specifies the type of alternative to be used. The argument `x` does not have a default value so we must past this to the machine. Suppose the we want to test hypotheses about the population mean result for exam1, μ_1 say.

```
>t.test(exam1)
```

will use the default values `mu = 0` and `alternative = "two.sided"`. The test that will be performed is $H_0 : \mu_1 = 0$ vs. $H_1 : \mu_1 \neq 0$. Specifying values of `mu` and `alternative` will have the obvious consequences.

```
> t.test(exam1, mu=30)
> t.test(exam1, alternative="greater")
> t.test(exam1, mu=31.5, alternative="less")
> t.test(mu=30, exam1)
> t.test(x=exam1, mu=30)
> detach()
```

In the first of these the argument `alternative` is not specified so the default value (`"two.sided"`) is used. In all except the last of these, the position of `exam1` (as the first unnamed argument) tells R this value is the `x` argument. We can use the name of the `x` argument if we choose. Note that the help system (*`?t.test`*) can be used to find the names and default values of arguments.

Paired t-test
Use help to find out about the other arguments of `t.test`. Using the `y` and `paired` arguments, test the hypothesis that the population mean marks for `exam1` and `exam2` are identical. What about `exam2` and `exam3`

2.2.3 `names` attribute

As mentioned in section 1.1.1, data frames have a `names` attribute that refers to the names of the variables in the data frame. These names can be listed using the `names(...)` function.

```
> names(mkmsc)
[1] "courseA" "courseB" "courseC"
```

The operator `$` allows us to access the sub-objects listed by `names(...)` directly. We can change the names and change the contents of the sub-objects by assignment. Suppose that the marks actually come from ST402, ST419 and ST422. We want to assign these names to the variables, look at the ST419 marks and then add 2 to each of these marks.

```
> names(mkmsc) <- c("ST402", "ST419", "ST422")
> mkmsc$ST419
[1] 67 80 55 76 49 61
> mkmsc$ST419 <- mkmsc$ST419 + 2
> mkmsc$ST419
[1] 69 82 57 78 51 63
```

2.2.4 Other non-intrinsic attributes

Non-intrinsic attributes other than `names` can be changed by assignment. For example, in a data frame, we may want to change the default row names into something more informative (in our example the names of the students is an obvious choice):

```
> row.names(mkmsc)
[1] "1" "2" "3" "4" "5" "6"

> row.names(mkmsc) <- c("Ali", "Bet", "Cal", "Dan", "Eli", "Foo")
> mkmsc
     ST402  ST419  ST422
 Ali    52     69     71
 Bet    71     82     84
 Cal    44     57     55
 Dan    90     78     68
 Eli    23     51     52
 Foo    66     63     61

# Names used are fictional and do no refer to any person
# living or dead; any resemblance is purely coincidental.
```

In the data set of marks for the second year exams, the first column is a column of candidate numbers. These identify each row and should really be the row names. A column in the data file can be used as row names by specifying a number for the `row.names` argument in a `read.table(...)` function; this number refers to the column in which row names can be found. We will redefine our `mk2nd` data frame.

```
> mk2nd <- read.table("marks2.dat", header=TRUE, row.names=1)
> attributes(mk2nd)
```

2.2.5 Regular sequences

A regular sequence is a sequence of numbers or characters that follow a fixed pattern. These are useful for selecting portions of a vector and in generating values for categorical variables. We can use a number of different methods for generating sequences. First we investigate the : sequence generator.

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
> 2*1:10
[1] 2 4 6 8 10 12 14 16 18 20
> 1:10 + 1:20
[1] 2 4 6 8 10 12 14 16 18 20 12 14 16 18 20 22 24 26 28 30
> 1:10-1
[1] 0 1 2 3 4 5 6 7 8 9
> 1:(10-1)
[1] 1 2 3 4 5 6 7 8 9
```

```
    # Notice that :  takes precedence over arithmetic operations.
```

The `seq` function allows a greater degree of sophistication in generating sequences. The function definition is

```
seq(from, to, by, length, along)
```

The arguments `from` and `to` are self explanatory. `by` gives the increment for the sequence and `length` the number of entries that we want to appear. Notice that if we include all of these arguments there will be some redundancy and an error message will be given. `along` allows a vector to be specified whose length is the desired length of our sequence. Notice how the named arguments are used below. By playing around with the command, see whether you can work out what the default values for the arguments are.

```
> seq(1,10)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(to=10, from=1)
[1] 1 2 3 4 5 6 7 8 9 10
> seq(1,10,by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
[16] 8.5 9.0 9.5 10.0
> seq(1,10,length=19)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
[16] 8.5 9.0 9.5 10.0
> seq(1,10,length=19,by=0.25)
Error in seq.default(1, 10, length = 19, by = 0.25) :
Too many arguments
> seq(1,by=2,length=6)
[1] 1 3 5 7 9 11
> seq(to=30,length=13)
[1] 18 19 20 21 22 23 24 25 26 27 28 29 30
> seq(to=30)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30
```

Finally you can use the function `rep`. Find out about `rep` using the R help system then experiment with it.

```
> ?rep
> rep(1, times = 3)
[1] 1 1 1
> rep((1:3), each =5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

2.2.6 Logical vectors

The logical values in R are `TRUE` and `FALSE`. Comparisons are performed using `==` for equality, `!=` for inequality, `>` for greater than, `>=` for greater than or equal, `<` for less than and `<=` for less than or equal to. The logical operators are `&` for AND, `|` for OR and `!` for NOT. When logical values are combined the result is another logical value (either `TRUE` or `FALSE`). For example, `TRUE & TRUE = TRUE`; two true statements joined with AND results in a true statement – "pandas eat bamboo" AND "sheep eat grass", as a combination of two true statements, is also true. However, `TRUE & FALSE = FALSE` – "statisticians are joyful" AND "pandas eat cheese" is false since (at least) one of these statements is false. Note that the scalar logical operators `||` (OR) and `&&` (AND) are available. These are faster but do not yield vector return values.

For any logical operator, we can work out the outcome of every possible combination of `TRUE` and `FALSE` to generate a truth table. We can do this in R by creating logical vectors.

```
> tf1 <- c(TRUE,TRUE,FALSE,FALSE)
> tf2 <- c(TRUE,FALSE,TRUE,FALSE)
> tf1 & tf2
[1] TRUE FALSE FALSE FALSE
> tf1 | tf2
[1] TRUE TRUE TRUE FALSE
> (tf1 & !tf2) | (!tf1 & tf2)
[1] FALSE TRUE TRUE FALSE
```

The last one of these is sometimes treated as a logical operator in its own right know as EXCLUSIVE OR (XOR). Try to work out what is going on here (a Venn diagram may help).

We can generate logical vectors from data. We use the marks example below to generate some logical vectors. In each case give an English version of the logical statement; for example, the first statement is ST419 marks are over 70 and R tells us that this is true for the second and fourth students.

```
> attach(mkmsc)
> ST419>70
[1] FALSE TRUE FALSE TRUE FALSE FALSE
> ST419>70 & ST402>70
[1] FALSE TRUE FALSE TRUE FALSE FALSE
> ST419>70 & ST402>80
[1] FALSE FALSE FALSE TRUE FALSE FALSE
> row.names(mkmsc)=="Ali"
[1] TRUE FALSE FALSE FALSE FALSE FALSE
```

2.2.7 Indexing vectors and subset selection

We often only want to use a portion of the the data or a subset whose members satisfy a particular criteria. The notation for referring to a part of a vector is the square brackets `[]`.

This is known as the *subscripting operator*. The contents of the brackets is a vector, referring to as an *indexing vector*. If the indexing vector is integer valued (often a regular sequence), the elements with indices corresponding to the values of the indexing vector are selected. If there is a minus sign in front then all elements except those indexed by the sequence will be selected.

```
> ST402
[1] 52 71 44 90 23 66
> ST402[1]
[1] 52
> ST402[c(1,2,6)]
[1] 52 71 66
> ST402[1:4]
[1] 52 71 44 90
> ST402[-(1:4)]
[1] 23 66
> ST402[seq(6,by=-2)]
[1] 66 90 71
> ST402[rep((1:3),each=2)]
[1] 52 52 71 71 44 44
> row.names(mkmsc)[1:3]
[1] "Ali" "Bet" "Cal"
```

If the contents of the `[]` brackets is a logical vector of the same length as our original, the result is just those elements for which the value of the logical vector is true. We can now work things like:

- the ST419 marks which were higher than 70,
- the ST402 marks for people who got less than 65 in ST419,
- the ST422 marks for people for people whose total marks are less than 200.

These are implemented below. Try to work out and English statement for what the fourth command is doing (note 50 is the pass mark for these exams).

```
> ST419[ST419>70]
[1] 82 78
> ST402[ST419<65]
[1] 44 23 66
> ST422[(ST402+ST419+ST422)<200]
[1] 71 55 52 61
> ST422[ST402>50 & ST419>50 & ST422>50]
[1] 71 84 68 61
```

In practice we may be more interested in the names of the students with ST419 marks over 70 rather than the marks themselves. Try to work out what the last two of the commands below are doing.

```
> row.names(mkmsc)[ST419>70]
[1] "Bet" "Dan"
> row.names(mkmsc)[ST402<50 | ST419<50 | ST422<50]
[1] "Cal" "Eli"
> names(mkmsc)[c(sum(ST402), sum(ST419), sum(ST422)) > 350]
[1] "ST419" "ST422"
> detach()
```

Backing out of a command
If you make a mistake while typing a command, you can correct it by moving the cursor to the spot where the mistake occurred and replacing it with the correct text. Alternatively, you can back out completely and return to the command prompt by pressing the escape key `Esc`.

2.2.8 Character sequences

We have encountered sequences of characters (often referred to as *strings*) as names and as arguments passed to functions, such as `"exam1"`, `"H:/ST419chapter2"` and `"greater"`. The double quotes `""` tell R that the content should be treated as a characters. Combinations of characters and numeric values may be helpful for labeling. For example, suppose we have pairs of points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_5, y_5)\}$. The `paste(...)` function combines its arguments (using the recycling rule where necessary) a returns a vector of mode `"character"`. The `sep` argument can be used specify what we want between the arguments (often we want nothing, `sep = ""`).

```
> paste(c("x","y"), rep(1:5,each=2), sep="")
[1] "x1" "y1" "x2" "y2" "x3" "y3" "x4" "y4" "x5" "y5"
# Not quite what we wanted

> str1 <- paste("(x", 1:5, sep="")
> str2 <- paste("y", 1:5, ")", sep="")
> label1 <- paste(str1,str2,sep=",")
> label1
[1] "(x1,y1)" "(x2,y2)" "(x3,y3)" "(x4,y4)" "(x5,y5)"
# Much better

> mode(label1)
[1] "character"
# Note that the result of paste is of mode "character"
```

† The functions `grep` and `regexpr` can be used to find specific patterns of characters. The wildcard `.` can represent any characters. Functions are also available for character substitution.

```
> cvec <- c("zaa", "aza", "aaz", "aaa", "zzz")
> grep("z", cvec)
[1] 1 2 3 5
> grep(".z", cvec)
```

```
[1] 2 3 5
> grep("z.", cvec)
[1] 1 2 5
> regexpr(".z.", cvec)
[1] -1 1 -1 -1 1
attr(,"match.length")
[1] -1 3 -1 -1 3
```

2.2.9 Lists*

A list is an ordered collection of objects. The objects in a list are referred to as *components*. The components may be of different modes. Lists can be generated using the function `list(...)`; the return value is a list with components made up of the arguments passed to the function. Below we create a list, look at the intrinsic attributes then display the contents of the list.

```
> mklst <- list("MSc exam marks", c(10,6,2005), mkmsc)
> mode(mklst)
[1] "list"
> length(mklst)
[1] 3
> mklst
```

Here we include a title, a date and our MSc exam marks data frame. We can use the indexing operator `[]` to access parts of the list.

```
> mklst[2]
[[1]]
[1] 10 6 2005

> mode(mklst[2])
[1] "list"

> length(mklst[2])
[1] 1
```

Note that the result of using indexing on a list is another list. If we want to add 4 to the day we cannot do this using `[]` indexing since `mklst[2]` is not numeric:

```
> mklst[2]+c(4,0,0)
Error in mklst[2] + c(4, 0, 0) :  non-numeric argument to binary operator
```

To avoid this problem we used a double square bracket subscripting operator `[[]]`. In our example, `mklst[[2]]` is a numeric vector.

```
> mklst[[2]]
[1] 10 6 2005
```

```
> mode(mklst[[2]])
[1] "numeric"
> mklst[[2]] <- mklst[[2]] + c(4,0,0)
> mklst
```

In fact, since `mklst[[2]]` is a numeric vector, we can refer to the first element directly (and add 4 to it).

```
> mklst[[2]][1] <- mklst[[2]][1] + 4
```

It is important to remember that, in general, the subscripting operator `[]` is applied to a list it returns a list. (This is a simplification, the `[]` operator responds to class, so the behaviour is different for data frames.) The operator `[[]]` returns a component of a list; the mode of the return value of `[[]]` is the mode of the component.

The default labels for the components of a list (`1, 2, 3,...`) bear no association to what the list contains. In the example above, we have to remember that the date was the second component of our list. This is no great hardship for a list with three elements but becomes very inconvenient in larger lists. We can associate names with the components of a list.

```
> names(mklst) <- c("title", "date", "marks")
```

We can now refer to components of the list using either the `[[]]` operator or the `$` operator.

```
> mklst[[2]]
[1] 18 6 2005
> mklst[["date"]]
[1] 18 6 2005
> mklst$date
[1] 18 6 2005
```

Note that in using the `$` operator we can refer to a name directly (we do not need to use the `""`). Names may also be assigned when we define the list; the same effect would have been achieved using:

```
> mklst <- list("title"="MSc exam marks", "date"=c(10,6,2005),
+ "marks"=mkmsc)
```

The `attach(...)` and `detach()` functions described in section 1.2.7 can be applied to any list with named components.

Commands taking up more than one line
If we hit return in R when a command is syntactically correct but incomplete, a `+` prompt will appear rather than the usual `>`. The `+` indicates that R is expecting us to finish the command. Spreading commands over more than one line is useful for long commands and does not affect the outcome in any way.

2.3 Basic R Graphics

R has powerful and flexible graphics capabilities. This section provides a flavour of the sorts of things that are possible rather than a comprehensive treatment.

2.3.1 Some descriptive plots

On of the simplest ways to get a feel for the distribution of data is to generate a histogram. This is done using the `hist(...)` command. By setting the value of arguments of `hist(...)` we can alter the appearance of the histogram; setting `probability = TRUE` will give relative frequencies, `nclass` allows us to suggest the number of classes to use and `breaks` allows the precise break points in the histogram to be specified.

```
> attach(mk2nd)
> hist(exam1)
> hist(exam1, probability = TRUE)
> hist(exam1, nclass=10)
> hist(exam1, breaks=c(0,20,25,30,40))
```

We can use a selection from a variable or data frame as the main argument in our `hist(...)` command. Notice that we cannot pass the data frame as an argument directly. I have used a trick selection (selecting the whole data frame) to get round this - there may well be a better way.

```
> hist(exam3[exam1+exam2 <50])
> hist(mk2nd[mk2nd>20])
> hist(mk2nd)
Error in hist.default(mk2nd) :  'x' must be numeric
> hist(mk2nd[mk2nd<=max(mk2nd)])
```

It is often useful to compare a data set to the normal distribution. The `qqnorm(...)` command plots the sample quantiles against the quantiles from a normal distribution. A `qqline(...)` command after `qqnorm(...)` will draw a straight line through the coordinates corresponding to the first and third quartiles. We would expect a sample from a normal to yield points on the qq-plot that are close to this line.

```
> qqnorm(exam2)
> qqline(exam2)
> qqnorm(mk2nd[mk2nd<=max(mk2nd)])
> qqline(mk2nd[mk2nd<=max(mk2nd)])
```

Boxplots provide another mechanism for getting a feel for for the distribution of data. Parallel boxplots are useful for comparison. The full name is a box-and-whisker plot. The box is made up by connecting three horizontal lines: the lower quartile, median and upper quartile. In the default set up, the whiskers extend to any data points that are within 1.5 times the inter-quartile range of the edge of the box.

```
> boxplot(exam1,exam2,exam3)
# The label here are not very informative

> boxplot(mk2nd)
# Using the data frame as an argument gives a better result

> boxplot(mk2nd, main="Boxplot of exam scores", ylab="Scores")
# A version with a title and proper y-axis label
```

2.3.2 Plot types, graphics parameters and interactivity

The default plot type in R is usually quite sensible but we may on occasion want to change it. This is done using the `type` argument in the plotting function. Below is a (rather silly) illustration of some of the different types of plot.

```
> plot(exam1,exam2)
> plot(exam1, exam2, type="p") # points (the default)
> plot(exam1, exam2, type="l") # lines
> plot(exam1, exam2, type="b") # both lines and points
> plot(exam1, exam2, type="o") # overlaid lines on points
> plot(exam1, exam2, type="h") # high density (vertical lines)
```

If we look at the original scatter plot you notice that the individual points are marked by small circles. This is not necessarily what we want. The symbol used to mark the points can be changed by altering a graphics parameter `pch` (plotting character). There are a huge number of graphics parameters than can be altered but this is one of the most useful to be able to change.

```
> plot(exam1, exam2, pch="+")
> plot(exam1, exam2, pch="x")
> plot(exam1, exam2, pch=2)
> plot(exam1, exam2, pch=3)
> plot(exam1, exam2, pch=4)
# Note:  the numbers in the final three commands
refer to predefined plotting character.
```

You might want to make a permanent change to a graphics parameter. This is done using the `par(...)` command. This command can also be used to list the graphics parameters.

```
> par() # list the graphics parameters and defaults
> par(c("pch", "col")) # defaults for plotting character and colour
> plot(exam1, exam2)
> par(pch="*") # changing plotting character to "*"
> plot(exam1, exam2)
```

R has a number of interactive graphics capabilities. One of the most useful is the `identify(...)` command. This allows us to label interesting points on the plot. After an `identify(...)` command, R will wait while the users selects points on the plot using the mouse. The process is stopped using the right mouse button.

```
> plot(exam1,exam2)
> identify(exam1,exam2)
> identify(exam1,exam2,row.names(mk2nd))
# Note the default marks are the position (row number) of the point in
the data frame.  Using row names may be more informative.
```

R allows you to put more than one plot on the page by setting the `mfrow` parameter. The value that `mfrow` is set to is an integer vector of length 2 giving the number of rows and the number of columns.

```
> par(mfrow=c(3,2))
> hist(exam1)
> qqnorm(exam1)
> hist(exam2)
> qqnorm(exam2)
> hist(exam3)
> qqnorm(exam3)
> par(mfrow=c(1,1))
```

You could use `par(mfcol=c(3,2))` to get the same 3×2 multi-figure plot. Work out the difference between `mfrow` and `mfcol`. R also allows you to change the tick marks and labels, the borders around plots and the space allocated for titles – more can be found in Venables's *et. al.* (see supplementary reading section 2.6.2).

2.3.3 Low level graphics*

R allows you to build plots up more or less from scratch. The commands `text(...)` and `lines(...)` can be used to add text and lines to a plot. In the following example we plot the positions of five cities (using their longitude and latitude) and then connect them using lines.

```
> citynames <- c("Athens", "Jo'burg", "London", "NYC", "Shanghai")
> longitude <- c(23.72, 28.07, -0.08, -73.47, 121.47)
> latitude <- c(37.97, -26.20, 53.42, 40.78, 31.17)
> cities <- data.frame(latitude,longitude)
> row.names(cities) <- citynames
> cities
> rm(latitude,longitude,citynames)
> attach(cities)
> plot(longitude,latitude,type="n")
> points(longitude,latitude,pch=".")
> text(longitude[1], latitude[1], row.names(cities)[1],pos=1)
> text(longitude[2], latitude[2], row.names(cities)[2],pos=4)
> text(longitude[3], latitude[3], row.names(cities)[3],pos=4)
> text(longitude[4], latitude[4], row.names(cities)[4],pos=4)
> text(longitude[5], latitude[5], row.names(cities)[5],pos=2)
> lines(c(longitude[1],longitude[3]), c(latitude[1],latitude[3]))
> lines(c(longitude[2],longitude[3]), c(latitude[2],latitude[3]))
> lines(c(longitude[4],longitude[3]), c(latitude[4],latitude[3]))
> lines(c(longitude[5],longitude[3]), c(latitude[5],latitude[3]))
> detach()
```

Things to note in this example:

- The command `plot(longitude,latitude,type="n")` will plot the appropriate axes but `type = "n"` indicates that nothing else should be plotted.
- The `pos` argument can take the value 1 for below, 2 for left, 3 for below and 4 for right. In the final plot the position of the text is not great. We could do better using the `adj` and `offset` commands. You can find out about these using the help system `?text`.

