MAS115 R programming, Lab Class 1

Bryony Moody (Original notes by Prof. P. G. Blackwell)

2020-21

1 First steps

1.1 Creating a workspace and a script file

It is very important that all the programming that you do is filed neatly on your computer so that you can access it easily at a later date. You should create a separate folder for each significantly different bit of work and store all the relevant scripts, datasets and plots within that folder. All of these files should have memorable names indicating what they contain. If you do not do this then you will soon end up unable to remember the purpose of large numbers of your files.

1.2 Starting your session

1. Before loading R please do the following:
   - Use Windows Explorer to create a new folder in your `U://My Documents` drive entitled `MAS115`.
   - Within this new folder create a sub-folder called `RExerciseLabs`
   - Within this new folder create a sub-folder called `Lab1`

   We will use the `RExerciseLabs` folder to do all the R work for these practicals and create separate subfolders for each specific class. It will store all the code we write in script files (or in R markdown files—see below) along with any output we create.
2. Start **RStudio** from the application menu.
3. We want to work in the folder that we just created. To do this we need to change what is known as the working directory.

   Within **RStudio** select **Session** > **Set working directory** > **Choose directory ...** on the menu bar. Browse to the folder you just created and select it.

   N.B. To check which directory we are currently working in we can use the command `getwd()`.
4. Create a script window by selecting **File** > **New file** > **R script** from the menu bar. We will write *all* of our commands in this editor and not directly into the command window. To run any command or selection of commands, highlight the desired commands using the mouse and either press `Ctrl+R` or click on the `Run` button at the top of the script window. We will want to save this script file as we edit it and probably reload it later.
   - To save a script, select the script window and go to **File** > **Save as...** on the menu bar. Call the script `Exercises1.R`. When entering any script filename, it is best to add the .R extension at the end, as it will make the script easier to locate.
   - If you wish to load an existing script then this can be done by selecting **File** > **Open File...** on the menu bar.

We are now ready to proceed with the rest of the class. Again, all code that you type should go into a script window; very rarely should you be typing directly to the command line. (Getting help, using `?command`, is a possible exception.) You should use your newly created script `Exercises1.R` for the exercises. The code actually used in this handout has been (automatically) extracted as a script `MAS115Lab1.R` which you can use to reproduce the results shown.

2 Documenting your work

Whenever you program, your code must be documented so that someone—often a future 'you'—can look at it and figure out what the various parts do. We will look at two simple ways of achieving this.

2.1 Commenting

A script file, as described above, can be documented by including text comments in the script itself. In order to create a comment in R, you use the `#` character. Anything occurring later on the same line of code will not be read by the command line. For example try running the following commands; you will find that the second line doesn't work—R gives you an error message.

```
x <- 1:3
x <- x + 1  Add 1 to every element in x
x
```

```
Error: <text>:2:13: unexpected symbol
1: x <- 1:3
2: x <- x + 1  Add
               ^
```

However, if instead you insert a `#` character when you want to add a comment then it will run. Try this out.

```
x <- 1:3
x <- x + 1 # Add 1 to every element in x
x
```

```
[1] 2 3 4
```

2.2 Markdown

A more sophisticated approach is to produce a document that combines code, explanation and results in an appropriate way. There are several ways of doing this; to start with, we will look at one approach that is well suited to tasks that combine mathematics and programming, and which we will use for assessment in this module.

We will use an *R markdown* file, with the `.Rmd` suffix, which is a mixture of explanation and code; the explanatory text can use LaTeX to present mathematical ideas, and the code consists of *chunks*, each of which is like a mini R script. When the document is 'knitted' together—just like compiling a LaTeX document—the R code is executed and the results inserted into the PDF document that is produced. Thus a single `.Rmd` file both produces and documents the work. Some of you will have seen the idea before in MAS113, very briefly. The document you are reading was produced in that way; some very simple examples of R markdown files are given on the module website as explained below.

Layout in an R markdown file can be achieved in two ways. One way is to use LaTeX commands for that too. For example, this document uses `\emph` for emphasis, and `\section` commands etc to define its structure. The other way is to use some simple *markdown* tags, which are intended to make the unprocessed file easy to read; for example, `*this*` emphasizes *this*, and `#, ##,...` (starting on a new line) define sections, subsections

etc. You can use either; it is generally better not to mix them in a single document, as it makes the source `.Rmd` file harder to read.

