STATS240.1 A short course in R

Gunnar Stefánsson, Ásta Jenný Sigurðardóttir and Lorna Taylor

October 17, 2012

ACKNOWLEDGEMENTS

This course has been developed in part as it has been taught at the Univ. Iceland Dept. of Biology http://www.hi.is

Contents

1 Introduction and installation **3**

1.1 About R . . . 3

1.2 Introduction . . . 4

1.3 Documentation . . . 4

1.4 Working with R . . . 4

2 Introduction to data analysis in R **6**

2.1 Starting up . . . 6

2.2 Before you begin . . . 6

2.3 Data vectors in R . . . 8

2.4 Deleting and listing objects . . . 9

2.5 Entering data into a file . . . 10

2.6 Reading data into R . . . 11

2.7 Data summaries in R . . . 12

2.8 Random numbers in R . . . 13

2.9 Simple data plots in R . . . 13

2.10 R and emacs . . . 14

3 Command files **17**

3.1 Repeated commands . . . 17

3.2 Command files . . . 17

3.3 Running commands stored in a file . . . 18

3.4 Batch runs . . . 18

4 Functions in R **19**

4.1 Introduction to functions in R . . . 19

4.2 Functions in command files . . . 19

4.3 Plotting functions . . . 20

4.4 Run commands . . . 21

5 Statistical models **22**

5.1 Linear statistical models . . . 22

5.2 Nonlinear statistical models . . . 24

5.3 Miscellaneous statistical models . . . 25
5.4 Further reading . . . 26

6 Data structures in R **27**
6.1 Vectors . . . 27
6.2 Naming vector elements . . . 27
6.3 Indexing vectors . . . 28
6.4 Arrays and matrices . . . 28
6.5 Indexing arrays and matrices . . . 29
6.6 Names of rows and columns . . . 30
6.7 Lists . . . 30
6.8 Data frames . . . 31

7 Advanced data manipulation in R **36**
7.1 Tabular summaries in R . . . 36
7.2 Frequency tables . . . 36
7.3 Row and column summaries . . . 37
7.4 Operations on data columns . . . 38
7.5 Other tabular functions . . . 38

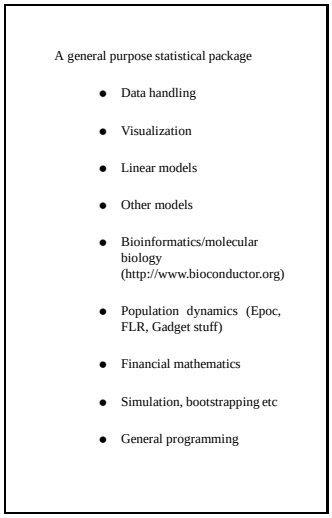
8 Plotting with R **40**
8.1 Some common plot commands . . . 40
8.2 More plot commands . . . 41

9 Programming R **43**
9.1 The if statement . . . 43
9.2 Loops - for . . . 43
9.3 Storing loop results . . . 44
9.4 Loops - while . . . 45

10 Further reading **47**
10.1 Reading material . . . 47

1 Introduction and installation

1.1 About R



The R package is free software, copyrighted by the GNU General Public License.

It can be obtained from the R project web page: http://www.r-project.org/ as a binary or source code.

Installation should be a trivial exercise.

Note that R is available for most operating system. The followin notes emphasize running R under Linux, but the differences only appear when R connects to the operating system (e.g. through reading data files).

R is an open source extensible general purpose statistical package. It can thus be used for simple and elaborate data analysis, and R excels at visualization.

Several web pages are dedicated to R. These include the R project homepage, http://www.r-project.org.

Since R is extensible and has its own programming language, it can in principle be used for any arbitrary computations.

Many scientific fields have chosen R as the primary tool. The reason for this is the combination of extensibility with access to a large collection of tools such as statistical and plotting routines.

In fishery science, Epoc is an ecosystem simulator and FLR is a general stock assessment tool, both written in R. Output from the ecosystem toolbox Gadget is traditionally analysed using R and a simulator has been written in R to generate simple versions of Gadget populations. R has also been used to simulate the effects of management measures such as marine protected areas etc.

Statistics in molecular biology is a field which extensively uses R and has a large web site dedicated to using R for the analysis of data on gene expression as well as very many other uses. It is even possible to find an interesting DNA-sequence and use R to find abstracts of publications which refer to this sequence.

Statistical models in R include the usual linear models but also include nonlinear models along with models with error structures other than Gaussian, e.g. generalized linear models (GLMs), generalized additive models (GAMs) etc.

Naturally R has simulation routines and can generate random numbers from all sorts of distributions. Combined with the programming environment this makes R well suited as a simulation tool. In this case R is also easily implemented on computer clusters where simulations are commonly run in parallel on

dozens of computers simultaneously.

1.2 Introduction

Obtain R from http://www.r-project.org/
Read instructions and install

The R package is free software, copyrighted by the GNU General Public License.

It can be obtained from the R project web page: http://www.r-project.org/ as a binary or source code.

Installation should be a trivial exercise.

Note that R is available for most operating system. The followin notes emphasize running R under Linux, but the differences only appear when R connects to the operating system (e.g. through reading data files).

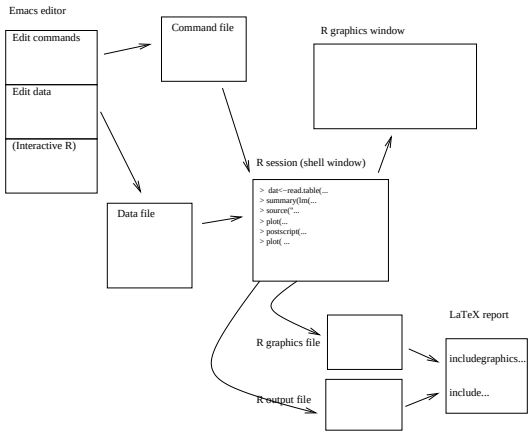
1.3 Documentation

See http://cran.r-project.org/manuals.html
or various books on R.

The R project home page contains several manuals and links to further material (see http://cran.r-project.org/manuals.html or various books on R).

Documentation on Splus is also applicable to R as most commands are exactly the same.

1.4 Working with R



Although R can be used in many environments, by far the most productive way to use R is by running R under Linux, using emacs for editing command files and using LaTeX for all word processing.

The details of how analyses are undertaken and how reporting is done will vary. However, in order to work in a structured and organized manner it is essential to keep a record of the commands used in the analysis. Hence command files are needed to store commands to be re-used.

Similarly, graphical output is best stored in files rather than only viewed on-screen. Thus, normally plots are first generated interactively on-screen and later, as the proper plots appear, these get stored in file. Commands to generate the plots are also stored in files and thus it is possible to revisit the entire analysis at a later stage.

Command files are used to keep a record of work and to store frequently-used commands. For example, when developing models, the approach is to first develop models interactively until an appropriate model is found. Then the corresponding commands are stored in a file so as to provide a record of precisely which model was used. Similarly, often-complex input commands are stored in files. A "source" command is used to execute the commands in the file. There are several reasons for this:

- Avoiding repetition
- Typographical errors are minimized
- A formal record is kept of the commands used

Similarly, graphical output is best stored in graphics files rather than only viewed on-screen. Thus, normally plots are first generated interactively on-screen and later, as the proper plots appear, these get stored in file. Commands to generate the plots are also stored in files and thus it is possible to revisit the entire analysis at a later stage.

R users should **never** use copy and paste to take a plot from a plotting window into a word processor. There are simply much better ways to do this. R can generate plots in a wide variety of different formats and these are best stored in files once the plot is considered complete and the word processor is then asked to use the plot in the file.

An experienced R user will always have a complete record of how every plot was generated: A command file will exist to show exactly how the plot was made and placed in a plot file. The word processing file will contain a link to the plot in a plot file.

References

The R project on-line manuals http://cran.r-project.org/manuals.html

%T An Introduction to R %A W. N. Venables, D. M. Smith %D 2002 %I Network Theory Ltd. %P 156pp ISBN: 0954161742

%T Introductory Statistics with R %A Peter Dalgaard %D 2002 %I Springer-Verlag %P 288pp ISBN: 0387954759

2 Introduction to data analysis in R

2.1 Starting up

Linux: Enter "R" at the shell prompt
Windows: Find R in the startup menu

Under Linux, start up a command (shell) window and then give the command "R" (note the upper case).

Under Windows, find R in the startup menu or as an icon somewhere (e.g. on the desktop).

R will then start up and wait for the user to enter commands.

2.2 Before you begin

“>” is the R prompt

Comments are marked with #

To get detailed information about a function, type:

> help("function name")

Notice that most R commands are actually calls to **functions** which implies that they are followed by parentheses.

The prompt from R is “>”, which means that R is ready to accept a command from the user. If you type in examples from the text be sure to omit the prompt character.
Comments are marked with #

To get detailed information about a function, type:

> help("function name")

R is normally command-orientated. This means that R expects the user to type in a command from the keyboard, rather than clicking with the mouse at buttons in various places on the screen.

These commands are entered at the R command prompt (or stored in command files). When a command has been given it is possible to re-run the same command by using the arrow keys to "back up" through the command history and hit enter at the command to be re-run. It is also possible to edit the commands in the history list and run the modified command. This is the recommended method of running R interactively.

In particular, the user should **never enter a complex command twice**. Rather, the previously run version should simply be edited. This greatly reduces the probability of new errors being entered.

R in Linux

Under Linux it is customary to begin in a command (shell) window. This can normally be started up by right-clicking on the background and selecting the appropriate item, or manually firing up gnome-terminal (better than xterm).

