

Biological Computing in R: Silwood Masters (CMEE, EEC, EA, NHM)

Imperial College London, 2014–15

Samraat Pawar (s.pawar@imperial.ac.uk),
(with inputs from many others!)

November 2, 2014

Contents

0	Introduction	1
0.1	What is this document all about?	1
0.2	Outline of the this module	1
0.3	Conventions used in this document	2
0.4	A note on being organized	2
0.5	Readings	2
1	Introduction to R	3
1.1	What is R?	3
1.2	Would you ever need anything other than R?	3
1.3	Installing R	3
1.4	Getting started	4
1.5	Useful R commands	5
1.6	Some Basics	5
1.6.1	Variable names and Tabbing	5
1.6.2	R likes E Notation	6
1.6.3	Operators	6
1.7	Sequences	6
1.8	Strings and Pasting	7
1.9	Indices and Indexing	7
1.10	Basic vector-matrix operations	8
1.11	Data types	8
1.11.1	Vectors	8
1.11.2	Matrices and arrays	9
1.11.3	Data frames	10
1.11.4	Lists	11
1.12	The Working Directory	11
1.13	The R analysis workflow	12
1.14	Importing and Exporting Data	13
1.15	Control statements	13
1.16	Running R code	14
1.17	Type Conversion and Special Values	14
1.18	Writing Functions	15
1.19	Practical 1	15
1.20	Vectorize it!	16
1.21	Useful R Functions	17
1.22	Packages	18
1.23	Launching R in batch mode	18
1.24	Readings	18

Chapter 0

Introduction

Donald Knuth, 1995: *Science is what we understand well enough to explain to a computer. Art is everything else we do*

0.1 What is this document all about?

This document contains the content of the Imperial College London Department of Life Sciences Msc/MRes Modules on Scientific Computing in R statistics modules, workshops and practicals. The content of this document will be covered over the week. This document is accompanied by data and code on which you can practice your skills in your own time and during the Practical sessions.

This document does not tell you every single thing you need to know to perform the exercises in it. In programming, you learn far more from trying to solve problems than from reading about how others have solved them. It is important that you work through the exercises and problems in each chapter, particularly as some of the questions ask you to find out about functions not introduced in the chapter's text itself, but which will be relied on in later chapters.

0.2 Outline of the this module

The content and structure of this week is geared towards the following objectives:

- Give you a introduction to R syntax and programming conventions assuming you have never set your eyes on R or any other programming language before — we will breeze through this as you have already been introduced to R in the Stats Week!
- Teach you principles of clean and efficient programming in R, including delightful things like vectorization and debugging.
- Teach you how to generate publication quality graphics in R — publication quality is thesis quality!
- Teach you how to develop efficient and reproducible data analyses “work flows” so (or anybody else) run and re-run your analyses, graphics outputs and all, in R

You will use R a lot during the rest of the course and probably your thesis and career — the aim is to lay down the foundations for you to become very comfortable with it!

0.3 Conventions used in this document

You will find all R commandline/console arguments, code snippets and output in colored boxes like this:

```
> ls()
```

Here `>` is the R prompt, and will type the commands/code that you see from this document into the R command line (or copy-paste, but not recommended!). I have aimed to make the content of this module computer platform (Mac, PC or Linux) independent because many of you are probably working (or later will be) with R on personal laptops or desktops. Indeed, platform-independence of data analyses is one of the main reasons why you are using R! Finally, note that:

★ In all subsequent chapters, lines marked with a star like this are things for you to do.

0.4 A note on being organized

In this module, you will write plenty of R code and deal with different data files. Please keep all your code, data inputs and results outputs organized in separate directories named `Code`, `Data`, `Results` (or equivalent) respectively. The CMEEs are already implementing this along with version control in git (if you are intrigued, please look up CMEE Week 1 lectures).

Note that R, as in UNIX like systems (Mac, Linux), uses `/` instead of the `\` used in Windows for directory path specification. Also, in general, we will be using relative paths throughout the exercises and practicals (more on this later).

0.5 Readings

(More will appear in different sections/chapters)

- Use the internet! Google “R tutorial”, and plenty will pop up. Choose one that seems the most intuitive to you.
- Bolker, B. M.: Ecological Models and Data in R (eBook and Hardcover available).
- Beckerman, A. P. & Petchey, O. L. (2012) Getting started with R: an introduction for biologists. Oxford, Oxford University Press.
Good, short, general introduction
- Crawley, R. (2013) The R book. 2nd edition. Chichester, Wiley.
Excellent but enormous reference book, code and data available from www.bio.ic.ac.uk/research/mjcraw/therbook/index.htm

Chapter 1

Introduction to R

1.1 What is R?

R is a freely available statistical software with strong programming capabilities. R has become incredibly popular in biology due to several factors: i) many packages are available to perform all sorts of statistical and mathematical analyses; ii) it has been developed and scrutinised by top level academic statisticians; iii) it can produce beautiful, publication-quality graphics; iv) it has a very good support for matrix-algebra.