A simple example of an R markdown file is given on the module website in two versions, with the layout defined either purely using LaTeX (`MAS115miniLaTeX.Rmd`) or using markdown tags as far as possible (`MAS115miniMarkdown.Rmd`) with only the maths in LaTeX. You can take either one and use it as a template for work that you hand in.

3 R programming tasks

In what follows, you actually need to run and understand the examples given before starting the tasks.[1]

There is no need type in the lines that begin with `[1]`—this is simply the output you should see when you run the code. You will see later why it has this form.

3.1 Creating simple objects

If we simply type a command e.g.

```
1/3
```

```
[1] 0.3333333
```

we see that the output is printed directly to the screen. To create objects we use the assignment operator `<-`. If we want to print the answer then we either type the name of the object we created or place parentheses around the command e.g.

```
x <- 1/3
x
```

```
[1] 0.3333333
```

```
(x <- 1/3)
```

```
[1] 0.3333333
```

Actually the operator `=` will work for assignment as well, but is discouraged because it is used for other things in R too, and because assignment is not the same as equality, and is not symmetric.

3.2 Variable modes

As with Python, R has several different modes of variable that it can store in memory, for example

- integer e.g. 0, 1, ... stored specifically as an integer;
- double/numeric e.g. 1.1, $\sqrt{2}$, 2, ... stored with double precision;
- character e.g. `"Treatment A"`, `"Bob"` or `"Kate"`;
- logical i.e. `TRUE` or `FALSE`.

{Note: R will store e.g. 3 as a double unless you specifically tell it not to.}

You can tell R which mode you want to store a variable in you when you create it. After creating some of the variables suggested later in the session, try looking at the *structure* of the variable created by typing e.g. `str(A)`. Can you recognise the output and where it tells you the mode?

[1] These tasks are inspired by the APTS course of Dr. R Ripley, Oxford University

3.3 Vectors

A vector is a collection of objects, all of the same type (e.g. double, integer, logical, character). Vectors are extremely important in R; in many ways they are the default data structure, and many aspects of R are designed round this idea. Important: a vector in R is more like a NumPy array than a list in Python.

To create a vector we often use a simple function such as `c, rep, seq`. The `c` stands for *combine* or *concatenate*. For example:

```
(x <- c(0, 1.2, 8/5))
```

```
[1] 0.0 1.2 1.6
```

```
(y <- seq(0, 1, 0.1))
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

The operator `":"` creates a vector of consecutive integers, a very useful special case (but note that the second endpoint *is* included). `seq` is a much more general way of creating regular sequences.

```
(z <- -3:6)
```

```
 [1] -3 -2 -1  0  1  2  3  4  5  6
```

For character variables we often use double quotes, `c`, and `paste`.

```
# R will normally try and work out what mode it thinks the variable is automatically
# This can be good but R can get it wrong, in which case you need to override it

## A vector of double precision numbers
A <- c(1.2,2,3.4) # The c() command simply tacks the numbers together

## To create a vector of integers you need to force it
B <- as.integer(c(1,2,3))

## A vector of characters (needs quotes around each value)
C1 <- c("Tim", "Jane", "Kate") # R recognises that the quotes imply characters
C2 <- as.character( c("Tim", "Jane", "Kate")) # Or again you can tell it for sure

## A vector of logicals (again R recognises these automatically)
D <- c(TRUE, FALSE, TRUE)
```

3.4 Tasks

1. Create a numeric/double object called `Score` with the value 10.4.
2. Create a logical object called `isMale` with the value `TRUE`.
3. Create an object called `Name` containing the character string `Tim`.
4. R treats even single objects as a vector of length 1. This is very important as it allows lots of coding efficiencies we will describe later. To show this use the object `Score` you previously created and type `is.vector(Score)`.
5. Using the `rep()` command, create a vector of character strings `Pest` containing 10 values with the first 3 being `"A"` and the last seven being `"B"`. You'll probably need to look at the help file `?rep`.
6. Using the `paste()` and the `rep()` command create a vector of character strings `Flower` containing 10 values with the first 8 being `"Helianthus debilis"` and the last two being `"Helianthus annuus"`. (Look at the `paste()` help page.)

Note: variables that can only take a few values, such as `Pest` and `Flower` above, can be stored more efficiently in R using a more sophisticated mode of variable, a *factor*; this is particularly important for some kinds of statistical modelling. We will look at this in detail later.

3.5 Random vectors

For generating *random* values, there are many built-in functions. `runif` generates real values between 0 and 1, by default. `sample` can generate many things; the default is a permutation, but it is also a way to generate samples from any discrete set of values.

To allow your (pseudo-)random values to be repeatable, you can set the *seed* for the generator using `set.seed(`n`)`, where n is an integer. You can turn this off using `set.seed(NULL)`. Experiment!

3.6 Tasks

1. Create a vector `height` of 10 random integers between 150 and 160, allowing repeats, using `sample()`. You might need to look at the help file for this command using `?sample`.
2. Generate some random samples—for example repeating the previous task—using various values with `set.seed`, to ensure you understand what is happening.
3. Experiment with other uses of `sample`. How would you simulate a series of coin tosses? Optional: what if the coin is biased?

3.7 Manipulating vectors

Since R considers vectors so fundamental, many operations with them are simple to carry out. Suppose we have three vectors (note the different lengths).

