

BIOL 3551: Introduction to R

The purpose of this document is to introduce you to R. Subsequent workshops will use R to illustrate some of the material covered in lectures but, for now, it's useful for you to get an idea of what R is, how you use it to do simple mathematical operations, how to iterate operations, and how to do useful things like reading in (or exporting) data, or producing graphs.

1. Why use R?

There are many ways to assess the implications of simple mathematical models of population dynamics and to conduct simulations of model outcomes. Knowledge of anything from Excel to low-level programming languages (like Fortran) can enable population simulations to be run. Which you opt for depends on multiple trade-offs, such as that between speed of coding (which is typically better using high-level software like Excel) and speed of execution (which is substantially better using low-level languages like Fortran or C). R lies somewhere in between those extremes but towards the Excel end of the spectrum. It is particularly useful as an introductory piece of software for population modelling for several reasons:

- Unlike Excel, R is free to download. Consequently, it is in widespread use and is rapidly becoming the software of choice for researchers in a wide range of disciplines.
- R is 'open source', meaning that anyone can contribute code. That means that, unless every aspect of the problem you are working on is completely unique, someone will probably have encountered it before and might well have written code to solve it.
- R is a 'one-stop shop' for many research requirements. You can use it for statistics, graphics, modelling and automating tasks. You can also use it for mapping and dealing with spatial data.
- R is particularly useful because you write commands for everything you do. Although that can be time-consuming the first time you do it, you can save the commands for use later. That means that there is never any question about how you generated a particular result, because the entire set of steps is recorded. More importantly, when your data change, your assumptions change or you get new data, you can repeat an analysis without having to change anything but the data file. That is particularly useful if you realise that your original data were in error.
- R runs on any platform (Windows, Macs, Linux) and so code can be easily shared with others.
- R has a large library of built-in functions to do things rapidly. Examples include that you can do matrix mathematics with single commands (very useful, as we'll see below), and that you can perform a calculation for lots of different numbers with a single command. If you were to do the same in Excel, you would need to have all your numbers in one column, then to copy a formula down next to those numbers. In R, a single formula can refer to an entire list of numbers and the result of running that formula will list the outcome for each. That 'vectorisation' can make it quick to code multiple calculations in a way that would take longer in a lower-level programming language (because you would need to write loops to