2.3.4 3-D graphics†

We can generate various types of three dimensional plot in R. The input for this type of plotting function is two vectors (values from the x and y axes) and a matrix (values from the surface that is to be plotted). We are going to generate 3-D plots for the surface

$$f(x, y) = x^2y^3$$

for $x \in [-1, 1]$ and $y \in [-1, 1]$. In order to do this we first generate regular sequence of 50 points on the interval $[-1, 1]$ and then use the `outer` function to give the 50×50 matrix (outer product) of points on the surface (more on matrices in section 3.2.7).

```
> x <- seq(-1,1,length=50)
> y <- seq(-1,1,length=50)
> z <- outer(x^2,y^3)
> contour(x,y,z)
> image(x,y,z)
> persp(x,y,z)
# Note The appearance of these plots can be improved by changing some of
the default value of the arguments.
```

2.3.5 Trellis graphics†

Trellis graphics provide graphical output with a uniform style. The functions are written in R and are slow and memory intensive. A graph produced using trellis graphics cannot be altered so everything we want to include must be specified in the initial command. Trellis graphics are implemented using the `lattice` library (more on the use of libraries in the next chapter). We will not use trellis graphics in this course. However, if you are interested in producing sophisticated conditional plots, you may want to look at the trellis graphics section of Venables and Ripley (see supplementary reading section 2.6.2).

2.4 Glossary for chapter 2

- **Regular sequences** `:` sequence generator gives sequence with unit steps between left and right arguments
  `seq(...)` more general regular sequence generator, can set step (`by`) and/or length of sequence (`length`)
  `rep(...)` for generating sequence with repeated values
- **Logical values** `TRUE` and `FALSE`
- **Comparison and logical operators** `==` equality
  `!=` inequality
  `<` less than
  `>` greater than
  `<=` less than or equal to
  `>=` greater than or equal to
  `!` negation
  `&` and (`&& scalar version`)
  `|` or (`|| scalar version`)
- **Accessing elements** `[...]` contents of brackets a vector of integers, a vector of logical values or a vector of names.
- **Accessing a list component** `[[...]]` contents of brackets a number or a name.
- **Accessing a named component** `$`
- **Referring to list components directly** `attach(...)` brings components of a list into view; use detach() to tidy up.
- **Generating a vector of strings** `paste(...)` converts any non-character arguments into characters and puts them together into a vector of mode character.
- **Generating lists** `list(...)` put arguments (which may be named) into a list.
- **A selection of plots** `hist(...)` histogram
  `qqnorm(...)` normal qq-plot
  `boxplot(...)`
  `contour(...)` 3D contours
  `image(...)` 3D coloured blocks
  `persp(...)` 3D wire frame
- **Some low level plotting commands** `points(...)`
  `lines(...)`
  `text(...)`
- **Some graphics parameters** `pch` plotting character
  `col` colour
  `mfrow` multi-figure page specification `mfcol`

2.5 Exercises

The file IntelNASDAQdated.dat contains closing prices of Intel stocks and the NASDAQ100 index from 3rd January 2000 to 9th September 2005. The variables in this file are `Day`, `Month`, `Year`, `Intel`, `NASDAQ`, `IntelLR` and `NASDAQLR`. Here, `LR` denotes log returns. The months are specified by their first three letter, that is `Jan`, `Feb`, ... Import these data into an appropriately named data frame.

1. Identify outlying points: An application of R interactive graphics features.
   (a) Plot the Intel log returns against NASDAQ log returns.
   (b) Identify the outliers in the plot generated above.
   (c) Use the `Day`, `Month` and `Year` variables to construct row names for the data frame.
   (d) Repeat the plot and identification stages above using the row names you have just generated to label the points.
   (e) What do the outliers you have identified correspond to.
2. Comparing 2000 with 2003: Selection allows us to construct subvectors and compare them.
   (a) Using appropriate selection construct a data frame containing log returns for both series for 2000.
   (b) Construct another data frame with log returns for both series for 2003.
   (c) Generate a scatter plot of Intel log returns against NASDAQ log returns for the year 2000.
   (d) Add points for 2003 to the plot generated above using a differ colour and plot character.
   (e) Comment on this plot noting any similarities and differences between 2000 and 2003.
   (f) Fit linear models with Intel log returns as the response variable and NASDAQ log returns as the explanatory variable for 2000. Repeat for 2003. Compare the resulting models.
3. Histograms: A first attempt to compare distributions.
   (a) Change the appropriate graphics parameters to get four plots to a page.
   (b) Generate histograms for Intel log returns and NASDAQ log returns for 2000 and 2003 (four plots in all).
   (c) The plots just generated are not much good for comparison. Redo this plot setting the breaks in the histograms so that they are all drawn on the same horizontal scale.
   (d) Reset the appropriate graphics parameters to get one plot per page.
4. Boxplots: Often a good way of doing quick comparisons between several variables.
   (a) Generate a boxplot of the four variables of interest.
   (b) Use help to find out about the named arguments of the boxplot command. Use them to generate a boxplot with better labels.

5. Test of variances: It seems clear from the descriptive plots that 2003 differs from 2000 quite dramatically. We can perform hypothesis tests to check whether the statistics support this.
   (a) Find out about the R function `var.test(...)` using help.
   (b) Test the hypothesis of equality of variance for the Intel log returns for 2000 and 2003.
   (c) Repeat the test for NASDAQ log returns.
6. Extracting information from data: Use selection to answer the following.
   (a) Find the mean price of Intel stock on your birthday for the period.
   (b) Find the date on which the NASDAQ index reached its lowest point for the period. Repeat for Intel and comment on the results.
   (c) Find the standard deviation of Intel log returns for the single month that contains the highest price of Intel stock.
7. † A tricky selection problem:
   (a) Create a vector containing the standard deviations of Intel log returns for each month from January 2000 to September 2004. [Hint: look up the use of factors – chapter 4 of An Introduction to R.]
   (b) Plot the values of the standard deviations against time.

2.6 Reading

2.6.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 2 for vectors and selection, chapter 6 for data frames, lists and chapter 7 for reading in data and chapter 12 for graphics.]
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition, Springer. [Chapter 2 (excluding 2.4) for data manipulation (includes some topics we have not covered yet), chapter 3 up to 3.4 for more on language elements and chapter 4 for graphics.]

2.6.2 Supplementary reading†

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Section 12.5 for details of plot spacing and tick marks, chapter 4 for factors]
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition, Springer. [Section 4.5 for trellis graphics]

Chapter 3

Writing functions in R

3.1 Key ideas

3.1.1 Good programming practice

A program is a set of instructions for a computer to follow. Putting a set of instructions together in a program means that we do not have to rewrite them every time we want to execute them. Programming a computer is a demanding (but potentially rewarding) task. The process is made simpler and the end product more effective by following the simple guidelines set out below.

- **Problem specification**: The starting point for any program should be specification of the problem that we would like to solve. We need to have a clear idea of what we want the program to do before we start trying to write it. In particular, the available inputs (arguments) and the desired output (return value) should be specified.
- **Code planning**: In the early stage of writing a program it is best to stay away from the machine. This is hard to do, but time sketching out a rough version of the program with pen and paper is almost always time well spent. Developing your own pseudo-code, some thing between natural and programming language, often helps this process.
- **Identifying constants**: If you see a number cropping up repeated in the same context, it is good practice to give this value to an identifier at the start of the program. The advantage is one of maintenance; if we need to change the value it can be done with a single alteration to the program.
- **Program documentation**: A good program should be self-documenting. The code should be laid out in a clear and logical fashion; appropriate use of indentation adds enormously to readability of the code. Comments should be used to describe how the more complicated parts of the program work. Writing comments is for your own benefit as much as anyone else's. Even though a program may seem trivial when you have finished writing it, a few weeks later you may be surprised to have to spend time unraveling the

'obvious code'. Just a few simple comments about what the program is for and what each part does will help.

- **Solving runtime problems**: Having carefully crafted and typed in your program, more often than not, the computer will refuse to run it at first time of asking. You may get some vaguely helpful suggestion as to where you have gone wrong. We will not be writing big programs so we do not require very sophisticated debugging tools. If the error message does not lead you straight to the solution of the problem, it is possible to comment out sections of the program to try to isolate the code which is causing the problem.
- **Program verification**: Once the program runs, we need to make sure it does what it is intended to do. It is possible to automate the process of program verification but we will usually be satisfied with checking some cases for which the correct output is known.

3.1.2 Flow control

Computers are very good at performing repetitive tasks. If we want a set of operations to be repeated several times we use what is known as a *loop*. The computer will execute the instructions in the loop a specified number of times or until a specified condition is met. Once the loop is complete, the computer moves on to the section of code immediately following the loop as illustrated below.

program section	`A`
	`loop B`
	`C`
order of execution	`A B B ...B C`

There are three types of loop common to most (C-like) programming languages: the `for` loop, the `while` loop and the `repeat` loop. These types of loop are equivalent in the sense that a loop constructed using one type could also be constructed using either of the other two. Details of the R implementation can be found in sections 3.5.1 and 3.5.3. In general loops are implemented very inefficiently in R; we discuss ways of avoiding loops in section 3.5.4. However, a loop is sometimes the only way to achieve the result we want.

Often there are parts of the program that we only want to execute if certain conditions are met. *Conditional statements* are used to provide branches in computer programs. The simplest structure is common to most programming languages;

`if (`*`condition`*`)` *`ifbranch`*

program section	`A`
	`if (`*`condition`*`) B`
	`D`
order of execution	if the condition is true `A B D`
	if the condition is false `A D`

In this example, `B` forms the if branch of our program. Another common conditional statement takes the form

```
if (condition) ifbranch else elsebranch
```

program section	`A`
	`if (condition) B`
	`else C`
	`D`
order of execution	if the condition is true `A B D`
	if the condition is false `A C D`

In R it is also possible to construct vector conditional statements. Conditioning in R is discussed in section 3.5.2

3.1.3 Pseudo-random numbers

Mathematical models fall into two categories, deterministic and stochastic. For a deterministic model, if we know the input, we can determine exactly what the output will be. This is not true for a stochastic model; the output of a stochastic model is a random variable. Remember that we are talking about the properties of models; discussions of the deterministic or stochastic nature of real-life phenomena often involve very muddled thinking and should be left to the philosophers. Simple deterministic models can produce remarkably complex and beautiful behaviour. A example is the logisitic map,

$$x_{n+1} = rx_n(1 - x_n), \tag{3.1}$$

for $n = 1, 2, \ldots$ and starting value $x_0 \in (0, 1)$. For some values of the parameter r, this map will produce a sequence of numbers $x_1, x_2, \ldots$ that look random. Put more precisely, we would not reject a null hypothesis of independence if we were to test these values. We call sequences of this sort *pseudo-random*. Pseudo-random number generators play an important role in the study of model properties using simulation (see chapter 4).

3.2 R essentials

3.2.1 Saving, loading and running R commands

R maintains a record of command history so that we can scroll through previous commands using ↑ and ↓ keys. It may sometimes be useful to save commands for future use. We can save the command history at any stage using

```
> savehistory("filename.Rhistory")
```

The commands entered so far will be saved to file in a plain text format. All of the sessions and my solutions to the exercises are available as `.Rhistory` files on the webpage and public folders. In order to restore saved commands we use

```
> loadhistory("filename.Rhistory")
```

The commands will not be run but will be made available via ↑ key. In order to run a saved set of commands or commands in plain text format from some other source we use

```
> source("filename", echo=TRUE)
```

The echo command tells R to print the results to screen. All of the commands in the file will be executed using this approach. It is also possible to paste commands (that have been copied to the Windows clipboard) at the R command prompt.

3.2.2 Packages

A package in R (library section in S) is a file containing a collection of objects which have some common purpose. For example, the library `MASS` contains objects associated with the Venables and Ripley's *Modern Applied Statistics with S.* We can generate a list of available packages, make the objects in a package available and get a list of the contents of a package using the `library(...)` command.

```
> library()  # lists available packages
> library(MASS)  # loads MASS package
> library(help=MASS)  # lists contents of MASS package
```

If we are specifically interested in the data sets available in a package we can use the `data(...)` command to list them and add them to our workspace.

```
> data()  # lists data sets in loaded packages
> data(package=nlme)  # lists data sets in mixed effects package
> data(Alfalfa, package=nlme)  # adds Alfalfa data set to workspace
> summary(Alfalfa)
```

Information on recommended packages (in the very large R Reference Index) along with a guide to writing R extensions can be found at `http://cran.r-project.org/manuals.html`

3.2.3 The search path

When we pass an identifier to a function R looks for an associated object. The first place R looks is in the workspace (also known as the global environment `.GlobalEnv`). If the object is not found in the workspace, R will search any attached lists (including data frames), packages loaded by the user and automatically loaded packages. The real function of `attach(...)` is now clear; `attach(...)` places a list in the search path and `detach(...)` removes it. The function `search()` returns the search path. Passing the name of any environments or its position to the objects function will list the objects in that environment.

```
> search()
> objects(package:MASS)
> objects(4)
```

3.2.4 Formatting text output

For anything other than the simplest functions, the format of the output that is printed to screen must be carefully managed; a function is of little use if its results are not easy to interpret. The function `cat(...)` concatenates its arguments and prints them as a character string. The function `substring(...)` is use to specify part of a sequence of characters.

```
> str3 <- "ST419 Computational Statistics"
> cat(substring(str3,1,13), substring(str3,26,29), "\n", sep="")
```

Note that `"\n"` is the new line character. The function `format(...)` allows greater flexibility including setting the accuracy to which numerical results are displayed. A combination of `cat(...)` and `format(...)` is often useful.

```
> roots.of.two <- 2^(1/1:10)
> roots.of.two
> format(roots.of.two)
> format(roots.of.two, digits=3)
> cat("Roots of two:  ", format(roots.of.two, digits=3),"\n")
```

3.2.5 Missing and not computable values

We have already encountered the special value `NA` denoting not available. This is used to represent missing values. In general, computations involving `NA`s result in `NA`. In order to get the result we require, it may be necessary to specify how we want missing values to be treated. Many function have named arguments in which the treatment of missing values can be specified. For example, for many statistical functions, using the argument `na.rm = TRUE` instructs R to perform the calculation as if the missing values were not there. R has a representation of infinity (`Inf`). However, the results of some expression are not computable, for example `0/0` or `Inf - Inf`). R uses `NaN` to denote not a number.

† The value `NA` is of mode `"logical"`. Despite the assurances of sections 2.1.1, R can actually handle three valued logic. Try the following, construct the truth tables and try to work out what is going on.

```
> lvec1 <- rep(c(TRUE, FALSE, NA), times=3)
> lvec2 <- rep(c(TRUE, FALSE, NA), each=3)
> lvec1 & lvec2
> lvec1 | lvec2
```

3.2.6 Coercion and class checking (`as` and `is`)

Some functions will only accept objects of a particular mode or class. It is useful to be able to convert (*coerce*) an object from one class to another. The generic function for this purpose is `as(...)`. The first argument is the object that we would like to convert and the second is the class we would like to convert to. There are a number of *ad hoc* functions of the form `as.`*x* that perform a similar task. Particularly useful are `as.numeric(...)` and `as.vector(...)`. The function `is(...)` and similar `is.`*x*`(...)` functions check that an object is of the specified class or type, for example `is.logical(...)` and `is.numeric(...)`. [There are two functions of the form `is.`*x*`(...)` command that do not fit this pattern: `is.element(...)` checks for membership of a set; `is.na(...)` checks for missing values element by element.]

```
> is.vector(roots.of.two)
> is.character(roots.of.two)
> is(roots.of.two, "character")
> as.character(roots.of.two)
> is.logical(lvec1)
> as(lvec1, "numeric")
> is(as(lvec1, "numeric"), "numeric")
```

3.2.7 Arrays and matrices

An array in R consists of a data vector that provides the contents of the matrix and a dimension vector. A matrix is a two dimensional array. The dimension vector has two elements the number of rows and the number of columns. A matrix can be generated by the `array(...)` function with the `dim` argument set to be a vector of length 2. Alternatively, we can use the `matrix(...)` command or provide a dim attribute for a vector. Notice the direction in which the matrix is filled (columns first)

```
> m1 <- array(1:15,dim = c(3,5))
> m1
> m2 <- matrix(1:15,3,5)
> m2
> m3 <- 1:15
> dim(m3) <- c(3,5)
> m3
> c(class(m1), class(m2), class(m3))
> array(1:15, dim=c(3,6))  # recycling rule used
> array(1:15, dim=c(2,4))  # extra values discarded
```

Using arithmetic operators on similar arrays will result in element by element calculations being performed. This may be what we want for matrix addition. However, it does not correspond to the usual mathematical definition of matrix multiplication.

```
> m1+m1
> m1*m1
```

Conformal matrices (that is, matrices for which the number of columns of the first is equal to the number of rows of the second) can be multiplied using the operator `%*%`. Matrix inversion and (equivalent) solution of linear systems of equations can be performed using the function `solve(...)`. For example, to solve the system $A\boldsymbol{x} = \boldsymbol{b}$, we would use `solve(A, b)`. If a single matrix argument is passed to `solve(...)`, it will return the inverse of the matrix.

```
> m4 <- array(1:3, c(4,2))
> m5 <- array(3:8, c(2,3))
> m4 %*% m5
> m6 <- array(c(1,3,2,1),c(2,2))
> m6
> v1 <- array(c(1,0), c(2,1))
> solve(m6,v1)
> solve(m6)  # inverts m6
> solve(m6) %*% v1  # does the same as solve(m6,v1)
```

The outer product of two vectors is the matrix generated by looking at every possible combination of their elements. The first two arguments of the `outer(...)` function are the vectors. The third argument is the operation that we would like to perform (the default is multiplication). This is a simplification since outer product works on arrays of general dimension (not just vectors). The following example is taken from Venables *et. al.*'s *An Introduction to R* page. The aim is to generate the mass function for the determinant of a 2×2 matrix whose elements are chosen from a discrete uniform distribution on the set $\{0, \dots, 5\}$. The determinant is of the form $AD - BC$ where A, B, C and D are uniformly distributed. The mass function can be calculated by enumerating all possible outcomes. It exploits the `table(...)` function that forms a frequency table of its argument (notice how `plot(...)` works for objects of class `"table"`.

```
> m7 <- outer(0:5,0:5)  # every possible value of AD and BC
> freq <- table(outer(m7,m7,"-"))  # frequency for all values of AD-BC
> freq
> plot(freq/sum(freq), xlab="Determinant value", ylab = "Probability")
```

Various other matrix operations are available in R: `t(...)` will take the transpose, `nrow(...)` will give the number of rows and `ncol(...)` the number of columns. The function `rbind(...)` (`cbind(...)`) will bind together the rows (columns) of matrices with the same number of columns (rows). We can get the eigenvectors and eigenvalues of a symmetric matrix using `eigen(...)` and perform singular value decomposition using `svd(...)`. Investigating the use of these commands is left as an exercise.

3.3 Writing your own functions

We have seen in the previous two sections that there are a large number of useful function built into R; these include `mean(...)`, `plot(...)` and `lm(...)`. Explicitly telling the computer to add up all of the values in a vector and then divide by the length every time we wanted to calculate the mean would be extremely tiresome. Fortunately, R provides the `mean(...)` function so we do not have to do long winded calculations. One of the most powerful features of R is that the user can write their own functions. This allows complicated procedures to be built with relative ease.

