Lab 1: Introduction to R

Instructions

Dr Mercedes Torres Torres

Contents

1 Introduction

'R' is a programming environment for data analysis and graphics, developed as an open source application. It is a powerful and flexible tool with a vast number of toolboxes. You can find copies of R on the windows machines in A32.

In this lab, we will cover how to install R and RStudio, our editor of choice for this module. We will also cover different sources of information about R that can help you in your course, as well as how to set up the RStudio environment so we can start programming.

In these first few labs, you will enter sometimes unfamiliar/obscure R syntax into the command window. In order to get the best out of the lab sessions, you should try to understand how they work and what they do before moving on.

For a comprehensive reference manual on all things R, please refer to *An Introduction to R* by Venables et al., which can be found in our Moodle page and here: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

I recommend that you print it and have it with you while you are working on the exercises.

Thorough the labs, we will be using *An Introduction to R* as a reference manual. Chapters or sections of interest will be signalled with *Reading:* in the Instruction sheets.

After this lab session, you will be able to use R to:

- Set up R scripts
- Set up R Markdown notebooks
- Use the R help function
- Perform simple arithmetic operations
- Perform simple vector calculations
- Perform simple matrix calculations
- Draw simple plots
- Install and use packages

2 Installing R

R is freely available open-source software, which is distributed in easy-to-install binary packages for most popular current operating systems, including Microsoft Windows, Apple Mac OS, and Linux. You can easily install R on your own computer by obtaining the current version from http://www.r-project.org.

2.1 RStudio

RStudio is a powerful Integrated Development Environment (IDE) for R. You can download RStudio Desktop from https://www.rstudio.com.

The typical RStudio interface is shown in Figure 1. It is made up of different windows:

1. Script/Editor window (top left): Scripts are collections of commands that can be saved and edited. To run a command or a script in RStudio, you will need to get it into the command window. To do this, you can select the instructions that you want to execute and click Run or CTRL+ENTER. To open a new script, go to File -> New -> R script.
2. Console/Command window (bottom left): You can type R commands in this window. Results from your instructions will also appear in this window. The command line starts with > . You will type your commands after this symbol.

3. Workspace/History window (top right): The Workspace tab has information about the data and functions that R has in memory. The History window has everything that has been typed or sent to the command window before.

4. Files/Plots/Packages/Help window (bottom right): One of the most useful features of RStudio, this window allows you to open files, view graphs, and install and load packages.

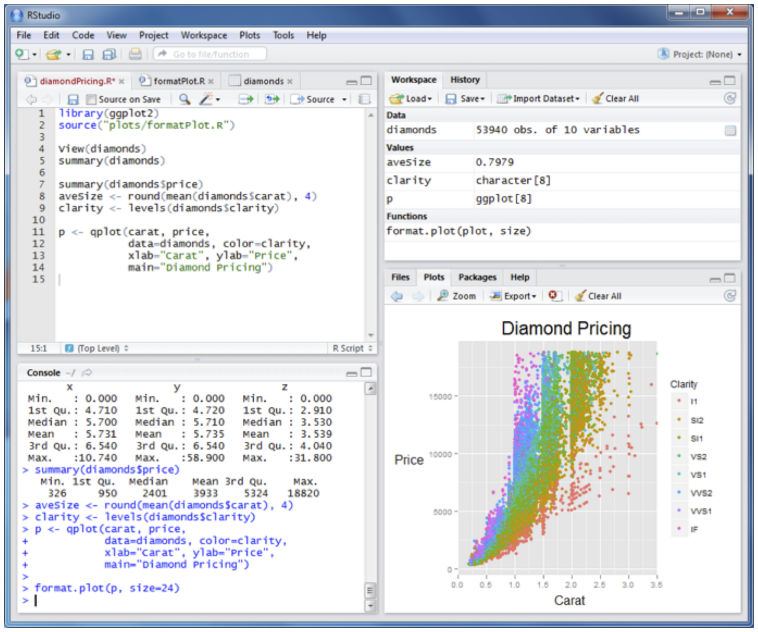

Figure 1: RStudio Interface

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

Instead of using the Console window to type code, you can save all your code in a script. Scripts are files with the extension ".r". Scripts will allow you to edit, expand, and rerun your code. Furthermore, if you want to keep all your work, you can save the workspace through the application (under 'workspace') or through the command prompt.

Alternatively, you can also create an R Markdown Notebook in RStudio, such as this one.

2.1.1 R Markdown Notebooks

R Markdown Notebooks are powerful files that allow you to combine text, code, outputs, and visualisations within the same document. They have the extension .Rmd

R Markdown Notebooks generates a new file that contains selected text, code, and results from the .Rmd file. This new file can be a finished web page, PDF, MS Word document, slide show, notebook, handout, book, dashboard, package vignette, or other format. Instead of *compiling* your code, you will *Knit* it into your desired format. This file that you are reading right now is the output of a R Markdown Notebook *"knitted"* into a PDF file.

In this module, we will use R Markdown Notebooks to generate HTML pages that combine text, code chunks, code outputs and visualisations in the same file. Lab exercises will be given out in .Rmd files, and you will be able to solve the questions in the same file and then *knit* them to visualise them.

Using R Markdown Notebooks takes some practice, so don't despair! Once you gain dexterity with them, you will be able create sophisticated documents that combine information in different formats and from different sources.