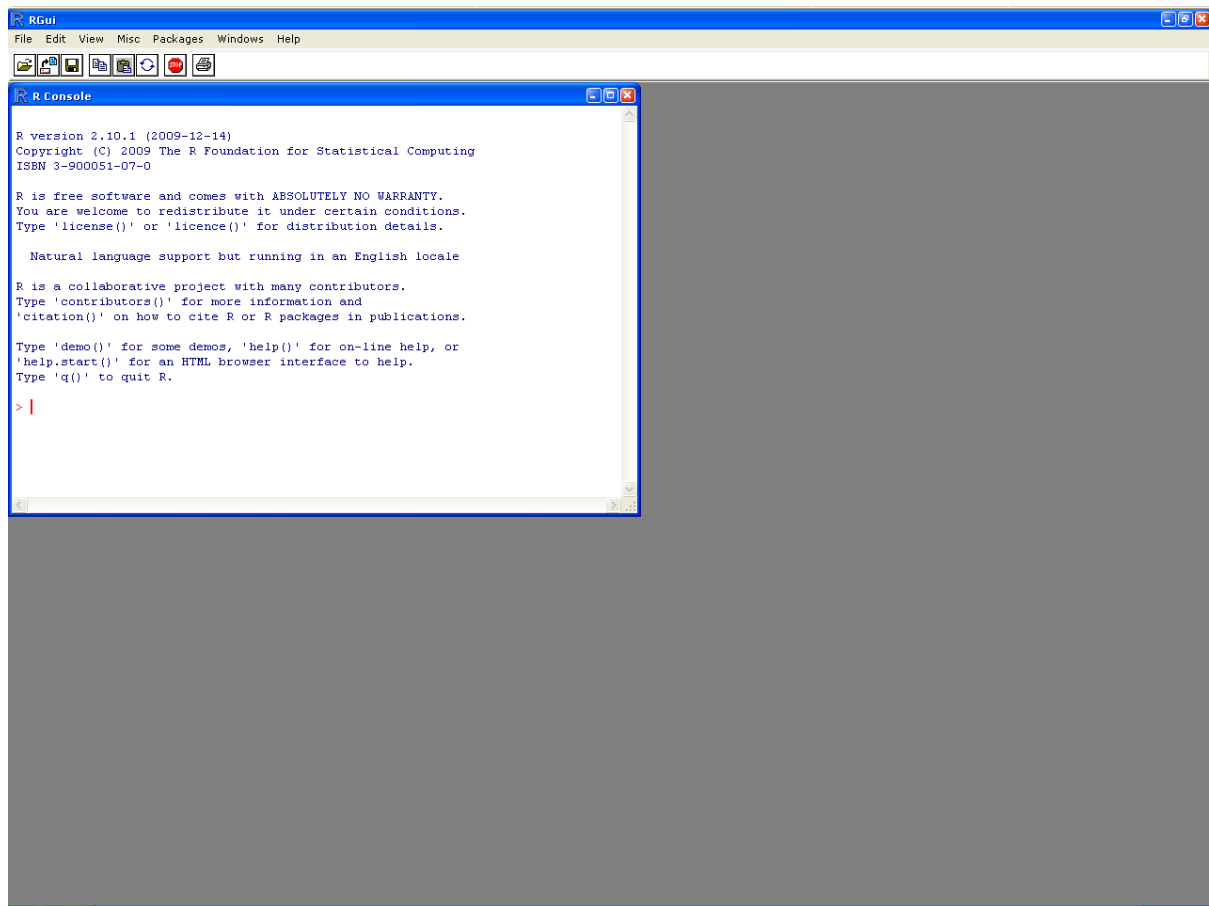
evaluate the formula repeatedly) and longer in Excel, because you would need to copy the formula repeatedly within your spreadsheet.

2. Installing and opening R

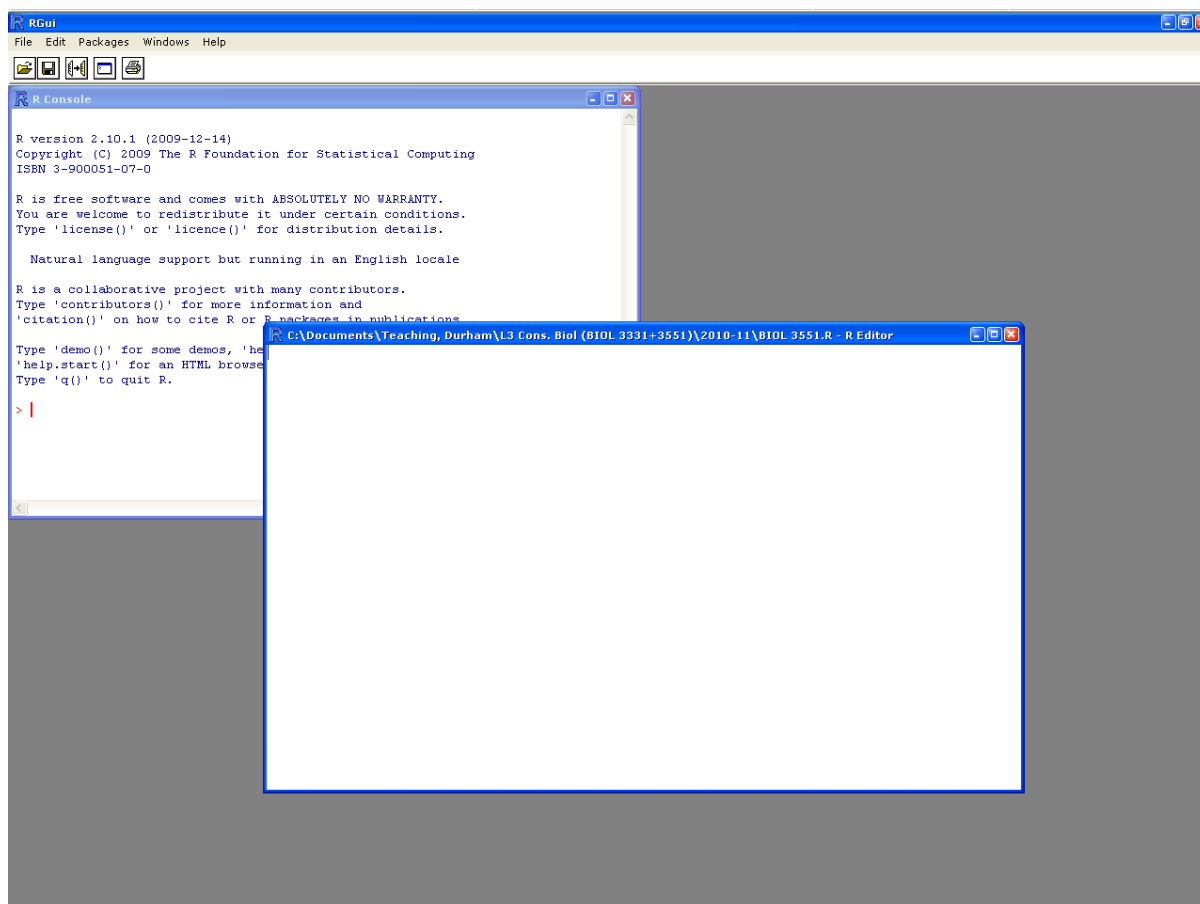
If you want to download R and install it on your own computer, you can do so by following the links at <http://www.r-project.org/>. This website contains a wide range of useful links and information, so it is also worth exploring to see what's available.

