CMM535 Data Science Development Lab Exercises

Worksheet 1 - Introduction to R programming

Worksheet Aim and Objectives

This is the first in a series of lab worksheets that will introduce you to a software tool called **R**.

- **R** is a programmable software environment for statistical and graphical computing
- We shall work with R via an integrated development environment (IDE) called **RStudio**

After completing this worksheet you should be:

- familiar with the layout and main features of the RStudio IDE
- able to use R for performing basic arithmetic and logical calculations
- able to use variables in R for storing results of calculations
- familiar with the various datatypes which can be used in R.
- able to load data from (csv) files.
- able to use some to R's built-in functions to perform calculations
- able to create vectors of values
- able to create simple plots of data stored in pairs of vectors using the **plot** function
- able to install and load packages (libraries) in R.

Using *R* and *RStudio*

R and **RStudio** are already installed on all School of Computing PCs (whether logged in directly at the machine, or logging in remotely to a specific machine remotely via **MyApps>DESKTOPS**, see https://myapps.rgu.ac.uk/).

You do not need to use MyApps when in the School Labs!

R is the *backend* programming language. It does have a console for accessing it directly, but it is far more flexible and convenient to use **RStudio** as a *frontend* for connecting to **R** and managing files.

Note that a University IT Service managed version of **R** and **RStudio** can be accessed remotely via RGU **MyApps>APPS**, but that this has an older version than the one installed on School of Computing machines.

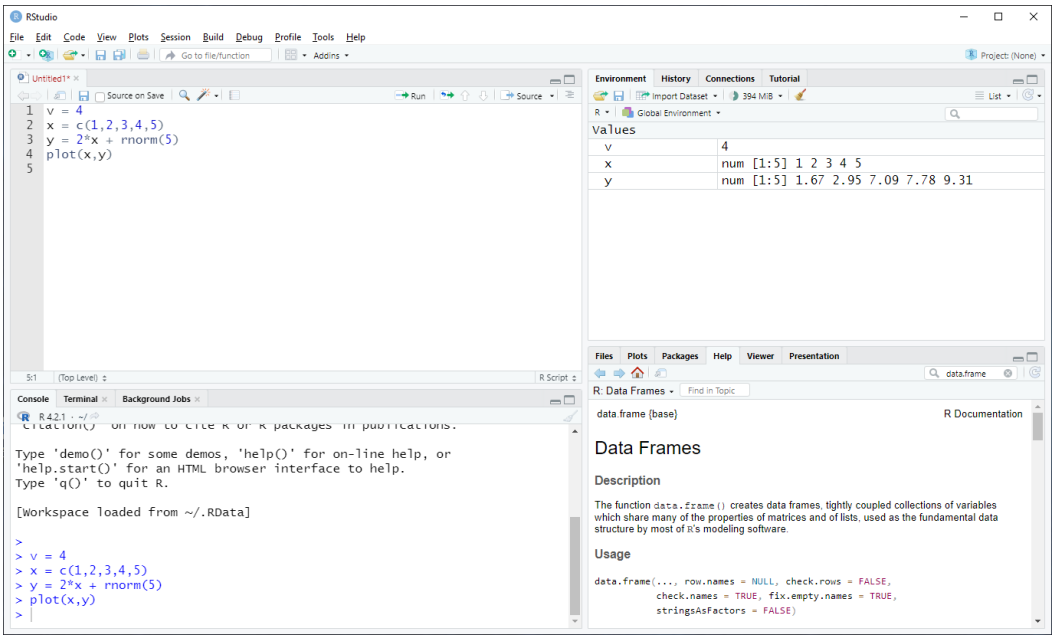
You can also install **R** and **RStudio** on your own personal devices. **But during lab sessions please use the School machines** (as you will need to get used to them for the Practical Exams). Both **R** and **RStudio** are open source and can be downloaded for free.

- If doing so, then install R first. R can be downloaded from https://cran.rstudio.com/
- Then install RStudio.
- RStudio can be downloaded from https://posit.co/download/rstudio-desktop/
- Note that if working on your own machine you will have to install libraries whenever you first use them. In the School machines, nearly all libraries you will need are already installed

1.1 Starting RStudio

RStudio is a program that allows us to interact with the R calculation engine and language. It is possible to *run* R directly via the R Gui. But **RStudio** offers a far more user friendly and feature rich user-interface.