Let's go through some instructions on how to generate, save, and edit R Markdown Notebooks. For additional information, you can also read R Markdown: The Definitve Guide

2.1.1.1 Managing .Rmd files

To create a new R Markdown Notebook, got to "File -> New File -> R notebook".

You can save your R Markdown Notebook as you would any other R file: either using ctrl+S (cmd+S in Mac), or clicking the disk icon at the top of RStudio's interface.

If you are using a computer with different encoding, **make sure you create/save your notebooks with UTF-8 encoding**.

Instead of *compiling* your code, you will *Knit* it into your desired format. When you *Knit* a *.Rmd* file, a new file in your desired format will be created. In this module, you may use PDF or HTML. We will cover how to choose this format in the next section.

You may also choose to preview your file before knitting. For this, you can use the *"Preview Notebook"* option. If you click on the wheel icon next to the *Knit* button, you can see different preview options you can choose.

For more information on how to to save R Notebooks check out this link

2.1.1.2 Editing .Rmd files

When you open a new R Notebook file, you will see it already has some dummy information to help you get started as shown in this Figure:

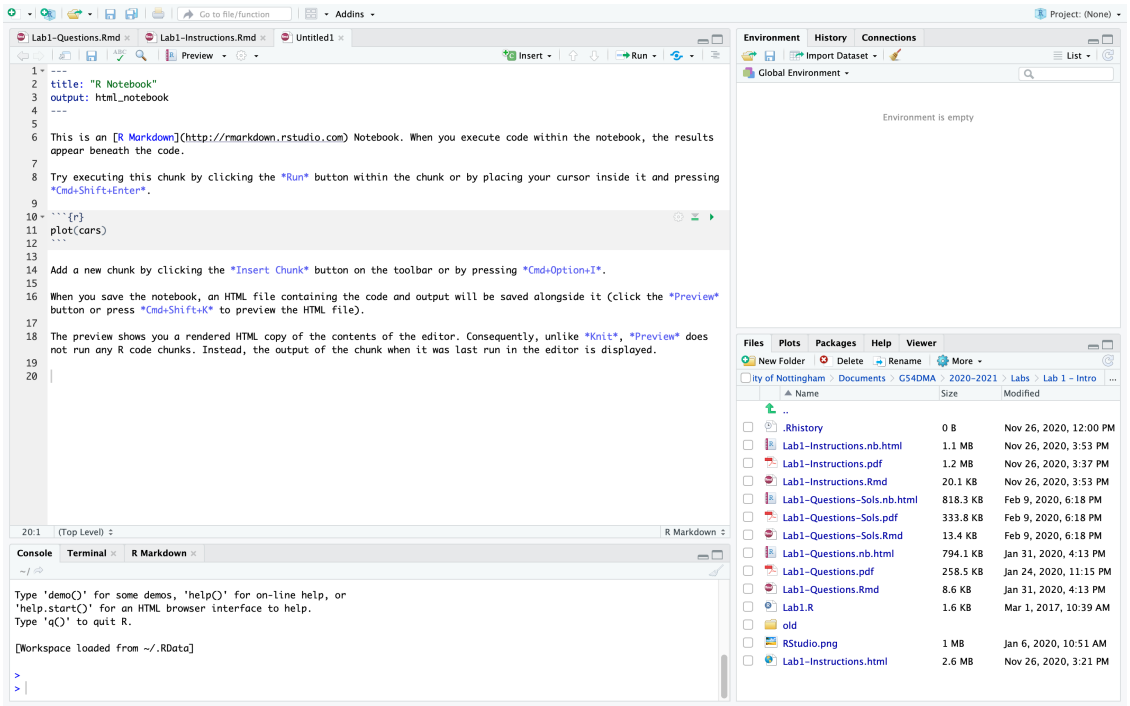

Figure 2: A brand new R Markdown Notebook

R notebooks are structured into three components, as shown in this Figure

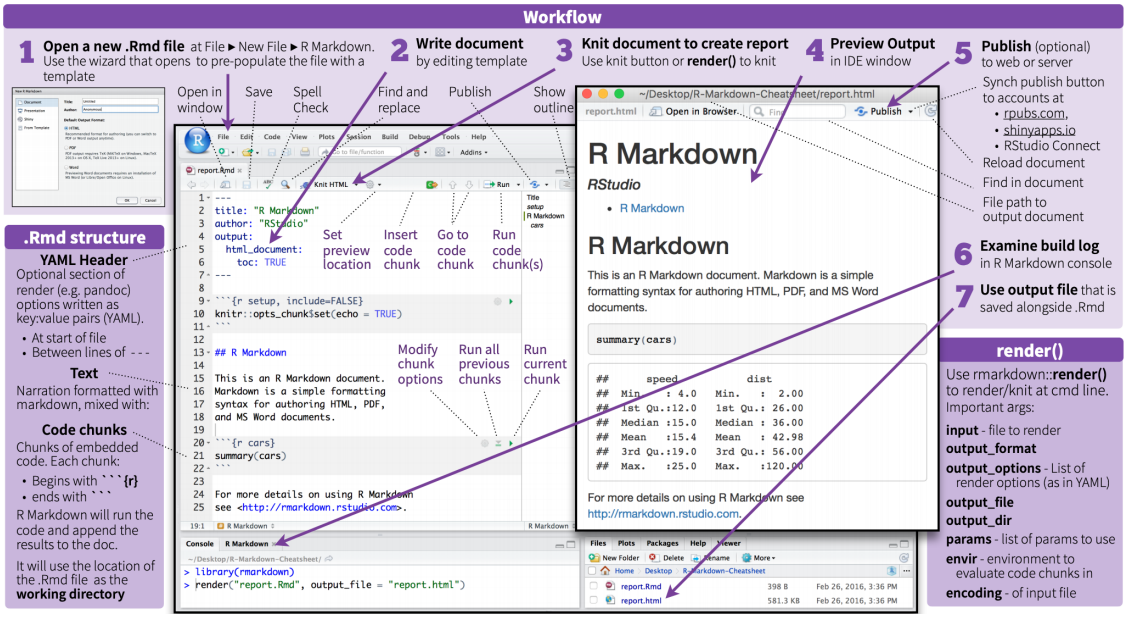

Figure 3: R Notebook components - Cheat Sheet

1. YAML metadata: This section stores information about the title, subtitle, author, and date of the document. It also includes options on how to knit your document. For information on how to write your YAML metadata for HTML, check out this link. YAML information for PDF can be found here. As a basic structure, you can use (remember to take out the comments with #):

```
---
title: 'Your title goes here'
subtitle: 'Your subtitle'
author: 'Your name'
output:
  pdf_document:           #this opens up the section for PDF knitting
    toc: yes                #table of contents: yes/no
    toc_depth: 3            #depth of the table of contents: how many levels do you want to show?
    fig_caption: yes        #do you want to add captions to figures? yes/no
    number_sections: true   #do you want automatic numbering in sections? true/false
  html_document:        #this opens up the section for HTML knitting
    df_print: kable         #this option selects how to display tables (dataframes)
    toc: yes
    toc_depth: 2
  html_notebook:          #this allows the preview option to work
    toc: yes
    toc_depth: 2
    toc_float:              #option to float the table of contents to the left of the document
      collapsed: no
      smooth_scroll: no
---
```

Figure 4: Standard YAML options

2. Text: R Markdown syntax, which is what is used in R Notebooks, is very simple. You can check syntax options in the first page of the R Markdown Reference Guide here.

3. R Code chunks: these chunks are sections in your file where you can write and execute R code separately. They can be inserted quickly using the keyboard shortcut Ctrl + Alt + I (macOS: Cmd + Option + I), or via the Insert menu in the editor toolbar. Because all of a chunk's output appears beneath the chunk

(not alongside the statement which emitted the output, as it does in the rendered R Markdown output), it is often helpful to split chunks that produce multiple outputs into two or more chunks which each produce only one output. You can insert options into a code chunk, just write them in the openning statement, after *r* and before the closing bracket. These are the three most common options you can use:

- eval = FALSE: Do not evaluate (or run) this code chunk when knitting the RMD document. The code in this chunk will still render in our knitted html output, however it will not be evaluated or run by R.
- echo=FALSE: Hide the code in the output. The code is evaluated when the Rmd file is knit, however only the output is rendered on the output document.
- results=hide: The code chunk will be evaluated but the results or the code will not be rendered on the output document. This is useful if you are viewing the structure of a large object (e.g. outputs of a large data.frame which is the equivalent of a spreadsheet in R).

This is an example of a chunk called *test-chunk* (chunks do not need to be named, but their names have to be distinct):

```
#Inside this chunk, R syntax is followed.
test = 2 + 2
test
```

```
## [1] 4
```

Let's see effect of the options:

- EVAL:

```
#Inside this chunk, R syntax is followed.
test = 2 + 2
test
```

There is no output associated with this chunk because the *eval* option is set to *FALSE*.

- ECHO

```
## [1] 4
```

The code for the previous chunk does not appear in the PDF, nor in the HTML! This is because the *echo* option is set to *FALSE*.

- RESULTS:

```
#Inside this chunk, R syntax is followed.
test = 2 + 2
test
```

The output of this chunk is not shown, because *results* is *FALSE*

For more information about using R Notebooks, check out this link.

You may also want to save a copy of these:

- R Markdown Cheatsheet: link
- R Markdown Reference Guide: link

3 R Resources

Due to its popularity, it is easy to find support for R questions. There are three main ways you can find support:

3.1 Built-in help

R offers a very powerful help function. The R help files are a useful source of information. Simply type your keyword into the following syntax:

```
help(plot)
```

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

```
help(help)
```

3.2 Online (tutorials, websites, and forums)

These are four of the most tutorials and websites with information, examples, tutorials and articles on R and some of its most popular packages.

- RStudio Online Learning: Includes numerous tutorials, cheat sheets, and examples. You can find it here.
- R Journal: an open access, refereed journal of the R project, especialised in statistical computing. It features articles on topics that could be of interest. You can find it here.
- Stack Overflow: an open community with questions and answers to all kinds of coding questions. You can find the R channel here: https://stackoverflow.com/tags/r/info

3.3 Offline (guides and books)

There are numerous books and guides on how to use R for data analysis, visualisation and machine learning. These are some recommendations:

- W. N. Venables, D. M. Smith and the R Core Team, 2019, *An Introduction to R (Notes on R: A Programming Environment for Data Analysis and Graphics)*: free resource with examples on everything you could need with regards to base R. Appendix A, in which you are shown *A sample session*, is particularly hepful. You can find it here: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
- Grolemund, G., 2014. *Hands-On Programming with R: Write Your Own Functions and Simulations*. O'Reilly Media, Inc.
- Wickham, H. and Grolemund, G., 2016. *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.
- Lantz, B., 2013. *Machine learning with R*. Packt Publishing Ltd.