To find R on networked computers at Durham, use the Start button, then go to Programs and Academic Software. The latest version of R should usually be found under either the Biological Sciences or Mathematics folders.

Opening R up, you should see something like this:

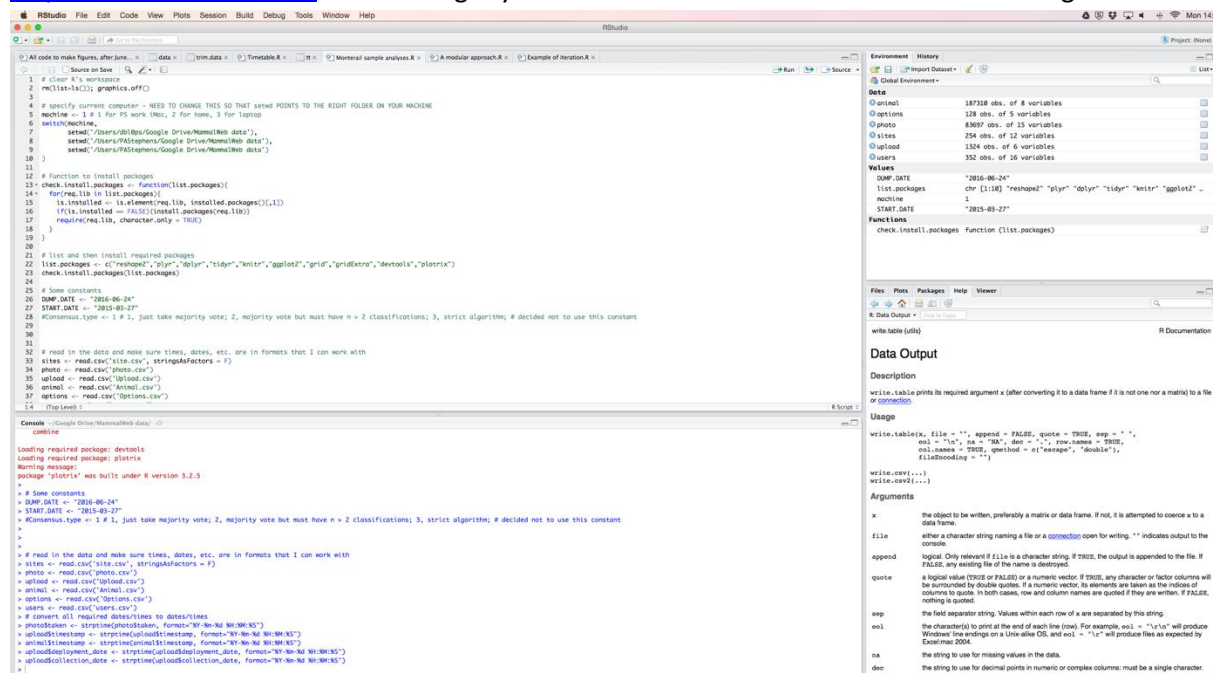


The window with the red “command prompt” (>) is known as the ‘Console’ window. You can type commands directly into the Console window but they won’t be saved. It’s good practice, from the outset, to make use of a ‘script file’ that can be saved, edited and re-run. To create a script file, select **File > New script** from the menu. You should save the script file immediately using **File > Save**. Give the file a meaningful name and finish it with .R. Here’s what the screen looks like if I save a file as “BIOL 3551.R”:



Having created a script file, you can type commands directly into it, remembering to save it occasionally (using CTRL-S).

An alternative to R, which is available on many of the university's computers, is RStudio. RStudio is a (relatively) user-friendly environment for using R. It is also freely-available, from <https://www.rstudio.com>. It has a slightly different look to the basic R environment – e.g.:



Here, the screen is divided into 4 components: the top left is an open script file (this doesn't appear until you open a script or go to File > New file > R script); the bottom left is the console window; the top right shows current variables and other objects in R's memory; and the bottom right is a useful viewer for help text, for graphics that you make, or for various other elements of the environment.

3. Running scripts and getting help in R

Some computers seem to be configured to provide help online, whilst others open text files within R. In either case, the commands are the same. The most basic kind of help can be found by typing **help()** into your script file and then running it. There are a number of ways to run it. One is to ensure that your cursor is somewhere on the line of text that you just typed and to press CTRL-R. The same effect can be achieved by highlighting the text and pressing CTRL-R. Obviously, that is more efficient if you want to run multiple lines of code at once. If you want to run your whole script, you can use CTRL-A to highlight the whole thing, and CTRL-R to run it. Notice that when you run some code, it appears in the 'Console' window, as though you had typed it at the command prompt.

More informative help files will be accessed by looking for more specific items. For example, for help on statistics commands, you could start by typing **help(stats)** or, to get the same effect, **?stats**. If your help function is html based, this will open a navigable window. If not, it will suggest other commands to further your search [e.g. **library(help="stats")**]. That will list functions within the 'stats' library. Browsing down the list of functions, you might see some that you could imagine being of use to you, such as **t.test**, **kruskal.test**, and **lm**. You can get further help on any of these by typing, for example, **?t.test**, or **?lm**. Help files take some getting used to but it is usually worth looking at the examples given at the bottom of each help page.

4. Mathematical operators and variables

At its most basic, R can be used as a calculator. For example, try typing **3+5** in the script window and running it. As before, '3+5' will appear in the Console window, as though you had typed it at the command prompt. Below it will be the answer to the calculation, appearing thus: [1] 8.

The [1] is meaningless in this context. However, it illustrates that R's default is to 'think' in terms of vectors. A vector is a list of numbers. The answer to the calculation '3+5' is a list of numbers of length 1. i.e. it is a 'scalar' (a single number). R gives the whole list (albeit that there is only one number in the list) and identifies the location of the first number in the list using the '[1]' notation. The reasons for this will become more obvious when you use vectors.

Other mathematical operators can be used in the same way. Experiment with +, -, *, and /, for addition, subtraction, multiplication and division. You can also use functions like **sqrt()**, **exp()**, **log()**, **min()**, **max()**, **mean()** and **var()**. To raise a number to a power, you can use the ^ symbol; for example, **5^3** (to give 5 cubed).