1.2 Would you ever need anything other than R?

Although we can technically program in R, the programming environment is not the greatest: especially the way types are managed is problematic (e.g., often your matrix will become a vector if it has only one column/row!), and the way errors and warnings are handled and displayed (often unintelligibly!).

Nevertheless, being able to program R means you can develop and automate your statistical analyses and the generation of figures into a reproducible work flow (there's that that again!). However, if your work also includes extensive numerical simulations, manipulation of very large matrices, bioinformatics, or complex workflows including databases, you will be much better off if you also know another programming language that is more versatile, computationally efficient (like python, PERL or C)). But for most of you, R will do the job, so you may revel in that knowledge! In particular, Python is recommended, as it is reasonably efficient, and can embed your R analysis work flow inside a more complex meta-workflow, for example, one that includes interfacing with the internet, manipulating/querying databases, and compiling a L^AT_EX document.

1.3 Installing R

Linux/Ubuntu: run the following in terminal

```
$ sudo apt-get install r-base r-base-dev
```

Mac OS X: download and install from

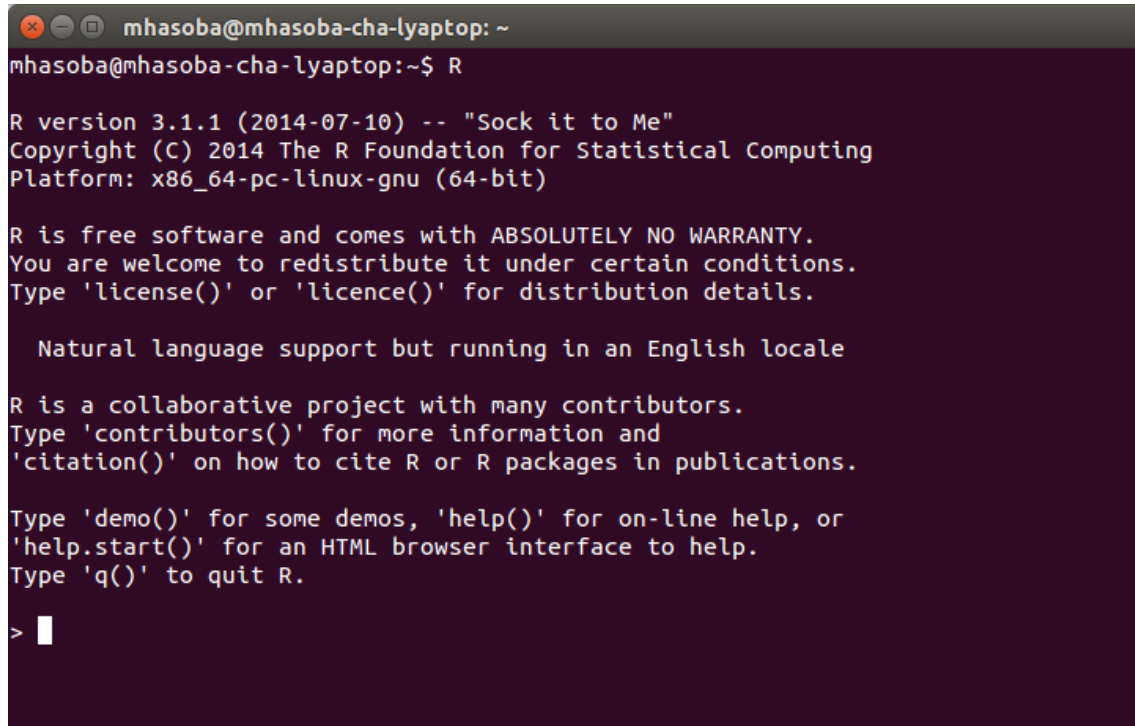
<http://cran.r-project.org/bin/macosx/>

Windows: download and install from

<http://cran.r-project.org/bin/windows/base/>

1.4 Getting started

Launch R (From Applications menu on Window or Mac, from terminal in Linux/Ubuntu) — it should look something like this (on Linux/Ubuntu or Mac terminal):

