Fuzzy Sets and Systems

Laboratory Session

The 'R' Environment and R Fuzzy Toolkit

'R' is a programming environment for data analysis and graphics, developed as an open source application. It is a powerful and flexible package which has a number of tool -boxes. You are able to find copies of R on the windows machines in the A32 Terminal Room.

In this lab, you will enter sometimes unfamiliar/obscure R syntax into the command window. Many of these commands will be useful to you and so in order to get the best out of the lab sessions, you should try to understand how they work and what they do before moving on.

R is freely available open-source software, which is distributed in easy to install binary packages for most popular current operating systems, including Microsoft Windows, Apple Mac OS X, and versions of Linux. You can easily install R on your own computer, if you wish, by obtaining the current version from:

http://www.r-project.org

Part 1

Learning outcomes

After this part you will be able to use R to:

- **perform simple matrix calculations**
- **save data to a file**
- **load matrices from file**
- **use the R help function**
- **draw simple plots using R.**

Double click the R icon to get R up and running. Feel free to peruse the menu-driven help section at your leisure – for now we just want to use the command window and to do some calculations for us.

All the instructions below for some time refer to the command window. The command line starts with `>`, so type your commands after this symbol.

For the purpose of this worksheet, the commands you should type are in ***bold italic*** with the instructions you should follow in regular typeface. Sometimes R will return to you values which are slightly different to those in this worksheet. Do not be concerned about this: it is the nature of using random processes.

In order to enter a variable for use in R, variables can be assigned by using the following syntax. Here 'num' is assigned the value of 2.

```
num <- 2
```

or

```
num = 3
```

The R language has its origins in mathematical and statistical computing, and its main type of variable is the 'matrix'. A matrix is a two dimensional list of numbers, similar to a two-dimensional array in other programming languages, but with properties similar to the mathematical notion of a matrix.

To input the matrix $\begin{pmatrix} 1, 2 \\ 3, 4 \end{pmatrix}$ input the following.

```
a <- matrix(c(1,2,3,4), nrow = 2, ncol = 2, byrow=TRUE)
```

where 'matrix' instructs R to accept a matrix, nrow/ncol are the number of rows and columns respectively, and 'byrow' is if the input is by row (true) or by column (false).
If you want to find out more about this function, type ***help(matrix)***

Type 'a' and the output should look something like:

```
> a
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

A number of R commands use the <- notation for assignment. However, in nearly all cases, an equals sign (=) will also work. As with many open source packages like this there are often a number of ways to perform the same operation. Have a look at how the following commands can be used to enter a matrix. What are the differences if any?

```
a = rbind(c(1,2), c(3,4))
a = cbind(c(1,2), c(3,4))
a = matrix(c(1,2,3,4), nrow=2)
 a = c(1,2,3,4); dim(a)=c(2,2)
```

Now enter another matrix into variable b:

```
b = rbind(c(1,3), c(1,4))
```

We want to check that R knows how to multiply and add matrices correctly. The syntax for adding is simply '+'. The syntax for multiplying whole matrices is %*% – enter the following commands and check the answers.

```
a + b
```

```
a %*% b
```

Let's create another new matrix c:

```
c= rbind(c(1,2,3), c(3,4,5))
```

Recall that we can't always multiply matrices if they are incompatible in size. Lets try to multiply c*a. Does it return an error?

```
c %*% a
```

Can we multiply a * c though?

```
a %*% c
```

In order to multiply matrices element-by-element, the notation is different – simply a*a.

Now try the following and make sure you understand the differences between a*a and a %*% a :

```
a * a
```

Similarly for powers of the numbers in the matrix:

```
a^3
```

If you want to store the answer, another variable, say y, can be set to be the answer. As R is a programming language, variable assignment is possible in the usual manner.