Often, we want to use 'variables' in formulae instead of numbers. A variable is an 'object' to which you can temporarily assign a value. We can base computations on the variables but the outcomes will be sensitive to the value that has been assigned to the variable at the time that we evaluate the computation. This can be quite powerful as you'll see later, because we can update the values of variables to find out how they will change over time. For now, however, consider only that a variable needs a name and a value. A statement (in our script file) to assign the value 3 to a variable

called A would look like: **A=3**. Similarly, we could write **B=5**. We can then evaluate the equation **A+B** by typing it in to the script file and running it. We can either type each statement on a different line, select all 3 lines and run them, or we can type the three statements on a single line, separated by semi-colons (i.e. enter the commands: **A=3; B=5; A+B**), and run that line. In either case, we should get the same output as before: [1] 8.

There's no reason why we should have called our variables A and B. They could have had any names and, in general, the more meaningful, the better. We could also have assigned the outcome of the computation to another variable – e.g. used code like **Outcome=A+B**. If you run that line, you will find that no value is returned in the console window. Instead, the value is stored for later use (whenever you invoke the variable 'Outcome'). One way to see the resultant value and to store it to a variable is to surround the command with brackets – e.g. try typing and running: **(Outcome=A+B)**.

5. Making use of vectors

Up to now, we have done nothing that a calculator couldn't do. To see how vectorisation can make things faster than using a calculator, we can try using vectors. For example, try typing and running **sampleVector=1:10**. Now, just double click on the name of your vector (so that you select only the part of your script that says 'sampleVector' and use CTRL-R to find out what value has been assigned to this object. You can see that the colon is used to indicate 'all the integers between'. This is a quick way of generating a vector. Notice, also, that only the first position in the vector is indicated by its element number. This is because all the elements of the vector fit on a single line. If you generate a larger vector [e.g. by typing **(sampleVector=1:100)**], you'll see that R identifies the position of each element that starts a new line in your console window. Two other ways to make a vector are: (1) to specify the elements individually using the **c()** function [e.g. try typing **(vector2=c(1,2,5,10,100))**]; or (2) by generating a sequence using the **seq()** function [e.g. try typing **(vector3=seq(3,25,0.5))**]. With the sequence command, you are supplying the start (3) and end (25) numbers in the sequence, as well as the increment size (0.5). As always, if you wanted more information about the function, you could type **?seq**.

You should be aware that you can see the value at any particular location in a vector by referring to the element's numerical index. For example, if you declare a vector using **A=c(10,3,100,6,17)**, then **A[1]** refers to the value at the first position within A (i.e. the value 10). **A[4]** refers to the value of the 4th element in the vector (i.e. 6). **A[3:5]** refers to the elements at positions 3 to 5 in the vector A (i.e. it will return the vector 100,6,17).

The mathematical operations referred to in section 4 can all be used with vectors. For example, try the following code:

```
A=seq(0.5,1.5,0.05)
(B=A^3)
```

You can also perform mathematical operations on two vectors. Assuming that the vectors are of equal length, R will typically evaluate the result using the elements in corresponding positions within the two vectors. For example, try

```
L=2:5
M=6:9
```

(N=L*M)
(P=L^(1/M))

6. The assignment operator and commenting on your code

Up to this point, I've been assigning values to variables using the '=' sign. For various reasons, not all of which are clear to me, users of R tend to avoid using the '=' sign in that context. Instead, they use the assignment operator, <-. Thus, you will often see code that looks more like **ZZ<-seq(1,2,0.01)**. I try to remember to use this notation and will try to continue to do so. Where you do see the '=' sign is typically within a function. Where a function requires that information is assigned to one or more variables, the '=' sign is the default (and the assignment operator might not work).

Another useful habit to get into is commenting on your code so that, if you return to it later, you can remember why you did what you did. Comments can be put anywhere in amongst your code, as long as they are preceded by the # symbol. Given that R will ignore anything that follows the # sign, obviously you shouldn't try putting commands that you want R to follow, anywhere on a line after a #.