Start **RStudio.** Once the program starts you should see a window similar to the one below:



In the figure we can see that RStudio offers us:

- A **Console** (on the bottom left) in which we can enter individual R commands
- A **Source** pane (on the top left, once we create a file) in which we can edit and run R scripts and R Markdown files
- In the top right there are several tabs:
  - An **Environment** section in which we will be able to see data variables and their values as they are calculated
  - A **History** tab that will allow us to see and re-run commands we have run
  - A **Connections** tab that can facilitates connections to external data sources
  - A **Tutorial** tab that allows access to a variety of in-built R tutorials
- At the bottom right there are also several tabs:
  - A **Help** tab to access help on R language commands, package definitions and examples
  - A **Packages** tabs where we can load and install libraries of extra functionality
  - **Files, Plot and Viewer** tabs though which we can see plots, and navigate files
- Along the top a **Menu Bar** with a variety of menus e.g. File, Edit, Session, Tools

Take a few moments to navigate around the RStudio window to familiarise yourself with it

Important Notes

- R is case sensitive, so always bear that in mind when entering variables and function names: e.g. `a` and `A` will be different variables to R; R will not recognise `Length(x)` if the function is `length(x)`
- If you cut and paste something containing quotes, you may need to retype the quotes as these don't always paste in the correct format.

Next, to get a feeling for what R can do, work through the following sections, typing the blue command into RStudio and observe the outputs.

1.2 Using R as a calculator

Let's start by entering values directly into the **Console**. When doing so we are interacting one line at a time directly with the **R** interpreter; it runs the command and either returns a result in the console, or saves the return value in a variable if the instruction is an assignment.

When we enter arithmetic calculations directly into the **Console**, **R** will perform the calculation and display the result. Use R to perform the following basic arithmetic calculations

```
2+2
6*7
13^2
19/2
```

R adopts the standard rules of precedence in calculations (powers before multiplication/division before addition/subtraction). Test this by calculating:

```
23 + 3 * 2
3 * 5 ^ 3 - 1
```

Use brackets to dictate the order of calculation in compound expressions. Test this by calculating:

```
(23 + 3) * 2
(3 * 5) ^ (3 - 1)
```

R also has all the standard mathematical functions built-in. Try a few examples:

```
sin(30)          # things with brackets are functions
log(100)
log10(100)
sqrt(36)
pi               # no brackets here since pi is a variable
```

Note that when in the console, pressing the **up** and **down** buttons on the keyboard will let you scroll through the most recent commands that you have entered in the current session. This can save some typing if you wish to run the same command again, or wish to edit and re-run a command.

R has lots of functions that basic calculators do not have, e.g. try

```
runif(6)   # this produces 6 random numbers between 0 and 1
print("Hello")
Sys.time()
Sys.Date()
max(10/3, pi, 3.2)
```

Note that anything in a line after a `#` is ignored by R, so we can use this to add comments into our code

1.3 Saving results in variables in R

Often we want to save the results of a calculation for use later. We can save calculation in named **variables**. For example, type the following:

```
x = 7

p = 14 *5

q = sqrt(10)
```

Notice that when a value is assigned to a variable, that variable and its value appear in the **Environment** window in the top-right pane of the RStudio window.

If a variable has been created, we can then use the variables in subsequent calculations:

```
8 + q

q^2

p * q
```

Note that R is case sensitive, so

```
g = p + q

G = p + q
```

creates two entirely distinct variables.

Note that in R we have a choice of symbols for assigning values to variables

```
f = g^3

F <- g^3
```

`=` and `<-` are both interpreted as assignment instructions.

Purists might consider **<-** to be a better choice, because it makes clearer the action that is being performed in an assignment operation. i.e. the expression on the right is being evaluated and the resulting value is assigned to the variable on the left.

We can also perform an assignment in the other direction

```
g^3  ->  h
```

However, as is common with many programming languages, when using **=** for assignment, the direction is unambiguously right to left.

And we should most definitely NOT interpret **=** as a creating a mathematical equation. In the context of coding, **=** is an assignment operation.

Thus far we have been typing commands one by one. This is fine for single line calculations, or if we don't need a permanent record of the commands. But usually we will have multi-command code, and we will want a permanent record of what we have done. R has several types of files. **From this point onwards avoid using the console; it is better to save work in an R script file, or an R Markdown file**

1.4 R script files

Create a new file using: **File > New File > R Script**

This should start a new empty file in the **Source** panel. In a **R Script** file we can type, edit, save and load multiple lines of **R** code. In a R Script file the code does not run immediately, since this is just an editor.

A **R script** file is designed for containing R code. It can have text as *code comments*.

