R Tutorial #1. March 25, 2011

[1] Getting R

http://www.r-project.org/

You will also find many tutorials, and further documentation there. The standard R distribution runs on all common computer platforms. Add-ons, called packages, are also there for special purpose applications.

* manipulate data sets flexibly
* operate on data, preforming complex calculations, including matrix ops
* many many tools for exploring and analyzing data
* advanced and highly customizable graphical facilities for plots
* full programming language, including user-defined functions
* written by statisticians

Download R and install it. Place the executable somewhere sensible. On Windoze machines, this is RGui.exe. On macs, this is R.app.

[2] First usage

Be organized. When you have a well defined project, create a dedicated folder for that project. Make sure the folder is somewhere you can find it again afterwards.

When you start R, navigate to the folder you want to work in. On mac, this is “Misc -> Change Working Directory” (or Cmd-D). On windows this may be “File -> Change Directory”.

When you exit R, it will ask you if you want to “Save workspace image?”. If you say yes, it will create two files, “.Rhistory” and “.RData”. Double clicking the “.RData” icon will restart R in the current directory with your variables etc all loaded. Stupidly, both Windoze and Macs make it hard to find, or even see, files beginning with a period. Whatever.

[3] Commands

The most basic way of using R is to type commands at the command prompt (>). When you type a command, hit ‘Return’ to have it executed. Incomplete commands will generate a continuation symbol (+). Normally, if you type a parenthesis or bracket or double quote that needs to be completed, R will automatically provide the closing element. This can be annoying.

Try the following. Where a line begins with ‘>’, type everything after the prompt and hit return. Where there is no initial ‘>’, I have provided what you might see produced by R. Sometimes I will omit this. Where a line begins with ‘#’, those are just my comments. Don’t type them, and don’t expect them to be produced.

```
# Lets check there is nothing known called ‘x’
> x
Error: object ‘x’ not found
```

```
# simple assignment: store the rhs in the variable given on the lhs
 > x <- 3

# Now repeat the first step
> x
[1] 3

# The simple value returned is just one number.  More generally, it may be a complex
# object with many parts.  Hence the need to index them with [1]

> x + 4
[1] 7

# note x is unchanged:
> x
[1] 3

# A list of numbers can be made using the ‘c’ function (from ‘concatenate)
# First a simple vector:

> x <- c(3,2,5,4,3)

# applying a function
> mean(x)

# element-wise operation
> x + 3

# raising to a power (exponentiation):

> x^2

# pre-defined constants

> pi

# getting help (menus also offer searchable help documentation)

> help(mean)
```

[4] Scripting

Entering commands one at a time is clumsy and laborious. For complex analyses it is absolutely ridiculous. You will need to learn to write your commands to a file, and then to have them read in and executed by R. The file must be a plain text file. This means not a word processing file, no Word .doc file, nothing with rtf extension, no fancy fonts. Just a plain old text file. There are several editors you can use. WordPad might work on a Windoze box. For Mac users, TextEdit will work, but you may have to bully it a bit to save your work as plain text.

Copy the following lines into a file called test.R. Include the comment lines beginning with #

****** Copy text below this line *****

```
# generate 100 random numbers randomly distributed over the interval [0,10]
x <- runif(100, 0, 10)

# generate 100 random numbers from the standard normal distribution (twice)

y <- rnorm(100, 0, 1)

z <- rnorm(100, 0, 1)

# sort y and z

ysorted <- sort(y)
zsorted <- sort(z)
```

***** End of text to be copied ******

Now read in those commands thus:

```
> source("test.R")
```

[5] Plotting

Lets try some simple plotting.

```
> plot(y,z)

> plot(ysorted, zsorted)

# the default is a scatter plot.  You might like to plot with lines.  Try these:

> plot(ysorted, zsorted, type="l")
```

or

```
> plot(ysorted, zsorted, type="b")

# How about a histogram?

> hist(x)
```

You can determine where the breaks lie in your histogram. First, learn how 'seq' works:

```
> seq(0,10,2)
```

Now try this:

```
> hist(x, breaks=seq(0,10,2))
```

Save you rplot by making sure the window has focus, and choose "File-> Save as"

You can be much more precise in dictating what kind of file to save as (pdf, jpeg, etc) and what size it should be. To do this, you open a new graphical device, repeat your plot command, and close the device again. Example:

```
> pdf("figure1.pdf", height=6cm, width=8cm)
> hist(x, breaks=seq(0,10,2))
> dev.off(dev.cur())
```

We will return to plots in a bit.