4 Working Directory

The working directory is a file path on your computer that sets the default location of any files you read into or save out of R. If you ask R to import or export data it will assume that the file is inside of your working directory, unless you write its full path.

The first thing you need to do at the beginning of each session is to set your working directory to the folder where you want to store your code and graphs.

You can check where your current working directory is with:

```
getwd()
```

And you can change your working directory with:

```
setwd("/Path/to/RCode")
```

Alternatively, you can use the Files tab in RStudio (bottom right window) to set your working directory. First, in the Files tab, navigate towards your desired working directory. Once there, click on "More -> Set as working directory".

Reading: For more information about the R environment, read *Chapter 1* of *An Introduction to R* by Venables et al.

5 Installing packages

From time to time, you might need to extend the functionality present in R with code from third parties. Part of the reason R has become so popular is the vast array of packages available at the CRAN and Bioconductor repositories. In the last few years, the number of packages has grown exponentially!

The five most popular R packages are:

- *dplyr*: a grammar of data manipulation
- *devtools*: a collection of package development tools
- *foreign*: read data stored by Minitab, S, SAS, SPSS, Stata, and more
- *cluster*: methods for cluster analysis
- *ggplot2*: an implementation of the grammar of graphics in R

We will introduce *dplyr*, and *ggplot2* in later lab sessions, and we will cover *cluster* in Lecture 9.

To install a package, you can use the *install.pacakages()* function:

```
install.packages("name")
```

Then, to load the package and use it, use the *library()* function (the location of the library can sometimes be omitted). You only need to install a package once but, in order to use it, **you will need to load it at the beginning of each session.**

```
library ("name", "location")
```

You only need to install a package once but, in order to use it, *you will need to load packages at the beginning of each session.*

Reading: For more information about Packages, read Chapter 13 of An Introduction to R by Venables et al.

6 Basic Operations in R

6.1 Arithmetic Operations

Variables can be assigned by using the following syntax:

```
num <- 2
num = 2
```

Here *num* is assigned the value of 2. A number of R commands use the <- notation for assignment. However, in nearly all cases, an equal sign (=) will also work.

Examples of the most common arithmetic operations are shown below:

```
a = 7
b = 4

a + b # addition
```

```
## [1] 11
```

```
a - b # subtraction
```

```
## [1] 3
```

```
a * b # multiplication
```

```
## [1] 28
```

```
a / b # division
```

```
## [1] 1.75
```

```
a ** b # power
```

```
## [1] 2401
```

```
a ^ b # power (alternative version)
```

```
## [1] 2401
```

```
a %% b # modulus (remainder)
```

```
## [1] 3
```

```
a %/% b # integer  division
```

```
## [1] 1
```

```
sqrt(a) # square root
```

```
## [1] 2.645751
```

```
log2(b) # logarithm with base 2
```

```
## [1] 2
```

```
log(b,2) # alternative logarithm with base 2
```

```
## [1] 2
```

```
log10(b) # logarithm with base 10
```

```
## [1] 0.60206
```

```
log(b, 10) # alternative logarithm with base 10
```

```
## [1] 0.60206
```

```
exp(a) # exponential function
```

```
## [1] 1096.633
```

6.2 Logical Operations

Examples of the most common logical operations are shown below

```
x = 13
y = 15
w = FALSE
z = TRUE

x < y # less than
```

```
## [1] TRUE
```

```
x < y - 3 # atithmetic operations (y-3) takes priority
```

```
## [1] FALSE
```

```
x <= y # less than or equal to
```

```
## [1] TRUE
```

```
x > y # greater than
```

```
## [1] FALSE
```

```
x >= y # greater than or equal to
```

```
## [1] FALSE
```

```
x == y # equal to
```

```
## [1] FALSE
```

```
x != y # not equal to
```

```
## [1] TRUE
```

```
x != y-2 # not equal to
```

```
## [1] FALSE
```

```
!w # not
```

```
## [1] TRUE
```

```
w | z  # or
```

```
## [1] TRUE
```

```
w & z # and
```

```
## [1] FALSE
```

```
isTRUE(x==x)
```

```
## [1] TRUE
```

7 Simple Data Structures

R has a wide variety of data structures. In this section, we will cover the most basic ones:

7.1 Vectors and Sequences

The simplest data structure is a numeric vector: a single entity consisting of an ordered collection of numbers.