To run code we can either press the **Run** button 

, or press **Ctrl-Enter** on the keyboard, and we can:

- Run code one line at a time by Clicking on an individual line,
- or run multiple lines at once by highlighting multiple lines with the cursor

Type the following lines into a **R script** and then run them

```
n = 1:10

p = n^2

# create a plot

plot(n,p)
```

Notice that variables n and p will be created, and a plot should appear. The comment line is purely for information – it is good practice to comment your code with an explanation of what it does if you plan to use it in the future or share it with colleagues.

To **save a file** go to **File > Save** or press the save icon  . If the file already has a name then it will just save, but if saving for the first time it will ask for a filename.

1.5 Types of variables in R

Like any programming language, **R** can store a variety of different types of data. Basic variables types include integer, numeric, logical, character amongst others.

```
i = 5             # numeric integer

z = 12.56         # numeric

name = "David"    # character type stores text

choice = TRUE     # logical type stores only TRUE or FALSE
```

The `class()` function can be used to determine the type of any given variable, e.g. `class(z)`

R can perform arithmetic on types that are compatible

```
i + z             # makes sense since both are numeric

i + choice        # gives a result (R assumes TRUE=1 and FALSE=0)

i + name          # fails since R can't sensibly combine them
```

Where is makes sense, R can **coerce** variables into another type using **as.***type* functions:

```
I = as.character(i)          # I is now the character "5"
```

A non-obvious data type that's important in data science is a **factor**. This is a way of storing categorical data in an efficient manner (assuming a list has lots of values that are the same).

e.g.
```
nation = c("UK","Spain","UK","Germany","Spain","UK","UK","Spain")
class(nation)
Nation = as.factor(nation)
class(Nation)
```

To see some benefits to this data type, run the following

```
summary(nation)
summary(Nation)
levels(Nation)
unclass(Nation)        # to see how the factor is encoded
```

1.6 The Working Directory

The working directory is the file location that R is *currently pointed at*. So R Studio will save a file, or try to load a file, or read a data file, from the **working directory**.

To determine the current working directory type

```
getwd()
```

To check the files in the current directory type

```
dir()
```

The School PC's are set up such that the default working directory should be the base folder of your own H: drive. This is fine for today, but we can change it if we want to organise your files.

To set your current working directory to match the file location we can use

Session > Set Working Directory > To Source File Location

To set your current working directory within the code use setwd(). e.g., to work on your H: drive in a folder called CMM535

```
setwd("H:/CMM535")
```

VERY IMPORTANT NOTES – SAVING FILES ON SCHOOL MACHINES

- always save files direct to your H: drive which is your own personal network space that can be accessed from any school machine and remotely
- do not save anything on the C: drive, which is the local drive of the PC)
- And do not save using windows shorcut **Windows created folders** such as **Documents**, **Desktop** or **Downloads**. Whilst ok to do so on your own personal machine at home, these folders can cause big problems when used on a network such as the University network. Some software, such as R Studio, does not work well with the network path names that lie behind these predefined User folders. Saving or accessing R files from these folders can freeze R Studio – so avoid them.

1.7 R Markdown files

Create a new file using **File > New File > R Markdown**

It will give a pop-up. Change the Title. Leave the Default Output Format as HTML.

This should start a new *markdown* file in the **Source** panel.

A **R Markdown** file is designed for creating reproducible reports that embed R code and R results in a document. It can have text as *markdown text*. In a **R Markdown** file we can type, edit, save and load multiple lines of **R** code, but also include, sections, headers, formatted text and comments to create a nicely formatted report.

In a R Markdown file the code does not run immediately, again since this is just an editor.

In the new file is some sample code. Before proceeding save the file, and then press the **Knit** button 

. This should cause some processing to appear in the Console, and then a separate window to open up containing the **knitted** document. You can see that text appears as text, and R code and output is embedded in the text.