This Linux command window is now ready to accept any commands from the user. Usually this is indicates by a dollar-sign. For example one can give the command “ls” to list files, “mkdir newdir” to

create a new directory (folder) etc. Select a directory for R work and go to that directory, for example:

```
$ cd
$ mkdir test
$ cd test
$ mkdir R
$ cd R
```

The next step is to start up emacs or xemacs. For this tutorial xemacs will be used.

```
$ xemacs test.r &
```

This command tells xemacs to start up a new window where the file "test.r" is to be edited. This will be opened as a new file if it does not already exist. The apersand at the end of the line is an indication to the Linux shell that xemacs is to run in background. If this is not done, then the shell will wait for xemacs to complete before we can give any new commands.

When reading in data from files (or running scripts) it is often convenient to start R in, or close to, the directory the files are stored in.

To quit R:
`q()`

You will be given 3 options: `y/n/c`

`y` saves the 'objects' you have created and exits.
`n` just exits.
`c` cancels the quit command.

For now you can just use `n`.

In R you can repeat, and recall and edit previous commands using the up and down arrows.

Help

When in R, for help on individual commands do e.g.:

```
help(mean)
?  mean
```

within help use

<space>	to page down the help
q	to quit help
f	to move forwards
b	to move backwards

At the end of help you will find examples of how the commands are used.

R in Windows

R can also be run in Windows with the R gui opened either from an icon or a menu.

To select the directory in which R is started:

File → Change dir ...

To quit R:
`q()`

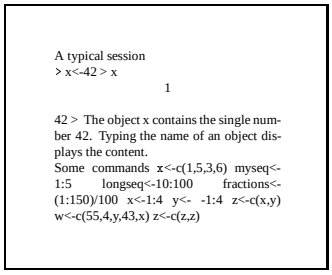
You will be given the option of saving the workspace.

*H*elp

When in R, for help on individual commands do eg:
`help(mean)`
`? mean`

At the end of help you will find examples of how the commands are used.

2.3 Data vectors in R



The symbol "<-" denotes assignment. Thus, "x<-1" means "assign the numerical value 1 to the object x". In simple words this means that "x" contains the single number "1" after the operation.

The assignment operator is used to assign numerical, character or more complex values to objects. If the object on the left hand side of the assignment exists it is overwritten. If it does not exist it will be overwritten.

Alternatively, one can use the equals sign for assignment in R. It is, however, better practice to use “<-” since this should be thought of as a left-arrow symbol and clearly can only mean an assignment. Earlier versions of S and Splus did not allow a single equals sign for this purpose but instead permitted the use of an underscore. The only symbol which works in all cases is “<-” and it is therefore the recommended version.

Example: To put a number into a variable enter the command

```
x<-1
```

To display the contents of such an object just type the name of the object. The entire session could look like this:

```
> x<-42
> x
[1] 42
>
```

To enter a series of numbers:

```
x<-c(1,5,3,6)
```

Shorthand for a sequence:

```
myseq<-1:5
longseq<-10:100
fractions<-(1:150)/100
```

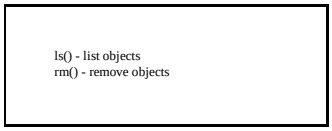
where the last line actually generates the numbers 0.01, 0.02, ..., 1.50.

The “c” command can be used to combine one or more series of numbers:

```
x<-1:4
y<- -1:4
z<-c(x,y)
w<-c(55,4,y,43,x)
z<-c(z,z)
```

The user should try all of these commands until they become completely natural.

2.4 Deleting and listing objects



To get a list of available object use the “ls()”-command. To delete files, use the “rm()”-command.

Notice that most R commands are actually calls to **functions** which implies that they are followed by parentheses, as in “rm(x)” to delete the object “x”.

Notice also that merely typing the name of a function:

```
> ls
```

just outputs the “ls” function itself - not the list of objects.

Note: A very common source of confusion is where commands are given: R commands can only be given to R, normally at the R prompt. Linux shell commands are given to a Linux shell only.

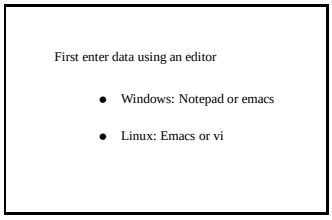
Some of these commands are similar enough that beginners give R commands to the shell and shell commands to R. It is important to remember that R commands can only be given while R is running and this is always indicated by the “>” prompt.

Commands can be continued onto a second line if needed. R will do this automatically and assume a continuation line is intended when a command is obviously not complete. For example, the following may describe a session:

```
> x<-c(1,2,
+ 3,4)
> x
[1] 1 2 3 4
> y<-1:4
> x+
+ y
[1] 2 4 6 8
>
```

Notice, therefore that if a user incorrectly enters e.g. an arithmetic symbol at the end of a line, then R will assume that a continuation is required and start the next line with a plus-symbol. In the case us such an error, the user should hit control-C to cancel the command.

2.5 Entering data into a file



Data are normally first entered into data files and subsequently read into the program which is to be used to analyse the data. It is almost always a poor idea to enter data directly into an analysis program without maintaining an external data file.

Small data sets are commonly entered by hand using a text editor. Formatting programs such as Word, WordPerfect or StarOffice should not be used for data entry. The data file should contain only data, though a single line at the top of the file, describing what is in each column, can be useful. For the same reason, spreadsheets should not be used to enter data.

A data file should thus never contain formatting commands or colored blocks of text etc. It should contain only numbers or codes to be used in analysis. When this simple rule is adhered to the data can be entered into any package whatsoever and can be validated using standard programs.

To enter data under **Windows** one would normally use Notepad, though a much better approach is to obtain some version of emacs for Windows.

A still better approach is to install a more data-orientated operating system such as Linux (or any Unix variant, but Linux happens to be free). The reason for this preference is that the sheer number of small programs for simple data manipulation under Linux far outweighs the time taken to install the operating system, if the intent is to do considerable data analysis.

With the advent of R it is possible to run the same analysis system under several operating systems, thus outweighing some of the inconveniences of unstable operating systems. Nonetheless, for larger data sets it quickly becomes difficult or impossible to analyse them except on systems which are designed for handling large amounts of data.

Data entry by hand on a **Linux** system normally consists of first starting up a terminal window followed by starting a text editor and indicating which file is supposed to contain the data.

For the “vi” editor this becomes the sequence:

vi mydata.dat	start the editor
i	go into insert mode
1 4.5	enter data, one
2 5	line at a time
<esc>	leave insert mode
:wq	write the file and quit editing

The emacs editor does not have command- or insert-modes but accepts commands as control-sequences:

emacs mydata.dat	start the editor
1 4.5	enter data, one
2 5	line at a time
C-XC-S	save the file
C-XC-C	exit the editor

A note on file names:

File names should be “short and sweet”. There is never any good reason to use 52 character file names containing spaces and international characters. The “ending” of a file name is usually 1-4 characters

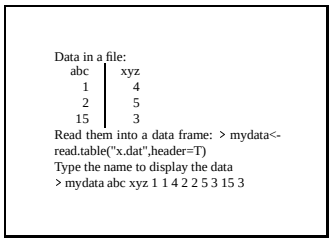
and is normally used to indicate the type of file in question. Thus one could use file names such as

```
mystuff.dat   |  a data file
plotting.r    |  a file with R plotting commands
hakewts.dat   |  the obvious thing
```

The use of spaces is filenames (not to mention directory names) is an extremely bad habit and the same applies to international characters. These **will** eventually cause problems when files are exchanged between people using different types or versions of computers, operating systems or languages.

In particular, it is a really, really bad idea to use spaces or international characters when using a system such as R which is designed to work on multiple platforms.

2.6 Reading data into R



Assuming that data exists in a simple text file, the next step is to read it into the program to be used for analysis.

The easiest way to read data into R is to use a single line at the top to indicate column names.
Note: To get a good data set for the following examples, reopen mydata.dat and add lines so it looks like this:

x	y
1	4.5
2	5
9	3
2	1
1	5.5

Then use the R command read.table()

Note that read.table() has many options and can handle a variety of different file formats.

Example: Note that the "> " is the prompt from R and is not entered by the user.

In the following R session the user tells R to read the file mydata.dat into R calling the result mydata.

```
> mydata<-read.table("mydata.dat",header=T)
```

Since the first line contains the name of the columns we choose header=T

To see the data in R type the name of the object containing the data set:

```
> mydata
    x    y
1   1   4.5
2   2   5
3   9   3
4   2   1
5   1   5.5
```

The resulting “mydata” which contains the data is a data frame (see chapter 6.1 on data structures).

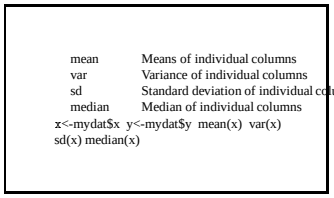
Several methods exist to enter data into R. The most common method for multi-column data is the above “read.table” command, whereas “scan” is easier to use for individual data columns.

The “scan”-function can also be used to read a single column of numbers from the keyboard:

```
> x<-scan()
1: 1 2 3
4: 2
5: 3
6: 7 5 4
9:
Read 8 items
> x
[1] 1 2 3 2 3 7 5 4
>
```

The “read.table” command returns a “data frame” (more on this later). Each column in the data appears as a column in the data fram and can be referenced using the dollar sign, e.g. with “mydata$x”.