You can create a vector using the *c()* function, which stands for *concatenate*:

```
a <- c(6, -1, 3, -12, 3.4, 7.05) # numeric vector
b <- c("one","car","apple","banana") # character vector
c <- c(TRUE,FALSE,TRUE,FALSE,FALSE) #logical vector
```

Entering data by hand can be tedious so R allows creating uniformly spaced data points, known as sequences. Sequences are created with the *:* operator, or the *seq(from=, to=, by=)* function.

```
d = 0:100
d1 = seq(1,100)
d2 = seq(from=1, to=100)
```

```
d3 = seq(to=100, from=1) # d, d1, d2, d3 generate the same sequence

e = seq(from=0, to=1, by =.01) #by determines the interval
f = seq(1,100,20) # you do not need to add the names of the parameters
g = seq(1,15, length.out=8) #length.out determines the length of the output
h =  seq(15, 1)
```

If you need to create a vector with the same repeated value, you can also use *rep(x, times=)* or *rep(x, each=)*:

```
rep(4, 3) # repeats 4 three times
```

```
## [1] 4 4 4
```

```
rep(c(1,2,3), times=2) # the whole vector is repeated
```

```
## [1] 1 2 3 1 2 3
```

```
rep(c(1,2,3), each=2) # each value is repeated
```

```
## [1] 1 1 2 2 3 3
```

You can index all of the elements of a vector using subscripts. R is extremely versatile:

```
a[1]# 1st element
```

```
## [1] 6
```

```
a[-1] # drops the 1st element
```

```
## [1]  -1.00   3.00 -12.00   3.40   7.05
```

```
b[-3] #drops the 3rd element
```

```
## [1] "one"    "car"    "banana"
```

```
a[c(2,4)] # 2nd and 4th elements of vector. The c() functions stands for "concatenate".
```

```
## [1]  -1 -12
```

```
# The : operator gives the lower and upper limits of an interval
b[2:4]  # from 2nd to 4th element.
```

```
## [1] "car"    "apple"  "banana"
```

```
a[a>5] # applies the condition inside the brackets
```

```
## [1] 6.00 7.05
```

You can read that last instructions as: "those elements in *a* such as a is larger than 5".

```
#What is the difference between these two instructions?
a[a>5]
```

```
## [1] 6.00 7.05
```

```
a>5
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE  TRUE
```

You can also check how many elements a vector has with *length()* and carry out any of the arithmetic and logic operations we saw in the previous section. In most cases, these operations will be performed element by element:

```
n1 = seq(1, 15, by=2)
n2 = c(2, 4, 5, 1, 7, 3, 9, 1)

length(n1)
```

```
## [1] 8
```

```
n1 + 3 # add 3 to each element
```

```
## [1]  4  6  8 10 12 14 16 18
```

```
n1 - n2 # element-by-element subtraction
```

```
## [1] -1 -1  0  6  2  8  4 14
```

```
n1 * n2 # element-by-element multiplication
```

```
## [1]   2  12  25   7  63  33 117  15
```

```
n1 ^ 2 # each element to the power of 2
```

```
## [1]   1   9  25  49  81 121 169 225
```

```
n1 / n2 # element-by-element division
```

```
## [1]  0.500000  0.750000  1.000000  7.000000  1.285714  3.666667  1.444444
## [8] 15.000000
```

```
sqrt(n1) # square root of each elemement
```

```
## [1] 1.000000 1.732051 2.236068 2.645751 3.000000 3.316625 3.605551 3.872983
```

```
n1 == n2 # element-by-element equal to
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
n1 < n2
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

You can also carry out simple statistics:

```
mean(n1) # mean
```

```
## [1] 8
```

```
sd(n1) # standard deviaton
```

```
## [1] 4.898979
```

```
median(n1) # median
```

```
## [1] 8
```

```
min(n1) # minimum value
```

```
## [1] 1
```

```
max(n1) # maximnum value
```

```
## [1] 15
```

```
summary(n1) # min, max, mean, med, quartile information
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     4.5     8.0     8.0    11.5    15.0
```

We will cover more on statistics in Lab 4.

7.2 Matrices

All columns in a matrix must be of the same type and the same dimensions. The general call will use the function *matrix(. . .)*:

```
matrix_name <- matrix(vector, nrow=r, ncol=c, byrow=FALSE,
   dimnames=list(char_vector_rownames, char_vector_colnames))
```

byrow=TRUE indicates that the matrix should be filled by rows. byrow=FALSE indicates that the matrix should be filled by columns (the default). dimnames provides optional labels for the columns and rows.

Let's see an example:

```
# generates 3 x 4 numeric matrix
X = matrix(1:12, nrow=3,ncol=4)
X
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
#you can also build a matrix by binding (e.g. appending) vectors together
X2 = rbind(c(1,2,4,5), c(3,4,9,1)) #binds by row
X2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    4    5
## [2,]    3    4    9    1
```

```
X3 = cbind(c(1,2,4), c(3,5,1), c(5,6,11)) #binds by column
X3
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    5    6
## [3,]    4    1   11
```

```
# another example
values <- c(1,34,5,99)
rnames <- c("R1", "R2")
cnames <- c("C1", "C2")
Y <- matrix(values, nrow=2, ncol=2, byrow=TRUE,
  dimnames=list(rnames, cnames))
Y
```

```
##    C1 C2
## R1  1 34
## R2  5 99
```

Similarly to vectors, we can index all the values in a matrix. We can index rows, columns or elements using subscripts.