The general syntax for defining a function is

```
name <- function(arg1, arg2, ...)  expr1
```

The function is called by using

```
name(...)
```

We will discuss the possible forms of the argument list in section 3.6. When the function is called the statements that make up *expr1* are executed. The final line of *expr1* gives the return value.

Consider the logistic map (3.1). The map could be rewritten as follows

$$x_{n+1} = g(x_n),$$

where

$$g(x) = rx(1-x).$$

Clearly, the quadratic function g plays an important role in the map. We can write R code for this function; note that we follow the usual computing convention and give this function the descriptive name `logistic` rather than calling it g.

```
> logistic <- function(r,x) r*x*(1-x)
```

Now that we have defined the function, we can use it to evaluate the logistic function for different values of r and x (including vector values).

```
> logistic(3,0)
[1] 0
> logistic(3,0.4)
[1] 0.72
> logistic(2,0.4)
[1] 0.48
> logistic(3.5, seq(0,1,length=6))
[1] 0.00 0.56 0.84 0.84 0.56 0.00
```

The expression whose statements are executed by a call to `logistic(...)` is just the single line `r*x*(1-x)`. This is also the return value of the function. The expression in a function may run to several lines. In this case the expression is enclosed in curly braces { } and the final line of the expression determines the return value. In the following function we use the `logistic(...)` function that we have just defined.

```
> lmap1 <- function(r)
+ {  temp <- logistic(r, 0.6)
+    logistic(r, temp)
+ }
> lmap1(1)
[1] 0.1824
> lmap1(3)
[1] 0.6048
```

More than one command on a line
Separate commands can be placed on a single line separated by ; for example, we could have written `temp <- logistic(r, 0.6); logistic(r, temp)`. This is best avoided unless the commands are very short.

This function `lmap1` performs a single iteration in the logistic map. Note that the argument name `r` is just a label. We can name this argument what we like.

```
> lmap1a <- function(turin)
+ {  galileo <- logistic(turin,0.6)
+    logistic(turin,galileo)
+ }
> lmap1a(1)
[1] 0.1824
> lmap1a(3)
[1] 0.6048
```

This is a silly example to prove a point; in practice it is best to use sensible names that you will be able to understand when you come back to look at the code.

We may construct a vector return value for a function.

```
> lmapvec <- function(x)
+ {  ans <- x
+    ans <- c(ans, logistic(3,ans))
+    ans
+ }
> lmapvec(0.2)
[1] 0.20 0.48
> lmapvec(0.48)
[1] 0.4800 0.7488
```

Making changes to a function
We often want to correct or make additions to an existing function. One of the easiest ways to do this is using `fix(...)`. For example, we could edit our `lmapvec(...)` function using `fix(lmapvec)`. This will open an edit window. Make the changes then close the window clicking `Yes` to save any alterations you have made.

3.4 Using side effects

In many instances we are more interested in the side effects of a function than its return value. This type of function is written in exactly the same way as other functions. The only difference is that we do not need to worry about the final line. [In fact it is polite to return a value that indicates whether the function has been successful or not but this will not concern us for now.]

```
> logisticplot <- function(r,lower,upper,npoints)
+ {  x <- seq(lower,upper,length=npoints)
+    plot(x,logistic(r,x), type = "l")
+ }
> logisticplot(3,0,1,10) # a bit ropey (not enough points)
> logisticplot(3,0,1,100) # better
```

3.5 Flow control

3.5.1 Loops – `for`

A `for` loop often provides the most obvious implementation.

`for (`*`loopvariable`* `in` *`sequence`*`)` *`expr1`*

Here `sequence` is actually any vector expression but usually takes the form of a regular sequence such as `1:5`. The statements of ***`expr1`*** are executed for each value of the loop variable in the sequence. An couple of examples follow.

```
> for (i in 1:5) print(i)

> load("moreRessentials.Rdata")
> attach(mk2nd)
> for (i in 1:length(exam1))
+ {  ans <- exam1[i] + exam2[i] + exam3[i]
+    cat(row.names(mk2nd)[i], " total:  ", ans, "\n")
+ }
```

3.5.2 Conditional statements – `if`

The `if` statement in R follows the standard syntax given in section 3.1.2.

`if (`*`condition`*`)` *`ifbranch`*
`if (`*`condition`*`)` *`ifbranch`* `else` *`elsebranch`*

Here the ***`condition`*** is an expression that yields a logical value (`TRUE` or `FALSE`) when evaluated. This is typically a simple expression like `x > y` or `dog == cat`. A couple of examples follow to illustrate the difference between `if` and `if - else` statements.

```
> for (i in 1:5)
+ {  cat(i, ":  ")
+    if (i<3) cat("small")
+    cat(" number \n")
+ }

> for (i in 1:5)
+ {  cat(i, ":  ")
+    if (i<3) cat("small")
+    else cat("big")
+    cat(" number \n")
+ }
```

As a slightly more substantial example, suppose that in order to pass student must achieve a total mark of 60 or higher. We can easily write R code to tell us which students have passed.

```
> for (i in 1:length(exam1))
+ {  ans <- exam1[i] + exam2[i] + exam3[i]
+    cat(row.names(mk2nd)[i], ":  ")
+    if (ans >= 60) cat("PASS \n")
+    else cat("FAIL \n")
+ }
```

In section 3.1.2 we mention the fact that loops are not efficiently implemented in R. One way of avoiding the use of loops is to use commands that operate on whole objects. For example, `ifelse(...)` is a conditional statement that works on whole vectors (rather than requiring a loop to go through the elements).

`ifelse(`*`condition`*`,`*`vec1`*`,`*`vec2`*`)`

If `condition`, `vec1` and `vec2` are vectors of the same length, the return value is a vector whose i^{th} elements if `vec1[i]` if `condition[i]` is true and `vec2[i]` otherwise. If `condition`, `vec1` and `vec2` are of different lengths, the recycling rule is used. We exploit recycling in this version of the PASS/FAIL example above.

```
> pf <- ifelse(exam1+exam2+exam3>=60,"PASS","FAIL")
> pf
```

This is not quite what we want. In the previous example we had candidate numbers associated with the PASS/FAIL results. We will return to this example in section 3.5.4.

3.5.3 More loops – `while` and `repeat` †

We can achieve the same result as a `for` loop using either a `while` or `repeat` loop.

`while (condition)` *`expr`*
`repeat` *`expr`*

A `while` loop continues execution of the expression while the condition holds true. A `repeat` loop repeated executes the expression until explicitly terminated. We illustrate with the trivial example from section 3.5.1 that just prints out numbers from 1 to 5.

```
> j <- 1
> while (j<6)
+ {  print(j)
+    j <- j+1
+ }

> j <- 1
> repeat
+ {  print(j)
+    j <- j+1
+    if (j>5) break
+ }
```

Note that in these examples we have to explicitly initialise and increment the value of our loop variable `j`. The `break` command can be used in any type of loop to terminate and move execution to the final } of the loop expression. In a `repeat` loop, `break` is the only way to terminate the loop.

A `while` loop can be useful when we have a condition for termination of the loop rather than an exact point. For example, suppose that we want to get the first 10 pass marks from the `mk2nd` data. We do not know exactly how far through the data we must look to find 10 pass marks. We could write a for loop with a break statement but a while loop is neater.

```
> npass <- 0
> j <- 1
> tot <- exam1+exam2+exam3
> while (npass < 10)
+ {  if (tot[j] >= 60)
+    {  npass <- npass+1
+       cat(row.names(mk2nd)[j], ":  ", tot[j], "\n")
+    }
+    j <- j+1
+ }
```

3.5.4 Vectorization and avoiding loops

Loops are an important part of any programming language. However, as we have mentioned before, loops are very inefficiently implemented in R (S is no better). Having learnt about loops, it is now important to learn how to avoid them where ever possible. We return to previous examples to demonstrate how the same operations can be performed using whole vectors rather than running through the indices using a loop. First the example in section 3.5.1. It is easy to avoid using loops here by adding the vectors and using the `cat(...)` and `paste(...)` functions to get the correct output.

```
> ans <- exam1 + exam2 + exam3
> cat(paste(row.names(mk2nd), "total:", ans), fill=15)
```

Not the `fill` argument of the `paste(...)` determines the width of the output. This allows us to get the output as a column (of the appropriate width). In section 3.5.2 we had nearly solved the vectorization problem but the output was not quite what we wanted. This problem is solved below.

```
> pf <- ifelse(ans>=60, "PASS", "FAIL")
> cat(paste(row.names(mk2nd), ":", pf), fill=12)
```

We now consider an example that involves the use of two functions that we have not encountered before. The function `scan(...)` is used to read vectors in from file while `apply(...)` allows us to apply functions to whole sections of arrays (a good way of avoiding loops). The `apply` function has three arguments, the first is an array (or matrix) and the third is the function that we would like to apply. The second argument gives the index of the array we would like to apply the function to. In matrices an index value of `1` corresponds to rows while an index value of `2` corresponds to columns. In the following example we scan in a set of values of the NASDAQ index for 23 days in October 2003. For each day we have 39 reading (intradaily data, ten minute readings from 8:40 to 15:00). We construct a matrix of log returns, remove the first row (why?) and work out the standard deviation of the log returns for each of the remaining 38 time points (8:50 to 15:00). Try to interpret the resulting plot.

```
> nas <- scan("NDoct2003.dat")
> naslr <- c(NA,log(nas[1:length(nas)-1])-log(nas[2:length(nas)]))
> nasmat <- array(naslr, c(39,23))
> nasmat <- nasmat[2:39,1:23]
> timesd <- apply(nasmat,1,sd) # calculate st.dev.  of rows
> length(timesd)
> plot(timesd, type="l")
```

3.6 Functions with default values*

We can exploit the ideas of named arguments and default values from 2.2.2 in functions that we write ourselves. Consider a function to draw the binomial mass function for some values of number of trial n (`size`) and probability of success p (`prob`).

```
> binomplot <- function(size, prob, colour, outputvals)
+ {  x <- 0:size
+    y <- dbinom(x, size, prob)
+    plot(x, y, type="h", col=colour)
+    if (outputvals) y
+ }

> binomplot(20, 0.2, 2, TRUE)
```

The argument `colour` is a number to specify the colour of the plotted lines while `outputvals` is a logical value; if `outputvals` is `TRUE` R will print out the values used to generate the plot. In general, we may want green lines and not printing out all of the values used to generate the plot. If this is the case, we can specify default values for these arguments.

```
> binomplot <- function(size, prob=0.5, colour=3, outputvals=FALSE)
+ {  x <- 0:size
+    y <- dbinom(x, size, prob)
+    plot(x, y, type="h", col=colour)
+    if (outputvals) y
+ }
```

Notice that the body of the function is unchanged. You can make these alterations easily using `fix(binomplot)`. Experiment with passing different arguments to the function.

```
> binomplot(20, 0.2, 2, TRUE)
> binomplot(20)
> binomplot(100)
> binomplot(100, prob=0.9)
> binomplot(100, 0.9, 4)
> binomplot(55, outputvals=TRUE, colour=1)
```

3.7 Functions as arguments*

It is often useful to be able to pass functions as arguments. For example, we may want an R function that can plot any mathematical function (not just binomial mass). In R, we treat an argument which is a function in an identical way to any other argument. The general plotting function below will plot the values of a function `ftoplot` for a specified set of `x` values.

```
> genplot <- function(ftoplot, x=seq(-10,10,length=200),
+ ptype="l", colour=2)
+ {  y <- ftoplot(x)
+    plot(x, y, type=ptype, col=colour)
+ }

> genplot(sin)
> genplot(sin, ptype="h")
> cubfun <- function(x) x^3-6*x-6
> genplot(cubfun, x=seq(-3,2,length=500))
```

This works well for plotting a (mathematical) function with a single argument, however, in many instances we need to pass more information to the (mathematical) function to be plotted. For example, to generate binomial mass using a general function like `genplot(...)`, we would like to be able to pass the values of n and p. This can be done by using a `...` argument. Any arguments that cannot be identified as arguments of the main function will be passed to the location of the `...`

```
> genplot <- function(ftoplot, x=seq(-10,10,length=200),
+ ptype="l", colour=2, ...)
+ {  y <- ftoplot(x, ...)
+    plot(x, y, type=ptype, col=colour)
+ }

> genplot(cubfun, x=seq(-4,3,length=500))
> genplot(dbinom, x=(0:20), prob=0.8, size=20, ptype="h")
```

3.8 Binary operators*

Binary operators are familiar from mathematics. The usual arithmetic operators such as $\times$ and $+$ could be viewed as functions with two arguments. Convention dictates that we write $1+2$ rather than $+(1,2)$. R syntax (sensibly) follows this convention; using binary operators with the arguments either side of the function name is much easier for most of us to understand. Other binary operators include matrix mulitplication `%*%` and outer product `%o%`. R allows us to write our own binary operators of the form `%name%`. Examples follow (note that the `""` around the function name are not used when the function is called).

```
> "%p%" <- function(y,x) plot(x, y, type="l", col=2)
> x <- seq(0.1, 20, length=400)
> log(x) %p% x
> (0.3*cos(x) + 0.7*sin(2*x)) %p% x

> "%r%" <- function(y,x)
+ {  lmstore <- lm(y ~ x)
+    lmstore$coefficients
+ }
> exam2 %r% exam1
```

3.9 Glossary for chapter 3

- **Storing commands** `savehistory(...)`
  `loadhistory(...)`
  `source(...`
- **Packages** `library(...)` list and load packages
  `data(...)` add data from a package to workspace
- **Search path** `search()` lists environments in the search path in the order in which they will be searched. Passing the position in search path or name of an environments to `objects(...)` lists objects in that environment.
- **User defined functions** `function(...){...}` the contents of the {} is a sequence of commands that form the function.

- **Formatting output** `cat(...)` prints as character string
  `format(...)` flexible formatting function
- **Coercion** `as.`*x*`(...)`
  `as(...)`
- **Class checking** `is.`*x*`(...)`
  `is(...)`
- **Loops** `for(...)`
  `while(...)`
  `repeat`
- **Arrays** `matrix(...)` to generate a matrix (2D array)
  `array(...)` to generate a general array
- **Conditional statements** `if (...)`
  `else`
  `ifelse(...)` vector conditioning

3.10 Exercises

1. Determinants for matrices with random components: In this example we generalize the example given at the end of section 3.2.7.
   (a) Write a function to generate a plot of the mass function for the determinant of a 2×2 matrix with elements that are uniformly distributed on the set $\{0, \dots, 5\}$.
   (b) Generalize the function written above so that it will work on an arbitrary set of integers.
   (c) Provide an argument to the function that allows us (if we choose) to print out the frequency table on which the plot is based.
2. Back-shift function: this is a useful function in time series analysis. It is defined as B where $BY_t = Y_{t-1}$ and $B^d Y_t = Y_{t-d}$.
   (a) Write a back-shift function that takes a vector as its argument and returns the back-shifted version of the vector. The first element of the returned vector should be `NA`. Do not use loops! Check that your function works.
   (b) Write a version of the function that back-shifts for an arbitrary number of times d. Set a default value for d.
3. Cobweb plot: One way to view the iterations of the logistic map is using a cobweb plot.
   (a) Write an R function to plot the logistic function and add the straight line $y = x$ to the plot (use the `lines()`) command.
   (b) Edit your function so that, from a starting point of $x = 0.1$, it draws a line up (vertically) to the logistic curve and then over (horizontally) to the line $y = x$.

(c) Include a loop to take a line vertically from the current point to the logistic curve then horizontally over to the line $y = x$ a fixed number of time (thus constructing the cobweb plot).

(d) Is there anyway to do this without using a loop?

4. Conditional likelihood for autoregressive process: A Gaussian autoregressive process of order 1 is a simple time series model in which the current observation is modelled explicitly as a function of the previous observation:

$$Y_t = \phi Y_{t-1} + \varepsilon_t,$$

where $\varepsilon \sim N(0, \sigma_\varepsilon^2)$. The log-likelihood for observations $\{y_1, \dots, y_n\}$ is approximated by

$$\ell(\phi, \sigma_\varepsilon^2) = -\frac{n}{2} \log \sigma_\varepsilon^2 - \frac{1}{2\sigma_\varepsilon^2} \sum_{i=1}^{n} (y_t - \phi y_{t-1})^2.$$

(a) The file (saved workspace) `arsim.Rdata` contains a data frame (`simdata`) of values simulated from an AR(1) with ϕ taking values between -0.9 and 0.9. Use the `load(...)` command to add the objects from the file to your workspace. Write a function that will generate a time series plot of any named columns from a data frame. Test out your function on the `simdata` data frame.

(b) Write a function that evaluates the likelihood for values of ϕ and σ_ε^2 for a specified data vector.

(c) Write a function that uses the function from part ii. to draw the likelihood surface for a given region of the parameter space.

5. † Bifurcation plot: The convergence properties of the logistic map for different values of r are summarized in a bifurcation plot. The bifurcation plot has values of r on the x axis and the values that the logistic map converges to on the y axis. For small values of r the map converges to a single value. As r increases we see period doubling and then chaotic regions. By repeated running the logistic map for a fixed number of iterations and plotting the last 50 values, write a function that will draw a bifurcation plot.

6. †† Mandelbrot set: The Mandelbrot set is a subset of the complex plane. Consider the map

$$z_{n+1} = z_n^2 + c \tag{3.2}$$

where $z_0 = c$. The Mandelbrot set consists of those points c in the complex plane for which the limit as n tends to infinity of $|z_n|$ is finite. Write an R function to construct a reasonable visual impression of the Mandelbrot set.

3.11 Reading

3.11.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 5 for arrays and matrices, chapter 9 for program control and chapter 10 for writing functions.]

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition, Springer. [Section 3.5 for formatting a printing.]

3.11.2 Supplementary reading†

- Venables, W. N. and Ripley, B. D. (2000) *S Programming*, Springer. [Much of what is said here works in R.]
- Bentley, J (2000) Programming Pearls, 2nd edition. Addison-Wesley. [An entertaining book that contains a lot of wisdom about programming.]

Chapter 4

Distributions and simulation

4.1 Key Ideas

4.1.1 Distribution, mass and density

The distribution of probability associated with a random variable Y, is given by the distribution function, $F_Y : \mathbb{R} \to [0,1]$ defined by

$$F_Y(y) = P(Y \leq y).$$

Distributions are often classified as either discrete (F_Y is a step function) or continuous (F_Y is a continuous function). Note that real observations are never made on a continuous scale.