2.7 Data summaries in R



For computing averages of data several approaches are available. in R, including the following commands:

mean	Mean of individual column
var	Variance of individual column

Example: If the data frame “mydata” contains “x” and “y” as columns, then these can first be extracted and the mean of “x” computed with the commands

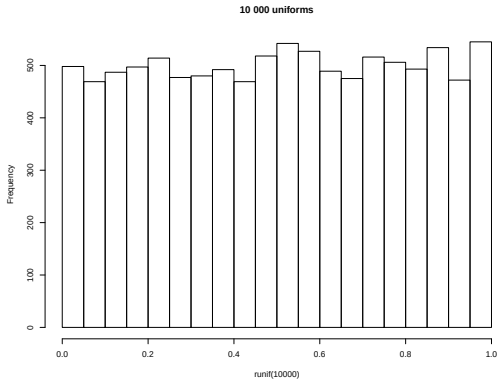
```
> x<-mydata$x
> x
[1] 1 2 9 2 1
> y<-mydata$y
> y
[1] 4.5 5.0 3.0 1.0 5.5
> mean(x)
[1] 3
```

The variance of the “x”-values is obtained with

```
> var(x)
[1] 11.5
```

and the standard deviation can be obtained with the “sd” function.

2.8 Random numbers in R



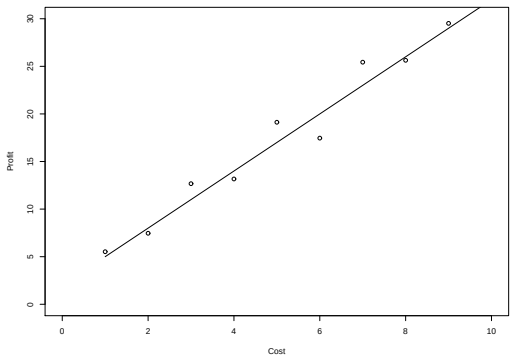
Random numbers are very useful for checking out properties, sampling schemes etc.
Common functions:
rnorm()
runif()

Some examples:

Random normal (Gaussian), single number:
rnorm() Uniform between 0 and 1, ten numbers:
runif(10) Single observation from a binomial distribution with 5 trials each a probability 1/2 of success:
rbinom(1,5,.5)

A typical way of looking at random numbers:
hist(runif(10000),main="10 000 uniforms",nclass=50)
Note the use of nclass.

2.9 Simple data plots in R



x<-1:10 z<-2+3*x y<-z+rnorm(10) plot(x,y, xlim=c(0,10),ylim=c(0,30), xlab="Cost",ylab="Profit") lines(x,z)

Figure 1: A typical scatterplot from R

The most illustrative plots of data include scatterplots and line plots.

Most spreadsheets do not provide proper line plots. Thus, the incorrect assumption is usually made that the data which go on the x-axis are categorical variables and the data is centered on these as if they were groups. One solution to this is to always ask for scatterplots when plotting data within spreadsheets. Within scatterplots options it is subsequently possible to ask that the points be joined by lines. A much better option is to use packages designed for visualising data in a scientific manner.

Within R the simplest family of plotting functions is the "plot" family, which includes the "plot", "lines" and "points" functions. Options to these are numerous and best viewed through the associated help files.

Before any model-based analysis, naturally the data should first be plotted in the form of a scatterplot in order to verify that a straight line makes sense, using

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

Note: The plot command has a long list of options and versions of the plot command exist for many different types of objects.

Usually a simple quick-and-dirty scatterplot is simply generated with "plot(x,y)", but usually one will want to modify this plot in various ways.

Suppose we have the following "x", "y" and "z"-vectors:

```
x<-1:10
z<-2+3*x
y<-z+rnorm(10)
```

where the x-variable simply contains the numbers 1, 2, ..., 10, the z-vector describes a straight line in x and y is a simulated set of numbers with Gaussian measurement errors around the line.

Some typical methods of plotting such pieces of information include:

```
plot(x,y)              # A point plot (scatterplot)
plot(x,z,type='l')     # Just the line
```

Adding a line to an existing plot is often useful

```
plot(x,y)              # A point plot (scatterplot)
lines(x,z)             # Add the line
```

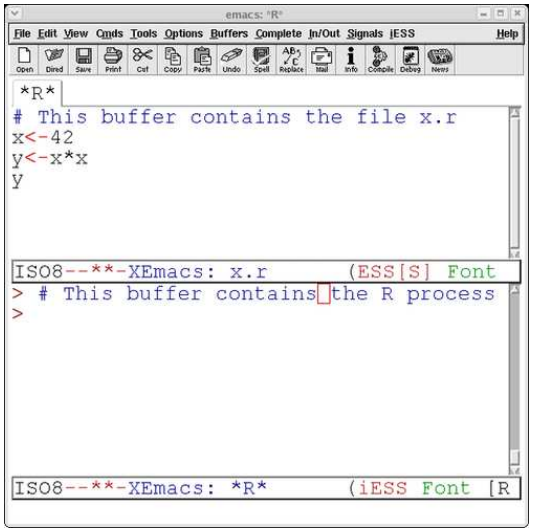
The default limits or labels on the axes are not always sensible and of course these can be changed:

```
plot(x,y,xlim=c(0,10),ylim=c(0,30),xlab="Cost",ylab="Profit")
lines(x,z)
```

2.10 R and emacs

There is considerable support within emacs for different computer languages and other programs. For example, emacs knows how to indent R functions, check that parentheses are closed etc.

Further, emacs can start up R so the program runs within an emacs buffer. This provides an easy way for copying and pasting R commands from one buffer into another. On the other hand emacs also provides editing functions which e.g. put the current line into the R buffer etc.



R/S-mode for .r-files
Interactive R/S startup within emacs!

Additional notes/practicals

R in Emacs

R is run either in the shell window or as a process within emacs/xemacs. The following describes the latter. Under Linux, start up emacs/xemacs from a terminal window. Under Windows, make sure your have started up xemacs with the appropriate additions and start up xemacs.

A new window should now be open, where xemacs waits for us to modify the file test.r. At the bottom of the window, xemacs display ESS[S] which indicates that xemacs is in ESS-**mode** where ESS is short-hand for Emacs Speaks Statistics. This mode has several useful features for editing R or Splus code.

If you have an xemacs/emacs window open, then you can split it in 2 parts, so you can have a record of you commands in one part of the window and run R in the other part.

The emacs and xemacs window commands are not quite the same, though most keyboard commands are. To start up R within xemacs, either select from the menu, Tools->Statistics->R Runtime or use the keyboard commands “M-X R”. The command M-X is emacs-speak, pronounced Meta-X and is usually implemented through <escape>-X (first hit the escape button and then x).

To do this:

File → Split window

If you do that you will also see the equivalent Ctrl command. What is it?

To start R you then do:

Tools → Statistics → R runtime

At the bottom of the Emacs window the directory to run R in is selected, to run R in the directory you are in hit return.

If you open a file with a `.splus` or `.r` extension, eg plot-run1.splus, in Emacs, it will highlight the text in different colours which may help with understanding the syntax. It is the file extension that tells Emacs the language you are using. Emacs also checks the parentheses (ie `(, ), {, }, [,]`) match which is very useful in longer commands and when writing functions.

Now xemacs has two main buffers, one contains the file test.r and the other contains the R process where R awaits your commands. You only see the R buffer, however. You can enter R commands manually into this buffer.

Normally one wants to view both the file and the R process at the same time. This is done by splitting the window using either the View->Split command or C-X2 (C-X is Control-X – hold down the control key while pressing x).

Then change buffer in one of the windows, using C-XB or by selecting the appropriate buffer in the Buffer menu.

You now have an R window and a window containing the R commands which can be either stored in the named file or executed.

Several web pages describe the ESS command set. The main point of these is htat when you type R commands into Emacs, they can be run in R using keyboard shortcuts.

The most useful are:

`Ctrl c Ctrl j`	runs the line
`Ctrl c Ctrl r`	runs a highlighted region
`Ctrl c Ctrl b`	runs the file (buffer)

Naturally, once the R commands are saved as a file, e.g. "test.r" has been saved one can use the R command

```
source("test.r")
```

to run the commands through R. This can be done anywhere, with R in emacs or from R running on the Linux shell command line.

In Emacs and Xemacs, help is opened in a buffer. If, after opening a help file in Emacs you cannot see the file you were working from you can recall it using the **Buffers** menu which lists all files open in the Emacs window.

References

%T An Introduction to R %A W. N. Venables, D. M. Smith %D 2002 %I Network Theory Ltd. %P 156pp ISBN: 0954161742

%T Introductory Statistics with R %A Peter Dalgaard %D 2002 %I Springer-Verlag %P 288pp ISBN: 0387954759

R Project home page http://www.r-project.org/

R manual page http://cran.r-project.org/manuals.html

Contributed R documentation http://cran.r-project.org/other-docs.html

R FAQ http://www.ugcs.caltech.edu/info/R/R-FAQ_toc.html

R newsletter http://cran.r-project.org/doc/Rnews/

ESS: Emacs Speack Statistics http://stat.ethz.ch/ESS/

Reference card for R within emacs (ESS) http://stat.ethz.ch/ESS/refcard.pdf

R and Emacs tutorial http://www.stat.vt.edu/ sbates/teaching/s4004/Handouts/RESS.pdf

Emacs commands http://www.utexas.edu/cc/docs/ccrl34.html

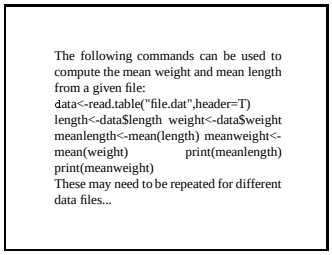
R for Windows FAQ http://www.stats.ox.ac.uk/pub/R/rw-FAQ.html

Xemacs for Windows http://www.xemacs.org/Download/win32/

Xemacs for Windows Download link http://ftp.dk.xemacs.org/pub/emacs/xemacs/xemacs-21.4-windows/

3 Command files

3.1 Repeated commands



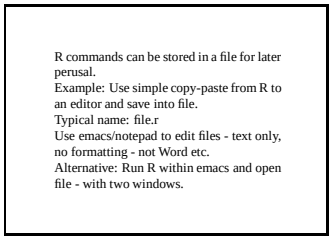
Few tasks are only done once. Most real-life projects involve several repetitions of the same or very similar computations. When these computations are more than single lines, it is very useful to store the commands as files which can be called upon to rerun the sequence with little effort.