A terminal window with a dark purple background and light green text. The window title is 'mhasoba@mhasoba-cha-lyaptop: ~'. The prompt is 'mhasoba@mhasoba-cha-lyaptop:~\$'. The output shows R version 3.1.1 (2014-07-10) with the slogan 'Sock it to Me'. It includes copyright information for The R Foundation for Statistical Computing and the platform 'x86_64-pc-linux-gnu (64-bit)'. It states that R is free software with absolutely no warranty and provides instructions on how to get help or quit. The prompt is now '>' with a white cursor.

```
mhasoba@mhasoba-cha-lyaptop: ~
mhasoba@mhasoba-cha-lyaptop:~$ R

R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

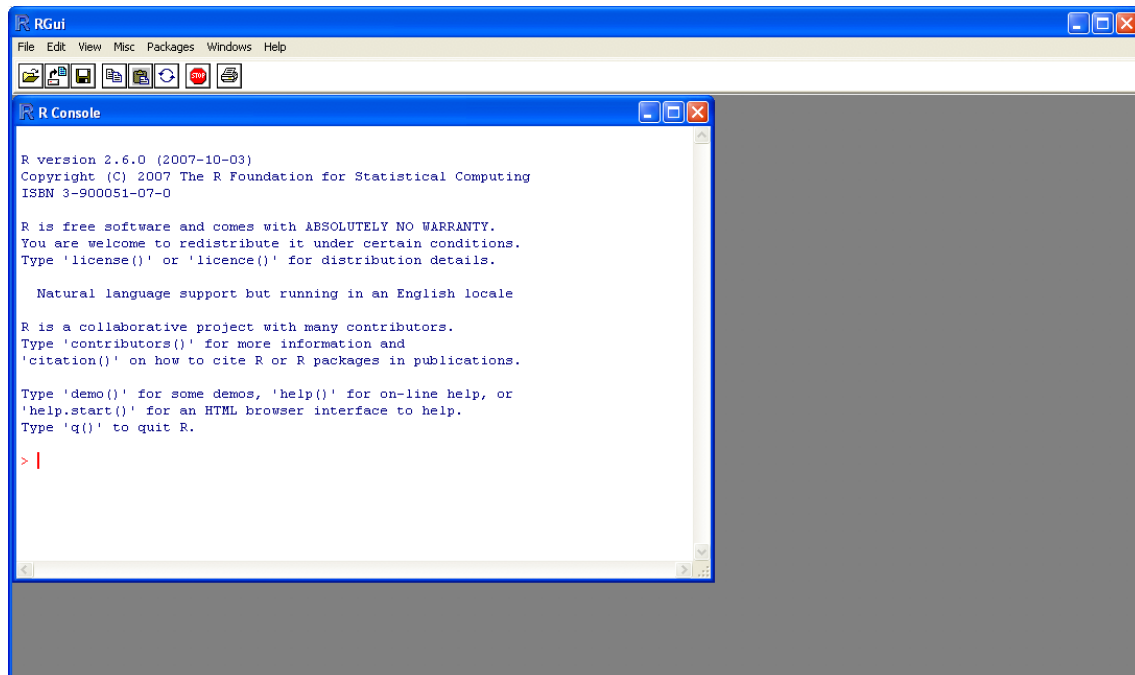
Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Or like this (Windows “console”, similar in Mac):

A screenshot of the RGui application window. The window has a blue title bar and a menu bar with 'File', 'Edit', 'View', 'Misc', 'Packages', 'Windows', and 'Help'. Below the menu bar is a toolbar with icons for file operations and help. The main area is titled 'R Console' and contains the same startup text as the terminal window, but with a red prompt character '>' and a red cursor. The background of the console area is white.

```
RGui
File Edit View Misc Packages Windows Help

R Console

R version 2.6.0 (2007-10-03)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

You can also use an IDE (Interactive Development Environment) that can offer delights like syntax highlighting (google it!), such as RStudio, geany, vim, etc.

1.5 Useful R commands

<code>ls()</code>	list all the variables in the work space
<code>rm('a', 'b')</code>	remove variable(s) a and b
<code>rm(list=ls())</code>	remove all variable(s)
<code>getwd()</code>	get current working directory
<code>setwd('Path')</code>	set working directory to Path
<code>q()</code>	quit R
<code>?Command</code>	show the documentation of Command
<code>??Keyword</code>	search the all packages/functions with Keyword, “fuzzy search”

1.6 Some Basics

Try out the following in the R console:

```
> a <- 4 # assignment
> a
[1] 4
> a*a # product
[1] 16
> a_squared <- a*a
> sqrt(a_squared) # square root
[1] 4
> v <- c(0, 1, 2, 3, 4) # c: "concatenate"
```

`c()` (concatenate) is one of the most commonly used functions — Don't forget it! (try `?c`)

Note that any text after a “#” is ignored by R — handy for commenting. In general, please comment your code and scripts, for *everybody's* sake!

```
> v # Display the vector variable you created
[1] 0 1 2 3 4
> is.vector(v) # check if it's a vector
[1] TRUE
> mean(v) # mean
[1] 2
> var(v) # variance
[1] 2.5
> median(v) # median
[1] 2
> sum(v) # sum all elements
[1] 10
> prod(v + 1) # multiply
[1] 120
> length(v) # length of vector
[1] 5
```

1.6.1 Variable names and Tabbing

```
> wing.width.cm <- 1.2 #Using dot notation
> wing.length.cm <- c(4.7, 5.2, 4.8)
```

Using tabbing with dot notation can be handy. Type:

```
> wing.
```

And then hit the `tab` key. This is handy, but good style and readability is more important than just convenient variable names. Variable names should be as obvious as possible, not over-long!