For a discrete distribution there is a countable set of possible values of the variable, $\{y_1, y_2, \ldots\}$ (the points at which the discontinuities in F_Y occur). The probability mass function is defined by

$$f_Y(y) = P(Y = y)$$

and takes non-zero values on $\{y_1, y_2, \ldots\}$. The distribution function can then be written as

$$F_Y(y) = \sum_{i: y_i \leq y} f_Y(y_i).$$

Some commonly used mass functions are given below:

binomial(n, p)	$f_Y(y) = \binom{n}{y} p^y (1-p)^{(n-y)}$ for $y = 0, \ldots, n$
Poisson(λ)	$f_Y(y) = \lambda^y e^{-\lambda}/y!$ for $y = 0, 1, \ldots$
discrete uniform$\{1, \ldots, m\}$	$f_Y(y) = 1/m$ for $y = 1, \ldots, m$

For a continuous distribution the density function, f_Y, is a positive real-valued function satisfying

$$F_Y(y) = \int_{-\infty}^{y} f_Y(u) du.$$

Probabilities can be calculated from the density function by integration:

$$P(a < Y \leq b) = \int_a^b f_Y(y)dy.$$

Some commonly used densities are given below:

normal(μ, σ^2)	$f_Y(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mu)^2/(2\sigma^2)}$ for $y \in \mathbb{R}$
exponential(θ)	$f_Y(y) = \theta e^{-\theta y}$ for $y > 0$
continuous uniform$[a, b]$	$f_Y(y) = 1/(b-a)$ for $y \in [a, b]$

4.1.2 Moments and the expectation operator

In what follows we assume that Y is a continuous random variable (in the discrete case, summation replaces integration). The expected value value of Y is defined as

$$E(Y) = \int_{-\infty}^{\infty} y f_Y(y)dy$$

whenever this integral exists. Expectation is linear in the sense that for any constants a and b

$$E(a + bY) = a + bE(Y).$$

For any reasonably well behaved function g we can define the expectation

$$E(g(Y)) = \int_{-\infty}^{\infty} g(y) f_Y(y)dy.$$

In particular, we may be interested in the moments $E(Y^2)$, $E(Y^3)$, ... and the central moments $E[(Y - E(Y))^2]$, $E[(Y - E(Y))^3]$, ... and so on. For example, the second central moment is the variance:

$$\text{Var}(Y) = E[(Y - E(Y))^2] = E(Y^2) - E(Y)^2.$$

The variance operator has the property

$$\text{Var}(a + bY) = b^2\text{Var}(Y).$$

4.1.3 Bivariate distributions, covariance and correlation

We are often interested in the relationship between random variables. For two random variables X and Y we define the joint distribution as

$$F_{X,Y}(x, y) = P(X \leq x \text{ and } Y \leq y).$$

The joint density is a positive real valued function, $f_{X,Y}$, satisfying

$$F_{X,Y}(x, y) = \int_{-\infty}^{y} \int_{-\infty}^{x} f_{X,Y}(u, v)dudv.$$

For functions of two variable, expectation is defined by

$$E(h(X,Y)) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(x,y) f_{X,Y}(x,y)dxdy.$$

It is clear from this definition that for any random variables with finite means,

$$E(X+Y) = E(X) + E(Y).$$

However,

$$\text{Var}(X+Y) = \text{Var}(X) + 2\text{Cov}(X,Y) + \text{Var}(Y),$$

where

$$\text{Cov}(X,Y) = E(X-E(X))(Y-E(Y)) = E(XY) - E(X)E(Y).$$

Covariance is used as a measure of the linear association between X and Y. It can be scaled to give correlation

$$\text{Corr}(X,Y) = \frac{\text{Cov}(X,Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}.$$

We say that X and Y are independent when there exist function F_X and F_Y such that

$$F_{X,Y}(x,y) = F_X(x)F_Y(y)$$

for all x, y. It then holds that

$$\text{Var}(X+Y) = \text{Var}(X) + \text{Var}(Y).$$

4.1.4 Samples, models and reality

An observed sample, $y_1, \ldots, y_n$, from a population is a collection of numbers. These numbers are often referred to as the *data*. The data can be thought of as one *instance* of a collections of random variables, $Y_1, \ldots, Y_n$. The random variables, $Y_1, \ldots, Y_n$, form our *model* for the data. The model is an attempt to represent mathematically the process that generates the data. It is important to remember that the model is something that we impose – the model is not the real data generating process. We focus initially on two models that are so commonly used it is easy to forget that they are just models.

i. A random sample from a population is a sample in which every member of the population has equal probability of being included. A random sample of size n from a population can be modelled by a set of independent and identically distributed random variable, $Y_1, \ldots, Y_n$. We often use Y (with no subscript) to denote a random variable the has the same distribution as all of the Y_is (this makes sense since the Y_is are identically distributed). Parameters of interest include the population mean, $\mu_Y = E(Y)$, and the population variance, $\sigma_Y^2 = \text{Var}(Y)$.

ii. A sample of size n from a population that is thought to depend linearly on fixed explanatory factors, $x_1, \ldots, x_n$, can be modelled by the simple regression,

$$Y_i = \alpha + \beta x_i + \varepsilon_i, \quad \{\varepsilon_t\} \sim iN(0, \sigma_\varepsilon^2). \tag{4.1}$$

where $iN(0, \sigma_\varepsilon^2)$ denotes normally and independently distributed with mean 0 and variance σ_ε^2. Parameters of interest include the intercept α, the slope β and the error variance σ_ε^2.

4.1.5 Properties of statistics

The value of a statistic is a function of the data; to represent this general relationship we use the notation

$$u = h(y_1, \dots, y_n).$$

The term statistic refers to the same function h applied to the random variables $Y_1, \dots, Y_n$,

$$U = h(Y_1, \dots, Y_n).$$

Just as the data, $y_1, \dots, y_n$, are thought of as one instance of the model, $Y_1, \dots, Y_n$, the value u is taken to be one instance of the statistic U. The table below gives some statistics and their sample values.

	statistic (U)	observed value (u)
sample mean	$\frac{1}{n}\sum_{i=1}^{n} Y_i \; (= \bar{Y})$	$\frac{1}{n}\sum_{i=1}^{n} y_i \; (= \bar{y})$
sample variance	$\frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \bar{Y})^2$	$\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2$
1st order statistic	$\min_i Y_i \; (= Y_{(1)})$	$\min_i y_i \; (= y_{(1)})$
rth order statistic	$Y_{(r)}$	$y_{(r)}$
sample median	$Y_{((n+1)/2)}$	$y_{((n+1)/2)}$
slope estimator ($\hat{\beta}$)	$\sum_{i=1}^{n}(Y_i - \bar{Y})(x_i - \bar{x})/(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2)$	$\sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x})/(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2)$

In all cases, U is a RANDOM VARIABLE. As such it makes sense to talk about the distribution of U and its moments, $E(U)$, $\text{Var}(U)$, etc. Many of the statistics that we are interested in are *estimators* of population parameters; referring to the table above, the sample mean, $\bar{Y}$, is used as an estimator of the population mean, μ_Y, the slope estimator, $\hat{\beta}$, is used as an estimator of the slope parameter, β. The observed value of the statistic is used as an estimate of the population parameter. Note the distinction, an estimat*or* is a model quantity (a random variable) while an estimat*e* is an observed quantity (a number).

We have an intuitive understanding of what constitutes a good estimator; a good estimator will provide estimates that are close to the true value. As statisticians, we would like to make this notion more precise. Suppose that U is an estimator for λ. We define two criteria that can be used to judge the quality of an estimator. Note that evaluation of these quantities is based on our model.

- $\text{Bias}(U) = E(U - \lambda) = E(U) - \lambda$. Bias measures how far our estimate will be from the true value on average. Unbiasedness ($E(U) = \lambda$) is usually taken to be a desirable property for an estimator.
- $\text{Var}(U) = E[(U - E(U))^2] = E(U^2) - E(U)^2$. Variance measures the spread of the estimator. An estimator with low variance (an *efficient* estimator) is usually desirable.

Neither bias nor variance alone provide an indication of the quality of an estimator; we do not want a unbiased estimator with a large variance, nor do we want an efficient estimator with a large bias. However, these measures can be combined to give a single measure of the quality of an estimator.

- $MSE(U) = E[(U-\lambda)^2] = (\text{Bias}(U))^2 + \text{Var}(U)$. Mean square error is a widely used measure of the quality of an estimator.

FTDSN – Measuring quality of estimators
The relationship between mean square error, bias and variance is easy to prove - try it. There are many other possible measures of the quality of estimators. Try to invent a sensible measure and then find out whether someone else has already thought of it.

The properties of some statistics are (at least partly) analytically tractable. Two well known examples follow.

Sample mean of a random sample: The mean and variance of the sample mean of a random sample are easy to evaluate under the usual model. Suppose that $Y_1, \ldots, Y_n$ are independent identically distributed random variables with mean μ_Y and variance σ_Y^2.

$$E(\bar{Y}) = E\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \frac{1}{n}\sum_{i=1}^{n} E(Y_i) = E(Y) = \mu_Y$$

The mean of the sample mean is just the population mean. Thus, the sample mean is an *unbiased estimator* of the population mean. The derivation of the variance exploits independence.

$$\begin{aligned} \text{Var}(\bar{Y}) &= \text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(Y_i) \text{ by independence} \\ &= \frac{1}{n}\text{Var}(Y) = \frac{1}{n}\sigma_Y^2. \end{aligned}$$

The variance of the sample mean decreases as the sample size increases, a desirable property for an unbiased estimator. The asymptotic distribution of the sample mean is described by a well known theorem.

The central limit theorem: Let $Y_1, \ldots, Y_n$ be independent, identically distributed random variables with mean $E(Y) = \mu_Y$ and finite variance $\text{Var}(Y) = \sigma_Y^2 < \infty$. Let

$$Z_n = \frac{\bar{Y} - \mu_Y}{\sigma_Y/\sqrt{n}},$$

then Z_n converges in distribution to a standard normal as $n \to \infty$.

There is a neat proof of this theorem using cumulant generating functions. The central limit theorem is used to provide an approximate distribution for the sample mean; in a large sample, the sample mean is approximately normally distributed regardless of the distribution of the population (or to be more precise, provided the population has finite variance). The central limit theorem is a remarkable result that is used routinely in testing hypotheses about the population mean. However, it is an asymptotic result; the normal distribution will only be a good approximation to the distribution of the sample mean in large samples. In section 4.6 we use simulation to investigate the small sample properties of the sample mean.

Least squares estimate of slope in regression For the simple regression model (4.1) the least squares estimator of the slope is

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(Y_i - \bar{Y})(x_i - \bar{x})}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}.$$

We can show (try it) that

$$E(\hat{\beta}) = \beta \text{ and } \operatorname{Var}(\hat{\beta}) = \frac{\sigma_\varepsilon^2}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}.$$

A useful statistic for tests involving the slope is

$$\frac{\hat{\beta} - \beta}{S_\varepsilon / \sqrt{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}} \sim t_{n-2}$$

where

$$S_\varepsilon^2 = \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{n-2} \text{ and } \hat{Y}_i = \hat{\alpha} + \hat{\beta}x_i.$$

Here $\hat{\alpha} = \bar{Y} - \hat{\beta}\bar{x}$ is the estimator of α.

4.1.6 Interval estimation

Suppose that we have a model, $Y_1, \ldots, Y_n$, parameterised by λ. In section 4.1.5 we discuss the properties of estimators that yield point estimates. It is also possible to generate interval estimates. For $\alpha \in [0,1]$, consider statistics V and W (functions of $Y_1, \ldots, Y_n$) for which

$$P(V > \lambda) = \alpha/2 \text{ and } P(W \leq \lambda) = \alpha/2.$$

It is clear that

$$P(V \leq \lambda < W) = 1 - P(V > \lambda) - P(W \leq \lambda) = 1 - \alpha.$$

If we replace V and W with their sample values v and w, the resulting interval $[v, w)$ is referred to as a two-sided $100(1-\alpha)\%$ confidence interval.

For example, consider $Y_1, \ldots, Y_n$ independent $N(\mu_Y, \sigma_Y^2)$ random variables. We know that

$$\frac{\bar{Y} - \mu_Y}{S_Y/\sqrt{n}} \sim t_{n-1}.$$

By definition

$$P\left(\frac{\bar{Y} - \mu_Y}{S_Y/\sqrt{n}} > t_{n-1,\alpha/2}\right) = \alpha/2 \text{ and } P\left(\frac{\bar{Y} - \mu_Y}{S_Y/\sqrt{n}} \leq -t_{n-1,\alpha/2}\right) = \alpha/2.$$

Rearranging yields

$$P\left(\bar{Y} - t_{n-1,\alpha/2}S_Y/\sqrt{n} > \mu_Y\right) = \alpha/2 \text{ and } P\left(\bar{Y} + t_{n-1,\alpha/2}S_Y/\sqrt{n} \leq \mu_Y\right) = \alpha/2.$$

Thus, the $100(1-\alpha)\%$ confidence interval for μ_Y is the familiar

$$[\bar{y} - t_{n-1,\alpha/2}S_y/\sqrt{n},\ \bar{y} + t_{n-1,\alpha/2}S_y/\sqrt{n}).$$

4.1.7 Principles of simulation

Simulation involves using a computer to generate observed samples from a model. In many fields this model is a complex deterministic system that does not have a tractable analytic solution, such as, the models of the atmosphere used in weather forecasting. A statisticians, we use simulation to provide insight into properties of statistics when these properties are not available from theory (it should be noted that simulation is usually a method of last resort). For simulated data we know the data generating process. The starting point for a simulation experiment is to simulate observations from the stochastic model of interest; for example,

i. instances, $y_1, \dots, y_n$, from a sequence of independent identically distributed random variables, $Y_1, \dots, Y_n$, with a given distribution (the usual model for a random sample),

ii. instances, $y_1, \dots, y_n$, from the sequence of random variables, $Y_1, \dots, Y_n$, defined by the simple regression model (4.1)

The simulated observations can be used to construct a simulated instance of a statistic; for example

i. sample mean, $\bar{y} = \sum_{i=1}^{n} y_i$,

ii. estimate of the slope parameter, $\hat{\beta} = \sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x})/(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2)$.

The process of simulating a sample can be repeated many times to build up a collection of samples known as simulated replications. Evaluating a statistic for each replication allows us to build up a picture of the distribution of the statistic. Consider repeatedly generating instances from the model and evaluating the statistic U for each replication. This process will yields

$$\begin{array}{lcl} y_1^{(1)}, \dots, y_n^{(1)} & \rightarrow & u^{(1)} \\ y_1^{(2)}, \dots, y_n^{(2)} & \rightarrow & u^{(2)} \\ \vdots & & \vdots \\ y_1^{(r)}, \dots, y_n^{(r)} & \rightarrow & u^{(r)} \end{array}$$

where $y_i^{(j)}$ is the jth replication of the ith sample member and $u^{(j)}$ is the jth replication of the statistic of interest. Here we use r to denote the total number of replications. The values $u^{(1)}, \dots, u^{(r)}$ can be thought of as r instances of the random variable U.

The design of a simulation experiment is crucial to its success; it is easy to waste time on inefficient or total meaningless simulations.

1. A simulation experiment should have a clearly stated objective. It is often helpful to think of a simple question (or set of questions) that we would like to answer. Interesting simulation experiments usually involve some sort of comparison; for example:

   - comparison of the relative merits of different estimators,

- comparison of the properties of a statistic for different values of the parameters of the population distribution.

2. As part of the design we must decide:

   - which factors we are going to make comparisons across,
   - the range of values of the factors that we are going to consider.

   Some possible sources of interesting comparison are listed below.

   (a) sample size: usually increasing the sample size will improve the properties of a statistic,

   (b) parameters of the population distribution: a statistic may perform better for certain parameter values.

   (c) the population distribution: a statistic's properties may change dramatically (or remain surprisingly stable) when the population distribution is changed.

3. The number of simulated replications: A large value of r should improve the accuracy of our simulation. However, large values of r will increase the computational cost and there may be better ways of improving our simulation.

4. Measuring the properties of simulated replications: Simulation generated a collection of values, $u^{(1)}, \dots, u^{(r)}$, from which we draw conclusions about the distribution of the statistic of interest, U. Some of the properties of U that we might want to attempt to measure using simulation are listed below.

   (a) If U is an estimator of a population parameter λ, the following approximations can be generated from our simulation:

   - Bias: $\frac{1}{r}\sum_{j=1}^{r}(u^{(j)} - \lambda) = \bar{u} - \lambda$
   - Variance: $\frac{1}{r}\sum_{j=1}^{r}(u^{(j)} - \bar{u})^2 = \frac{1}{r}\sum_{j=1}^{r}(u^{(j)})^2 - \bar{u}^2$
   - Mean square error: $\frac{1}{r}\sum_{j=1}^{r}(u^{(j)} - \lambda)^2$

   (b) We may want to test whether U has a particular distribution. The usual χ^2 goodness of fit tests can be applied to our simulated statistics, $u^{(1)}, \dots, u^{(r)}$. For some distributions, specific tests may be appropriate; for example, the Shapiro-Wilk test for normality.

   (c) Graphical comparisons are also useful. In particular, parallel box plots provide a good means of comparing two or more sets of simulated statistics.

4.2 Notation for distributions in R

R supports calculations for a large number of distributions. All of the commonly known distributions are handled. The following table is a selection from page 51 of Venables *et. al.*'s *An Introduction to R.*

Distribution	R name	additional arguments
binomial	`binom`	`size, prob`
chi-squared	`chisq`	`df, ncp`
exponential	`exp`	`rate`
gamma	`gamma`	`shape, scale`
normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
Student's t	`t`	`df, ncp`

The additional arguments are mostly self-explanatory. The `ncp` stands for non-centrality parameter and allows us to deal with non-central χ^2 and non-central t distributions.

4.3 Probability and quantile calculations

R can be used like a set of statistical tables, that is, for a random variable Y, work out the values of p and q associated with expressions of the form

$$P(Y \leq q) \geq p.$$

In order to calculate probabilities from points on a distribution, we put a `p` in front of the distribution name. The function

`p`*`distributionname`*`(q,...)`

will return the probability that the distribution named takes a value less than or equal to q. The follow simple example illustrates.

Suppose $X \sim \text{Bin}(85, 0.6)$ and $Y \sim N(12, 9)$ and that X and Y are independent. Calculate

i. $P(X \leq 60)$

ii. $P(X \geq 60)$

iii. $P(Y < 15)$

iv. $P(Y \leq 15)$

v. $P(X < 60, Y > 15)$

```
# i.
> pbinom(60,85,0.6)
[1] 0.9837345

# ii.
> 1 - pbinom(59,85,0.6)
[1] 0.02827591
```

```
# An alternative approach
> pbinom(59,85,0.6,lower.tail=FALSE)
[1] 0.02827591

# iii. and iv. (trick question - continuous distribution)
> pnorm(15,12,3)
[1] 0.8413447

# v. (using independence)
> pbinom(59,85,0.6) * pnorm(15,12,3,lower.tail=FALSE)
[1] 0.1541691
```

We can also go in the other direction. The quantile function

`q`*`distributionname`*`(p,...)`

will return the value which the distribution named has probability p of being below. For a discrete distribution it returns the smallest number with cdf value larger than p.

Suppose $X \sim \text{Bin}(85, 0.6)$ and $Y \sim N(12, 9)$ and that X and Y are independent. Solve

i. $P(X \leq q) \geq 0.8$

ii. $P(X \geq q) < 0.6$

iii. $P(Y < q) = 0.25$

```
# i. q = 55, 56, ..., 85
> qbinom(0.8,85,0.6)
[1] 55

# ii. P(X < q-1) ≥ 0.4 ⇒ q = 51, 52, ..., 85
> qbinom(0.4,85,0.6)
[1] 50

# iii.
> qnorm(0.25,12,3)
[1] 9.97653
```

4.4 Density and cumulative distribution

R allows us to calculate density function values (probability mass in the discrete case). The function

`d`*`distributionname`*`(x,...)`

will generate the value of the density function (probability mass function) for the named distribution at x. These values can be used to density function plots.; We use this function to investigate the effect of the parameter values on the t-distribution and gamma distribution below.

- Degree of freedom of the t distribution

  ```
  > x <- seq(-3,3,length=200)
  > plot(x, dnorm(x), type="l")
  > lines(x, dt(x,df=16), col=2)
  > lines(x, dt(x,df=8), col=3)
  > lines(x, dt(x,df=4), col=4)
  > lines(x, dt(x,df=2), col=5)
  > lines(x, dt(x,df=1), col=6)
  ```

- Shape and scale of gamma

  ```
  > x <- seq(0,10,length=200)
  > plot(x, dgamma(x,shape=1,scale=1), type="l")
  > lines(x, dgamma(x,shape=1,scale=2), col=2)
  > lines(x, dgamma(x,shape=1,scale=4), col=3)
  > plot(x, dgamma(x,shape=2,scale=1), type="l")
  > lines(x, dgamma(x,shape=2,scale=2), col=2)
  > lines(x, dgamma(x,shape=2,scale=4), col=3)
  ```

In determining which distribution is appropriate for a set of data, a plot comparing the empirical density (or empirical cumulative distribution) with the theoretical density is useful. This applies to data from simulation experiments as well as observed data.

```
> allmarks <- marks2[marks2<41]
> x <- seq(5,45,length=200)
> hist(allmarks, breaks=seq(5,40,length=14), probability = TRUE)
> lines(x,dnorm(x, mean=mean(allmarks), sd = sd(allmarks)),col=2)

> library(stepfun)
> plot(ecdf(allmarks))
> lines(x,pnorm(x, mean=mean(allmarks), sd = sd(allmarks)),col=2)
```

For discrete data we may want to use a slightly different plotting method. The probability mass function of a binomial is given below using vertical lines to indicate that we only have positive probability at discrete points.

```
> x <- 0:100
> plot(x,dbinom(x,100,0.25),type="h")
```

4.5 Generating (pseudo-)random samples

Computer programs cannot produce genuine random numbers; for given input we can, in principal, predict exactly what the output of a program will be. However, we can use a computer to generate a sequence of pseudo-random numbers (as defined in section 3.1.3). The distinction is somewhat artificial since no finite sequence of numbers is truly random; randomness is a property that can only really be attributed to mathematical models (random variables). From this point on we will drop the *pseudo* with the understanding that, whenever we use the word random in the context of numbers or samples, what we mean is something that appears random; see Knuth, *Seminumerical Algorithms* for an interesting discussion of this issue.