7. Simple graphics

Initially, producing graphics in R seems awkward. However, the flexibility, repeatability and customisation potential is infinitely preferable to most alternatives. The most basic plots are produced using the plot() function. For example, try the following code:

The following 3 lines will produce a simple graph

```
A<-1:10  
B<-A+(1.5*rnorm(10))  
plot(A,B)
```

Consider how B is derived. In particular, you could try seeing what is produced by the function 'rnorm(10)' (a vector of 10 random numbers, picked from a distribution with a mean of zero and a standard deviation of 1). What then is the result of '1.5*rnorm(10)'? Hopefully, you can see what underlies B, and why it is only roughly linear with A.

The plot derived above is rather uninspired but there are many ways that you can customise it. Key things to be aware of are shown by the following code:

The next line specifies more details of the graphic output

```
plot(A,B,xlab="Values in vector A",ylab="Values in vector B",pch=19,col="red",cex=1.3,bty="l")
```

Parts of this should be self-explanatory but parts might be less obvious. pch specifies the type of symbol for the points (the 'point character'), cex specifies its size (the character expansion, relative to the default of 1), whilst bty specifies the type of box to have around the figure (in this case, and L-shaped box). Further information can be gained by typing **?plot** and **?par**. Colour options can be seen by typing **colours()**. An alternative look could be produced by typing:

```
# type="l" is used to produce a line plot; type="b" would have both points and a line
plot(A,B,xlab="Values in vector A",ylab="Values in vector B",type="l",
     lwd=2,lty=2,col="blue",bty="n")
```

Experiment with producing graphs of varying appearance. Note that the axes ranges can be set using the `xlim` and `ylim` commands (see the help file under **?plot.default**). Greater flexibility is obtained by suppressing the axes altogether (using `xaxt="n"`, `yaxt="n"` or `axes=FALSE`) and drawing on the axes afterwards using the `axis()` function (type **?axis** for help on this function).

8. Simple statistics

As an example of how R can be used to perform simple statistical tests, we can perform a linear regression on the vectors used to produce the plot in section 7. Regression is conducted using the `lm()` function. One way to write this is:

```
# lm is used for a regression where y is a function of x, by writing y ~ x. e.g.
mod<-lm(B~A)
summary(mod)
```

Obviously, given that B is based on a vector of random numbers, everyone's results for this regression will vary. The most important part of my output for this appears as follows:

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-0.2088	0.7843	-0.266	0.797
A	1.1489	0.1264	9.089	1.72e-05 ***

This shows that the formula for the regression line through my 10 data points is given by $B = -0.2088 + 1.1489A$. The slope of the line is significant (with $t = 9.089$ and $p < 0.05$) suggesting, unsurprisingly, that there is a strong effect of A on B. We can add this 'best fit' line to our graph with the code:

```
plot(A,B,xlab="Values in vector A",ylab="Values in vector B",pch=5,col="red",cex=0.8,bty="l")
abline(mod,lty=2,col="blue")
```

9. Loops and iteration

Loops allow us to iterate processes repeatedly. They are, thus, one of the most important tools for modelling population dynamics. A classic way to illustrate the function of loops is using text output. For example, try running the code `print("hello")`. The output, `[1] "hello"`, illustrates that R treats vectors of text in the same way as it would treat vectors of numbers. We can assign text to variables in the same way that we assigned numbers to variables. Try `Some.text<-c("Hello","world")`. Run `Some.text` to see what it contains. Another way to see the content on the variable 'Some.text' is to use the function `print(Some.text)`. This prints out the whole content of the vector once. To print it out multiple times, try:

```
for (L in 1:10) print(Some.text)
```

Notice that this iterates the process of printing 10 times. More generally, a loop will iterate the commands that follow it, with the 'loop variable' (in this case denoted 'L') taking each specified value (in this case, each of the values in the vector 1:10), iterating the command, then taking the next value in the list. This principle might be more obvious if we use a numeric example.

```
for (loop.var in seq(0.5,5,0.5)){  
  X<-loop.var/2+1  
  print(X)  
}
```

Here, the loop variable "loop.var" takes the values 0.5, 1.0, 1.5 ... 5.0 in sequence. For each value that it takes, the two following statements (within the curly braces, { and }), are executed. As a default, a loop 'for' command will execute the single statement that follows it. The curly braces are used to specify additional statements to execute for each value of the loop variable.