```
X[1,3] # 3rd element of the 1st column
```

```
## [1] 7
```

To access all of the elements on one dimension (e.g. all elements of a column), leave the dimension blank. You can also use ranges and distinctive indexes:

```
X[,4] # 4th column of matrix
```

```
## [1] 10 11 12
```

```
X[3,] # 3rd row of matrix
```

```
## [1]  3  6  9 12
```

```
X[2:3,1:3] # rows 2,3 of columns 1,2,3
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
X[, c(2,4)] # 2nd and 4th column
```

```
##      [,1] [,2]
## [1,]    4   10
## [2,]    5   11
## [3,]    6   12
```

```
X[c(1,3), c(2,4)]
```

```
##      [,1] [,2]
## [1,]    4   10
## [2,]    6   12
```

You can also use negative indexes to ignore certain rows or columns:

```
X[-1,] # what does this do?
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    5    8   11
## [2,]    3    6    9   12
```

```
X[,-1] # what does this do?
```

```
##      [,1] [,2] [,3]
## [1,]    4    7   10
## [2,]    5    8   11
## [3,]    6    9   12
```

You can carry out arithmetic, logical and statistical operations in matrices as well. These operations will be carried out element by element most times:

```
A = rbind(c(1,2,3), c(9,8,7), c(10, 1, 3))
B = matrix(seq(1,9), ncol=3)
C = cbind(c(3, 4,5), c(1, 4, 1))

#Arithmetic operations
A + B
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]   11   13   15
## [3,]   13    7   12
```

```
A + C # error: different dimensions
```

```
## Error in A + C: non-conformable arrays
```

```
A - B
```

```
##      [,1] [,2] [,3]
## [1,]    0   -2   -4
## [2,]    7    3   -1
## [3,]    7   -5   -6
```

```
A * B # element-by-element product
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   21
## [2,]   18   40   56
## [3,]   30    6   27
```

```
A ^ 2
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   81   64   49
## [3,]  100    1    9
```

```
A / B
```

```
##          [,1]      [,2]      [,3]
## [1,] 1.000000 0.5000000 0.4285714
## [2,] 4.500000 1.6000000 0.8750000
## [3,] 3.333333 0.1666667 0.3333333
```

```
A %*% B # arithmetic matrix multiplication
```

```
##      [,1] [,2] [,3]
## [1,]   14   32   50
## [2,]   46  118  190
## [3,]   21   63  105
```

```
B %*% C # arithmetic matrix multiplication
```

```
##      [,1] [,2]
## [1,]   54   24
## [2,]   66   30
## [3,]   78   36
```

```
sqrt(A)
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.000000 1.414214 1.732051
## [2,] 3.000000 2.828427 2.645751
## [3,] 3.162278 1.000000 1.732051
```

```
#Logical operations

A == B
```

```
##       [,1]  [,2]  [,3]
## [1,]  TRUE FALSE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE
```

```
A != B
```

```
##       [,1] [,2] [,3]
```

```
## [1,] FALSE TRUE TRUE
## [2,]  TRUE TRUE TRUE
## [3,]  TRUE TRUE TRUE
```

```
A <= B
```

```
##       [,1]  [,2] [,3]
## [1,]  TRUE  TRUE TRUE
## [2,] FALSE FALSE TRUE
## [3,] FALSE  TRUE TRUE
```

```
A > B
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,]  TRUE  TRUE FALSE
## [3,]  TRUE FALSE FALSE
```

```
#Simple statistical operations

mean(A) # calculates mean across all values
```

```
## [1] 4.888889
```

```
colMeans(A) # mean by column
```

```
## [1] 6.666667 3.666667 4.333333
```

```
rowMeans(A) # mean by row
```

```
## [1] 2.000000 8.000000 4.666667
```

```
sd(A) # standard deviation across all values
```

```
## [1] 3.586239
```

```
length(A) #total number of elements
```

```
## [1] 9
```

```
ncol(C) # number of columns
```

```
## [1] 2
```

```
nrow(C) # number of rows
```

```
## [1] 3
```

```
rowSums(C) # adds elements per row
```

```
## [1] 4 8 6
```

```
colSums(C) # adds elements per column
```

```
## [1] 12  6
```

You can also access splits (i.e. subsets) of data by using conditions as indexes. Let's see some examples:

```
#Returns even numbers in A
A[A%%2==0]
```

```
## [1] 10  2  8
```

```
#What does this do?
A%%2==0
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE  TRUE FALSE
## [2,] FALSE  TRUE FALSE
## [3,]  TRUE FALSE FALSE
```

7.3 Arrays

Arrays are similar to matrices but can have more than two dimensions. To create an array, use the array() function, which takes vectors as input and uses the values in the dim parameter to create an array.

The following example creates an array of two 3x3 matrices:

```
vector1 = 1:9
vector2 = 13:21

# Take these vectors as input to the array.
res <- array(c(vector1,vector2),dim = c(3,3,2))
res
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   16   19
## [2,]   14   17   20
## [3,]   15   18   21
```

You can access different splits of the data using indeces:

```
res[1,,] # first row in all dimentions
```

```
##      [,1] [,2]
## [1,]    1   13
## [2,]    4   16
## [3,]    7   19
```

```
res[,1,] # first column in all dimensions
```

```
##      [,1] [,2]
## [1,]    1   13
## [2,]    2   14
## [3,]    3   15
```

```
res[,,1] # first matrix in the array
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