Using R we can generate random instances from any commonly used distribution. The function

`r`*`distributionname`*`(n,...)`

will return a random sample of size n from the named distribution. At the heart of this function, R uses some of the most recent innovations in random number generation. We illustrate by sampling from Poisson and normal distributions.

```
> poissamp <- rpois(400, lambda=2)
> hist(poissamp, breaks=0:10, probability=TRUE)

> normsamp <- rnorm(250, mean=10, sd=5)
> hist(normsamp, breaks=seq(-10,30,length=15), probability=TRUE)
> x <- seq(-10,30,length=200)
> lines(x, dnorm(x, mean=10, sd=5), col=2)
```

The random number generator works as an iterative process. Thus, consecutive identical commands will not give the same output.

```
> rnorm(5)
[1] 0.4874291 0.7383247 0.5757814 -0.3053884 1.5117812
> rnorm(5)
[1] 0.38984324 -0.62124058 -2.21469989 1.12493092 -0.04493361
```

The command `set.seed(...)` allows us to determine the starting point of the iterative process and thus ensure identical output from the random number generator. This is useful when developing the code for a simulation experiment.

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

The functions `sample(...)` can be used to generate random permutations and random samples from a data vector. The arguments to the function are the vector that we would like to sample from and the size of the vector (if the size is excluded a permutation of the vector is generated). Sampling with replacement is also possible using this command.

```
> nvec <- 10:19
> sample(nvec)
> sample(nvec, 5)
> sample(nvec, replace=TRUE)
> sample(nvec, 20, replace=TRUE)
> cvec <- c("Y","o","u","r","n","a","m","e")
> sample(cvec)
> cat(sample(cvec), "\n")
```

† Most random number generators exploit simple recursions such as the logistic map discussed in section 3.3. In fact, the logistic map with $r = 4$ has some desirable properties as a random number generator. Below we generate numbers from the logistic map and look at their properties.

```
> logistic <- function(r,x) r*x*(1-x)
> pdordm <- function(seed,length)
+ {  rparam <- 4
+    nextval <- logistic(rparam,seed)
+    ans <- nextval
+    for (i in 2:length)
+    {  nextval <- logistic(rparam,nextval)
+       ans <- c(ans,nextval)
+    }
+    ans
+ }
> rnums <- pdordm(0.123,10000)
> acf(rnums)
> hist(rnums,nclass=30)
```

The number generated by the logistic map appear uncorrelated but are not uniformly distributed. A common means of generating a random sample from a uniform distribution is to use a recursion of the form

$$x_{t+1} = ax_t (\text{mod } m).$$

These are known as multiplicative congruential generators (). We can implement this generator using a quick amendment to our `pdordm` function.

```
> mcong <- function(a, m, x) (a*x)%%m
> pdordm <- function(seed,length)
+ {  aparam <- 7^5
+    mparam <- 2^31 - 1
+    nextval <- mcong(aparam,mparam,seed)
+    ans <- nextval
+    for (i in 2:length)
+    {  nextval <- mcong(aparam,mparam,nextval)
+       ans <- c(ans,nextval)
+    }
+    ans/mparam
+ }
> rnums <- pdordm(12345,10000)
> acf(rnums)
> hist(rnums,nclass=30)
```

This looks much closer to uniformity than the output of the logistic map.

4.6 Simulation experiments - an example

To demonstrate how a simulation experiment works, we are going to look at the small sample properties of the sample mean for a Poisson population, a statistic whose asymptotic properties are well known. Consider a population that has a Poisson distribution with mean λ, so $Y \sim \text{Pois}(\lambda)$. We take a sample of size n. We would like to know whether the normal distribution will provide a reasonable approximation to the distribution of the sample mean. Following the principals laid down in section 4.1.7, we first consider the questions that we would like our simulation to answer:

1. How large does n need to be for the normal distribution to be a reasonable approximation?
2. What is the effect of the value of λ?

Writing down these questions makes clear the factors that we will need to make comparisons across:

- n, the sample size: the central limit theorem tells us that for large values of n the normal will be a reasonable approximation,
- λ, the parameter of the population distribution: we might expect the shape of the underlying Poisson distribution to have an effect on the distribution of the sample mean.

We will use the computer to generate simulated samples $(y_1^{(1)}, \dots, y_n^{(1)})$, $(y_1^{(2)}, \dots, y_n^{(2)})$, $\dots$, $(y_1^{(r)}, \dots, y_n^{(r)})$, where r is the number of simulated replications. For each of these simulates samples, we evaluate the sample mean to give a sequence $\bar{y}^{(1)}, \bar{y}^{(2)}, \dots, \bar{y}^{(r)}$ that can be viewed as instances of the statistic of interest $\bar{Y}$. The first step is to write a function to generate simulated samples and, from these, simulated values of the statistic. Our first attempt uses loops.

```
> poisSampMean1 <- function(n, lambda, r)
+ {  meanvec <- c()
+    for (j in 1:r)
+    {  sampvals <- rpois(n, lambda)
+       meanvec <- c(meanvec, mean(sampvals))
+    }
+    meanvec
+ }
> set.seed(1)
> poisSampMean1(10, 3, 6)
[1] 3.3 3.4 2.6 3.0 3.3 2.6
```

It is easy to alter this program to avoid the loop. We first put $n \times r$ simulated values into a vector. We then use the `matrix(...)` function to divide this vector into r columns (each column corresponds to a simulation replication). Using `colMeans(...)` to calculate the column means will then yield r simulated instances of the sample mean.

```
> poisSampMean2 <- function(n, lambda, r)
+ {  simvals <- rpois(n*r, lambda)
+    simvals <- matrix(simvals, n, r)
+    colMeans(simvals)
+ }
> set.seed(1)
> poisSampMean2(10, 3, 6)
[1] 3.3 3.4 2.6 3.0 3.3 2.6
```

To get a visual impression of the simulated sample means we write a function to draw a histogram and plot a normal distribution with the same mean and standard deviation.

```
> histNorm <- function(data, nbins=21)
+ {  hist(data, breaks=seq(min(data), max(data), length=nbins),
+        probability=TRUE, col=5)
+    x <- seq(min(data), max(data), length=200)
+    lines(x, dnorm(x, mean=mean(data), sd=sd(data)), col=2)
+ }
```

Try experimenting with this function with various values of n and λ with $r = 1000$. If this runs very slowly, try reducing r. If you get a histogram with strange gaps in, try changing the value of the `nbins` argument.

```
> histNorm(poisSampMean2(8,1,1000))
> histNorm(poisSampMean2(100,10,1000))
```

We can use a test for normality to provide some numerical indication of whether or not the normal distribution provides good fit. The Shapiro-Wilk test is a standard test for normality. First we need a function to extract the p-value from the output of the `shapiro.test(...)`.

```
> shapiro.p <- function(data) shapiro.test(data)$p.value
```

$*$ If we feed a vector parameter to the `rpois(...)` command, the recycling rule will be used in generating random values with each value of the vector used in turn and then repeated. We exploit this below to write a function that generates a long vector of simulated instances then splits them up using factors that we define and the `tapply(...)` function. Note that this is probably taking loop avoidance a little bit too far; the solution is easy to implement using a loop (try it) and my without loop solution causes memory problems for large values of r (can you improve on it?).

```
poisSW <- function(nvec, lambdavec, r)
+ {  numn <- length(nvec); ntot <- sum(nvec)
+    numlam <- length(lambdavec)
+    simvals <- rpois(ntot*numlam*r, lambdavec)
+    lamvals <- factor(rep(lambdavec, times=ntot*r))
+    samps <- factor(rep(1:(numn*r), times=rep((nvec*numlam), each=r)))
+    simres <- as.vector(tapply(simvals, list(lamvals, samps), mean))
+    lamvals2 <- factor(rep(lambdavec, times=numn*r))
+    sims <- factor(rep(nvec, each=numlam*r))
+    tapply(simres, list(lamvals2, sims), shapiro.p)
+ }
```

In order to generate test statistic values for $n = 5$, 10, 50 and 100, with $\lambda = 1$, 3 and 10 (total of 12 combinations) we apply the function defined above.

```
poisSW(c(5,10,50,100), c(1,3,10), 2000)
```

Comment on the output of this command.

4.7 Glossary for week 4

- **Probability, quantiles, density and random numbers**

`p...`	cumulative probability
`q...`	quantiles
`d...`	density (mass)
`r...`	random values

- **Distributions**

`beta`	beta
`binom`	binomial
`cauchy`	Cauchy
`chisq`	chi-squared
`exp`	exponential
`f`	F
`gamma`	gamma
`geom`	geometric
`lnorm`	log-normal
`logis`	logistic
`nbinom`	negative binomial
`norm`	normal
`pois`	Poisson
`t`	Student's t
`unif`	uniform
`weibull`	Weibull
`wilcox`	Wilcoxon

4.8 Group Exercise

Consider the simple regression model

$$Y_t = \alpha + \beta t + \varepsilon_t, \quad \{\varepsilon_t\} \sim iN(0, \sigma_\varepsilon^2), \tag{4.2}$$

for $t = 1, \dots, n$.

1. For $n = 15$, $\alpha = 5$, $\beta = 2$ and $\sigma^2 = 1$, write a function to simulate a single instance of the $Y_1, \dots, Y_n$ from model (8.3) (this is easy to do without using a loop). Plot these value against t.
2. Write a function that takes y and x as arguments and returns $\hat{\beta}$, the estimate of the slope from regression y on x.
3. Using the same parameter values as 1, write a function to simulation 1000 instances of the model ($r = 1000$). For each simulated instance use 2 to generate the estimate of the slope parameter. Return a vector of instances of $\hat{\beta}$.
4. Using the vector of simulated instances of $\hat{\beta}$ generated in 3:
   (a) approximate the bias of $\hat{\beta}$,
   (b) approximate the variance of $\hat{\beta}$,
   (c) approximate the mean square error of $\hat{\beta}$.

   Are these values what you expected?
5. Repeat questions 3 and 4 with $\alpha = 10$. How does this alter your analysis of $\hat{\beta}$.
6. † Now consider the case where $\varepsilon_t \sim \text{Pois}(\lambda) - \lambda$, for $t = 1, \dots, n$, that is, the errors in the regression have zero mean but are not normally distributed. Leaving the other parameters unchanged, write a function to simulate simple regression with Poisson errors, generate instances of the statistic $t = (\hat{\beta} - \beta)/\text{se}(\hat{\beta})$ and perform a goodness of fit test (check the Kolmogorov–Smirnov test `ks.test`) comparing these values with an appropriate t-distribution. Compare results across different values of λ.

4.9 Reading

4.9.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 8 for distributions]
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Section 5.1 for distributions and section 5.2 for random numbers]

4.9.2 Supplementary reading†

- Knuth, D. E. (1998) *The Art of Computer Programming, Volume 2 – Seminumerical Algorithms*, 3rd edition. Addison-Wesley. [Introduction to chapter 3 for a discussion a randomness.]

- Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (2002) *Numerical Recipes in C++ – The Art of Scientific Computing*, 2nd edition. Cambridge University Press. [Section 7 for stuff on random numbers.]

Chapter 5

Linear Models I

5.1 Key Ideas

5.1.1 Multiple regression

In regression problems we often want a model that represents our response variable in terms of several explanatory variables. A multiple regression model is

$$Y_i = \beta_0 + \beta_1 x_{1,i} + \ldots + \beta_p x_{p,i} + \varepsilon_i,$$

where $\varepsilon_i \sim iN(0, \sigma_\varepsilon^2)$ is an error term and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$ are p explanatory variables with $\boldsymbol{x}_j = (x_{j,1}, \ldots, x_{j,n})'$ for $j = 1, \ldots, p$. Normality of errors is required for statements about the distributions of test statistics made below to hold. A convenient representation of this model is the matrix form

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

where $\boldsymbol{Y} = (Y_1, \ldots, Y_n)'$, $X = (\boldsymbol{1}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_p)$, $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)'$ and $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_n)'$. The least squares estimator of $\boldsymbol{\beta}$ is given by (invertibility of $\boldsymbol{X}'\boldsymbol{X}$ is guaranteed when $\text{rank}(X) = p+1$)

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{Y}$$

and the estimate of the variance of the error term is

$$\hat{\sigma}_\varepsilon^2 = s_\varepsilon^2 = (\boldsymbol{Y} - \hat{\boldsymbol{Y}})'(\boldsymbol{Y} - \hat{\boldsymbol{Y}})/(n - p - 1).$$

The fitted values are

$$\hat{\boldsymbol{Y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}} = \boldsymbol{X}(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{Y} = \boldsymbol{H}\boldsymbol{Y}$$

where $\boldsymbol{H} = \boldsymbol{X}(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'$. We can show that $\hat{\boldsymbol{\beta}}$ is unbiased and that the covariance matrix of $\hat{\boldsymbol{\beta}}$ is given by

$$\text{Var}(\hat{\boldsymbol{\beta}}) = (\boldsymbol{X}'\boldsymbol{X})^{-1}\sigma_\varepsilon^2.$$

The usual statistic to test the null hypothesis $\beta_j = 0$, is given by

$$\hat{\beta}_j/\text{se}(\hat{\beta}_j) \sim t_{n-p-1},$$

where $\text{se}(\hat{\beta}_j)$ can be calculated from s^2_ε and the j^{th} diagonal element of $(\boldsymbol{X}'\boldsymbol{X})^{-1}$.

In determining the explanatory power of our regression model for these data the following quantities are useful:

$$\begin{aligned}
\text{SST} &= (\boldsymbol{Y}-\bar{\boldsymbol{Y}})'(\boldsymbol{Y}-\bar{\boldsymbol{Y}}) = \sum_{i=1}^{n}(Y_i-\bar{Y})^2 = S(\{\mathbf{1}\}) \\
\text{SSE} &= (\boldsymbol{Y}-\hat{\boldsymbol{Y}})'(\boldsymbol{Y}-\hat{\boldsymbol{Y}}) = \sum_{i=1}^{n}(Y_i-\hat{Y}_i)^2 = S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_p\}) \\
\text{SSREG} &= \text{SST}-\text{SSE} = S(\{\mathbf{1}\}) - S(\{1,\boldsymbol{x}_1,\ldots,\boldsymbol{x}_p\})
\end{aligned}$$

The total sum of squares (SST) measures the unexplained variability when we fit a model with just a constant term (the sample mean $\bar{Y}$). We use $S(\{\mathbf{1}\})$ to indicate that the sum of squares is calculated for a model with just a constant term. The error sum of squares (SSE) measures the unexplained variability when the full regression model is fitted. We use $S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_p\})$ to indicate that the sum of squares is calculated for a model with the p explanatory variables included. The regression sum of squares (SSREG) then measures the variability explained by our regression model. The significance of the whole regression is measured by comparing SSREG with SSE. In fact,

$$\frac{\text{SSREG}/p}{\text{SSE}/(n-p-1)} \sim F_{p,n-p-1}$$

under the null hypothesis that $\beta_1 = \ldots = \beta_p = 0$. The R^2 statistic is generated by comparing SSREG with SST

$$R^2 = \frac{\text{SSREG}}{\text{SST}}.$$

This is a popular measure but should be treated with caution – we do not have a sampling distribution for this statistic.

The sequential analysis of variance table is calculated by measuring the reduction in the error sum of squares as variables are added one by one. If we include variables in the order $\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_p$, the analysis of variance table gives us

Variable	df	Sum of Squares
x_1	1	$\text{SSREG}(\boldsymbol{x}_1) = S(\{\mathbf{1}\}) - S(\{\mathbf{1},\boldsymbol{x}_1\})$
x_2	1	$\text{SSREG}(\boldsymbol{x}_2\vert\boldsymbol{x}_1) = S(\{\mathbf{1},\boldsymbol{x}_1\}) - S(\{\mathbf{1},\boldsymbol{x}_1,\boldsymbol{x}_2\})$
$\vdots$	$\vdots$	$\vdots$
x_j	1	$\text{SSREG}(\boldsymbol{x}_j\vert\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{j-1}) = S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{j-1}\}) - S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_j\})$
$\vdots$	$\vdots$	$\vdots$
x_p	1	$\text{SSREG}(\boldsymbol{x}_p\vert\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{p-1}) = S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{p-1}\}) - S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_p\})$
Error	$n-p-1$	$\text{SSE} = S(\{\mathbf{1},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_p\})$

For the j^{th} entry, the sum of square measures the reduction is unexplained variability going from a model with explanatory variables $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{j-1}$ to $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_j$. The associated test statistic is

$$\frac{\text{SSREG}(\boldsymbol{x}_j\vert\boldsymbol{x}_1,\ldots,\boldsymbol{x}_{j-1})}{\text{SSE}/(n-p-1)} \sim F_{1,(n-p-1)}$$

under the null hypothesis that $\beta_j = 0$. If we square the t-statistic for testing $\beta_j = 0$, we get the F-statistic for testing $\beta_j = 0$ from a sequential analysis in which $\boldsymbol{x}_j$ is the last variable to enter the model.

5.1.2 Regression diagnostics

As with all statistical models, it is important that we check that our fitted regression model provides an adequate representation of the data. We also need to check the distributional assumptions, on which inference is based, are reasonable. A key quantity in model checking is the vector of residual values. Residuals are useful for detecting outlying points or other structure we may not have accounted for in our model. They are calculated from the difference between what is observed an what we expected based on our model:

$$e_i = Y_i - \hat{Y}_i.$$

The residuals, $e_1, \dots, e_n$ do not have constant variance. In fact $\text{Var}(e_i) = (1 - h_i)\sigma_\varepsilon^2$ where h_i is the i^{th} diagonal element of $\boldsymbol{H}$. The standardized residuals are scaled for by their estimated variances

$$r_i = \frac{e_i}{s_\varepsilon \sqrt{(1 - h_i)}}.$$

Sometimes individual observations can have a strong influence on the estimated variance. This may mask outliers. One approach to tackling this problem is to evaluate a deletion residual

$$r_i^* = \frac{e_i}{s_{(i)} \sqrt{(1 - h_i)}}.$$

where $s_{(i)}^2$ is the estimate of the error variance from fitting a model excluding point i. These are often referred to as studentized residuals.

The leverage h_i measures the distance of the explanatory variable values associated with point i, that is $x_{1,i}, \dots, x_{p,i}$, from the explanatory variable values associated with other points. A point with high leverage will exert a strong influence over the parameter estimates. Cook's distance measures the influence of point i by comparing $\hat{\beta}$ with $\hat{\beta}_{(i)}$, the parameter estimate that results from excluding i from the fitting process. There are many other measures of influence – equivalent measures are often given different names; see Atkinson (1985) for a clear treatment.

5.2 Multiple regression models

A model with several variables is constructed using the operator `+` in the model formula. In this context `+` denotes inclusion not addition. We can use the operator `-` is used for exclusion in model formulae. When models are updated, we use a `.` to denote contents of the original model. The use of these operators is best understood in the context of an example.