Example: Suppose there are several data files for length and weight and the mean length and mean weight are to be computed from the data in each file. This would indicate that the following commands need to be run for each file.

```
data<-read.table("file.dat",header=T)
length<-data$length
weight<-data$weight
meanlength<-mean(length)
meanweight<-mean(weight)
print(meanlength)
print(meanweight)
```

When these commands are re-run for different data sets, the only change is to the name of the data set, i.e. the first line. It is obvious that there are considerable savings if the commands do not need to be re-entered every time.

3.2 Command files

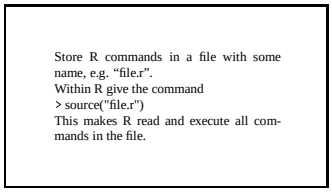


Storing the commands in a file can save substantial time and effort. Normally the name of such a command file would be of the form file.r.

A command file should contain only text, no formatting such as boldface or colors etc.

Command files are normally edited using emacs or xemacs since these editors know how to display and indent R code.

3.3 Running commands stored in a file



Giving the command

```
source("file.r")
```

causes R to execute all the commands in the file.

Notice that when R is running commands automatically in this manner, R does not print output from most commands. Thus, in order to see actual output, a formal "print" command (or equivalent) needs to be given within the file.

Such command files frequently contain commands to print results prettily rather than print inexplicable numbers alone.

Example: Suppose the file lw.dat contains the following data:

length	weight
34	3.2
42	3.5
37	3.0

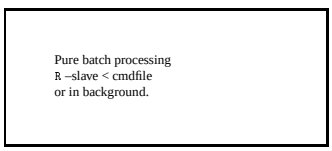
A second file, e.g. calcmean.r may contain the commands to analyse these data and to print some basic results:

```
data<-read.table("lw.dat",header=T)
length<-data$length
weight<-data$weight
meanlength<-mean(length)
meanweight<-mean(weight)
cat("The mean weight is: ",round(meanweight,2),"\n")
cat("The mean length is: ",round(meanlength,2),"\n")
```

The "source" command can now be given within R:

```
> source("calcmean.r")
The mean weight is:  3.23
The mean length is:  37.67
```

3.4 Batch runs



In addition to running R interactively, it can be run in non-interactive mode. A particularly useful approach under Linux is

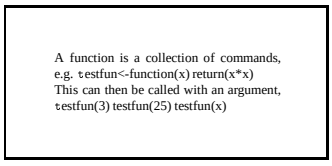
```
R --slave < cmdfile  &
```

When this approach is used, care must be taken to make sure that output is into text files and plot files rather than onto the screen.

If output is given to the screen, a command such as “script” can be given in the shell to store output.

4 Functions in R

4.1 Introduction to functions in R



A function is a collection of commands, e.g.

```
testfun<-function(x){
  return(x*x)
}
```

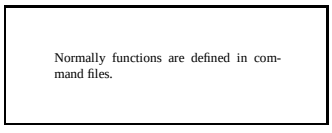
This can then be called with an argument,

```
testfun(3)
testfun(25)
testfun(x)
```

Example: The following function will calculate the sum of two numbers

```
> sum<-function(x,y){ # call the function sum and the input x and y
+  s<-x+y             # define s as the sum
+  return(s)          # let the function return the sum
+  }

> c<-sum(2,3)         # try the function
> c
[1] 5
```

4.2 Functions in command files



Although functions can be defined directly from the keyboard, this is usually not fruitful since typically several iterations are needed before a function has been correctly defined. A better approach is therefore to define the function within a command file.

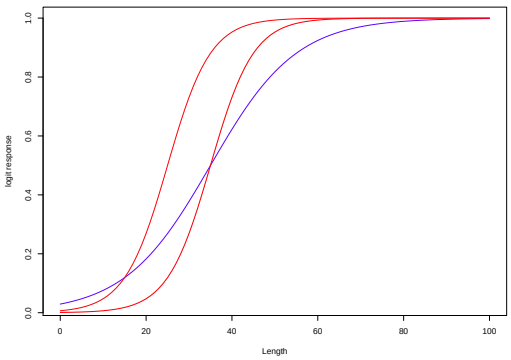
Example: Suppose the file cmd.r contains the following lines:

```
f<-function(x){
 y<-x+2
 s<-sum(y)
 return(s)
}
```

The following describes a typical use of this function

```
> source("cmd.r")
> f(2)
```

4.3 Plotting functions



Functions are commonly used for plotting

Example: Suppose we define a logistic function in the file “function.r”, i.e. the file contains the following lines:

```
f<-function(x,a,b){
  y<-1/(1+exp(-b*(x-a)))
}
```

This function can now be read into R using the command source("function.r"). This only **defines** the function, however. Note that the **arguments** to the function are the *x*-values in the vector “x” and the two parameters to the logistic.

In order to call the function we subsequently need a few more lines and a typical R session might be

```
source("function.r")
x<-0:100
y1<-f(x,35,.1)
y2<-f(x,35,.2)
y3<-f(x,25,.2)
plot(x,y1,type='l',col="blue",xlab="Length",ylab="logit response")
lines(x,y2,col="red",lwd=2)
lines(x,y3,col="red",lwd=1)
```

The above commands will typically be stored in another file, e.g. “run.r”.

Giving the command source("run.r") from the command line in R will then first read the file “function.r”, which defines the function, then call the function three times in a row, with different arguments and finally plot the function.

An expanded version of the logistic function could be

```
f<-function(x,a=0,b=1){
  y<-1/(1+exp(-b*(x-a)))
  return(y)
}
```

Suppose this code is stored in the file “functions.r” in a Windows directory.

A typical file containing the commands to implement the plots could be the following “run.r” file:

```
source("c:/temp/functions.r")
x<-0:100
y1<-f(x,35,.1)
y2<-f(x,35,.2)
y3<-f(x,25,.2)
pdf("c:/temp/graph.pdf")
plot(x,y1,type='l',col="blue",xlab="Length",ylab="logit response")
lines(x,y2,col="red",lwd=2)
lines(x,y3,col="red",lwd=1)
dev.off()
```

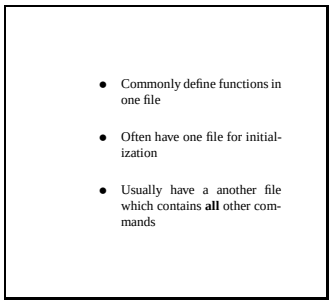
The interactive session

```
> source("run.r")
```

will then produce the plots into the file “graph.pdf”.

Alternative graphics formats abound. For example, the “png” command is used to generate PNG (portable network graphics) files, known to word processors, web browsers etc, “postscript” is used to generate postscript files and so forth.

4.4 Run commands



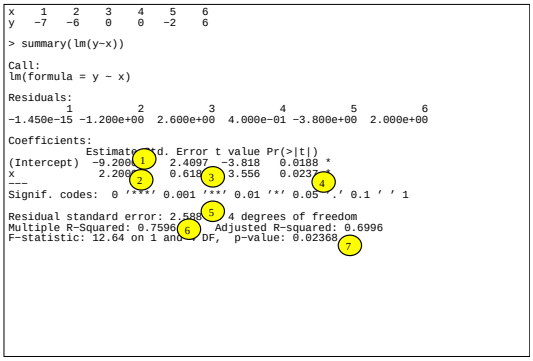
Most R jobs end up being split into three parts, function definition, setting or reading initial values and doing actual computation or plots.

Therefore there is usually a structure of files of the form:

- All functions in one file (e.g. functions.r)
- One file for initializing variables and reading data (e.g. init.r)
- Another file contains **all** other commands (e.g. run.r)

5 Statistical models

5.1 Linear statistical models



Mathematical model:

$$y = \alpha + \beta x + \varepsilon$$

R definition:

$$y \sim x$$

lm(y x)
Storing the output fm<-lm(y x).

Figure 2: Example output from a simple linear model fit of the form y=a+bx. Items (1)-(2) are the estimates of a and b respectively. The estimate of the standard error of b is given by (3). The P-value for testing whether the true (underlying) value of b is zero is in (4). Items (5)-(7) give the MSE, R-squared and P-value for the entire model, respectively.

Suppose that within R a user has two columns of data, “x” and “y”, which come in pairs and there is a need to fit a straight line through the data points.

Having plotted the data, this is followed by specifying the model, which should be of the form $y = \alpha + \beta x$. The model notation in R for this simple linear model is

$$y \sim x.$$

The tilde character ($\sim$) indicates that the left-hand side is a dependent variable and the model is on the right-hand side. On the right hand side it is implicitly assumed that there will be an intercept (α in the mathematical model) and therefore there is only a need to list the “dependent” variable(s), in this case only x.