```
a <- 1:3
b <- 6:8
c <- 1:2
```

If we add/multiply vectors of the same length then they are added/multiplied elementwise e.g.

```
a + b
```

```
[1]  7  9 11
```

```
a * b
```

```
[1]  6 14 24
```

Alternatively if one vector is shorter than the other they can still be added but the shorter vector is repeated until it is sufficiently long. Note that R will give you a warning message if the number of times you need to recycle the shorter vector is not an integer.

```
a + c
```

```
Warning in a + c: longer object length is not a multiple of shorter object
length
```

```
[1] 2 4 4
```

```
a + 1
```

```
[1] 2 3 4
```

We can use elementwise calculations on vectors to build up more complicated objects. For example, let `x` and `y` be numeric vectors.

```
t1 <- x > 2 # What does this do? What is the structure of t1? Can you see why?
t2 <- y == 4 # Similarly what does this do?
```

3.8 Subsetting vectors

You can extract certain elements of vectors by using the built-in indexing. Note that indices start at 1 in R. What do the following commands produce?

```
d <- 11:20
d[2]
```

```
[1] 12
```

```
d[-2] # Note that this is d without its 2nd element
```

```
[1] 11 13 14 15 16 17 18 19 20
```

```
d[1:7]
```

```
[1] 11 12 13 14 15 16 17
```

If you want you can also change elements using the same indexing e.g.