To illustrate consider the `Cars93` data from the MASS package. These are the values of variables recorded on 93 cars in the USA in 1993. Consider the `MPG.highway` (fuel consumption in highway driving) variable. It is reasonable to suppose that fuel consumption may, in part, be determined by the size of the vehicle and by characteristics of the engine. We start by using `pairs(...)` to generate a scatter plot matrix for four of the variables in the data set. A linear model is fitted using the `lm(...)` function (notice we use the named argument `data` to specify the data set as an alternative to attaching the data).

```
> library(MASS)
> ?Cars93
> names(Cars93)
> pairs(Cars93[c("MPG.highway","Horsepower","RPM","Weight")], col=2)
> lmMPG1 <- lm(MPG.highway ~ Horsepower + RPM + Weight, data=Cars93)
> summary(lmMPG1)
```

The `anova(...)` function gives us the (sequential) analysis of variance table using the order in which the variables are specified.

```
> anova(lmMPG1)
```

In the analysis of variance table for this example, adding each variable in the order specified gives a significant reduction in the error sum of squares. However, the explanatory variables are highly correlated; changing the order in which they are included alters the analysis of variance table.

```
> lmMPG2 <- lm(MPG.highway ~ Weight + Horsepower + RPM, data=Cars93)
> anova(lmMPG2)
```

Notice that we could have achieved the same outcome by updating our `lmMPG1` model object. Below we use `(.-Weight)` to denote the existing right-hand-side of the model formula with the variable `Weight` removed.

```
> lmMPG2 <- update(lmMPG1, ~ Weight + (.-Weight), data=Cars93)
> anova(lmMPG2)
```

When explanatory variables are correlated, inclusion or exclusion of one variable will affect the significance (as measured by the t-statistic) of the other variables. For example, if we exclude the `Weight` variable, both `Horsepower` and `RPM` become highly significant. Variables cannot be chosen simply on the basis of their significance in a larger model. This is the subject of the next section.

5.3 Model selection

The MASS package provides a number of functions that are useful in the process of variable selection. Consider the situation where we are interested in constructing the best (according to some criteria that we will specify later) linear model of `MPG.highway` in terms of ten of the other variables in model.

5.3.1 Forward search

In a forward search, from some starting model, we include variables one by one. We have established that `MPG.highway` is reasonably well explained by `Weight`. A model with just the `Weight` variable is a reasonable starting point for our forward search.

```
> lmMPG3 <- lm(MPG.highway ∼ Weight, data=Cars93)
> summary(lmMPG3)
```

We may want to consider what the effect of adding another term to this model is likely to be. We can do this using the `addterm(...)` function.

```
> addterm(lmMPG3, ∼ .+EngineSize + Horsepower + RPM + Rev.per.mile +
+ Fuel.tank.capacity + Length + Wheelbase + Width + Turn.circle,
+ test="F")
```

This function adds a single term from those listed and displays the corresponding F statistic. The second argument is a model formula; the term ∼ . denotes our existing model. If we select variables according to their significance (most significant first) the variable `Length` would be the next to be included.

```
> lmMPG4 <- update(lmMPG3, ∼ .+Length)
> summary(lmMPG4)
```

This process of variable inclusion can be repeated until there are no further significant variables to include. Notice something strange here; the coefficient for `Length` is positive. Does this seem counter intuitive? What happens when we remove `Weight` from the model?

```
> summary(update(lmMPG4, ∼ .-Weight))
```

Can you explain this?

5.3.2 Backwards elimination

An alternative approach is to start with a large model and remove terms one by one. The function `dropterm(...)` allows us to see the impact of removing variables from a model. We start by including all ten candidate explanatory variables.

```
> lmMPG5 <- lm(MPG.highway ∼ Weight + EngineSize + Horsepower + RPM +
+ Rev.per.mile + Fuel.tank.capacity + Length + Wheelbase +
+ Width + Turn.circle, data=Cars93)
> dropterm(lmMPG5, test="F")
```

Rather surprisingly, it is clear from this output that, in the presence of the other variables, `Horsepower` is not significant. `Horsepower` can be removed using the `update(...)` function and repeat the process.

```
> lmMPG6 <- update(lmMPG5, ~ .-Horsepower)
> dropterm(lmMPG6, test="F")
```

Which variable does the output of this command suggest should be dropped next?

5.3.3 Step-wise selection

The process of model selection can be automated using the `step(...)` function. This uses the Akaike information criteria (AIC) to select models. AIC is a measure of goodness of fit that penalises models with too many parameters; low AIC is desirable. The function can be used to perform a forward search.

```
> lmMPG7 <- lm(MPG.highway ~ 1, data=Cars93)
> step(lmMPG7, scope=list(upper=lmMPG5), direction="forward")
```

Here the starting model (`lmMPG7`) just contains a constant term. The `scope=list(upper=lmMPG5)` argument tells R the largest model that we are willing to consider. The process stops when the model with the smallest AIC is that resulting from adding no further variables to the model. Using a similar strategy we can automate backwards selection.

```
> step(lmMPG5, scope=list(lower=lmMPG7), direction="backward")
```

True step-wise regression allows us to go in both directions; variables may be both included and removed (if these actions result in a reduction of AIC).

```
> step(lmMPG7, scope=list(upper=lmMPG5))
```

At each stage the output shows the AIC resulting from removing variables that are in the model, including variables that are outside the model or doing nothing `<none>`. The result is a model with four of original ten variables included.

Model selection procedures are based on arbitrary criteria. There is no guarantee that the resulting model will be a good model (in the sense of giving good predictions) or that is will be sensible (in terms of what we know about the economics/physics/chemistry/. . . of the process under consideration). The procedures discussed in this section may also produce undesirable results in the presence of outliers or influential points.

5.4 Diagnostic plots

We work with the model suggested by step-wise regression.

```
> lmMPG8 <- lm(MPG.highway ~ Weight + Wheelbase +
+ Fuel.tank.capacity + Width, data=Cars93)
> summary(lmMPG8)
```

R will produce a set of four diagnostic plots when a linear model object is passed as an argument to the `plot(...)` function. These are two plots of residuals against fitted values, a normal qq-plot and an index plot of Cook's distance.

```
> par(mfrow=c(2,2))
> plot(lmMPG8)
> par(mfrow=c(1,1))
```

We may want to generate other diagnostic plots. For example, a plot of studentised residuals against fitted values with outlying values identified by name.

```
> plot(fitted(lmMPG8), rstudent(lmMPG8), col=2)
> identify(fitted(lmMPG8), rstudent(lmMPG8), label=Cars93$Make)
```

The leverages are generated using the `lm.influence(...)` function. Below we generate an index plot of leverage a label the high leverage (influential) points using vehicle type.

```
> plot(lm.influence(lmMPG8)$hat, type = "h", col=2)
> identify(lm.influence(lmMPG8)$hat, label=Cars93$Type)
```

We can fit a model that excludes certain observations by using the `subset` argument. For example, in the leverage plot generated by the commands above, it is clear that many of the influential points are vans and sporty cars. It is reasonable to suppose that a model built including these types of vehicles may not provide the best representation of fuel consumption for normal cars. The commands below update the model excluding vans and sporty cars.

```
> lmMPG8 <- update(lmMPG8, subset = (Type != "Van" & Type != "Sporty"))
> summary(lmMPG8)
```

The argument `subset` can also be used with the `lm(...)` function. Note that the `subset` is just a logical vector; in this example the values of the vector will be true when the corresponding vehicle is neither a van nor a sporty car. The result of excluding these types is a substantial alteration in parameter values. In fact, it would be appropriate to repeat the process of model selection if we are going to restrict our attention to a subset of observations.

Residuals scaled for constant variance are given by the function `rstandard(...)` applied to a linear model objects, while `cooks.distance(...)` will give Cook's distance (no surprises there). Other measures of influence are available; `dffits(...)` gives influence on fitted values and `dfbetas(...)` measures impact on individual parameter estimates.

† Colour and shading can be used to good effect in diagnostic plots. To illustrate consider the regression model including weight alone. We draw a plot in which the influence (according to Cook's distance) determines the darkness of shading of points.

```
> lmMPG9 <- lm(MPG.highway ~ Weight, data=Cars93)
> summary(lmMPG9)
> cdist <- cooks.distance(lmMPG9)
> inf.col <- gray(1-cdist/max(cdist))
> plot(Cars93$Weight, Cars93$MPG.highway, bg=inf.col, pch=21, cex=1.5)
```

Here `gray(...)` is a vector of values between 0 and 1 on a gray scale (where 0 is black and 1 is white). The named argument `bg` gives the colours to shade points and `cex` determines the size of the plotting character. Other colour commands include `rgb(...)` (for red, green, blue) and `hsv` (for hue, saturation, value).

5.5 Transformations

So far we have considered linear models for the observations. In many instances, a simple transformation of the response variable improves the fit of a linear model. The use of transformations requires caution; over-zealous application can result in a model that fits very well but tells us nothing about the real world. The Box-Cox family of transformations is often used in practice:

$$y^{(\lambda)} = \begin{cases} (y^\lambda - 1)/\lambda, & \lambda \neq 0 \\ \log(y), & \lambda = 0. \end{cases}$$

We may need to add a constant to the response variable values to ensure that they are positive before applying this transformation.

The MASS package contains the function `boxcox(...)` that draws the profile likelihood values against λ. This allows us to choose the transformation most appropriate for our data. To illustrate we consider the model with just one explanatory variable, `Weight`.

```
> boxcox(lmMPG9, lambda=seq(-2,2,by=0.1))
> attach(Cars93)
> MPGtrans = -(1/(MPG.highway)-1)
> lmMPG10 <- lm(MPGtrans ~ Weight)
> summary(lmMPG10)
> plot(lmMPG10)
```

The likelihood plot suggests that $\lambda = -1$ (reciprocal transformation) may be appropriate. Applying this transformation then refitting the model results in marked improvement in the properties of the residuals.

5.6 Polynomial regression

We have considered models that are linear in both the parameters and the explanatory variables. Higher order terms in the explanatory variables are readily included in a regression model. A regression model with a quadratic term is

$$Y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \varepsilon_i.$$

In polynomial regression the lower order terms are referred to as being marginal. For example, x is marginal to x^2. If the marginal term is absent, a constraint is imposed on the fitted function;

if x is excluded from quadratic regression, the regression curve is constrained to be symmetric about the origin (which is not usually what we want). In a variable selection problem, removal of marginal terms is not usually considered.

A quadratic regression model is fitted in R by including a quadratic term in the model formula.

```
> lmMPG11 <- lm(MPG.highway ~ Weight+I(Weight^2), data=Cars93)
> summary(lmMPG11)
```

The `I(...)` is necessary to prevent ^ from being interpreted as part of the model formula. This function forces R to interpret an object in its simplest form – it is also useful for preventing character vectors from being interpreted as factors. The commands below draw the fitted curve.

```
> cf <- as.vector(lmMPG11$coefficients)
> fline <- function(x) cf[1] + cf[2]*x + cf[3]*x^2
> x <- seq(min(Cars93$Weight), max(Cars93$Weight), length=300)
> plot(Cars93$Weight, Cars93$MPG.highway)
> lines(x, fline(x), col=2)
```

We can fit higher order terms in multiple regression. These often take the form of interactions. For example, in the model

$$Y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \varepsilon_i,$$

the β_3 parameter measures the strength of the interaction between x_1 and x_2. This model can be fitted using

```
> summary(lm(MPG.highway ~ Weight+Wheelbase+I(Weight*Wheelbase), data=Cars93))
```

or equivalently

```
> summary(lm(MPG.highway ~ Weight*Wheelbase, data=Cars93))
```

Note that in the second of these commands, R fits the marginal terms automatically. More on interactions next week.

5.7 Harmonic regression †

The explanatory variables in a regression model may be vectors that we construct. We have already seen an example of this in the simple time trend model

$$Y_t = \alpha + \beta t + \varepsilon_t$$

where $t = 1, \ldots, n$ is the explanatory variable. These constructed variables are often referred to as dummy variables. In harmonic regression the dummy variables are constructed from trigonometric functions. This is useful for modelling seasonal behaviour in time series. Suppose that we have a time series with seasonal period s ($s = 4$ for quarterly data, $s = 12$ for monthly data and $s = 365.2422$ for daily data). A simple harmonic regression model consists of a cyclical function with the fundamental frequency, $\omega = 2\pi/s$, plus noise:

$$Y_t = \alpha_1 \sin(\omega t) + \beta_1 \cos(\omega t) + \varepsilon_t.$$

In order to represent more complex seasonal patterns we may include harmonics. The model below consists of a linear trend and a trigonometric representations of seasonality with m frequencies (fundamental and $m-1$ harmonics),

$$Y_t = \alpha_0 + \beta_0 t + \sum_{j=1}^{m} (\alpha_j \sin(\omega j t) + \beta_j \cos(\omega j t)) + \varepsilon_t$$

where m is a number that is smaller than the integer part of $s/2$ (why?).

5.8 Exercise

It is 1993. A well know US manufacturer has developed a new make of sporty car – Vehicle X. The sales director wants to get some idea of how Vehicle X should be priced. Using the `Cars93` data set, our aim is to find the regression which best explains price in terms of the other numeric and integer variables (obviously not using the other price variables). To find out the class of the variables try

```
> for (j in 1:27) cat(names(Cars93)[j], ":  ", class(Cars93[[j]]), "\n")
```

1. Using the variable selection method of your choice propose an initial model for `Price`.
2. Construct diagnostic plots and comment on the quality of the model that you have proposed.
3. Attempt model improvement; you might like to consider excluding certain points, transformations or the inclusion of higher order terms. Be careful not to make your model too complicated.
4. The sales director does not trust any model with more than three explanatory variables in it. Reformulate your model (if necessary) in light of this information.
5. Vehicle X has CityMPG 25, HighwayMPG 35, 2.4 litre engine, 130 horsepower, 5500 rpm, 1500 revs per mile, 11.3 litre fuel tank, passenger capacity 2, length 175 inches, wheelbase 98 inches, width 64 inches, turn circle 40 feet, no rear seats or luggage room, and weight 2700 pounds. Using the model you feel most appropriate, suggest a price range for Vehicle X. Do you have any reservations about this prediction?
6. Write a function that takes a linear model object as input and draws a selection of diagnostic plots (different from those generated by the generic `plot(...)` command).
7. † Write a function that takes a data frame and a numeric variable from this data frame (the response) as arguments and does step wise selection using all of the other numeric variables as candidate explanatory variables (the aim of this function is to avoid having to type out a big list of variables when we want to do stepwise selection).

8. † Simulate a harmonic regression model for monthly data ($s = 12$) in with two frequencies ($m = 2$). For each simulated replication fit a model that has three frequencies ($m = 3$). What are the properties of α_3 and β_3?

5.9 Reading

5.9.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 11 for statistical models]
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Section 6.3 for regression diagnostics and p172 for model selection]

5.9.2 Supplementary reading†

- Dalgaard, P. (2002) *Introductory Statistics with R*, Springer. [Chapter 9 and 10 for some gentle stuff (all R no theory)]
- Atkinson, A. C. (1985) *Plots, Transformations and Regression*, Oxford University Press. [Professor Atkinson's comprehensive treatment]

Chapter 6

Linear Models II

6.1 Key ideas

Consider a situation in which we take measurements of some attribute Y on two distinct group. We want to know whether the mean of group 1, μ_1, is different from that of group 2, μ_2. The null hypothesis is equality, that is, $H_0 : \mu_1 = \mu_2$ or $H_0 : \mu_1 - \mu_2 = 0$. Suppose that we sample n_1 individuals from group 1 and n_2 individuals from group 2. A model for these data is, for $i = 1, 2$ and $j = 1, \ldots, n_i$,

$$Y_{i,j} = \mu_i + \varepsilon_{i,j}, \quad \{\varepsilon_{i,j}\} \sim iN(0, \sigma_\varepsilon^2). \tag{6.1}$$

Our observations can be treated as coming from a single population. All of our response variable values can be stacked in a single vector $(Y_1, \ldots, Y_n)'$ where $n = n_1 + n_2$ and $Y_i = Y_{1,i}$ for $i = 1, \ldots, n_1$ and $Y_i = Y_{2,i-n_1}$ for $i = n_1 + 1, \ldots, n$. We also set up an additional variable d; this is an indicator for being in group 2, so $d_i = 0$ for $i = 1, \ldots, n_1$ and $d_i = 1$ for $i = n_1 + 1, \ldots, n$. Model (6.1) is then equivalent to

$$Y_i = \mu + \lambda d_i + \varepsilon_i, \quad \{\varepsilon_i\} \sim iN(0, \sigma_\varepsilon^2),$$

where $\mu = \mu_1$ and $\lambda = \mu_2 - \mu_1$. Testing $H_0 : \mu_1 = \mu_2$ is then equivalent to testing $H_0 : \lambda = 0$. We can generate a test statistic for this hypothesis by regressing Y on the dummy variable d. Notice that we need one dummy variable and the constant term to completely characterise a problem with two groups.

The linear model framework described above is readily generalised to other analysis of variance problems. For example, suppose that we are interested in determining the effect on our response variable, Y, of a factor with k levels. We could use a model

$$Y_i = \lambda_1 + \lambda_2 d_{2,i} + \ldots + \lambda_k d_{k,i} + \varepsilon_i, \quad \{\varepsilon_i\} \sim iN(0, \sigma_\varepsilon^2),$$

where d_r is an indicator for the level r of the factor; thus, $d_{r,i} = 1$ if the level of the factor is r for observation i, and $d_{r,i} = 0$ otherwise. Notice that there is no d_1 variable. For a factor with k levels we require a constant and $k-1$ dummy variables to completely characterise the problem.

6.2 Factors

Factors are variables that have a number of different levels rather than numeric values (these levels may have numbers as labels but these are not interpreted as numeric values). For example, in the `Cars93` data set the `Type` variable is a factor; vehicles in the study are of type `"Compact"`, `"Large"`, `"Midsize"`, `"Small"`, `"Sporty"` or `"Van"`.

```
> library(MASS)
> class(Cars93$Type)
> levels(Cars93$Type)
```

Note that `Cylinders` is also a factor. Some of the levels of cylinders are labeled by number but these are not interpreted as numeric values in model fitting.

```
> class(Cars93$Cylinders)
> levels(Cars93$Cylinders)
```

Factors are used to identify groups within a data set. For example, in order to work out the mean of a variable for each type of vehicle, we would use the function `tapply(...)`. This function takes three arguments: the variable of interest, the factor (or list of factors) and the function that we would like to apply. A few examples follow.

```
> tapply(MPG.highway, Type, mean)
> tapply(MPG.highway, Type, length)
> tapply(Price, list(Type, AirBags), mean)
```

In order to create a factor variable, we use the function `factor(...)`. This returns an object with class `"factor"`. We can specify an order for this factor using the `levels` argument and `ordered=TRUE`. To illustrate, consider question 7 from the exercise in week 2. We have daily readings of log returns and we would like to group these by month and year (that is, Jan 2000, Feb 2000, . . . , Sep 2004) and work out the standard deviation.

```
> load("IntNas.Rdata")
> attach(intnas2)
> Monthf <- factor(Month, levels=c("Jan","Feb","Mar","Apr","May","Jun",
+ "Jul","Aug","Sep","Oct","Nov","Dec"), ordered=TRUE)
> IntelLRsds <- tapply(IntelLR, list(Monthf,Year), sd, na.rm=TRUE)
> plot(as.vector(IntelLRsds),type="l")
```

6.3 Linear models using factors

The results from fitting linear models in which the explanatory variables are constructed from factors are often referred to as analysis of variance (ANOVA). Venables and Ripley, section 6.2, gives the details of the construction of explanatory variables from factors; this material is technical and is not required for the course. In most instances the R default for constructing the model matrix $\boldsymbol{X}$ from factors is sensible. We will build up the complexity of our models using examples.