And you can carry out arithmetic, logical and statistical operations as well:

```r
ar1 = array(1:27, dim=c(3,3,3))

# If you need to generate random numbers, use the sample() function
# sample.int(a,b) generates b integer nums in the range [0,a]
ar2 = array(sample.int(100,27), dim=c(3,3,3))

# Arithmetic operations
ar1 + 2
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    3    6    9
## [2,]    4    7   10
## [3,]    5    8   11
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   12   15   18
## [2,]   13   16   19
## [3,]   14   17   20
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   21   24   27
## [2,]   22   25   28
## [3,]   23   26   29
```

```r
ar1 - ar2
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]   -1  -66  -46
## [2,]  -27  -47   -5
## [3,]  -87  -38   -5
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  -64  -72  -39
## [2,]  -82  -74  -70
## [3,]  -16  -10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]  -22  -78  -46
## [2,]  -71   12  -20
## [3,]  -37  -57  -27
```

```r
ar1 * ar2
```

```
## , , 1
```

```
##
##      [,1] [,2] [,3]
## [1,]    2  280  371
## [2,]   58  260  104
## [3,]  270  264  126
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]  740 1105  880
## [2,] 1023 1232 1479
## [3,]  336  375  108
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]  779 2200 1775
## [2,] 1820  253 1196
## [3,] 1218 1944 1458
```

```
sqrt(ar1)
```

```
## , , 1
##
##          [,1]     [,2]     [,3]
## [1,] 1.000000 2.000000 2.645751
## [2,] 1.414214 2.236068 2.828427
## [3,] 1.732051 2.449490 3.000000
##
## , , 2
##
##          [,1]     [,2]     [,3]
## [1,] 3.162278 3.605551 4.000000
## [2,] 3.316625 3.741657 4.123106
## [3,] 3.464102 3.872983 4.242641
##
## , , 3
##
##          [,1]     [,2]     [,3]
## [1,] 4.358899 4.690416 5.000000
## [2,] 4.472136 4.795832 5.099020
## [3,] 4.582576 4.898979 5.196152
```

```
# Logical operations

ar1!=ar2
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE
##
## , , 2
##
```

```
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,] TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE
```

```
ar1>ar2
```

```
## , , 1
##
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE FALSE FALSE
##
## , , 2
##
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE FALSE  TRUE
##
## , , 3
##
##       [,1]  [,2]  [,3]
## [1,] FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE
## [3,] FALSE FALSE FALSE
```

```
# Statistical operations

mean(ar1)
```

```
## [1] 14
```

```
rowMeans(ar1)
```

```
## [1] 13 14 15
```

```
colMeans(ar1)
```

```
##      [,1] [,2] [,3]
## [1,]    2   11   20
## [2,]    5   14   23
## [3,]    8   17   26
```

```
rowSums(ar1) #sums across all dimensions
```

```
## [1] 117 126 135
```

```
colSums(ar1)
```

```
##      [,1] [,2] [,3]
```

```
## [1,]    6   33   60
## [2,]   15   42   69
## [3,]   24   51   78
```

See *help(array)* for more details.

8 Simple Plots

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

```
a = seq(1,15)
c = 2*a+1
plot(a, c)
```

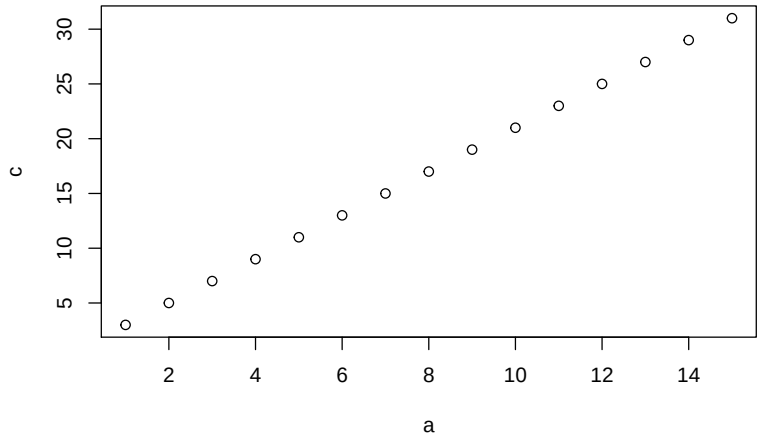

Figure 5: Basic Plot

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

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

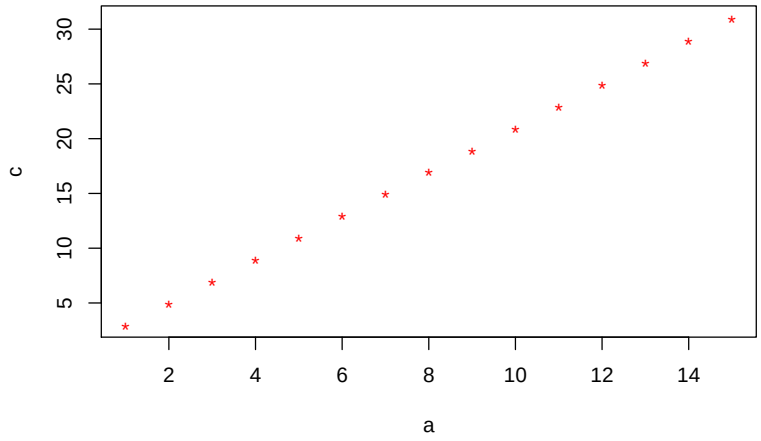

Figure 6: Customised Basic Plot

Let's add a title:

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

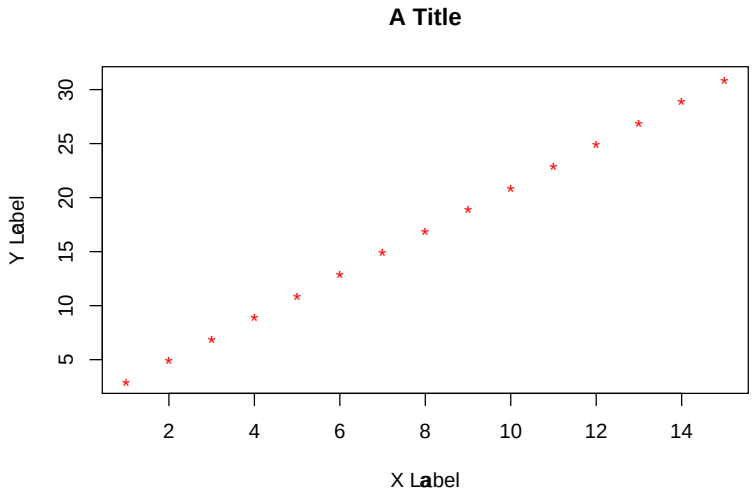

Figure 7: Customised Basic Plot With Title

You can actually do this all within the plot command.

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

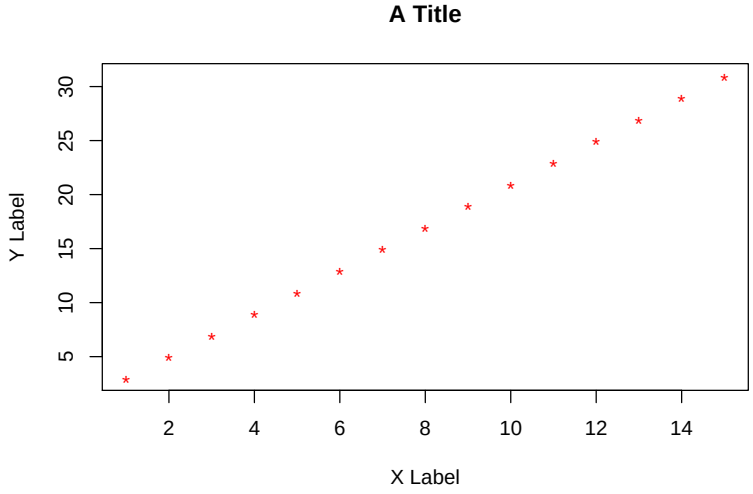

Figure 8: Customised Basic Plot With Title

We can use the *points* or *lines* command to plot more than one data series on one figure . For this, use the functions *points(x,y,col)* or *lines(x,y,col)*:

```
a = seq(1,15)
c = 2*a+1
b = a + 2
plot(a, c)
points(a, b, col='blue')
```

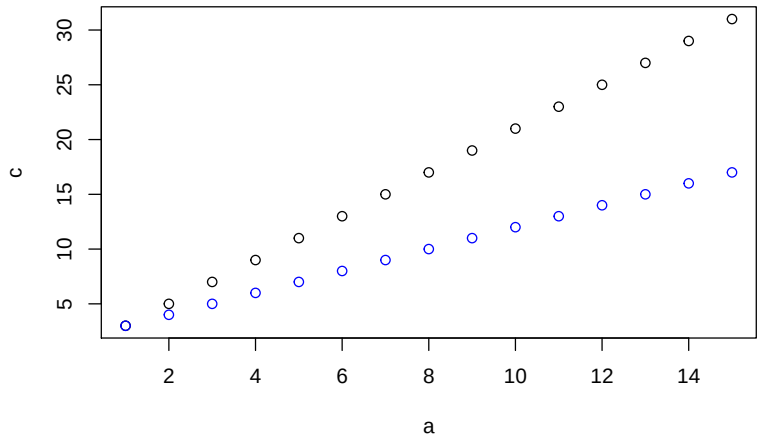

Figure 9: Two Sequences Using Points

```
a = seq(1,15)
c = 2*a+1
b = a + 2
plot(a, c, type = 'l',col='red')
lines(a, b, col='blue')
```

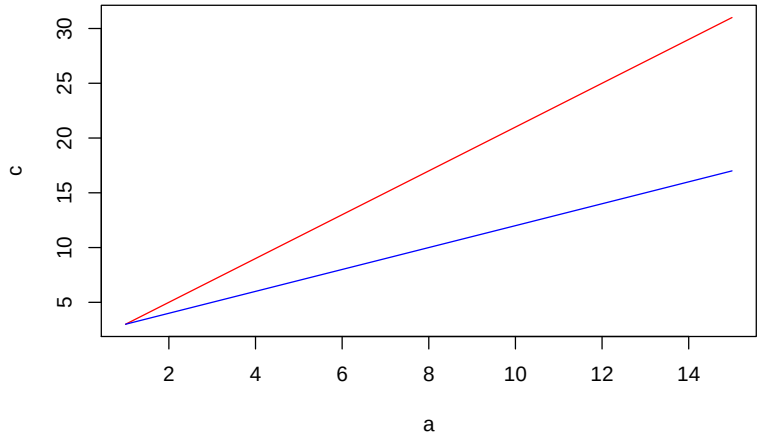

Figure 10: Two Sequences using Lines

Reading: For more information about the *plot()* function, read points 12.1.1, 12.4, and 12.5 of *An Introduction to R* by Venables et al (we will cover more advanced plotting commands in the upcoming weeks).