1.6.2 R likes E Notation

```
> 1E4
[1] 10000
> 1e4
[1] 10000
> 5e-2
[1] 0.05
```

R uses *E* notation to print very large or small numbers:

```
> 1E4 ^ 2
[1] 1e+08
> 1 / 3 / 1e8
[1] 3.333333e-09
```

1.6.3 Operators

The usual operators are available in R (slight differences from `python`):

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power
%%	Modulo
%/ %	Integer division
==	Equals
!=	Differs
>	Greater
>=	Greater or equal
&	Logical and
	Logical or
!	Logical not

1.7 Sequences

The `:` operator creates vectors of sequential integers:

```
> years <- 1990:2009
> years
[1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
[11] 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

> years <- 2009:1990 # or in reverse order
> years
[1] 2009 2008 2007 2006 2005 2004 2003 2002 2001 2000
[11] 1999 1998 1997 1996 1995 1994 1993 1992 1991 1990
```

For sequences of fractional numbers, you have to use `seq()` :

```
> seq(1, 10, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
[16] 8.5 9.0 9.5 10.0
```

You can also `seq(from=1,to=10, by=0.5)` OR `seq(from=1, by=0.5, to=10)` with the same effect (try it) — this explicit, “argument matching” approach is partly why R is so popular.

1.8 Strings and Pasting

R’s string handling ain’t elegant or pretty (unlike python!), but it works:

```
> species.name <- "Quercus robur" #double quotes
> species.name
[1] "Quercus robur"
> species.name <- 'Fraxinus excelsior' #single quotes
> species.name
[1] "Fraxinus excelsior"
> paste("Quercus", "robur")
[1] "Quercus robur"
> paste("Quercus", "robur", sep = "") #Get rid of space
"Quercusrobur"
> paste("Quercus", "robur", sep = ", ") #insert comma to separate
```

And as is the case with so many R functions, pasting works on vectors:

```
> paste('Year is:', 1990:2000)
[1] "Year is: 1990" "Year is: 1991" "Year is: 1992" "Year is: 1993"
[5] "Year is: 1994" "Year is: 1995" "Year is: 1996" "Year is: 1997"
[9] "Year is: 1998" "Year is: 1999" "Year is: 2000"
```

Note that this last example creates a vector of 11 strings.

1.9 Indices and Indexing

Every element of a vector in R has an order: the first value, second, third, etc. To illustrate this, type:

```
> MyVar <- c( 'a' , 'b' , 'c' , 'd' , 'e' ) # create a simple vector
```

Then, square brackets extract values based on their position in the order:

```
> MyVar[1] # Show element in first position
[1] "a"
> MyVar[4]
[1] "d" # Show element in fourth position
```

The values in square brackets are called “indices” — they give the index (position) of the required value. We can also select sets of values in different orders, or repeat values:

```
> MyVar[c(3,2,1)] # reverse order
[1] "c" "b" "a"
MyVar[c(1,1,5,5)] # repeat indices
[1] "a" "a" "e" "e"
```

So you can manipulate vectors by indexing:

```
> v <- c(0, 1, 2, 3, 4) # Re-create the vector variable v
> v[3] # access one element
[1] 2
> v[1:3] # access sequential elements
[1] 0 1 2
> v[-3] # remove elements
[1] 0 1 3 4
> v[c(1, 4)] # access non-sequential
[1] 0 3
```

1.10 Basic vector-matrix operations

```
> v2 <- v
> v2 <- v2*2 # whole-vector operation
> v2
[1] 0 2 4 6 8
> v * v2 # product element-wise
[1] 0 2 8 18 32
> t(v) # transpose the vector
[,1] [,2] [,3] [,4] [,5]
[1,] 0 1 2 3 4
> v %*% t(v) # matrix/vector product
[,1] [,2] [,3] [,4] [,5]
[1,] 0 0 0 0 0
[2,] 0 1 2 3 4
[3,] 0 2 4 6 8
[4,] 0 3 6 9 12
[5,] 0 4 8 12 16
> v3 <- 1:7 # assign using sequence
> v3
[1] 1 2 3 4 5 6 7
> v4 <- c(v2, v3) # concatenate vectors
> v4
[1] 0 2 4 6 8 1 2 3 4 5 6 7
> q() # quit
```

1.11 Data types

Like python (why python? Ask the CMEEs!), R comes with data-types. Mastering these will help you write better, more efficient programs and also handle diverse between datasets. Now get back into R (if you quit R using `q()`), and type:

1.11.1 Vectors

```
> a <- 5
> is.vector(a)
[1] TRUE
```

```
> v1 <- c(0.02, 0.5, 1)
> v2 <- c("a", "bc", "def", "ghij")
> v3 <- c(TRUE, TRUE, FALSE)
```