```
d <- 11:20
d[1] <- 10
d[2:5] <- 9:6
d[8:9] <- d[9:8]
d
```

```
 [1] 10  9  8  7  6 16 17 19 18 20
```

3.9 Tasks

1. Why is the command `a+1` an example of recycling?
2. From your earlier object `height`, using an operator (i.e. not just entering it by hand), create a logical vector `isTall` which is `TRUE` if the corresponding value of `height` is 155 or more, and `FALSE` otherwise.
3. What will by produced by the following command? `height[isTall]`
4. Create a vector `Aheight` containing just those values in your vector `height` which have corresponding `Flower` values `"Helianthus annuus"`.
5. Do the following:
   (a) Create a vector `Even` of the first 100 even numbers in order.
   (b) By using indexing (with the `-` index), create a sub-vector `EvNoFirst` which has removed the first element of `Even` i.e. the integer 2.
   (c) Similarly, create a sub-vector `EvNoLast` which has removed the last element of `Even` i.e. the integer 200.
   (d) Calculate the difference `EvNoFirst-EvNoLast`. Is it what you expected it to be?
6. What does the function `table` do to a vector of coin-tosses from an earlier task?

4 Homework

Due practical class week 2. Your solutions must be in the form of a PDF document produced from an R markdown file, including a suitable title and your name in the header and code, results and explanation for each task. It should read as a proper document - split by question with explanatory text. Please submit your homework by 12pm Thursday 18th February 2021.

Tasks

1. R has built-in functions which enable us to sample from common random distributions. One such is `rgamma()` which enables us to sample from a Gamma random variable with density

$$f(x) = \frac{r^a}{\Gamma(a)} x^{(a-1)} e^{-rx}.$$

   The command `rgamma(10, 1, 5)` will produce 10 random observations from the gamma distribution with shape $a = 1$ and rate $r = 5$. Do the following:

   (a) By looking at the help file (using the command `?rgamma`), create 10000 numbers from the gamma distribution with shape parameter 2 and *scale* 4. Store them in a vector `x`.

   (b) Find the mean and standard deviation of this sample.

   (c) Find the mean of all the entries in `x` which are strictly greater than 2.5.

   (d) What does the following command do? `sum(x > 2.5)`

2. **Estimating π.** Georges is sitting in a French café after his lunch break playing with his toothpick. While he is trying to solve a particularly difficult maths problem his toothpick falls onto his notepad on his table. His notepad happens to contain horizontal lines exactly 4 cm apart while (after much use) his toothpick is exactly 2 cm long. The toothpick happens to fall so that it crosses one of the lines in his book.

   He starts to wonder how likely it is that his toothpick would have fallen in such a way. To investigate this he repeatedly throws his toothpick onto his page so that the location of the centre of the toothpick has a uniform distribution anywhere on the page and the angle it makes with the vertical is also uniformly distributed.

   (a) Georges first works out that the distance between the center of the toothpick on one of his throws and the nearest notepad line is uniformly distributed between 0 and 2 (can you explain why?). Using `runif()` create 10000 such random distances and store them in a vector called `CentDist`.

   (b) By default, R uses radians to measure angles. Georges calculates that the angle from the vertical at which his toothpick falls is uniformly distributed between 0 and $\pi/2$. Using `runif` again, create 10000 such angles and store them in a vector `Angles`.

   (c) We can plot an example of the toothpick and a line as shown below in Figure 1. Simple trigonometry tells us that the height d that the toothpick extends vertically above its mid-point is $\cos(\theta)$, where θ is the angle from the vertical.

   Now if the centre of the toothpick landed x cm from nearest line then it will cross that line if $x < d = \cos(\theta)$. Transform your variable `Angles` into a vector `d` working out the vertical height that each of the sample toothpicks will extend above its centre.

   (d) Using R calculate the proportion of Georges' 10,000 sampled toothpicks which cross lines and store it as the variable `p`.

   (e) Calculate 1/`p`. Does it look like a number you recognise?

(f) *Additional Challenge - Optional* Can you show formally that if he were to keep throwing his toothpick eventually this fraction would tend to a simple function of π?

[*Hint*: We need to work out the $P(X < f(\Theta))$ for your function $f(\cdot)$ defined earlier. The problem is that both X and Θ are random variables. Try and condition on the value of Θ using the ideas from MAS113 i.e.

$$P(X < f(\Theta)) = \int P(X < f(\Theta)|\Theta = \theta)p_\Theta(\theta)d\theta$$

where $p_\Theta(\cdot)$ is the density of Θ.]

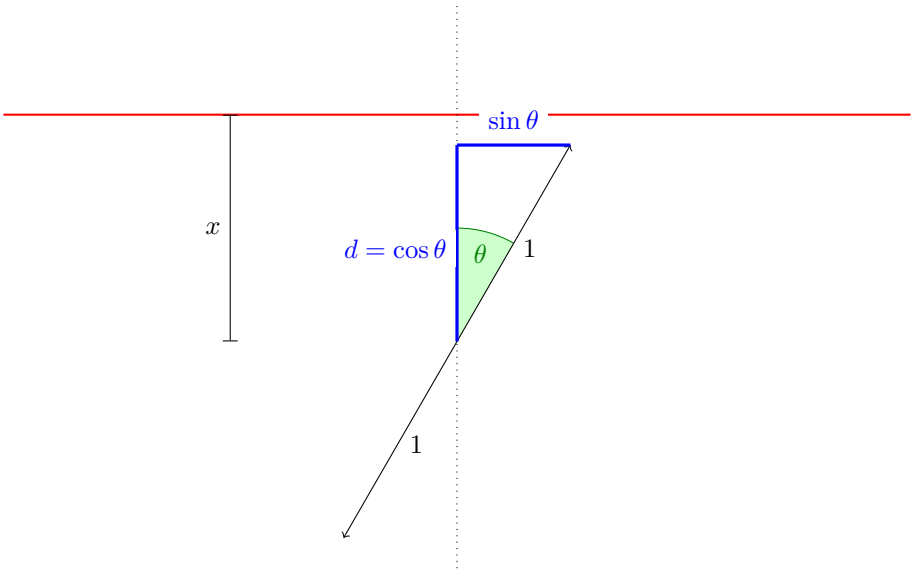

Figure 1: Plot illustrating a sample toothpick