To fit the actual model the “lm” function is used (lm being short for “linear model”):

```
lm(y~x)
```

In order to process the model results, the fitted model is stored under some name, e.g. “fm”:

```
fm<-lm(y~x).
```

Example: Suppose the data are given by

```
x     1     2     3     4     5     6
y    -7    -6     0     0    -2     6
```

A simple linear model can be fitted to the data and the results output using:

```
> summary(lm(y~x))
```

The results are shown in the figure.

Note: The output from the various lm-related programs is quite detailed and although a statistics course can be designed around the interpretation of the results, some basic knowledge is essential.

Consider the output given in the figure.

Example: Consider a data set with a dependent variable y, an independent variable x and a factor, f:

```
   x  f         y
1  1  A  6.367151
2  2  A 10.783743
3  3  A 11.528125
4  4  A 15.564471
5  5  A 18.509431
6  1  B  4.608247
7  2  B  6.849981
8  3  B 12.301949
9  4  B 14.251640
10 5  B 16.483796
11 1  C  6.293174
12 2  C  7.905664
13 3  C 10.640212
14 4  C 15.881404
15 5  C 16.679703
```

If this data set is read in using read.table, the f-column will automatically become a factor and can be used directly in a model such as

```
lm(y~f+x)
```

```
> summary(lm(y~x))

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.8277 -0.9488 -0.1151  0.7969  2.1061

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.7466     0.6992   3.928  0.00173 **
x             2.9656     0.2108  14.066 3.04e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.155 on 13 degrees of freedom
Multiple R-Squared: 0.9383,     Adjusted R-squared: 0.9336
F-statistic: 197.9 on 1 and 13 DF,  p-value: 3.043e-09
```

```
> fm<-lm(y~f+x)
> drop1(fm,test="F")
Single term deletions

Model:
y ~ ff + x
       Df Sum of Sq     RSS     AIC  F value     Pr(F)
<none>               10.317   2.386
 f      2     7.018  17.335   6.170   3.7414    0.0576 .
x       1   263.837 274.153  49.585 281.3080 3.499e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Use the resid function to extract residuals, then plot these and standardize to test for normality etc.

Use anova(fm1,fm2) to compare two models.

Having obtained the model, the coefficients can now be obtained, summary statistics of the model can be listed and the analysis of variance corresponding to the model is obtained:

```
fm<-lm(y~x).
summary(fm)         # General summary of model fit
anova(fm)           # Additional variation explained by each effect
drop1(fm)           # Marginal test of each effect in a model
coef(fm)            # Extract coefficients of fitted model
resid(fm)           # Extract residual
fitted(fm)          # Extract fitted values
```

5.2 Nonlinear statistical models

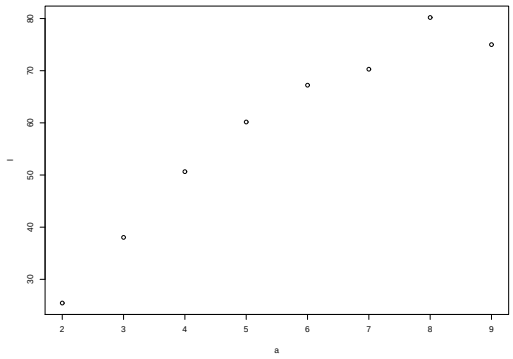


Figure 3: Example of a potential nonlinear relationship (length and age).

Nonlinear statistical models involve some nonlinear combinations of the parameters themselves (i.e. not the independent variables, so e.g. $y = \alpha + \beta x^2$ is in fact a linear model). Nonlinear estimation methods are therefore needed.

Example: Suppose the data are

```
   x    y
 1 3.25
 2 2.86
 3 2.48
 4 2.22
 5 2.02
 6 1.83
 7 1.71
 8 1.68
 9 1.45
10 1.41
```

and it is known that the process generating the data is of the form $y = \alpha + e^{1-x/K} + \varepsilon$ where ε can be assumed to come from a normal distribution. (These data were in fact generated using y<-round(1+exp(1-x/5)+rnorm(10)*0.05,2).)

In this case it is natural to consider the sum of squared deviations:

$$S = \sum_{i=1}^{n} \left(y_i - \left(\alpha + e^{1-x_i/K} \right) \right)^2$$

and estimate the parameters by minimizing S.

Suppose the function S defines the sum of squares as a function of the two parameters. Normally, this function would be stored in a file, e.g. s.r and the file would be read in using "source("s.r")".

Next estimate the parameters using nlm:

```
nlm(S,c(0,1))
$minimum
[1] 0.01447913

$estimate
[1] 1.051933 4.867273

$gradient
[1] -1.116759e-09  2.067758e-10

$code
[1] 1
```

In order to define a sum of squares as a function of two parameters, the following form can be used.

```
S<-function(beta)
  {
    alpha<-beta[1]
    K<-beta[2]
    yhat<-alpha+exp(1-x/K)
    S<-sum((y-yhat)^2)
    return(S)
  }
```

Notice the "trick" of using a vector to store the two parameters. This is needed when using a generic routine such as nlm since nlm will assume that the parameters are all stored in a single vector.

5.3 Miscellaneous statistical models

Usual assumptions: Linear, Gaussian errors, constant variance, independence.
Alternatively: Nonlinear, non-Gaussian, heterogeneity, non-independence.
Examples: Length-weight relationships, spatial correlations etc

Many issues arise with common fisheries data, indicating deviations from the usual assumptions of linearity and Gaussian distributions.

Thus various nonlinear models have been used for length-weight relationships in fishery science and a large number of other biological relationship.

To fit a generalized linear model, use a function called glm.

5.4 Further reading

Extensive references exist for statistical models in R (or Splus).

For simple statistical analyses the statistical basis can be obtained from any introductory book (such as Moore and McCabe's).

For linear models in R, consult any corresponding textbook, (such as Fox et al).

For more detailed regression analysis a comprehensive book on linear models (such as John Neter et al) is needed.

References

%T An Introduction to R %A W. N. Venables, D. M. Smith %D 2002 %I Network Theory Ltd. %P 156pp ISBN: 0954161742

%T Introductory Statistics with R %A Peter Dalgaard %D 2002 %I Springer-Verlag %P 288pp ISBN: 0387954759

%T An R and S Plus Companion to Applied Regression %A John Fox, Georges Monette %D 2002 %I SAGE Publications %P 312pp ISBN: 0761922806

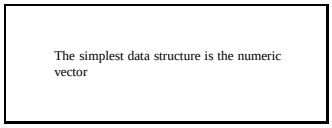
%T The analysis of variance. %A Scheffe, H. %D 1959 %I John Wiley and Sons, Inc, New York. %P 477pp. ISBN: 0471758345

%T Introduction to the practice of statistics. %A Moore D. S. and McCabe, G. P. %D 1999 %I Freeman and Company %P 825pp ISBN: 0716796570

%T Applied linear statistical models. %A Neter, J., A Kutner, M. H., Nachtsheim, C. J. and Wasserman, W. %D 1996 %I McGraw-Hill %P 1408pp. ISBN: 0256117365

6 Data structures in R

6.1 Vectors



The simplest data structure is the numeric vector.
Vectors can be used in arithmetic expressions (operations are performed element by element).

If the vectors do not have the same length then the shorter vectors are reused until they match the length of the longest vector.

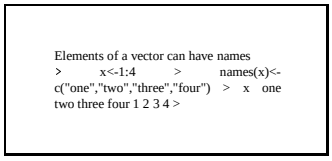
Examples of available arithmetic functions and operators: $+$,$-$, $*$, , $\hat{}$, *log*(), *exp*(), *sin*(), *cos*(), *tan*(), *sqrt*()

Useful vector operations: max(), min(), length(), sum(), prod(), mean(), var(), sort()

Example:

```
> x<-c(1,2,4,6)   # define the vector x
> x*3             # multiply each number in x by 3
> x[2]            # the 2nd component of x
> length(x)       # shows the length of x
```

6.2 Naming vector elements



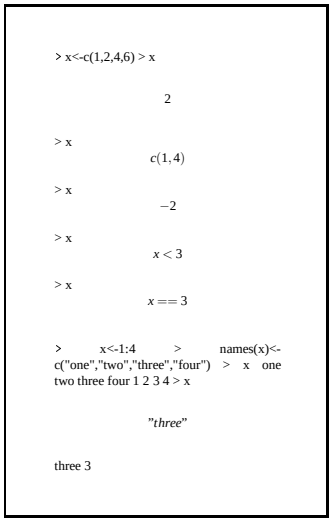
Naming the elements of a vector can be useful in several regards. Firtly, the elements may correspond to locations or treatments and thus naming them provides useful identification.

Secondly, as will be seen below, naming can be useful for extracting the appropriate element.

Suppose we give names to the elements of a vector:

```
> x<-1:4
> names(x)<-c("one","two","three","four")
> x
  one   two three  four
    1     2     3     4
```

6.3 Indexing vectors



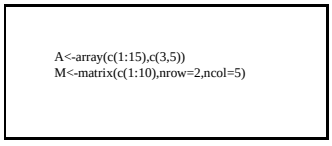
Square brackets are used to denote an **index** to an element of a vector.

- The usual method to extract one or more elements is to simply list the locations of the items of interest, with vec[12], vec[c(3,5)], vec[2:6] etc.
- It is also possible to use a logical expression to obtain a vector of true/false values of the same length and use the logical vector as an index.
- If a numerical index vector contains negative values then they will be **omitted** from the result.
- Finally, if the elements have names, then the vector can be indexed with a sequence of (character) names to be extracted.

Example:

```
> x<-c(1,2,4,6) # define the vector x
> x[2]          # the 2nd component of x
> x[c(1,4)]     # gives elements 1 and 4
> x[-2]         # gives all elements in x except no 2
> x<3           # returns a logical vector
> x[x<3]        # all elements of x which are less than 3
> x[x==3]       # all elements of x which equal 3
> x["three"]    # the element named "three"
```

6.4 Arrays and matrices



An array is a data construct that can be though of as a matrix with multiple dimensions.

A matrix in R is an array with 2 subscripts. R contains a variety of operators and functions which work with matrices.

Matrices can be added and subtracted from each other using + and - as expected, if the matrices are of the appropriate dimensions.

Care must be taken with the * operator. If A and B are matrices then A*B is a matrix of element by element products but A%*%B is a matrix multiplication.

In addition to simple operators several functions are available for matrix manipulation. Useful matrix functions include: t, nrow, ncol, diag.

Example:

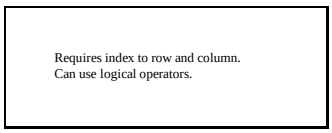
```
> A<-array(c(1:20),c(5,2,2))      # make the 3-dimensional array A_{i,j,k}
> A[5,1,1]                  # A_{5,1,1}
[1] 5
> A[1,2,1]
[1] 6
```

```
>M<-matrix(c(1:6),nrow=2,ncol=3)       # make the 2 × 3 matrix M
```

```
> M
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> t(M)          # transpose M

     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

```
> diag(3,2)          # make a 2 × 2 matrix with 3 on the diagonal

     [,1] [,2]
[1,]    3    0
[2,]    0    3
```

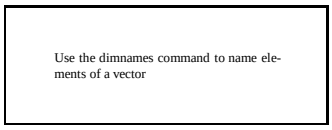
6.5 Indexing arrays and matrices



Example:

```
> A[2,3]     # returns the element in row 2, column 3
> A[2,]      # returns all elements in row 2
> A[A<2]     # returns all elements that are less than 2
```

6.6 Names of rows and columns

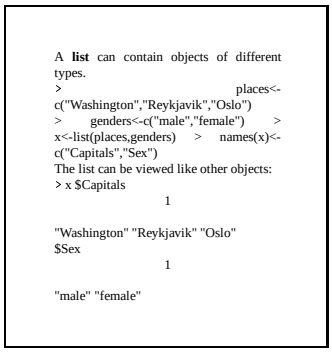


Example: Paired data can be read in using read.table and then the command

```
dimnames(x)<-list(sex=c("male","female"),length=as.character(1:25))
```

can be given.

6.7 Lists



Objects of different types can be aggregated together into a new object called a **list**.

Example: A somewhat abstract list might contain information about age, gender and height.

```
> a<-list(age=10,gender="female",height=c(150,135,143,127,149))
```

This list consists of three components. As with any R object, typing the name of the list reveals all of its contents.

```
> a
$age
[1] 10

$gender
[1] "female"

$length
[1] 150 135 143 127 149
```

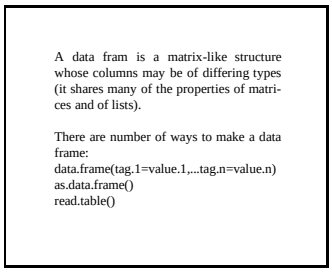
The components of the list can be referred to in several different ways, notably by index or by component name:

```
> a[[1]]   #use the index to get the age
[1] 10
> a$age    #use the components name to get the age
[1] 10
```

In some cases it is useful to loop over the names and use them somewhat like a vector reference, in which case the reference is of the form listname[[itemname]]:

```
a[["height"]]
```

6.8 Data frames



A dataframe is a special type of list which is organized as a matrix-like structure whose columns may be of differing types. It is thus a list that satisfies certain conditions and can e.g. be displayed in a matrix form.

R includes several commands which return dataframes.

data.frame(tag.1=value.1,...tag.n=value.n)	makes a data frame from values no 1 to n
as.data.frame()	coerces its arguments to a data frame
read.table()	read an entire data set from an external file

Additional notes and practicals: R commands

Vectors

The simplest data structure in R is a vector.

To create a vector of 2.3, 4.2, 5.7, 9.2, 4.7.

```
x <- c(2.3, 4.2, 5.7, 9.2, 4.7)
```

`<-` means assigned the value, do not use `=` .

To see x.
```
x
```

You can then use x.
```
1/x
y <- c(x, x)
y
```

In R, `x` is known as an object.

To list all the objects
```
ls()
```
and to remove individual objects
```
rm(x)
rm(y)
```
or
```
rm(x,y)
```

These are similar to the shell commands you learned earlier.

Vector arithmetic

Sequences of numbers can be generated as follows:

```
x <- 1:10
y <- 5:1
z <- seq(2,10,2)
w <- seq(10,5,-0.5)
```

Q. Which command returns a sequence of numbers from 0 to 5 with intervals of 0.5?

In operations the colon has the highest priority e.g.

```
2*1:10
```

compare

```
n <- 5
1:(n-1)
1:n-1
```

To repeat blocks of numbers.

```
rep(2, times=5)
rep(2,5)
rep(1:2,5)
rep(1:2,c(5,5))
c(rep(1,2), rep(3,3))
rep(c(1,3),c(2,3))
```

Q. Which commands return the following sequences of numbers?
2 4 2 4 2 4
1 1 1 1 5 5 5 5 5

Elementary arithmetic operators are the usual

```
 +, -,*, /, ^
```

Other functions include: `log, exp, sin, cos, tan, sqrt` e.g.

```
x^2
x^(0.5)
```

square root

```
sqrt(x)
```

Natural logarithm (ln) and exponential function

```
log(5)
log(x)
exp(3)
```

Log base 10

```
log(5,10)
log(x,10)
```

Minimum, maximum and range of a vector.

```
min()
max()
range()
```

Number of elements in a vector.

```
length(x)
```

Sum

```
sum(x)
```

Mean, variance and standard deviation

```
mean(x)
var(x)
sd(x)
```

Quantiles

```
median(x)
quantile(x)
```

Absolute value

```
abs(-5)
```

Indexing vectors R is a very useful language for manipulating data, which is a very important tool for exploratory statistical analyses and plotting data. To use this feature of R you need to learn how the data are indexed. These are very important commands for you to understand.

In the simple case of a vector:

```
x <- 1:20
x
x[1:5]
x[10:12]
x[-1:-5]
x[-c(1:5)]
x[length(x)]
x[length(x)-1]
```

Q. Which command returns the 15th number in x?
Q. Which command returns the second last number in x?

Matrices

Other data objects include matrices.

```
x <- 1:10
y <- x*x
z <- cbind(x,y)
```

creates a matrix called z with 2 columns.

Q. What does z look like?

Elements of z can be extracted in a similar way as for a vector. The row and column need to be identified.

```
z[1,]    # is the first row of z
z[,1]    # is the first column of z
z[1,2]   # what does this do?
```

For the dimensions of a matrix or array, the number of rows and number of columns.

```
dim(z)          # row and column dimensions
dim(z)[[1]]     # the first element of dim ie the number of rows
dim(z)[[2]]     # the second element of dim ie the number of columns
nrow(z)
ncol(z)
```

Q. How many rows and columns are there in z?
Q. What is the command to return the number in the 2nd column and second row of z?

Operations can be carried out on matrices as they are for vectors.

```
z*2
z[,1] - sqrt(z[,2])
```

2 (or more) matrices with appropriate dimensions can be joined using `rbind` and `cbind`.

```
rbind(z,z)
cbind(z,z)
```

Or with another matrix `w` possibly like:

```
rbind(z,w)
cbind(z,w)
```

Q. Create a matrix `w` and try these. If it doesn't work, what was wrong with `w`?

Matrices can also be created using using the command `matrix`.

```
z1 <- matrix(1:10, ncol=2)
z2 <- matrix(1:10, byrow=T, ncol=2)
z3 <- matrix(1:10, byrow=T, ncol=5)
```

Q. What do byrow and ncol do?

Text manipulation

`paste()` is used to create a string either converting numbers into characters or by joining text and/or numbers. When joining, the separator can be selected eg:

```
years <- 1990:1994
paste(years)
paste("year",years)
x <- paste("year",years, sep="")
paste("len", seq(4,30,2), sep="")
paste("len", seq(4,30,2), sep=".")
```

Q. What is the difference between `1990:1994` and `paste(1990:1994)`?

`substring()` extracts part of a string eg with `x` from the previous example:

```
substring(x,1,4)
as.numeric(substring(x,5,8))
```

Vector and matrix dimension names

The dimensions (rows and columns) of a matrix can be named.

*V*ectors For a vector.

```
age.vec <- c(10,42,65,46,30)
```

To return the names:
`names(age.vec)`

To create the names
`names(age.vec) <- paste(2:6)`

The names can be used to select an element of the vector.
`age.vec[names(age.vec)=="5"]`
`age.vec["5"]`

The names can be changed:
`names(age.vec) <- paste("age",2:6, sep="")`

Then
`age.vec[names(age.vec)=="age5"]`

*M*atrices

Matrices have 2 dimensions and the command `dimnames` is used.