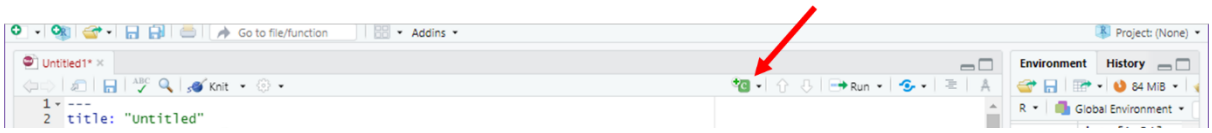
We shall return in later worksheets to learn a lot more about R Markdown files and how to edit and format them. But for now, delete the sample content, i.e. delete everything **except the header content** shown to the right.

```
---
title: "Untitled"
author: "David"
date: "2022-09-19"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

Take care to retain the ``` under the knitr command. This is important syntax that starts and ends *code chunks*.

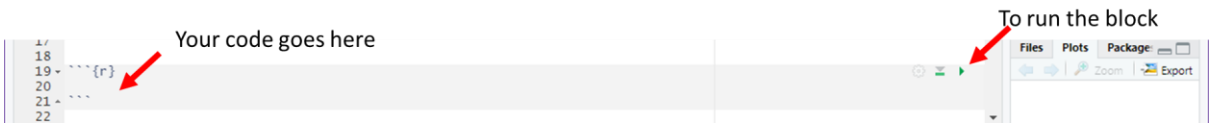
Start a code chunk by clicking on the small green box which appears at the top of the window and select "R".



This will insert a R chunk.

Enter the following code into the code block.

```
n = 1:10
p = n^2
# create a plot
plot(n,p)
```



Then run the block.

Text can be added. Text must go outside of code blocks. Code must go inside code blocks

It is good practice to create code blocks for sections of related code. We don't need a separate code block for every line, and don't use just a single code block for long lengthy code.

Each code block can be run individually to see the output and to help with debugging.

For the remaining exercises in this lab sheet you can choose to work with a simple R script, or with a R Markdown file, whichever you prefer. But over the next few labs, gradually get familiar with use of R Markdown files, as you will need t use them in the assessment for this module.

1.8 Vectors of values in R

R allows us to define, store and manipulate ordered sets of values. The resulting object is called either a **vector**, or a **list.**

Technically speaking these are different types of things: a **vector** can only contain items that are the same type as each other; whereas a **list** can contain items that are of different types, can have multiple columns/rows. But in many situations we will use the two words interchangeably (since **R** converts vectors to lists when it needs to).

The individual values in the vector/list are called its **elements**. To create a **list** we can use the ***combines*** function c()

```
n = c(0,1,2,3,4,5)
```

To display the content of a list, just enter it's variable name

```
n
```

An alternative is to create a **vector** using the **range** notation

```
N = c(1:5)
```

Calculations can be performed on the vector. This is one reason that **R** is great for stats and machine learning – operations can be applied to entire lists, rather than having to use loops to repeat operation as we might in other programming languages.

Whatever operation we apply to the vector is applied element by element. Calculate the following, and observe the results:

```
4*n
n^2
sqrt(n)
```

We can access individual elements from within a vector by specifying the **index** or position at which the value is located:

```
n[2]
```

We can access sub-vectors from within a vector by specifying the range of **indices** at which the values are located:

```
n[2:4]
```

We can ask the vector how many elements it contains:

```
length(n)
```

Rather than type every element individually, it is often useful to create a vector using a **sequence** from a specified initial and final value

```
m = seq(1,20)
```

If we want a sequence of numbers with a spacing other than 1, we can use the seq() function with a third parameter that specifies the spacing between values

```
t = seq(0.0, 20.0, 0.1)
```

We can define new vectors in terms of other variables

```
r = 4.0 * t - 5
```

Vectors are not restricted to numerical values

```
power = c("oil", "gas", "wood", "sun", "wind", "wave", "hydro", "nuclear")
```

Lists of the same length can be combined together

```
a = c(1:6)
b = a^2
Table1 = rbind(a, b)          # rbind combines lists as rows of a matrix
Table2 = cbind(a,b)           # cbind combines lists as columns of a matrix
```

1.9 Logical operators

R can make choices and comparisons using **logical operations**

Logical operators available in **R** include

<	>	for less than/greater than
<=	>=	for less/greater than or equal
==		test for equality (is equal to)
!=		test for inequality (is different to)
&		element by element AND operator
|		element by element OR operator
!		element by element NOT operator

For example:

```
numbers = c(1,2,3,4,5,6,7,8,9,10)

(numbers > 7)          # to identify numbers are greater than 7

(numbers > 7) | (numbers< 2)

(numbers%%2 == 0)      # to identify the even numbers
```

1.10 Simple Plots using R

For any pair of vectors with the same length, we can plot a 2D graph of one variable against the other.

e.g. Let's plot some points on a straight line:

```
x = seq(0.0, 20.0, 0.1)

y = 2 * x – 5

plot(x, y)
```

We can change the type of point by setting the(point character) **pch** argument