```
y= a%*%c
```

and now you can use y in a calculation

```
y + c(1, 2, 3) + c(1, 2, 5)
```

R has many built-in capabilities to perform various manipulations on matrices, such as the ability to transpose matrices. This can be performed using the aperm() function – this is a useful function with a range of transposition options. However, the quick notation for the plain transpose is :

```
t(y)
```

So the transpose flips rows and columns. This is useful in situations where the parameters to a function require a column vector when our data is a row vector.

If we want to add an additional column to an existing matrix we can use the cbind( ) command. Alternatively we can do this using the matrix editor through the fix() command.

```
z= cbind(c, c(1,2))
```

The matrix a is not a big matrix – but it might be and we might want to save it to a file. The first thing to do is to ensure that the current directory is one we can write into and read from. It is best if this is a directory on the H: drive.

We can find which directory is the current working directory with getwd() and change it with setwd(), or we can use the File menu. Note that the forward slash '/' is used within R as the directory separator, so we might do something like[1]:

```
setwd("H:/Temp")
```

Now matrix a can be written to a file with:

```
write.table(a, file="data.txt")
```

Look at this file to see what you have got.

We can load files as well – be careful with the quote marks. They are essential.

```
z= read.table("data.txt")
```

There are also commands which read and write delimited text if you want other than space delimiters

```
L= read.table("a.txt", header=FALSE, sep=":")
```

Colon delimited files are used on some unix systems – it is more usual to use commas or tabs on windows systems.

The R help files are a useful source of information – simply type your keyword into the following syntax:

```
help(plot)
```

or with keywords:

```
help.search("line")
```

If you want to find out more about the help function then use:

```
help(help)
```

If you want to keep track of what you are doing try the history command (use the help to find out about it :) .

[1] The type of pathname you use will obviously depend on the operating system you are using.

If you want to keep all your working you can save the workspace either through the application (under 'workspace') or through the command prompt – again see help on save for the details.

To exit the program using the command line use the quit() command.

Entering data by hand can be tedious so R allows creating uniformly spaced data points

```
a= 0:100
```

Sequences of non-integer numbers can be done using this:

```
b= seq(0,1,.01)
```

You have probably realized by now that the output of assignments is automatically suppressed. However, just type 'b' now to show you the variable. Now, just make something to plot, e.g.:

```
c= b ^ 2
```

R comes with built in graphics capability and is a useful tool for investigating data and for plotting advanced graphs and charts, including a wide variety of 3D plots.

```
plot(a, c)
```

See the help on plot to see the various variants of plotting. For example, see the effect of the following:

```
plot(a, c, col="red", pch="*")
```

While the graphical window is still open, then add a title and axes labels using the following command.

```
title(main = "A Title", xlab = "X Label", ylab = "Y Label")
```

You can actually do this all within the plot command.

```
plot(a, c, col="red", pch="*", main = "A Title", xlab = "X Label",
ylab = "Y Label")
```

We can use the lines command to plot more than one data series on one figure (or, as in this case, add a second set of symbols for the same data series):

```
lines(a, c, col="blue")
```

Part 2

Learning outcomes

In this session you will :

- **learn how to access columns and rows of data**
- **how to do basic 2D plots**
- **how to use control structures**
- **meet the idea of an r-file.**

Ensure that you have a datafile, "a.txt" containing the following matrix:

A B
2 4
3 5
4 6

Use the read.table command to get the data from the file data.txt that you created last time into a matrix called ab. Check that the data has been read correctly.