[6] Objects, vectors, lists

Numbers are simple objects. Most things R deals with are more complex. Lets look at an arbitrary object.

```
> myhist <- hist(x, breaks=seq(0,10,2))

> names(myhist)

> myhist$counts

> myhist$breaks
```

Ok. Now you know how to find out what the named components of an arbitrary objectare, and how to extract them. Try this too:

```
> str(myhist)
```

Often we deal with vector or matrices. Single elements are extracted by providing their index:

```
> x[10]
```

We can also extract a range of values:

```
> x[4:10]
```

Now lets make a 5 x 5 matrix, and ensure that we can get at any part of it at any time:

```
> a <- c(2,4,3,5,4)
> b <- ... etc. Make five rows, a-e.
```

Now make a matrix:

```
> mymx <- rbind(a,b,c,d,e)

> mymx
```

Matrix indices are provided in the order <row, column>, thus:

```
> mymx[3,2]
```

Look at a row:

> mymx[2,]

Look at a column:

> mymx[,4]

Look at a subset:

> mymx[2:3, 2:4]

You need to get good at subsetting your data, extracting or referring to precisely those bits you want.

To get the dimensions of your data:

> dim(mymx)

[7] Reading in data: read.table

Download the small file at this URL: http://cspeech.ucd.ie/Statistics/data/smallcor.dat. save it, e.g. in a file called smallcor.dat.

Look at the data in your text editor. Note that there are two columns. Each row corresponds to a single observation, in this case, an <x,y> pair.

Read them in using this command:

> mydat <- read.table("smallcor.dat",header=T)

The qualification "header=T" tells the read.table function that the first row is not data, but is the name of the columns. This allows you to use either matrix or named component forms for accessing your data:

> mydat$X

or

> mydat[, 2]

Usually, it is a good idea to arrange your data in this fashion, with one row per observation. You may have very many columns, depending on the complexity of your observation.

[8] More on built in functions (also called 'methods')

Many functions can take multiple arguments, as above, where we provided the file name and a value for "header". In this instance, the first argument is obligatory (otherwise, what file should it read in?), but the second argument is optional. The default is 'F'. The values 'T' and 'F' stand for 'true' and 'false', respectively.

Investigate the following functions:

rnorm, runif, rep, seq, max, min, mean, var, length

[9] Plotting and 'par'

Many aspects of a plot are dictated by setting 'graphical parameters'. There are very many of these. Some of them may be set when you issue the plot command, and some of them must be set before you plot things. E.g. specifying axis labels will obviously be done when plotting, while specifying how many plots are to appear in the plotting window must be done before you do any individual plot.

The function 'par' is used to set many aspects of the plotting environment before you do individual plots. Example:

```
# produce plots in two rows and three columns = 6 plots per window/page

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

You can get a list of all the current high-level graphical parameters thus:

```
> p <- par()

> p$names
```

As you can see, there are many of them. Help(par) will be your friend here. They collectively control the number of plots per window/page, the inner margins (mar), and the outer margins (oma) around each plot, etc.

Often, if you have a sequence of plotting commands, you might like to store the present set of graphical parameters first, do your plotting, and then restore things to the way they were before. This can be done using the following sequence:

```
# store a copy of the current values while simultaneously setting new values
> oldpar <- par(mfrow=c(2,3),oma=c(2,2,3,5),mar=c(0,0,1,0),...)

> ... your plotting commands here

# restore the original values
> par(oldpar)
```

Many of the graphical parameters are set when you actually do the plot.

[10] Major learning exercise

Have a go at recreating the plot provided as an exercise here: http://cspeech.ucd.ie/Statistics/lab2.php**.** The basic plot is a 'boxplot'. I used a trick to group the data into pairs, by inserting some dummy data between each pair. Do this incrementally, and place your commands in a text file, so you can modify them gradually. Work on one sub-problem at a time.