The more observant among you will have noticed that the previous code could have been executed more simply using, for example,

```
LV<-seq(0.5,5,0.5)  
(X<-LV/2+1)
```

That is relevant and reminds us that we should always seek the simplest and most elegant code to achieve our ends. However, here, the loop code serves only to illustrate its use. Loops are more necessary when one computation depends on the outcome of the previous computation. For example:

```
N<-rep(0,10)          # sets up a vector of 10 zeros that we can change using calculations  
N[1]<-100  
for (t in 2:10)  N[t]<-N[t-1]*(1+rnorm(1)/10)  
plot(1:10,N,type="l")
```

If you managed to graph that code, congratulations! You've just run your first population model in R (a so-called "random walk" model, in which population size one year is the same as last year's population size, \pm some random 'shock'). Notice that the model could not have been run using a vector approach, because we can only calculate each year's population size once we know the population size the year before. Therefore, we have to step through the calculations one at a time. Notice, also, that if you run the code again, you will not get the same result. Thus, this is a stochastic population model.

10. From vectors to matrices and data frames

Vectors can be thought of as a single list of values of any given type (e.g. numbers, character strings, etc.), storing the kind of information that we might store in a single column within Excel. Very often, however, we are interested in the relationship between two or more lists of values. In these situations, it helps to be familiar with matrices and data frames.

A matrix is an 'array' of values, like a 2-dimensional vector. The information within a matrix must all be of the same type (e.g. all numerical). We can either declare a matrix just as a lot of zeros (and

populate it later with values derived from elsewhere) or, if we know the values we want to put in the matrix from the outset, we can populate it at the same time as declaring it. In the first case, we could write something like:

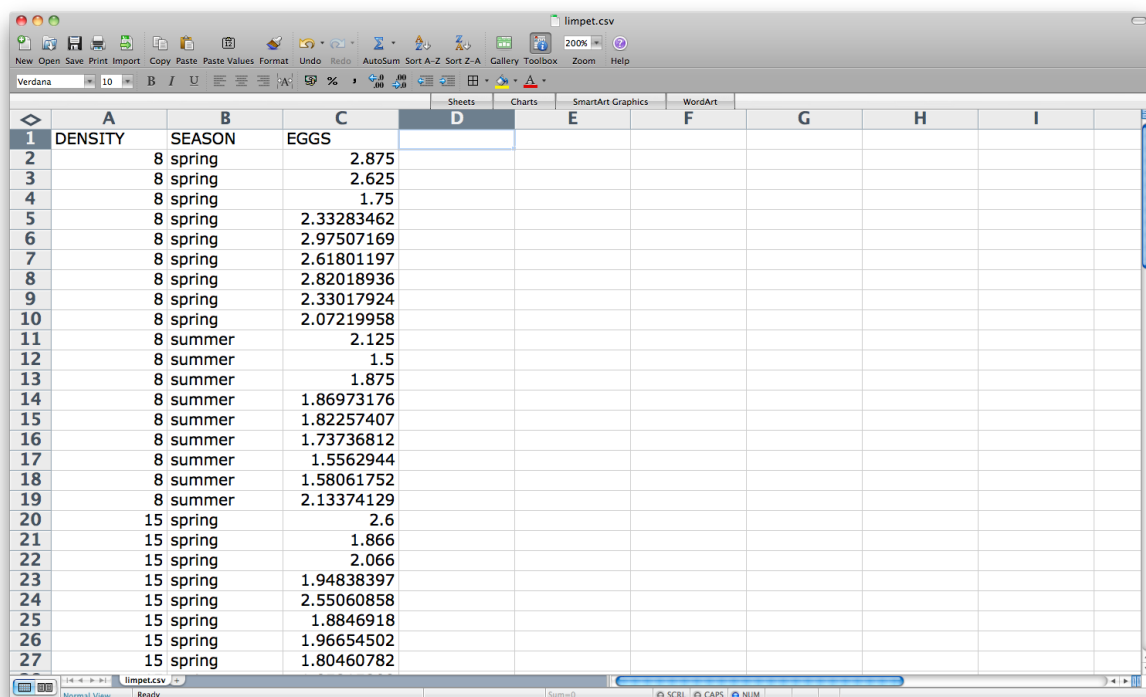
```
my.matrix<-matrix(nrow=5,ncol=3,0)
```