1.11.2 Matrices and arrays

R has many functions to manipulate matrices (two-dimensional vectors) and arrays (multi-dimensional vectors).

```
> m1 <- matrix(1:25, 5, 5)
> m1
  [,1] [,2] [,3] [,4] [,5]
[1,]   1   6  11  16  21
[2,]   2   7  12  17  22
[3,]   3   8  13  18  23
[4,]   4   9  14  19  24
[5,]   5  10  15  20  25
> m1 <- matrix(1:25, 5, 5, byrow=TRUE)
> m1
  [,1] [,2] [,3] [,4] [,5]
[1,]   1   2   3   4   5
[2,]   6   7   8   9  10
[3,]  11  12  13  14  15
[4,]  16  17  18  19  20
[5,]  21  22  23  24  25
> dim(m1)
[1] 5 5
> m1[1,2]
[1] 2
> m1[1,2:4]
[1] 2 3 4
> m1[1:2,2:4]
  [,1] [,2] [,3]
[1,]   2   3   4
[2,]   7   8   9
> arr1 <- array(1:50, c(5, 5, 2))
> arr1
, , 1

  [,1] [,2] [,3] [,4] [,5]
[1,]   1   6  11  16  21
[2,]   2   7  12  17  22
[3,]   3   8  13  18  23
[4,]   4   9  14  19  24
[5,]   5  10  15  20  25

, , 2

  [,1] [,2] [,3] [,4] [,5]
[1,]  26  31  36  41  46
[2,]  27  32  37  42  47
[3,]  28  33  38  43  48
[4,]  29  34  39  44  49
[5,]  30  35  40  45  50
```

1.11.3 Data frames

This is a very important data type that is peculiar to R. It is great for storing your data. Basically, it's a two-dimensional table in which each column can contain a different data type (e.g., numbers, strings, boolean). You can think of a dataframe as a spreadsheet. Data frames are great for plotting your data, performing regressions and such. Later (Chapter 2) you will see that fancy plotting using `ggplot` demands the use of dataframes. Now try the following:

```
> Col1 <- 1:10
> Col1
[1] 1 2 3 4 5 6 7 8 9 10
> Col2 <- LETTERS[1:10]
> Col2
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
> Col3 <- runif(10) # 10 random num. from uniform
> Col3
[1] 0.29109 0.91495 0.64962 0.95503 0.26589 0.02482 0.59718
[8] 0.99134 0.98786 0.86168
> MyDF <- data.frame(Col1, Col2, Col3)
> MyDF
  Col1 Col2      Col3
1     1    A 0.2910981
2     2    B 0.9149558
3     3    C 0.6496248
4     4    D 0.9550331
5     5    E 0.2658936
6     6    F 0.0248217
7     7    G 0.5971868
8     8    H 0.9913407
9     9    I 0.9878679
10    10    J 0.8616854
> names(MyDF) <- c("A.name", "another", "another.one")
> MyDF
  A.name another another.one
1      1      A    0.2910981
2      2      B    0.9149558
3      3      C    0.6496248
4      4      D    0.9550331
5      5      E    0.2658936
6      6      F    0.0248217
7      7      G    0.5971868
8      8      H    0.9913407
9      9      I    0.9878679
10     10      J    0.8616854
> MyDF$A.name
[1] 1 2 3 4 5 6 7 8 9 10
> MyDF[,1]
[1] 1 2 3 4 5 6 7 8 9 10
> MyDF[c("A.name", "another")]
  A.name another
1      1      A
2      2      B
3      3      C
4      4      D
5      5      E
6      6      F
7      7      G
8      8      H
9      9      I
10     10      J
```

```
> class(MyDF)
[1] "data.frame"
> str(MyDF) # a very useful command!
'data.frame': 10 obs. of 3 variables:
 $ A.name      : int  1 2 3 4 5 6 7 8 9 10
 $ another     : Factor w/ 10 levels "A","B","C","D",...
 $ another.one : num  0.291 0.915 0.65 0.955 0.266 ...
```

1.11.4 Lists

A list is simply an ordered collection of objects (that can be other variables) (a bit like python lists!). You can build lists of lists too.

```
> l1 <- list(names=c("Fred","Bob"), ages=c(42, 77, 13, 91))
> l1
$names
[1] "Fred" "Bob"

$ages
[1] 42 77 13 91

> l1[[1]]
[1] "Fred" "Bob"
> l1[[2]]
[1] 42 77 13 91
> l1[["ages"]]
[1] 42 77 13 91
> l1$ages
[1] 42 77 13 91
```

1.12 The Working Directory

Before we go any further, let's get ourselves organized. In R, type:

```
> getwd()
```

This tells you what the current “working directory” is.