6.3.1 Example A: One-way ANOVA

Consider the following question: does the provision of airbags affect the maximum price that people are willing to pay for a car? We can test this using a linear model with a single factor (one-way analysis of variance). The null hypothesis is that there is no effect. We start by plotting `Max.Price` against `AirBags` notice the type of plot that is generated. Try to interpret the output of the following commands.

```
> plot(AirBags,Max.Price)
> lmMaxP1 <- lm(Max.Price ~ AirBags, data=Cars93)
> summary(lmMaxP1)
> plot(lmMaxP1)
```

Notice that R ensures that the columns of the model matrix are not linearly dependent by excluding one level from the linear model. The validity of analysis of variance results is dependent on constant variance within groups. We can see from the diagnostic plots that this is not entirely unreasonable for these data. We could check using the function `var.test(...)`

```
> MaxP0 <- Cars93$Max.Price[Cars93$AirBags=="None"]
> MaxP1 <- Cars93$Max.Price[Cars93$AirBags=="Driver only"]
> MaxP2 <- Cars93$Max.Price[Cars93$AirBags=="Driver & Passenger"]
> var.test(MaxP0, MaxP1)
> var.test(MaxP0, MaxP2)
> var.test(MaxP1, MaxP2)
```

6.3.2 Example B: Two-way ANOVA

A linear model with two factors is referred to as two-way analysis of variance. We can use this technique to test the effect of two factors simultaneously. For example, we may ask whether, in addition to provision of airbags, the availability of manual transmission explains differences in maximum price.

```
> lmMaxP2 <- lm(Max.Price~ AirBags+Man.trans.avail, data=Cars93)
> anova(lmMaxP2)
```

As with multiple regression models, the sequential analysis of variance table is affected by the order in which variables are included.

```
> lmMaxP3 <- lm(Max.Price~ Man.trans.avail+AirBags, data=Cars93)
> anova(lmMaxP3)
```

We can readily include interaction terms in our model. A term of the form `a:b` is used to denote an interaction. We can include interactions and all lower order terms by the notation `a*b`. The following example illustrates.

```
> anova(lm(Max.Price~ AirBags+Man.trans.avail+AirBags:Man.trans.avail,
+ data=Cars93))
> anova(lm(Max.Price~ AirBags*Man.trans.avail, data=Cars93))
```

Note that the output of these commands is identical. What do you conclude about the interaction between provision of airbags and manual transmission in determining maximum price?

6.3.3 Example C: Factorial Design

Hines and Montgomery (1990, p.416) give the following results for an experiment in integrated circuit manufacture in which arsenic is deposited on silicon wafers. The factors are deposition time (A) and arsenic flow rate (B). Although both variables are quantitative, measurements are only taken at a high (labeled 1) and low level (labeled 0) of each. The purpose is to find out whether the factors have an effect and, if so, to estimate the sign and magnitude of the effect. The response is the thickness of the deposited layer.

Treatment combination	A	B	Thickness
(1)	0	0	14.037, 14.165, 13.972, 13.907
a	1	0	14.821, 14.757, 14.843, 14.878
b	0	1	13.880, 13.860, 14.032, 13.914
ab	1	1	14.888, 14.921, 14.415, 14.932

The figures for thickness are stored in a single column row by row in the file `arsenic.dat`. We will create a data frame for these data and then generate variables to represent the levels of the two factors A and B.

```
> arsenic <- read.table("arsenic.dat", header=TRUE)
> A <- rep(0:1, each=4, times=2)
> B <- rep(0:1, each=8)
> A
> B
> arsenic$A <- A
> arsenic$B <- B
> rm(A,B)
> fix(arsenic)
> anova(lm(Thickness~ A*B, data=arsenic))
```

Having established from this that neither B (the flow rate) nor the interaction term are significant, we fit a model with just A the deposition time and generate some diagnostic plots.

```
> lmArsenic <- lm(Thickness~ A, data=arsenic)
> summary(lmArsenic)
> par(mfrow=c(2,2))
> plot(lmArsenic)
> par(mfrow=c(1,1))
```

6.3.4 Example D: Analysis of covariance

In an analysis of covariance we are interested in the relationship between the response and covariates (explanatory variables in the usual sense) for different levels of a set of factors. For example, suppose we are interested in the behaviour of the relationship between fuel consumption in city driving and vehicle weight, comparing vehicles where manual transmission is available with those where it is not. We start with a plot of `MPG.city` against `Weight` with the level of `Man.trans.avail` identified for each point. The `legend(...)` function pauses while we choose a location for the legend on the plot.

```
> plot(Weight, MPG.city, pch=as.numeric(Man.trans.avail),
+ col=as.numeric(Man.trans.avail)+1)
> legend(locator(n=1), legend=c("Not available", "Available"),
+ pch=1:2, col=2:3)
```

Initially we fit a model in which the slope is the same for both level of the factor.

```
> lmMPGadd <- lm(MPG.city ~ Weight+Man.trans.avail, data=Cars93)
> summary(lmMPGadd)
```

This indicates that, if the same slope is fitted for each level of the factor then the difference intercepts is not significant. We can allow for an interaction between `Weight` and `Man.trans.avail`.

```
> lmMPGint <- lm(MPG.city ~ Weight*Man.trans.avail, data=Cars93)
> summary(lmMPGint)
```

With the interaction term included, both the `Man.trans.avail` variable and the interaction between `Weight` and `Man.trans.avail` are significant. This indicates that both slope and intercept are significantly different when the levels of `Man.trans.avail` are taken into account.

```
> lmMPGmt0 <- lm(MPG.city ~ Weight, subset = (Man.trans.avail=="No"))
> lmMPGmt1 <- lm(MPG.city~ Weight, subset = (Man.trans.avail=="Yes"))
> summary(lmMPGmt0)
> summary(lmMPGmt1)
> abline(lmMPGmt0, col=2)
> abline(lmMPGmt1, col=3)
```

Note that the difference between the slope estimates of these two models is precisely the interaction term from `lmMPGint`.

We can fit these models simultaneously using a model formula of the form `a/x -1` where `a` is a factor. This will fit the separate regression models. The -1 is to remove the overall intercept term that we do not want.

```
> lmMPGsep <- lm(MPG.city ~ Man.trans.avail/Weight-1, data=Cars93)
> summary(lmMPGsep)
```

6.4 Exercise

1. This question illustrates the importance of including marginal terms. Fit quadratic regression models for `MPG.highway` with `Horsepower` as an explanatory variable. First of all just with the quadratic term and them with the marginal term included. The commands are:

   ```
   lmMPG1 <- lm(MPG.highway ~ I(Horsepower)^2, data=Cars93)
   lmMPG2 <- update(lmMPG1, ~ Horsepower + .)
   ```

   Plot `MPG.highway` against `Horsepower` and add the fitted lines generated by the two regressions. Which do you think is more appropriate.

2. We wish to determine whether the log price is affected by the origin of the vehicle. Why might we consider performing a log transformation? Create a variable `logPrice` by taking logs of the `Price` variable.

   (a) Carry out one-way analysis of variance with `Origin` as a factor. What conclusions do you draw?

   (b) Now include `Type` in the model. Can you explain the resulting changes?

   (c) Is there a significant interaction between price and origin? [You will need to fit another model to answer this question.]

   (d) Draw a plot of `logPrice` against `as.numeric(Type)` with the colour of each point determined by its origin. [You will need to use `col = as.numeric(Origin)`.] Create a legend for this plot. Does this help to explain the results of our models?

3. Write a function that takes two variables and the data frame that contains these variables as arguments. The function should perform the quadratic regression of first variable on the second and generate a plot of the response against the explanatory variable with the fitted curve added to the plot.

4. † Consider the zero-mean auto-regressive model of order 1, AR(1):

   $$w_t = \phi w_{t-1} + \varepsilon_t, \quad \{\varepsilon_t\} \sim iN(0,1).$$

   Write a function to simulate a series of n values of an AR(1). Now consider a simple regression model with time as the explanatory variable and autoregressive errors:

   $$Y_t = \alpha + \beta t + w_t, \quad \{w_t\} \sim AR(1).$$

   Write a function that simulates n values from this regression model and returns $\hat{\beta}$. Write a function that simulates r instances of $\hat{\beta}$ and use it to investigate the MSE of the associated estimator. How does this compare with (analytic) results for the case of uncorrelated errors.

6.5 Reading

6.5.1 Directly related reading

- Venables, W. N. *et. al.* (2001) *An Introduction to R.* [Chapter 4 for stuff on factors]

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Section 6.1 for analysis of covariance example and section 6.7 for factorial design]

6.5.2 Supplementary reading†

- Hines, W. W. and Montgomery, D. C. (1990) *Probability and Statistics in Engineering and Management Science*, 3rd edition. Wiley. [For arsenic example]
- Dalgaard, P. (2002) *Introductory Statistics with R*, Springer. [Section 10.3 dummy variables, 10.5 interaction, 10.6 two-way ANOVA and 10.7 analysis of covariance - all R no theory]
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Section 6.2 for stuff on contrasts]

Chapter 7

Time Series

7.1 Key ideas

7.1.1 Time series operators

1. Back-shift operator, B. Moves the time index of an observation back by one time interval:
$$BY_t = Y_{t-1}.$$
If we back-shift p times the result is
$$B^p Y_t = Y_{t-p}.$$

2. Differencing operator, $\Delta = 1 - B$. Takes the difference between an observation and the previous observation:
$$\Delta Y_t = (1-B)Y_t = Y_t - Y_{t-1}$$
The effect of differencing d times can be calculated by expanding the corresponding polynomial in B:
$$\Delta^d Y_t = (1-B)^d Y_t = \sum_{j=0}^{d} \binom{d}{j} (-1)^j Y_{t-j}.$$
The differencing operator is applied to models to reduce them to stationarity. It is often applied to data in an attempt to generate a series for which a stationary model is appropriate.

3. Seasonal differencing operator, $\Delta_s = 1 - B^s$. Takes the difference between two points in the same season:
$$\Delta_s Y_t = Y_t - Y_{t-s}.$$

4. Seasonal summation operator, $S(B) = 1 + B + B^2 + \ldots + B^{s-1}$. Adds points over the seasonal period:
$$S(B)Y_t = Y_t + Y_{t-1} + \ldots + Y_{t-s+1}$$

7.1.2 Covariance stationarity

Stationarity features prominently in the statistical theory of time series. A model $\{Y_t\}$ is covariance stationary if:

1. $E(Y_t) = \mu_Y$ a constant for all t,
2. $\text{Var}(Y_t) = \sigma_Y^2$ a constant for all t,
3. $\text{Corr}(Y_t, Y_{t-h})$ is a function of h alone for all t and integer h.

In summary, the mean, variance and the correlation structure of the series do not change over time.

7.1.3 Auto-correlation

The auto-correlation function (ACF) is of key interest in time series analysis. This function describes the strength of the relationships between different points in the series. For a covariance stationary time series model $\{Y_t\}$, the autocorrelation function (ACF), $\rho(.)$, is defined as

$$\rho(h) = \text{Corr}(Y_t, Y_{t-h}) \quad \text{for} \quad h = 0, 1, \ldots.$$

For a sample, $\{y_1, \ldots, y_n\}$ the sample estimate of the ACF is

$$\hat{\rho}(h) = r(h) = \frac{c(h)}{c(0)} \quad \text{where} \quad c(h) = \frac{1}{n} \sum_{i=h+1}^{n} (y_i - \bar{y})(y_{i-h} - \bar{y}).$$

7.1.4 Autoregressive moving average and related models

Models of the autoregressive moving average (ARMA) class are widely used to represent time series. They are not the only time series models available and are often not the most appropriate. However, they form a good starting point for the analysis of time series. In what follows we describe zero mean processes. The definitions are easily adapted to the non-zero mean case.

Time series are usually characterised by their dependence structure; what happens today is dependent on what happened yesterday. A simple way to model dependence on past observations is to use ideas from regression. We can build a simple regression model for our time series in which the explanatory variable is the observation immediately prior to our current observation. This is an autoregressive model of order 1, AR(1):

$$Y_t = \phi Y_{t-1} + \varepsilon_t, \quad \{\varepsilon_t\} \sim iN(0, \sigma_\varepsilon^2).$$

This idea can be extended to p previous observations, AR(p):

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \ldots + \phi_p Y_{t-p} + \varepsilon_t.$$

This can be written as

$$Y_t - \phi_1 Y_{t-1} - \phi_2 Y_{t-2} - \ldots - \phi_p Y_{t-p} = \varepsilon_t,$$

or

$$\phi(B) Y_t = \varepsilon_t,$$

where $\phi(B) = 1 - \phi_1 B - \ldots - \phi_p B^p$. We can specify conditions on the function $\phi(.)$ in order to ensure that this model is stationary.

A more general class of model is defined by including dependence on past error terms. These are referred to as autoregressive moving average (ARMA) models. If we include dependence on the q previous error term, the result is an ARMA(p, q):

$$Y_t - \phi_1 Y_{t-1} - \ldots - \phi_p Y_{t-p} = \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q},$$

or

$$\phi(B) Y_t = \theta(B) \varepsilon_t,$$

where $\theta(B) = 1 + \theta_1 B + \ldots + \theta_q B^q$.

Models which are stationary after differencing may be represented as autoregressive integrated moving averages (ARIMA). If $\{Y_t\} \sim$ ARIMA(p, d, q), then $\{\Delta^d Y_t\} \sim$ ARMA(p, q). This relationship is represented by the model equation

$$\phi(B) \Delta^d Y_t = \theta(B) \varepsilon_t.$$

In practice d is usually a small number ($d = 0$, 1 or 2).

The order of an ARIMA model is determined by three numbers p, d and q. If the process is seasonal, in addition to these numbers we have s the seasonal period, and P, D and Q the orders of the seasonal parts of the model. † The seasonal ARIMA$(p, d, q) \times (P, D, Q)_s$ model is

$$\Phi(B^s) \phi(B) \Delta_s^D \Delta^d Y_t = \Theta(B^s) \theta(B) \varepsilon_t,$$

where $\Phi(.)$ and $\Theta(.)$ are polynomials of order P and Q respectively.

7.1.5 Generalised autoregressive conditionally heteroskedastic models

Let $\{X_t\}$ be a zero mean time series and let $\mathcal{F}_t$ denote the the past of the series, $\{X_{t-i} : i = 0, 1, \ldots\}$. The generalised autoregressive conditionally heteroskedastic, GARCH(p, q), model is defined by

$$\begin{aligned} X_t &= \sigma_t \varepsilon_t, \\ \sigma_t^2 &= c + \sum_{i=1}^{p} b_i X_{t-i}^2 + \sum_{i=1}^{q} a_i \sigma_{t-i}^2, \end{aligned}$$

where $\sigma_t^2 = \mathrm{Var}(X_t | \mathcal{F}_{t-1}) = E(X_t^2 | \mathcal{F}_{t-1})$, and $\{\varepsilon_t\} \sim$ IID$(0, 1)$ with ε_t independent of $\mathcal{F}_{t-1}$. All of the parameters are non-negative, $c > 0$ and $\sum a_i + \sum b_i < 1$. In financial applications, GARCH models are routinely fitted to the (log) returns for the price of an asset. The popularity of the GARCH class is due, at least in part, to the ease with which the likelihood is constructed; if $\boldsymbol{x} = (x_1, \ldots, x_n)$ is a sequence of observed returns then, by prediction error decomposition,

$$\ell(\boldsymbol{\theta}) = \log f(\boldsymbol{x}|\boldsymbol{\theta}, \mathcal{F}_0) = \sum_{t=1}^{n} [\log \sigma_t - \log f_\varepsilon(x_t/\sigma_t)],$$

where the density function of the error terms, f_ε, is assumed known.

7.2 Time series objects

7.2.1 Creating and plotting time series objects

In order to illustrate some time series ideas we will use the data in the file `UKairpass.dat`. This file contains the monthly numbers of airline passengers (thousands) using UK airports from January 1995 to June 2003. It contains two series, one for domestic passengers and one for international passengers.

The function `ts(...)` provides a flexible mechanism for creating time series objects from vectors, matrices or data frames. We will illustrate using a data frame.

```
> UKairpass <- read.table("UKairpass.dat", header=TRUE)
> Domestic <- ts(UKairpass[1], start=1995, frequency=12)
> International <- ts(UKairpass[2], start=1995, frequency=12)
```

`Domestic` and `International` are now time series objects. The argument `frequency` refers to the seasonal period s. We can view time series objects by simply typing their name or use the function `tsp(...)` to extract the start, end and frequency. Notice how, since we specified `frequency = 12` when we created these objects, R assumes that the appropriate labels for the observations are month and year.

```
> Domestic
> tsp(Domestic)
```

The function `ts.plot(...)` will generate a time series plot. Below we add colour to differentiate between the series and include a legend.

```
> ts.plot(Domestic, International, col=c(2,3))
> legend(locator(n=1),legend=c("Domestic","International"), col=c(2,3),
+ lty=1)
```

7.2.2 Simple manipulation of time series objects

We can shift the series in time using the `lag(...)` function. This function has two arguments: the series that we want to shift and `k` the number of time steps we would like to push the series into the past (the default is 1). This function works in a counter intuitive way; if we want the series back shift the series in the usual sense (so that Jan 1995 reading is at Feb 1995, and so on) we have to use `k=-1`.

```
> lag(Domestic)
> lag(Domestic, k=-1)
> lag(Domestic, k=-12)
```

In order to take differences we use the `diff(...)` function. The `diff(...)` function takes two arguments: the series we would like to difference and `lag`, the back-shift that we would like to perform (default is 1). Here `lag` argument of the `diff(...)` function (not to be confused with the `lag(...)` function!) is interpreted in the usual way so `diff(Domestic, lag=12)` is equal to `Domestic - lag(Domestic, k=-12)`. Note that this is seasonal differencing, Δ_{12}, not order 12 differencing Δ^{12}.

```
> diff(Domestic)
> ts.plot(diff(Domestic, lag=12), diff(International, lag=12), col=c(4,6))
> legend(locator(n=1),legend=c("Domestic","International"), col=c(4,6),
+ lty=1)
```

We can perform simple arithmetic and transformations using time series objects. The results are also time series objects.

```
> Total <- Domestic + International
> attributes(Total)
> ts.plot(Domestic, International, Total, col=c(2,3,4))
> logDomest <- log(Domestic)
> attributes(logDomest)
> ts.plot(logDomest, col=2)
```

7.3 Descriptive analysis and model identification

The function to generate the ACF in R is `acf(...)`. This function returns an object of mode list and (unless we alter the default setting) plots the ACF.

```
> acf(International)
> acfInt <- acf(International, plot=FALSE)
> mode(acfInt)
> attributes(acfInt)
> acfInt$acf
```

Note that 1.0 on the x-axis in this plot, denotes 1 year. We can use the time series plots and ACF plots to guide the process of model identification. The time series plot of `International` suggests a trend and seasonal pattern. We may remove these by differencing and look at the ACF of the resulting data.

```
> diffInt <- diff(International)
> diffIntSeas <- diff(diffInt, lag=12)
> acf(diffInt)
> acf(diffIntSeas)
```

This final plot indicates that after differencing and seasonal differencing there is still some significant correlation of points one year apart. † We may tentatively identify an ARIMA$(0,1,0) \times (0,1,1)_{12}$ on the basis of this plot.

In order to generate a multivariate time series from two series that are observed at the same time points we may use the `ts.union(...)` function. Multivariate time series may be passed as arguments to plotting functions. Passing a multivariate time series to the `acf(...)` function will draw individual ACF plots and also the cross-correlations. We can also do perform simple manipulation operations on a multivariate series; the result is that each of the component series will be altered.

```
> bothSeries <- ts.union(Domestic, International)
> ts.plot(bothSeries, col=c(3,4))
> acf(bothSeries)
> diffBothSeries <- diff(diff(bothSeries), lag=12)
> acf(diffBothSeries)
```

7.4 Fitting ARIMA models

There are functions available in R to fit autoregressive integrated moving average (ARIMA) models. We proceed by examples.