For a matrix.
`age.mat <- matrix(c(10,42,65,46,30,12,40,64,48,34), ncol=2)`

To return the dimension names:
`dimnames(age.mat)`

To create the column names – columns are the second dimension
`dimnames(age.mat)[[2]] <- paste(2000:2001)`

To create the row names – rows are the first dimension
`dimnames(age.mat)[[1]] <- paste(2:6)`

It's better to name both at once:
`dimnames(age.mat) <- list(paste("age",2:6, sep=""), 2000:2001)`

The names can be used to select an element (or elements) of the matrix.
`age.mat[,dimnames(age.mat)[[2]]==2001]` # for a column
`age.mat[dimnames(age.mat)[[1]]=="age3",]` # for a row

The double square parentheses `[[]]` are used for matrix dimnames as they are a type of object called a `list`. Each element of a list can be a different length and the dimensions of a matrix are normally not equal.

Data frames

A data frames is, in some aspects, a more useful data format than a matrix. Data frames can contain columns of different types eg character and numeric. NB: Even though a data.frame has 2 dimensions `names` refers to the column names.

Data frames can also be created within R.

```
x <- seq(5,25,5)
y <- c(2,4,6,7,4)
ldat <- data.frame(len = x,num = y)
```

creates a data frame with two columns names len and num.

Columns of a data frame can be referred to by name eg:
`ldat$num`

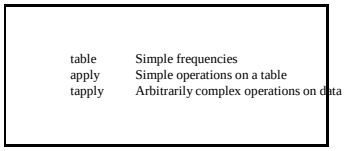
As with matrices, operations can be carried out on the columns. It is very easy to add new columns to a

data frame. eg
`ldat$num2 <- ldat$num*2`

Q. What does `ldat` look like now?
Q. Which command adds a column of zeros to `ldat`?

7 Advanced data manipulation in R

7.1 Tabular summaries in R



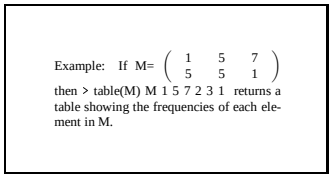
The most commonly requested data summaries include frequency tables, means and variances (or standard deviations). All of these are available in R.

mean, var	Means and variances of individual columns
table	Simple frequencies
apply	Simple operations on a table
tapply	Arbitrarily complex operations on data in columns

For example, if "mydata" contains "x" and "y" as columns, (like in the earlier examples) then we can e.g. do

```
x<-mydat$x
y<-mydat$y
mean(x)               # calculates the mean of x
tapply(y,x,mean)      # calculates the mean of the y-values
                      # related to each number in x
apply(mydata,2,mean) # calculates the mean of each column in mydata
                      # i.e. the mean of x and the mean of y
```

7.2 Frequency tables



The "table" command is used to count how often values appear.

Thus, table(X,...) makes a frequency table for the object X

Example: If $M = \begin{pmatrix} 1 & 5 & 7 \\ 5 & 5 & 1 \end{pmatrix}$ then

```
> table(M)
M
1 5 7
2 3 1
```

returns a table showing the frequencies of each element in M.

Example: If the vector R contains recruitment information and S contains levels of spawning stock in the year that the recruitment was generated, then the following command will give a tabular account of how often the recruitment was above or below the median according to the level of the spawning biomass

```
table(Rec=R<median(R),SSB=S<median(S))
```

Note the use of named argument to label the output table.

This can be expanded:

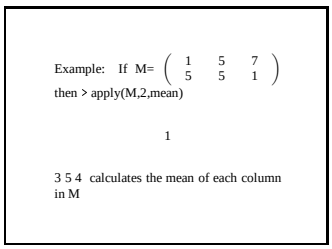
```
table(Rec=ifelse(R<median(R),"low","hi"),SSB=ifelse(S<median(S),"low","hi"))
```

which will give output of the form:

```
     SSB
Rec   hi low
  hi   3   5
  low  5   2
```

Naturally, only slight modifications to this will give output which can be automatically included in LaTeX documents, without any copying or pasting.

7.3 Row and column summaries



apply(X, MARGIN, FUN, ...) returns an object obtained by applying a function(FUN) to margins of an array(X).

Example: If M= $\begin{pmatrix} 1 & 5 & 7 \\ 5 & 5 & 1 \end{pmatrix}$ then

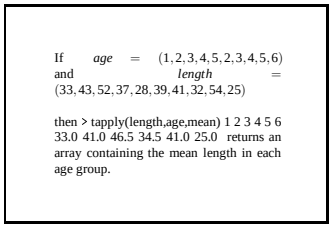
```
> apply(M,2,mean)
[1] 3 5 4
```

calculates the mean of each column in M

```
> apply(M,1,mean)
[1] 4.333333 3.666667
```

calculates the mean of each row in M

7.4 Operations on data columns



The **tapply** command is used in cases when one wants to compute statistics such as averages or sums of one variable using another variable to describe groupings.

Typical usage would be of the form

```
tapply(x,i,sum)
```

which computes the sum of the x-values within each level of the index vector i. Notably, x and i have to be of the same length.

The tapply command is not restricted to means or sums since any function can be used as long as it can be applied to such subsets of the x-vector. Some common possibilities include

Example: Suppose some ages and lengths of fish are given by

```
age<-c(1,2,3,4,5,2,3,4,5,6)
le<-c(33,43,52,37,28,39,41,32,54,25)
```

then

```
> tapply(le,age,mean)
   1    2    3    4    5    6
33.0 41.0 46.5 34.5 41.0 25.0
```

returns an array containing the mean length in each age group.

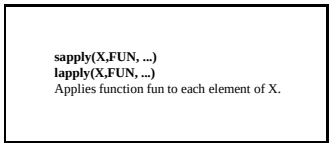
Basically, tapply groups the values in le svo that each group is associated with the corresponding age and then applies the mean function to each group.

age	1	2	3	4	5	6
le	33	43 39	52 41	37 32	28 54	25
mean length	33.0	41.0	46.5	34.5	41.0	25.0

Note that this can also be done age-by-age through a command sequence of the form

```
mean(le[age==1])
mean(le[age==2])
mean(le[age==3])
```

7.5 Other tabular functions



The **sapply** and **lapply** commands are used in somewhat more complex situations where an arbitrary function is to be used for each element of a vector (or structure).

Example: In fishery science it is common to write small functions to evaluate e.g. yield per recruit for a given level of fishing mortality. When plotting a yield-per-recruit curve one typically wants to evaluate this function for a range of fishing mortalities and this is best done using sapply, e.g.

```
yr<-sapply(Fvec,yrfun)
```

Aggregating data

`table`, `apply`, `tapply` and `aggregate` are commands which can be used to summarise and aggregate data. The examples below will explain much more than the descriptions.

`table` creates a table of the counts of each factor level
`tapply` applies a function to a ragged array and creates an array
`aggregate` is the same as tapply but writes the output to a data frame rather than an array
`apply` applies a function to an array, the seconds term defines the dimension on which the function operates (eg 1 = row, 2 = column).

Create some objects these functions can be applied to.

```
x <- c(45,55,45,35,45,35,50,50)
y <- rep(1:2,4)
z <- rep(c(10,20),rep(4,2))
dat <- matrix(c(1:12),ncol=3,byrow=T)
```

To count the number at each level:

```
table(x)
table(y)
```

Q. How many times does 50 occur in x?

The output of any command can be saved as an object.

```
tmp <- table(x)
```

Q. What are the names of tmp?

The number of times 50 occurs in x can be extracted automatically.

```
tmp[names(tmp)==50]
```

To apply a function:
To see how x, y and z relate to each other. `cbind(x,y,z)`

```
tapply(x,x,length)          # the number of each element of x by x
tapply(x,y,length)          # the number of each element of x by y
tapply(x,y,sum)             # sum the values of x in groups of y
tapply(x,y,mean)            # sum the values of x in groups of y
tapply(x,list(y,z),sum)     # sum the values of x in groups of y and z
```

Q. How many values of x correspond to y = 1?
Q. What is the mean value of x if y=1?
Q. What is the mean value of x if y=1 and z = 20?

To return the same information as `tapply` but in colums use `aggregate`.

```
aggregate(x,list(y),sum)
aggregate(x,list(y,z),sum)
```

To apply a function to the rows or columns of a matrix:

```
apply(dat, 1, sum)     # the sum of the rows of dat
apply(dat, 2, mean)    # the mean of the columns of dat
```

These functions can also be applied to data frames.

Create a small data frame of age and length data – the number by year, age and length:

```
y <- rep(2000:2001, rep(6,2))
a <- rep(rep(1:3,rep(2,3)),2)
l <- rep(c(4,5,5,6,6,7),2)
n <- sample(1:10, 12, replace=T)
ldat <- data.frame(year = y, age = a, len = l, num=n)
```

The number of fish by year:

```
tapply(ldat$num, ldat$year,sum)
```

The number of fish by age and year:

```
tapply(ldat$num, list(ldat$age,ldat$year),sum)
```

The dimension names of these objects can be used, eg extract the data for 2001 from the returned table:

```
tmp <- tapply(ldat$num, list(ldat$age,ldat$year),sum)
tmp[,dimnames(tmp)[[2]]==2001]
```

And to extract only data for age 3:

```
tmp[dimnames(tmp)[[1]]==3,]
```

For mean length at age by year:
total length (by year and age) / number of fish (by year and age)

Total length by age and year:

```
tlen <- tapply(ldat$num*ldat$len, list(ldat$age, ldat$year), sum)
```

Number of fish:

```
fnum <- tapply(ldat$num, list(ldat$age, ldat$year), sum)
```

Mean length at age:

```
tlen/fnum
```

8 Plotting with R

8.1 Some common plot commands

Plotting

*S*catter and line plots

Using the data:

```
x <- 1:10
y <- x*x
```

The simplest plot is

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

with lines

```
plot(x,y, type="l")
```

with lines and points

```
plot(x,y, type="b")
```

To relabel the axes:

```
plot(x,y, type="l", xlab = "x axis", ylab ="y axis")
```

```
title("simple plot")
```

Another way to add a title is:
```
plot(x,y, type="l", xlab = "x axis", ylab ="y axis", main="simple plot")
```

To control the bounds of the x and y axes:
```
plot(x,y, type="l", xlim = c(0,15), ylim = c(0,120))
```

To overlay another plot:
```
plot(x,y, type="l")
points(x, y+50)
```

To add a dashed horizontal line:
```
plot(x,y, type="l")
abline(h=2, lty=2)
```

To add a dashed vertical line (with a different line type):
```
abline(v=2, lty=5)
```

Adding points with a different colour and shape:
```
points(x, y-10, pch=3, col=2)
```

To plot more than one plot on the device:
For 2 rows and 3 columns of plots:
```
par(mfrow=c(2,3))
plot(x,y)
plot(x,y,type="l")
```

With more than one plot on a page it can be useful to have a title for the whole page. To do this a wider border is required.
```
par(mfrow=c(2,2), oma=c(2,1.5,2.5,1.5))
plot(x,y)
plot(x,y,type="l")
mtext("My plots", outer=T)
```

To save a plot to a file:
`dev.print(file="<`*filename.ps*`>")`
where you define <*filename*>.