In using R for an analysis, you will likely use and create several files. This means that it is sensible to create a folder (directory) to keep all code files together. You can then set R to work from this directory, so that files are easy to find and run — this will be your working directory. So now do the following:

- ★ Create a directory called `Week5` in an appropriate location (For CMEEs, in `CMEECourseWork` for others, some place you can remember!
- ★ Create subdirectories within `CMEECourseWork/Week5` called `Code`, `Data`, and `Results`

You can create directories using `dir.create()` within R

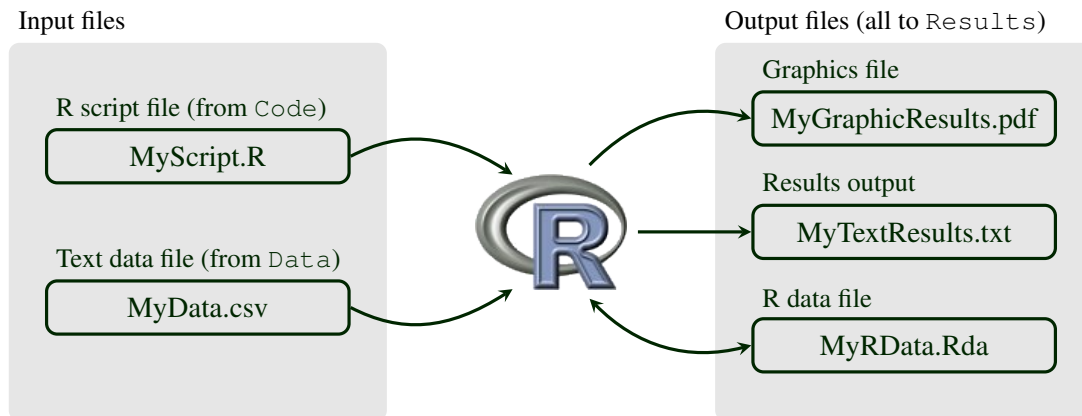
Use relative paths Using relative paths in your R scripts and code will make your code computer independent and your life better! For example, in R, `../Data/mydata.txt` specifies a file named `mydata.txt` located in the “parent” of the current directory. Now, set the working directory to be `Week5/Code`:

```
> setwd("FullPathUpToHere/Week5/Code")
> dir()
```

Note that `FullPathUptoHere` is not to be taken literally and entered! For example, if you created `Week5` in `H:`, you would use `setwd("\\H:/MyICStatsModules/Code")`. For CMEE, it would be `setwd("\\FullPathUptoHere/CMEECourseWork/Week5/Code")`.

1.13 The R analysis workflow

Your typical R analysis workflow will be as follows:



Some details on each kind of file:

R script files These are plain text files containing all the R code needed for an analysis. These should always be created with a simple text editor like Notepad (Windows), TextEdit (MacOS) or Geany (Linux) and saved with the extension `*.R`. We will use the built-in editor in R in this class. Alternatively, if RStudio is a good option because it also works across platforms. Try it. A big advantage of something like RStudio is that you will get syntax highlighting, which is very handy and will make R programming far more convenient and error-free. You should *never* use Word to save or edit these files as R can only read code from plain text files.

Text data files These are files of data in plain text format containing one or more columns of data (numbers, strings, or both). Although there are several format options, we will typically be using `csv` files, where the entries are separated by commas. These are easy to create and export from Excel (if that's what you use...)¹.

Results output files These are plain text files that contain your results, such as the summary of output of a regression or ANOVA analysis. Typically, you will put your results in a table format where the columns are separated by commas (`csv`) or tabs (tab-delimited).

Graphics files R can export graphics in a wide range of formats. This can be done automatically from R code and we will look at this later but you can also select a graphics window and click 'File > Save as...'

Rdata files You can save any data loaded or created in R, including model outputs and other things, into a single `Rdata` file. These are not plain text and can only be read by R, but can hold all the data from an analysis in a single handy location. I never use these, but you can, if you want.

¹If you are using a computer from elsewhere in the EU, Excel may use a comma ($\pi = 3,1416$) instead of a decimal point ($\pi = 3.1416$). In this case, `csv` files may use a semi-colon to separate columns and you can use the alternative function `read.csv2()` to read them into R.

1.14 Importing and Exporting Data

Now we are ready to see how to import and export data in R, typically the first step of your analysis. The best option is to have your data in a comma separated value text file or in a tab separated file. Then, you can use the function `read.csv` (or `read.table`) to import your data (note the relative paths!):

```
> MyData <- read.csv("../Data/trees.csv")
> head(MyData) # Have a quick look at the data frame
> str(MyData) # Have a quick look at the column types
> MyData <- read.csv("../Data/trees.csv", header = TRUE) # with headers
> MyData <- read.table("../Data/trees.csv", sep = ',',
header = TRUE) # A more general way
> head(MyData)
> MyData <- read.csv("../Data/trees.csv", skip = 5) # skip first 5 lines
```

Note that the resulting `MyData` in your workspace is a R dataframe. You can also save your data frames using `write.table` or `write.csv`:

```
> write.csv(MyData, "../Results/MyData.csv")
> dir("../Results/") # Check if it worked
> write.table(MyData[1,], file = "../Results/MyData.csv", append=TRUE) # append
> write.csv(MyData, "../Results/MyData.csv", row.names=TRUE) # write row names
> write.table(MyData, "../Results/MyData.csv", col.names=FALSE) # ignore col names
```

1.15 Control statements

In R, you can write `if`, `then`, `else` statements, and `for` and `while` loops like any programming language. However, loops are slow in R, so use them sparingly.