```
plot(x, y, pch = '*')
```

There are lots of other arguments that can be used to manipulate and enhance the plot

```
plot(x, y, pch = '*', col = "red")
```

Note that by default a plot is presented as discrete dots. But we can ask R to join the dots …

The plots created by are pretty basic. We shall see in later worksheets that more professional looking and flexible plots and visualisations can be created using libraries, such as **ggplot2.**

1.11 Using the Help facility in RStudio

The **Help** tab in RStudio allows you to access help information and language definition.

Once on the Help Tab enter the phrase or R function name in the search box. RStudio will helpfully present an autocomplete list of things it can help with.

Search for help on the **plot** function.

Alternatively you can enter the name of a function after a `?` at the console

```
?plot

?c
```

This should bring up information from the **R Documentation** that explains how the plot and function works, and what argument settings are available to adapt and refine the appearance of the plot.

Consult the documentation, and then experiment with different settings for the **type** argument

```
plot(x, y, type="p")
```

Then try replacing `type="p"` by `type="l", type="b", type="o"` and observe the effect of each.

1.12 Data input from (csv) file – data.frame

Download the file *consumption.csv* from CampusMoodle and save it on your H: drive.

Inspect this simple dataset in Excel.

Then, on the console window, type the following

```
data = read.csv("consumption.csv", header=T, stringsAsFactors=T)
```

This assumes that the file has been saved in the **working directory** that R Studio is pointing at.

If you get an error message such as

cannot open file 'consumption.csv': No such file or directory

then the file is not in the working directory. You can either move it to the working directory, or set the working directory to the file's location, or you can specify the full path to the file in the read.csv function.

```
data = read.csv("path/consumption.csv", header=T, stringsAsFactors=T)
```

where **path/** should be replaced by the path to the directory where you have downloaded the file e.g. **H:/cmm535/** , or if on your own machine at home e.g. **C:/cmm535/lab1/** , or whatever location you have used.

Setting the file locations is something you'll have to do in every single lab, so make sure you understand how to do so!

This dataset happens to be in comma separated format (csv), hence we used the **read.csv** function. Similar functions exist for other file types. The **read.csv** function reads the specified file and assigns it into whatever variable name we have specified, **data** in the case above.

This loaded data is stored in a type of object called a **data.frame**.

You will see that **data** appears on the Environment tab. If you click on it, a tabular representation of the data will appear on the top left panel.

Data.frame objects will be important in this and other modules. They are natural ways of storing **structured datasets** that consist of columns (representing data features or variables) and rows that containing the observations or instances.

We'll do more with data.frames in later modules and in CMM510, but just to experiment a little, ytry the following commands to see what happens:

```
summary(data)

mean(data$Gas)

plot(data$Year, data$Renewables)
```

Optional activity for private study

1.13 R's inbuilt tutorial system: The swirl Package

R has a set of in-built self-paced tutorials which you can investigate to learn about many the features and functions of the R environment and language. To access this, you will need package called ***swirl***.

If it is in the list of packages (packages tab, bottom right pane), simply select it so that R loads it. If it is not in the list of packages, then follow the instructions* below to install it, and then to load it check the checkbox beside it when it appears on the list of packages.

swirl allows you to learn R from within R with interactive tutorials.

Once it is loaded, using `library(swirl)`, then to start the tutorial, in the console type

```
swirl()
```

Once you enter your name, enter **1** to proceed, and then enter **1** to show the tutorial options on R Programming. You will be presented with several alternatives that you can choose from. Tutorials 1-4 would be useful to start with.

If you want to come out of the tutorial temporarily, you can type `play()`

To resume where you left type `ntx()`

To leave swirl completely type `bye()`

For more information search the Help facility for *swirl*.

*Downloading and installing packages

The R system has lots of additional packages of functionality. For accessing functionality that is not bundled with the core of the R language we may have to load one or more additional package. **The ones we need will generally be installed on the School machines already, but you may need to do this on your own PC.**

To install a package:

1. Select the *packages* tab on the bottom right pane and then select *install packages*. A new window will appear.
2. Type the name of the package which you would like installed in the *"Packages"* section and click on *Install*. Ensure the *"Install dependencies"* option is selected as it may save you lots of package installations.

Or if you know the name of the package, then use `install.packages` function with the package name in brackets, e.g. to install `swirl` use `install.packages("swirl")`

Installing a package creates a copy of the package files on your computer. To use it in the current **R** session you still have to load it into the current *R workspace* you still have to use `library(swirl)`