Introduction to R

All the practicals in the stock assessment course will use R. R is a free statistical package and is very similar to the commercial package Splus.

You will also have information on R in your lecture notes. It is very important that you understand what the commands are doing as you will need to use them throughout the rest of the course. In this handout, not all the commands are explained and I want you to work out what each command does. Some commands are just used to generate data sets ie `rnorm, sample` and you do not need to understand them for these examples.

It is not necessary to memorise all these commands. By doing the practicals you will learn the commands you use most frequently and if you keep a record of your work you can look back to see how to do things. If you use a text editor to store your work you will have a record of commands and you can use the same commands for different datasets. This makes it very easy to replicate work – much more so than in a spreadsheet. It is also easy to see exactly what you did.

Today is just an introduction to help you become familiar with some of the commands you will be using.

There is also an introduction to R available from www.tutor-web.net

R in Linux

Start R in the terminal by typing `R`. You can start R any directory. 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 Emacs

Another option is to run R in Emacs.

If you have an 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. 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 Emace 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.

If you type 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)

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.

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.

Help

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.

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.

Vectors

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"]`

Matrices

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`?

Plotting

Scatter 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*>.

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
```

More plots

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

Histograms

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])
```

Barplots

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)
```

Boxplots

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)
```

Reading in data

Data are normally read into R from a file. The command `read.table` reads in data from a file and creates a data frame, which is in some aspects a more useful data format than a matrix. Data frames can contain columns of different types eg character and numeric. Additional commands for `read.table` include `skip=X` which ignores the first `X` lines of the file and `header=T` which uses the first line read in as the column names.

To read in the file output.txt ignoring the first 3 lines eg:
`sdata <- read.table("output.txt",skip=3)`

The column names can also be named using eg:
`names(sdata) <- c("year","step","number")`

In Emacs create a file containing 2 columns of numbers of equal length. Assuming the dataset is in the same directory you have R open in and is called `fish.txt`:

In R:
`mydata <- read.table("fish.txt")`
will read in the data from your file.

To name the columns eg:
`names(mydata) <- c("le", "num")`

Alternatively, if there are names in the file for the columns. Then:
`mydata <- read.table("fish.txt", header=T)`

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
```