★ Type the following in a script file called `control.R` (save it in your Code directory)

```
1 ## If statement
  a <- TRUE
3 if (a == TRUE){
  print ("a is TRUE")
5 } else {
  print ("a is FALSE")
7 }

9 ## On a single line
  z = runif(1) ##random number
11 if (z <= 0.25) print ("Less than a quarter")

13 ## For loop using a sequence
  for (i in 1:100){
15     j <- i * i
    print(paste(i, " squared is", j ))
17 }

19 ## For loop over vector of strings
  for(species in c('Heliodoxa rubinoides',
21                  'Boissonneaua jardini',
                    'Sula nebouxii'))
23 {
  print(paste('The species is', species))
25 }
```

```

27 ## for loop using a vector
v1 <- c("a", "bc", "def")
29 for (element in v1){
    print(element)
31 }

33 ## While loop
i <- 0
35 while (i<100){
    i <- i+1
37    print(i^2)
}

```

1.16 Running R code

You can run the code you wrote in blocks to test and understand it:

- ★ Place the cursor on the first line of code and run it by pressing the keyboard shortcut (PC: ctrl+R, Mac: command+enter, Linux: ctrl+enter if you are using geany).

Of course, you write code so that you can just “run” it, which runs your full analysis and outputs all the results. The way to run *.R script/code from the command line is to `source` it. This causes R to accept code input from a named file and run it:

```
> source("control.R") # Assuming you are in Code directory!
```

Note that you will need to add the directory path to the file name (`control.R` in the above example), if the file is not in your working directory. For example, `../Code/control.R` if you are in, say, `Data`.

1.17 Type Conversion and Special Values

There are different kinds of data variable types such as integer, float (including real numbers), and string (e.g., text words). Beware of the difference between NA (Not Available) and NaN (Not a Number).

```

> as.integer(3.1)
[1] 3
> as.real(4)
[1] 4
> as.roman(155)
[1] CLV
> as.character(155)
[1] "155"
> as.logical(5)
[1] TRUE
> as.logical(0)
[1] FALSE
> b <- NA
> is.na(b)
[1] TRUE
> b <- 0./0.
> b
[1] NaN
> is.nan(b)

```

```
[1] TRUE
> b <- 5/0
> b
[1] Inf
> is.nan(b)
[1] FALSE
> is.infinite(b)
[1] TRUE
> is.finite(b)
[1] FALSE
> is.finite(0/0)
[1] FALSE
```

1.18 Writing Functions

R lets you write your own functions. The syntax is quite simple, with each function accepting arguments and returning a value:

```
MyFunction <- function(Arg1, Arg2){
2
  ## statements involving Arg1, Arg2
4
  return (ReturnValue)
6
}
```

★ Type the following in a script file called `TreeHeight.R`, save it in your Code directory and run it using `source`:

```
# This function calculates heights of trees
2 # from the angle of elevation and the distance
# from the base using the trigonometric formula
4 # height = distance * tan(radians)
#
6 # Arguments:
# degrees      The angle of elevation
8 # distance    The distance from base
#
10 # Output:
# The height of the tree, same units as "distance"
12
TreeHeight <- function(degrees, distance)
14 {
  radians <- degrees * pi / 180
16 height <- distance * tan(radians)
  print(paste("Tree height is:", height))
18 return (height)
20 }

TreeHeight(37, 40)
```

1.19 Practical 1

Modify the script `TreeHeight.R` so that it does the following:

- ★ Loads `trees.csv` and calculates tree heights for all trees in the data. Note that the distances have been measured in meters. (Hint: use relative paths))
- ★ Creates a csv output file called `TreeHts.csv` in `Results` that contains the calculated tree heights along with the original data in the following format (only first two rows and headers shown):

```
"Species", "Distance.m", "Angle.degrees", "Tree.Height.m"
"Populus tremula", 31.6658337740228, 41.2826361937914, 25.462680727681
"Quercus robur", 45.984992608428, 44.5359166583512, 46.094124200205
```

1.20 Vectorize it!

R is very slow at running cycles (`for` and `while` loops). This is because R is a “nimble” language: at execution time R does not know what you’re going to perform until it “reads” the code to perform. Compiled languages such as C, know exactly what the flow of the program is, as the code is compiled before execution. As a metaphor, C is a musician playing a score she has seen before – optimizing each passage, while R is playing it “a prima vista” (i.e., at first sight).

Hence, in R you should try to avoid loops like the plague. In practical terms, sometimes it is much easier to throw in a `for` loop, and then optimize the code to avoid the loop if the running time is not satisfactory. R has several functions that can operate on entire vectors and matrices.