```
ab= read.table("a.txt", header=T)
```

Check what is in ab in the R window.

To get the second column of ab try:

```
ab$A
```

and the second row

```
ab$2
```

Set the vector a to be the first column of ab but don't bother to display again

```
a= ab$A
```

If you just want to focus on this matrix and avoid the $ notation, then you can attach the data, so the variables can be accessed directly

```
attach(ab)
```

and when you're done with that matrix/dataset:

```
detach(ab)
```

Add a column to ab – described earlier (hint – you need to cbind or fix the matrix)

R has standard programming features like for-loops which can be used in a variety of ways. Here we also introduce rnorm() which is a way of generating normally distributed sequences of numbers in a random fashion. Also, note the notation for accessing separate elements of the matrix, using the square braces:

```
for(i in 1:nrow(ab))
{
  f = 0.5 + rnorm(ncol(ab));
  ab[i,] = ab[i,]+f
}
```

In order to get this to work, press return at the end of each line (which should return a '+' symbol). Don't forget a semicolon between each line, and remember to add curly braces. This piece of code has altered the rows of ab by adding random elements to them. R also has if statements:

```
if ( i == 5 ) {
  print('five!')
} else {
  print('not five!')
}
```

There are many other language constructs which you can use for programming – see the help for syntax. Now save your new version of ab in data.txt – you will be using this data later on.

```
write.table(ab, file="data.txt")
```

We can plot all the columns against the x values on one graph like this

```
plot(ab, a$B)
```

We can use the lines command to plot more than one data series on one figure:

```
lines(a$A ~ ab, col="blue")
```

Function files

A function file allows you to repeat a series of commands that you have created without having to type them all in again. This is particularly useful for conducting experiments where you want to repeat tests with only minor changes. Almost all R commands are actually function files.
Create a new source file by choosing File|New from the menu enter the following:

```
myplot <-function(){
  pdf("g1.pdf");
  a = seq(0,1,0.1);
  c = a * (a^2);
  plot(a, c, col="blue", pch="*");
  dev.off()
}
```

Save this as myplot.r . You can add this file to the project through the interface or by using the source() command. You will now be able to enter myplot() at the command line and see the effect (note that this file must be in the correct directory).

Part 3

Learning outcomes

In this session you will do more with data analysis functions, learning:

- **how to load and manipulate data**
- **how to describe data, including statistical summaries and cross-tabulations**
- **how to do more advanced types of plotting, including multiple plots**
- **how to do basic 3D plotting**

You can download data files to experiment with from the UCI Machine Learning Repository:
http://archive.ics.uci.edu/ml
R also has several built-in data sets available, including the famous 'iris' data. Type:

```
data()
```

to see a list of in-built data.

In this section, we will use the iris data, but will copy it into our own variable, ir, so that we can change names, etc.:

```
ir= iris
```

Technically, the type of the variable, or 'class' as it is known in R terminology, is not a matrix, but a data.frame:

```
class(ir)
```

However, it broadly behaves in a similar way to a 2D matrix. For detailed differences, see R help and online manuals.

Once the data is in such a variable, we can set, view and use the names of each of the columns, rather than using the numeric column indices:

```
names(ir) = c("sepal.length", "sepal.width",  "petal.length",
"petal.width", "class")
names(ir)
ir$sepal.length
```

Of course, the numeric indices can still be used:

```
ir[1,]
ir[1:3,]
ir[c(1,51,101),]
```

Have a play with other indexing to understand the capabilities of R.

There are various functions to calculate statistical properties of data, such as:

```
mean(ir$sepal.length)
median(ir$petal.width)
summary(ir)
```

And functions to perform tabulations and cross tabulations:

```
table(ir$class)
ir$sepal.length.class = ifelse(ir$sepal.length <6,  "setosa",
"other")
table(ir$sepal.length.class)
table(ir$sepal.length.class, ir$class)
```

R has many functions for more advanced graphs and plots, for example:

```
pie(table(ir$class))
pie(table(ir$class), col=c("white","blue","red"))

hist(ir$sepal.width)
hist(ir$sepal.width, breaks=30, col="blue",
  main="Histogram of sepal width", xlab="sepal width")
```

Also, the capability to add extra lines, legends, text, etc. to plots.