View the content of my.matrix to see what that statement does. The alternative approach (when we know the values we'd like to have in the matrix) is illustrated here:

```
my.matrix<-matrix(c(0,0.024,0.52,0.08,0.25,279.5,0,0.43),nrow=3)
```

Here, we have specified a vector of values that we want in our matrix. We have also specified the number of rows in the matrix. Consequently, R will put the values in to a matrix, column by column, assuming that each column contains the number of values specified by nrow. Obviously, the size of the vector needs to be a multiple of the number of rows in the matrix.

If we want to store different types of values in an object, we need a data frame, rather than a matrix. A data frame is essentially a table of information. It is perfectly possible to construct a data frame in R but often just as simple to import one that we made in other software. For example, imagine that we have a .csv file that we made in Excel (you can make one of these by entering information in Excel and using File > Save As to save it as a file of type 'comma separated values'). The data might look something like this:



	A	B	C
1	DENSITY		
2		8 spring	2.875
3		8 spring	2.625
4		8 spring	1.75
5		8 spring	2.33283462
6		8 spring	2.97507169
7		8 spring	2.61801197
8		8 spring	2.82018936
9		8 spring	2.33017924
10		8 spring	2.07219958
11		8 summer	2.125
12		8 summer	1.5
13		8 summer	1.875
14		8 summer	1.86973176
15		8 summer	1.82257407
16		8 summer	1.73736812
17		8 summer	1.5562944
18		8 summer	1.58061752
19		8 summer	2.13374129
20		15 spring	2.6
21		15 spring	1.866
22		15 spring	2.066
23		15 spring	1.94838397
24		15 spring	2.55060858
25		15 spring	1.8846918
26		15 spring	1.96654502
27		15 spring	1.80460782

We can easily import this into R using the command:

```
DF<-read.csv("limpet.csv",header=TRUE)
```

By specifying 'header=TRUE' (which can also be written 'header=T', because when R is expecting TRUE or FALSE, it will be quite happy with T or F), we tell R that there is a header row that can be

used for column headings (i.e. variable names). Note that the process will be made easier if you tell R which directory to look in for your input file. You should experiment with the functions `setwd()` and `getwd()` to find out about the working directory. Notice that R expects folder addresses to be in the format “C:/My Documents/R” – i.e. with a front-slash (/) between folder levels.

Data can easily be exported as a .csv file using the reverse process:

```
write.table(DF,"An_output_file.csv",row.names=F,sep=",")
```

This uses the first named object (DF) to create a file with the name given in inverted commas. The ‘row.names’ command tells it not to include row numbers in the output file and the ‘sep’ command ensures that values are comma separated. This information about data frames, input and output is surplus to the requirements of the current exercise but might be useful for future reference.

Recall that vectors are ‘indexed’ by referring to each value according to its place within the vector. The notation for matrices (and data frames) is very similar but we need to index both the row and the column. Using the ‘my.matrix’ object specified above, try the following commands to see how to index individual matrix elements, whole rows or columns, or groups of rows and columns.

```
my.matrix[1,3]  
my.matrix[3,2]  
my.matrix[2,]  
my.matrix[,2]  
my.matrix[1:2,]  
my.matrix[3,2:3]
```

Notice that, as with vectors, indexing matrices uses square brackets. The first term in the square brackets indexes the row (or rows) of interest, whilst the second indexes the column(s) of interest. Leaving one or other term out (i.e. having nothing either before or after the comma) is the same as specifying **all** of the corresponding rows or columns.