7.4.1 Simple AR and ARIMA for Nile data

For every year from 1871 to 1970 the volume of water flowing through the Nile at Aswan was recorded (I think the units are cubic kilometers). These 100 data points are in the file `nile.dat`.

```
> nile <- ts(read.table("nile.dat", header=TRUE), start=1871)
> ts.plot(nile, col=2)
> acf(nile, col=2)
```

We fit ARIMA models using the function `arima(...)`. The commands below fit an AR(1) and an ARIMA(0,1,1) to the Nile data. Note the brackets around the command; this is equivalent to performing the command without brackets then typing the name of the object just created. The `order` is a vector of three numbers (p, d, q). Which of these models do you think is preferable?

```
> (nileAR1 <- arima(nile, order=c(1,0,0)))
> (nileARIMA011 <- arima(nile, order=c(0,1,1)))
```

One of the key questions in time series modelling is, does our model account adequately for serial correlation in the observed series? The function `tsdiag(...)` will give an index plot of standardized residuals (over time), a plot of the sample ACF values of the residuals and a plot of the Ljung-Box statistic calculated from the residuals for a given number of lags (this number can be set using the `gof.lag` argument). Both sample ACF and Ljung-Box statistics are used to detect serial correlation in the residuals, that is, serial correlation that we have not accounted for using our model. The Ljung-Box statistic at lag h tests the hypothesis of no significant correlation below lag h. The commands below generate diagnostic plots for our two models. Does this change your opinion about which model is preferable?

```
> tsdiag(nileAR1)
> tsdiag(nileARIMA011)
```

One of the aims of time series modelling is to generate forecasts. In R this is done using the `predict(...)` function which takes a model object and a number (number of prediction) as arguments. The return value is a list with named components `pred` (the predictions) and `se` (the associated standard errors). Below we forecast 20 years beyond the end of the series for each of our models. What do you notice about these predictions? (A picture may help.) The predictions are very different; can you explain this?

```
> predict(nileAR1,20)$pred
> predict(nileARIMA011,20)$pred
```

7.4.2 Seasonal models for air passengers data

Seasonal ARIMA models are also fitted using the `arima(...)` function. The argument `seasonal` is a list with named elements `order` (the seasonal order of the process, $(P,\ D,\ Q)$) and `period` (the seasonal period). Earlier we indicated that an ARIMA$(0,1,0)\times(0,1,1)_{12}$ may be appropriate for the international air passengers data. We will fit this model and look at the properties of residuals. What do you notice in the index plot of the residuals?

```
> (intARIMA <- arima(International, order=c(0,1,0),
+ seasonal=list(order=c(0,1,1), period=12)))
> tsdiag(intARIMA)
```

A good way to represent forecasts is in a plot along with the observed data. Below we generate a plot of forecasts and an approximate 95% confidence interval for the forecasts.

```
> intfore <- predict(intARIMA, 36)
> ts.plot(International, intfore$pred, infore$pred+2*intfore$se,
+ infore$pred-2*intfore$se, lty=c(1,2,3,3), col=c(2,3,4,4))
```

7.5 Exercise

1. The file `rpi.dat` contains the monthly RPI for the UK (retail price index excluding housing costs) from January 1987 to August 2002 measured on a scale which takes the January

1987 figure to be 100. Generate a time series plot of the data. Generate an plot of the sample ACF for the original series, for the first differences and for the seasonal difference of the first difference series. Comment on these plots.

2. The file `passport.in7` contains the number of applications arriving at the UK passport service monthly from July 1993 to June 2002. Fit an ARIMA$(0, 1, 1) \times (0, 1, 1)_{12}$ model to these data and generate a plot of 5 years of forecasts. Is there anything in the residual plots for this model that makes you feel uneasy about your forecasts?

3. In this exercise I would like you to write some of your own time series functions.

   (a) The `lag(...)` function behaves counter intuitively (as described above). Write a simple function that takes a time series object and a positive integer as arguments and performs back-shifting operation in the normal way (should only take one line).

   (b) By default the `acf(...)` function plots included the autocorrelation at lag 1. This is irritating; we know that the ACF at lag 1 is equal to 1, we don't need it to be included on the plot. Try to write a function that plots the ACF from lag 2 upwards. [Hint: have a look at the attributes of the return value of the `acf(...) function`.]

   (c) Write a function that takes as arguments a time series and an integer, d, and difference the series d times, that is, performs the operation Δ^d.

   (d) † Write a function that fits several different ARIMA models to a data set a choose the one with minimum AIC.

4. A couple of interesting time series simulation problems (one easy, one tricky).

   (a) Simulate an AR(1), an ARIMA(1,1,0) and an ARIMA(1,2,0) with $n = 500$. Compare the time series plots and ACF plots for these series.

   (b) † Design and conduct a simulation experiments to find the properties of the parameter estimates of an ARMA(p, q) for different values of n, ϕ and θ.

7.6 Reading

7.6.1 Directly related reading

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Sections 14.1, 14.2 and 14.3 but not the spectral stuff.]

7.6.2 Supplementary reading†

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [The remainder of chapter 14 interesting spectral and financial time series stuff.]

- Brockwell, P. J. and Davis, R. A. (1996) *Introduction to Time Series and Forecasting.* Springer-Verlag. [One of the best time series introductions.]

Chapter 8

Exploratory Multivariate Analysis

8.1 Principal components analysis

8.1.1 Population principal components

We will consider principal components analysis as a data reduction technique. In this section we depart somewhat from our previous notation. We follow most of the conventions adopted in Mardia, Kent and Bibby (1979). In particular we consider a random $1 \times p$ row vector, $\boldsymbol{x}$, with mean $\boldsymbol{\mu}$ and variance matrix $\boldsymbol{\Sigma}$. The reasons for this notational convention will become apparent in §8.1.2.

Suppose that $\boldsymbol{\Sigma}$ has eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p \geq 0$ and that the corresponding eigenvectors are stacked in a $p \times p$ matrix $\boldsymbol{\Gamma} = (\boldsymbol{\gamma}_1, \ldots, \boldsymbol{\gamma}_p)$ so $\boldsymbol{\Gamma}'\boldsymbol{\Sigma}\boldsymbol{\Gamma} = \boldsymbol{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_p)$. Note that $\boldsymbol{\Gamma}$ is orthogonal so $\boldsymbol{\Gamma}' = \boldsymbol{\Gamma}^{-1}$. The principal components of $\boldsymbol{x}$ are the entries in the $1 \times p$ random vector $\boldsymbol{y}$ where

$$\boldsymbol{y} = (\boldsymbol{x} - \boldsymbol{\mu})\boldsymbol{\Gamma}. \tag{8.1}$$

The k^{th} principal component is

$$y_k = (\boldsymbol{x} - \boldsymbol{\mu})\boldsymbol{\gamma}_k.$$

We can readily show that $E(\boldsymbol{y}) = 0$ and $\text{Var}(\boldsymbol{y}) = \boldsymbol{\Lambda}$. As a consequence $\text{Var}(y_1) \geq \cdots \geq \text{Var}(y_p)$

The principal components are (standardized) linear combinations of our original variables. The first principal component combines the variables in a way that maximizes variance. This has important implications for the use of principal components as a data reduction technique. Reversing transformation (8.1) yields

$$\boldsymbol{x} = \boldsymbol{\mu} + \boldsymbol{\Gamma}'\boldsymbol{y}.$$

We know that y_1 has maximal variance, y_2 has the second largest variance and so on with y_p having the smallest variance. It may be reasonable to ignore principal components beyond some point $r < p$. This allows us to reduce the dimension of our problem and focus analysis on the first r principal components. As these r components account for the bulk of the variability, a reasonable approximation to $\boldsymbol{x}$ may be constructed as

$$\boldsymbol{x}^* = \boldsymbol{\mu} + \boldsymbol{\Gamma}^{*\prime}\boldsymbol{y}^*,$$

where $\boldsymbol{\Gamma}^*$ and $\boldsymbol{y}^*$ are respectively the first r columns of $\boldsymbol{\Gamma}$ and $\boldsymbol{y}$.

8.1.2 Sample principal components

Now consider taking a sample $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ where each $\boldsymbol{x}_i$ is a $1 \times p$ vector. We stack these vectors row-wise in the $n \times p$ data matrix $\boldsymbol{X} = (\boldsymbol{x}_1', \ldots, \boldsymbol{x}_n')'$. Another way to view $\boldsymbol{X}$ is as p column vectors of dimension n stacked column-wise $\boldsymbol{X} = (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n)$. We define the sample mean by $\bar{\boldsymbol{x}} = \frac{1}{n}\sum_{j=1}^n \boldsymbol{x}_j$, and the sample variance matrix by $\boldsymbol{S} = \frac{1}{n}\sum_{j=1}^n (\boldsymbol{x}_j - \bar{\boldsymbol{x}})'(\boldsymbol{x}_j - \bar{\boldsymbol{x}})$.

We perform a decomposition that is analogous to that described for the population variance matrix in §8.1.1. Thus, $\boldsymbol{G}'\boldsymbol{S}\boldsymbol{G} = \boldsymbol{L}$ where $\boldsymbol{L}$ is a matrix with the eigenvalues of $\boldsymbol{S}$ on the diagonal (in decreasing order of magnitude) and $\boldsymbol{G}$ is the column-wise stack of the corresponding eigenvectors, $\boldsymbol{G} = (\boldsymbol{g}_1, \ldots, \boldsymbol{g}_p)$. Defining $\mathbf{1}$ as an $n \times 1$ vector of 1s, the sample principal components are given by the columns of $\boldsymbol{Y}$ where

$$\boldsymbol{Y} = (\boldsymbol{X} - \mathbf{1}\bar{\boldsymbol{x}})\boldsymbol{G}.$$

The k^{th} principal component is an $n \times 1$ vector

$$\boldsymbol{Y}_k = (\boldsymbol{X} - \mathbf{1}\bar{\boldsymbol{x}})\boldsymbol{g}_k.$$

This is a linear combination of the vectors $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_p$. If we consider relevant variability is captured by the first r principal components, we may take

$$\boldsymbol{X}^* = \mathbf{1}\bar{\boldsymbol{x}} + \boldsymbol{G}^{*\prime}\boldsymbol{Y}^*$$

to be a reasonable approximation to $\boldsymbol{X}$. Here $\boldsymbol{G}^*$ and $\boldsymbol{Y}^*$ are respectively the first r columns of $\boldsymbol{G}$ and $\boldsymbol{Y}$.

8.1.3 Principal components in R

We will work initially with the data set `temperature.dat`. This contains observations of the average daily temperature at 15 locations from 1 January 1997 to 31 December 2001. We generate a data frame, check variable names and calculate the values p (number of variables) and n (number of observations) in the usual way.

```
> temps <- read.table("Temperature.dat", header=TRUE)
> names(temps)
> p <- length(temps)
> n <- length(temps$DAB)
```

It will be useful later to have our data in matrix format. We use `as.matrix()` to convert a data frame to a matrix, then work out the sample mean and covariance matrix

```
> X <- as.matrix(temps)
> xbar <- colMeans(X)
> S <- var(X)
```

We could determine the principal components directly by computing the eigenvalues and eigenvectors of $\boldsymbol{S}$. However, there are functions available in R to perform these calculations. The function `princomp(...)` takes a data matrix argument and returns a list (of class `princomp`). Information about the principal components can be extracted from this list; `summary(...)` and `plot(...)` provide the variance of each component while `loadings(...)` gives the values in the matrix $\boldsymbol{G}$ used to transform to principal components.

```
> temps.pca <- princomp(temps)
> summary(temps.pca)
> loadings(temps.pca)
> plot(temps.pca)
```

The principal components themselves are given by the `predict(...)` command

```
> temps.pr <- predict(temps.pca)
> Y <- as.matrix(temps.pr)
```

We gain some insight into the nature of the principal components by plotting

```
> plot(Y[,1], type="l", col=2)
> plot(Y[,2], type="l", col=2)
```

Which feature of the data does the first principal component represent? Is this surprising? What about the second principal component?

We can generate approximations to our data matrix from the principal components. We first set up the matrix of eigenvectors $\boldsymbol{G}$ and the matrix of mean values $\mathbf{1}\bar{\boldsymbol{x}}$ (note `temps.pca$center` and `colMeans(X)` are the same thing)

```
> G <- matrix(as.numeric(temps.pca$loadings),p,p)
> xbarvec <- matrix(rep(as.numeric(temps.pca$center),each=n), n, p)
```

Using just the first principal component

```
> Xpc1 <- xbarvec + matrix(Y[,1],n,1) %*% matrix(t(G)[1,],1,p)
> plot(X[,1], pch=19, cex=0.3, col=2)
> lines(1:n, Xpc1[,1], col=4)
```

Note the rather awkward matrix construction that we have to do here since R interprets `Y[,1]` as a vector. This is not the case when we use more than one principal component

```
> Xpc2 <- xbarvec + Y[,1:2] %*% t(G)[1:2,]
> plot(X[,4], pch=19, cex=0.3, col=2)
> lines(1:n, Xpc2[,4], col=4)
```

Do you think that using two principal components provides a reasonable approximation to the data?

8.2 Cluster analysis

In this section we will use data on the weekly closing prices of six stocks recording for the weeks starting 22nd November 2004 to 7th November 2005.

```
price <- read.table("price.dat", header=TRUE)
Pmat <- as.matrix(price)
```

We can calculate log return for the data matrix

```
lagPmat <- rbind(rep(NA,6),Pmat[1:(nrow(Pmat)-1),])
logRet <- log(Pmat) - log(lagPmat)
```

Using the notation of the previous section, suppose we observe $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ where each $\boldsymbol{x}_i$ is a $1 \times p$ vector. There are numerous approaches to clustering available; we consider an agglomerative clustering method. Starting with n clusters, points are brought into clusters one by one. The aim is to keep dissimilarity within clusters low compared to dissimilarity between clusters. We measure dissimilarity using the usual Euclidean distances between the rows of the data matrix calculated by `dist(...)`. The clustering method `hclust(...)` takes a matrix of distances as an argument and returns a list (of class `hclust`). The `plot(...)` command applied to the output of `hclust` produces a dendrogram illustrating how the clusters are put together. We can divide the data up into a given number of clusters using the `cutree(...)` command; this is illustrated below dividing the data into 4 clusters.

```
> h <- hclust(dist(logRet[2:51]))
> plot(h, cex=0.5)
> cutree(h,4)
```

Conventional wisdom is that financial data contains volatility clustering. One rather crude way to look for this is to perform cluster analysis on the square returns.

```
> sqrRet <- logRet*logRet
> h <- hclust(dist(sqrRet[2:51]))
> plot(h,cex=0.5)
> cutree(h,2)
```

Cluster analysis is not guaranteed to produce sensible clusters. We can sometime identify structure from simple graphics such as those generated by `stars(...)`. This produces a glyph for each $\boldsymbol{x}_i$ with shape determined by the values of the p variables.

```
> stars(sqrRet, key.loc=c(10,2))
```

8.3 Exercise

1. For the weather example in §8.1.3, find the difference between the data matrix and the approximation generated using just the first principal component. Is it reasonable to model this difference as IID normal? What about when 2 or 3 principal components are used?
2. † Our principal components in §8.1.3 are dominated by the first component. We might consider principal components analysis in series for which this dominant component has been removed. Using an appropriate model for seasonality, generate deseasonalised data and apply principal components. What is the dominant effect now?
3. † From finance.yahoo.com, collect a long series of daily highs and lows from a number of stocks in the same sector. The difference between high and low is a measure of volatility. Can you identify volatility clustering using these differences?

8.4 Revision exercise

1. The plain text data file `prices.dat` contains the daily closing prices ($) and log returns of Intel stock and the NASDAQ index from 1st February 2000 to 31st July 2000. The first line of the file gives the variable names. Create a data frame `inp` using:

   ```
   > inp <- read.table("prices.dat", header=TRUE)
   ```

   (a) List the names of the variables in `inp`.

   (b) Work out the mean, variance and minimum for the NASDAQ log returns.

   (c) Generate a scatterplot of Intel log returns against NASDAQ log returns. Identify the month and day of any unusual points and comment on the plot.

   (d) Generate a time series plot with Intel log returns and NASDAQ log returns in different colours on the same plot. Comment on this plot with reference to the observations made in part (c).

   (e) Using appropriate selection and creating additional objects where necessary, find the following using R [note that months are labeled using their first three letters]:

      i. the mean of Intel prices when the Intel log returns are greater than zero,

      ii. the standard deviation of Intel log returns for March,

      iii. the days in April where Intel log returns are higher than NASDAQ log returns,

      iv. the largest difference between Intel log returns and NASDAQ log returns for May and June,

   (f) At the start of the period (1st February) we have 1000 Intel shares and $0 in the bank. We adopt the following trading strategy:

      - if the closing price of Intel shares goes above $130 on any day we sell 10 shares (you may assume we receive the closing price on that day for each share),
      - if the closing price goes below $110 on any day we buy 10 shares (you may assume we pay the closing price on that day for the each of the shares and, if we have no money, we borrow interest free from the bank).

How many shares do we have and how much money do we have in the bank at the end of the period (31st July)?

2. It has been suggested that the Intel log returns can be represented using simple regression with the NASDAQ log returns as the explanatory variable. The model is

$$Y_t = \alpha + \beta x_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma^2) \tag{8.2}$$

where Y_t is Intel log return, x_t is NASDAQ log return and the ε_t is a noise term for $t = 1, \ldots, n$.

   (a) Fit model (8.2) using the Intel log returns and NASDAQ log returns from the `inp` data frame created in question 1. Give the parameter estimated and associated p-values. Give the null hypotheses and conclusions of the associated hypothesis test.

   (b) We would like to include month of the year as a factor in our linear model.

      i. How many explanatory variables are associated with the factor `Month`? When the factor is included in a linear model, what is the nature of the explanatory variables?

      ii. Include `Month` in our existing model for Intel log returns. Is month a significant explanatory factor? Give the reason for your answer.

      iii. Fit a model for Intel log returns that includes the interaction between month of the year and NASDAQ log returns along with all the marginal terms. Using the output from this model, write down the estimated linear association between Intel log returns and NASDAQ log returns for July.

   (c) Write a function that takes a linear model object as an argument and returns the date of the point with the largest (in absolute value) residual and the value of that residual. Apply this function to the model fitted in part (b)iii. and write down the date of the point with the largest absolute residual and the value of the residual.

   (d) Generate an indicator for the date given in part (c) (that is, a vector that takes the value 1 on that date and zero everywhere else). Include this indicator as an explanatory variable in the model generated in part (b)iii. Give the estimated coefficient for the indicator. What does this estimate represent? Compare the standard diagnostic plots for the model without the indicator to those with the indicator.

3. Consider the regression model

$$Y_t = \beta x_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma^2) \tag{8.3}$$

where $t = 1, \ldots, n$ and the ε_t are independent.

   (a) Write a function that takes three arguments, the slope parameter β, the error standard deviation σ, and a vector of explanatory variables x. The default values for β and σ should both be 1. The function should return simulated values $y_1, \ldots, y_n$ from model (8.3) where n is the length of the explanatory variable vector. Set the random number generator seed to 7 and test your function on $x = (1, 2, \ldots, 30)$ using the default values of β and σ. Give the minimum and maximum of the simulated y values.

(b) Write a function that uses the function in part (a) and returns a vector of r instances of the parameter estimate $\hat{\beta}$. Set the random number generator seed to 7 and test your function on $x = (1, 2, \ldots, 30)$ and $r = 50$ using the default values of β and σ. Give the mean and variance of the resulting vector of parameter estimates.

(c) Write a function that simulates instance of $\hat{\beta}$ for every possible combination of $\beta = 0.5$ and 1.5, and $\sigma = 1$, 2 and 4 and returns the approximated mean square error for each combination. Set the random number generator seed to 7 and use your function with the NASDAQ log returns from the data frame `inp` created in question 1 as the explanatory variable and $r = 100$.

8.5 Reading

8.5.1 Directly related reading

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [Parts of sections 11.1 and 11.2.]

8.5.2 Supplementary reading†

- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*, 4th edition. Springer. [The remainder of chapter 11.]
- Mardia, K. V., Kent, J. T. and Bibby, J. M. (1979) *Multivariate Analysis.* Springer-Verlag. [A good reference source.]