8.2 More plot commands

More plots

It is important to look at data before using it – for some understanding and to identify possible problems.

*H*istograms

Histograms plot the frequency of data.

A data frame with only year and number:
```
y <- rep(2000:2001, rep(20,2))
n <- round(abs(rnorm(40, 20,20)))
ndat <- data.frame(year = y, num = n)
```

Histogram of the number:
```
hist(ndat$n)
```
By year
```
hist(ndat$n[ndat$year==2000])
```

```
hist(ndat$n[ndat$year==2001])
```

*B*arplots

Barplots plot the number by category.

A data frame with only year and length:

```
y <- rep(2000:2001, rep(10,2))
l <- sample(4:8, 20, replace=T)
ldat2 <- data.frame(year = y, len = l)
```

Plot the number in each length group:

```
tmp <- table(ldat2$len)
barplot(tmp)
```

Or simply

```
barplot(table(ldat2$len))
```

By year

```
tmp <- table(ldat2$year, ldat2$len)
barplot(tmp[1,], main="2000")
barplot(tmp[1,], main="2001")
```

Alternatively:
Using:

```
x <- seq(5,25,5)
y <- c(2,4,6,7,4)
z <- c(3,4,7,6,3)
ldat3 <- data.frame(len = x,num = y, num2 = z)
```

Lines can be overlaid on the barplot by storing the position on the x axis of the bars.

```
x <- barplot(ldat3$num, names=ldat3$len)
lines(x, ldat3$num2)
```

*B*oxplots

Box (and whisker) plots summarise data – graphically providing information on the distribution of the data by category.

To plot summary statistics on the length distribution by year from ldat2:

```
boxplot(split(ldat2$len,ldat2$year), xlab="year")
```

To plot summary statistics on the number of fish by year, age and length in ldat:

```
par(mfrow=c(2,3))
boxplot(split(ldat$num,ldat$year), xlab="year")
boxplot(split(ldat$num,ldat$age), xlab="age")
boxplot(split(ldat$num,ldat$len), xlab="length")
```

Or

```
boxplot(num ~ year, data=ldat, xlab="year")
boxplot(num ~ age, data=ldat, xlab="age")
boxplot(num ~ len, data=ldat, xlab="length")
```

Linear regression

Create a dataset and plot it:

```
x <- 1:20
w <- 1 + sqrt(x)/2
y <- (2*x + rnorm(x)*w)
plot(x,y)
```

Fit a linear regression line through the data:
`rf <- lm(y ~ x)`

Plot the data and the fitted line:
```
plot(x,y)
abline(rf, col=2)
```

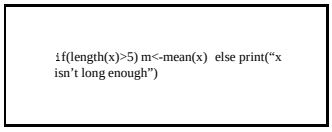
To see the details of the regression:
`summary(rf)`

From the linear regression we can look at the residuals - the difference between the line and the actual values.

```
plot(fitted(rf), resid(rf), xlab="fitted values", ylab="residuals")
abline(h=2, lty=2)
abline(h=-2, lty=2)
```

9 Programming R

9.1 The if statement



The "if" statement can be used to perform a computaton only when a certain condition is satisfied.

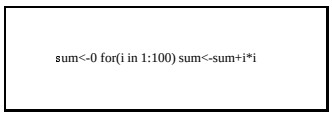
If the average of the elements of x is to be computed only when the length of x is larger then 5 then the following sequence can be used:

```
if(length(x)>5){
  m<-mean(x)
  }
```

Another statement that can be used with the if statement is the else-statement. To continue the above example, and in addition print error messages if x isn't long enough then an else-part can be added:

```
if(length(x)>5){
  m<-mean(x)
  } else{
  print(``x isn't long enough'')
  }
```

9.2 Loops - for



Loops are used when similar operations need to be performed. For example, when adding the squared integers 1*1, 2*2, 3*3, ... up to 100*100, this can either be done by

```
sum<-0
sum<-sum+1*1
sum<-sum+2*2
...
sum<-sum+100*100
```

or through a for-loop:

```
sum<-0
for(i in 1:100){
    sum<-sum+i*i
}
```

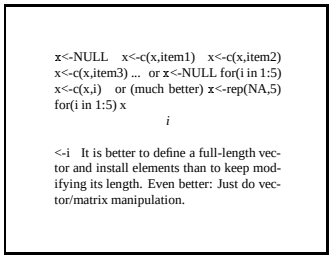
It should be noted that when a loop is used to add numbers, the variable containing the sum should be set to zero before starting the loop.

In the loop itself, 1:100 simply denotes the sequence of numbers 1, 2, 3, ... 100 and the notation

```
for(i in 1:100){
    ...
}
```

indicates that the commands within the brackets should be run 100 times, while the variable "i" is set first to 1, then 2 and so on.

9.3 Storing loop results



It is often useful to first initialise an object and then use it to collect results repeatedly.

Without a loop this kind of collection might proceed as follows.

```
x<-NULL
x<-c(x,item1)
x<-c(x,item2)
x<-c(x,item3)
...
```

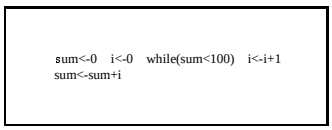
It is much more useful to use a loop for this purpose.

Example:

```
>row<-NULL          #  can also use row<-c()
>for(i in 1:5){
  row<-c(row,1:i) # append the numbers 1,..,i to the row
```

```
}
>row
[1] 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 # the result
```

9.4 Loops - while



A while loop can be used to perform operations until a certain condition is satisfied. For example to add the numbers 1,2,3,... until the sum reaches 100:

```
sum<-0
i<-0
while(sum<100){
  i<-i+1
  sum<-sum+i
  }
```

(This while-loop would stop when sum=105 and i=14)

When given a while command, R will run the sequence of commands in the brackets repeatedly until sum is greater than or equal to 100.

Practicals

Sourcing a file

It is not necessary to type R commands directly into R. Files can be written and 'sourced'. If all your commands are in a file (myfile.r) and R is open in the same directory as the file, the command:
`source("myfile.r")`
reads in the commands and runs them in R.

Create a file in Emacs called myfile.r containing:

```
x <- 1:10
y <- x*5
plot(x,y)
print(x)
```

`print(x)` returns `x` in the same way `x` does when typed directly into R.

Then, in R type:
`source("myfile.r")`

In Linux, files can be also run from the terminal.
`R -slave < myfile.r`

R — loops and functions

*L*oops – `for`

It is often useful to be able to repeat the same operation and one way to do this is using a for–loop.

- In a text file save this code:

  ```
  for(i in 1:5){
  ```

```
  print(i*i)
}
```

- Run the script in R.
- A loop can be used to compare samples from a Normal distribution. In the example below σ varies.

```
x <- seq(0.5,2.5, 0.5)
t <- 0
plot(density(rnorm(100, 0, 1)), ylim=c(0, 0.1), type="n")
for(i in x) {
  t <- t+1
  lines(density(rnorm(100, 0, i)), col=t)
}
```

- What does `type="n"` do?
- Compare samples from Normal distributions where you repeat the same sampling procedure. What happens as you decrease/increase *n*?
- Try the examples in your lecture notes.

*L*oops – `while`

It is also possible to write loops with the `while` command.

- Try this:

```
sumi <- 0
i <- 0
while(sumi < 10){
  i <- i+1
  sumi <- sumi+i
  print(c(sumi, i))
}
```

- a `while` loop repeats the commands between the {}'s until the while statement is met.

*T*he `if` statement

eg

- eg

```
x <- 1:10
if(length(x) > 5){
  print(mean(x))
}
```

- can also to `if` *cond* `else` eg

```
x <- 1:3
if(length(x) >= 5){
  print(mean(x))
 } else {
  print("Error: x < 5")
}
```

*W*riting a function in R

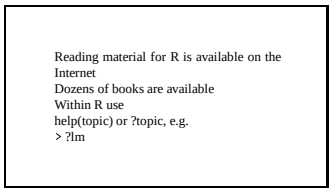
- Create the following function:

```
mymean <- function(data){
 m <- sum(data)/length(data)
 return(m)
}
```

- Create a small dataset `x`.
- Compare `mymean(x)` with `mean(x)`

10 Further reading

10.1 Reading material



Search the Internet on any topic, e.g. using Google to look for phrases such as

“linear models with R”

and this will typically return several useful links.