- ★ For example, type (save in Code) in `Vectorize1.R` and run it (it sums all elements of a matrix):

```
1 M <- matrix(runif(1000000), 1000, 1000)

3 SumAllElements <- function(M) {
4   Dimensions <- dim(M)
5   Tot <- 0
6   for (i in 1:Dimensions[1]){
7     for (j in 1:Dimensions[2]){
8       Tot <- Tot + M[i,j]
9     }
10  }
11  return (Tot)
12 }

13 ## This on my computer takes about 1 sec
15 print(system.time(SumAllElements(M)))
## While this takes about 0.01 sec
17 print(system.time(sum(M)))
```

Both approaches are correct, and will give you the right answer. However, one is 100 times faster than the other! Fortunately, R offers several ways of avoiding loops. Here are the main ones:

```
1 ## apply:
2 # applying the same function to rows/columns of a matrix

3 ## Build a random matrix
5 M <- matrix(rnorm(100), 10, 10)
## Take the mean of each row
7 RowMeans <- apply(M, 1, mean)
8 print (RowMeans)
9 ## Now the variance
10 RowVars <- apply(M, 1, var)
11 print (RowVars)
12 ## By column
13 ColMeans <- apply(M, 2, mean)
14 print (ColMeans)
15 ## You can use it to define your own functions
```

```

17 ## (What does this function do?)
   SomeOperation <- function(v){
19   if (sum(v) > 0){
       return (v * 100)
21   }
       return (v)
23 }
print (apply(M, 1, SomeOperation))

25
## by:
27 ## apply the function to a dataframe, using some factor
## to define the subsets

29
## import some data
31 attach(iris)
print (iris)

33
## use colMeans (as it is better for dataframes)
35 by(iris[,1:2], iris$Species, colMeans)
by(iris[,1:2], iris$Petal.Width, colMeans)

37
## There are many other methods: lapply, sapply, eapply, etc.
39 ## Each is best for a given data type (lapply -> lists)

41 ## replicate:
## this is quite useful to avoid a loop for function that typically
43 ## involve random number generation
print(replicate(10, runif(5)))

```

1.21 Useful R Functions

Mathematical

log(x)	Natural logarithm
log10(x)	Logarithm in base 10
exp(x)	e^x
abs(x)	Absolute value
floor(x)	Largest integer $< x$
ceiling(x)	Smallest integer $> x$
pi	π
sqrt(x)	\sqrt{x}
sin(x)	Sinus function

Strings

strsplit(x, ' ; ')	Split the string according to ' ; '
nchar(x)	Number of characters
toupper(x)	Set to upper case
tolower(x)	Set to lower case
paste(x1, x2, sep=' ; ')	Join the strings inserting ' ; '

Statistical

mean(x)	Compute mean (of a vector or matrix)
sd(x)	Standard deviation
var(x)	Variance
median(x)	Median
quantile(x, 0.05)	Compute the 0.05 quantile
range(x)	Range of the data
min(x)	Minimum
max(x)	Maximum
sum(x)	Sum all elements

Random number distributions

<code>rnorm(10, m=0, sd=1)</code>	Draw 10 normal random numbers with mean 0 and s.d. 1
<code>dnorm(x, m=0, sd=1)</code>	Density function
<code>qnorm(x, m=0, sd=1)</code>	Cumulative density function
<code>runif(20, min=0, max=2)</code>	Twenty random numbers from uniform [0,2]
<code>rpois(20, lambda=10)</code>	Twenty random numbers from Poisson(λ)

1.22 Packages

The main strength of R is that users can easily build packages and share them through `cran.r-project.org`. There are packages to do most statistical and mathematical analysis you might conceive, so check them out before reinventing the wheel! Visit `cran.r-project.org` and go to Packages to see a list and a brief description. To install a package, within R type `install.packages()` and choose the package to install.

1.23 Launching R in batch mode

Often, you may want to run the final analysis without opening R in interactive mode. In Mac or linux, you can do so by typing:

```
R CMD BATCH MyCode.R MyResults.Rout
```

This will create an `MyResults.Rout` file containing all the output. On Microsoft Windows, its more complicated — (change the path to `R.exe` and output file as needed:

```
\C:\Program Files\R\R-3.1.1\bin\R.exe" CMD BATCH --vanilla --slave \C:\PathToMyResults\Resul
```

1.24 Readings

- The Use R! series (the yellow books) by Springer are really good. In particular, consider: “A Beginner’s Guide to R”, “R by Example”, “Numerical Ecology With R”, “ggplot2” (we’ll see this in Chapter 2), “A Primer of Ecology with R”, “Nonlinear Regression with R”, “Analysis of Phylogenetics and Evolution with R”.
- For more focus on dynamical models: Soetaert & Herman. 2009 “A practical guide to ecological modelling: using R as a simulation platform”.
- There are excellent websites besides `cran`. In particular, check out `www.statmethods.net` and `http://en.wikibooks.org/wiki/R_Programming`.