```
abline(v=mean(ir$sepal.width),col="red", lwd=3)
abline(v=median(ir$sepal.width),col="green", lwd=3)

legend("topright",c("mean","median"),col=c("red","green"),
  pch=16)
```

R can easily plot multiple plots on one 'window':

```
par(mfrow=c(2,2))
boxplot(ir[,1:4], las=3, main="whole data")
boxplot(ir[ir$class=="setosa",1:4], las=3,
  main="setosa")
boxplot(ir[ir$class=="versicolor",1:4], las=3,
  main="versicolor")
boxplot(ir[ir$class=="virginica",1:4], las=3,
  main="virginica")
```

Finally, the latest versions of R have introduced interactive 3-dimensional plotting capabilities:

```
library(rgl)
?plot3d
plot3d(ir$sepal.length,ir$sepal.width,ir$petal.length,
  col=ir$class, xlab="sl",ylab="sw",zlab="pl", size=8)
```

If these commands do not work (if you get a form of "library RGL not found" error message), then you may need to download and install the RGL library package. You may not have permissions on the lab machines to do this, but can experiment on your own computer.

Part 4

Learning outcomes

In this session you will practice the use of R by creating a tipper system that does not rely on fuzzy logic:

- **how to create a complex function for the tipper problem in R**
- **illustrate the effect of your tipper function by plotting a 3-d surface**

We are going to work through the 'Fuzzy vs. Non-Fuzzy' tutorial that you met before, but this time actually implementing the non-fuzzy tipper in R. The description is in the 'fuzzy-approach' worksheet.

First of all, let's create the service and food variables, which each range from 0 to 10 in steps of 0.1:

```
service= seq(0, 10, 0.5)
food= seq(0, 10, 0.5)
```

Now, working from the worksheet, let's create a fixed tip of 15% (0.15) and create a plot of the tip (on the y-axis) against the service (on the x-axis).

```
tip= 0.15
plot(service, tip, type= 'l')
```

Actually, the plot command is not quite correct, because we are trying to plot a single fixed number (the tip) against a range (service from 0 to 10). The simplest was of getting around this for now is to use a trick in R:

```
tip= 0.15
plot(cbind(service, tip), type= 'l')
```

We can make it look even more like the MATLAB equivalent by colouring the line blue:

```
plot(cbind(service, tip), type= 'l', col='blue')
```

This relationship does not take into account the quality of the service, so you need to add a new term to the equation. Because service is rated on a scale of 0 to 10, you might have the tip go linearly from 5% if the service is bad to 25% if the service is excellent. You can achieve this will the following formula:

```
tip= 0.20/10 * service + 0.05
```

And (now the tip variable is the same length as the service variable) the relationship can be seen in the following plot:

```
plot(service, tip, type= 'l', col='blue')
```

The formula does what you want it to do, and is straightforward. However, you may want the tip to reflect the quality of the food as well. This extension of the problem is defined as follows.
The Extended Tipping Problem. Given two sets of numbers between 0 and 10 (where 10 is excellent) that respectively represent the quality of the service and the quality of the food at a restaurant, what should the tip be?

First, we need to create a function to map service and food to tip, We do this by:

```
sf= function(s,f) 0.20/20 * (s + f) + 0.05
```

And then, we calculate this function over a matrix of service and food values by:

```
tip = outer(service, food, sf)
```

To see this relationship, we need to plot a 3D graph of tip against both service and food. This can be done in R as follows:

```
persp3d(service, food, tip, col=tip*40)
```

You can now extend this function along the same lines as described in the MATLAB tutorial, simply by extending the definition of 'sf'. I recommend doing this by defining a proper multi-line function in a text editor, cuting and pasting into R as you go. So, for example:

```
sf= function(s,f) {
  servRatio= 0.8;
  t= servRatio * (0.2/10*s + 0.05) +
    (1 - servRatio) * (0.2/10*f + 0.05)
}
tip = outer(service, food, sf)
persp3d(service, food, tip, col=tip*40)
```

See if you can extend it to the full complex function at the end of the tutorial.