# Notes for Statistical Computing module

Compiled by: Res Altwegg, Birgit Erni, Neil Watson, Allan Clark

2023-01-23

# Contents

# Chapter 1

# Preamble

Welcome to the module "Introduction to statistical computing with R". The goal of this module is to give you an introduction to program R (if you are new to it) or a refresher (if you are already familiar with R), and how it is used for statistical computing.

By the end of this module, you should be able to:

- manipulate/organize data and data files
- use R for summary statistics, fitting statistical model
- use R to create quality graphics
- use the R language for programming/writing your own custom functions
- use simulations to solve some statistical problems (Monte Carlo and bootstrap)
- optimize functions using R
- do parallel computing in R

R is the most widely used software package for statistical computing. I consists of a set of core packages and a very large number of contributed packages. Anyone can write an R package and make it available to the community. As a result, the R project has grown enormously and contains much more than you are ever likely to use. Starting to use R is a bit like moving to a new city. You will first become familiar with the main facilities such as main roads and the most popular places. You will explore the area around where you live and work in more detail and find all the facilities you need, such as shops, restaurants, sports fields, etc. There will be a lot of places in this city that you will never visit even if you stay for many years. These may be neighbourhoods far from where you live, but other people live there and build their lives around those places. Nobody knows every nook and cranny of this big city.

R is a bit similar. You will learn basic principles and procedures first. Then, you will explore R's functionality around the type of problems you want to

solve. There are always alternative ways of accomplishing something, like there are always alternative shops where you can buy your food. You will become familiar with certain R functions and packages and over time develop expertise in using them; nobody is an expert on all aspects of R. So if you have already used R before, you will learn new aspects even with this relatively basic course. And we may very well learn new tricks from you!

## 1.1 Admin

### 1.1.1 Lecturers

- Sulaiman Salau (30 Jan - 2 Feb 2023)
- Mzabalzo Ngwenya (2 Feb - 10 Feb 2023)

### 1.1.2 Assessment

- The average across all your practicals counts 20%;
- the two assignments count 10% each; and
- a 3-hour practical exam counts 60%; the date of the exam is still to be determined.

# Chapter 2

# Basics and vectors

In this section, we will cover some of the basics of R, what it is and how to get started with using it. We are going to touch on the following subjects:

- R and RStudio
- basic operations
- script files, R Markdown
- data types
- data objects: vectors, matrices and data frames

## 2.1 Why R?

### 2.1.1 Brief R History

R had it's beginning in 1992 when Ross Ihaka and Robert Gentleman began developing language for statistical computing that was based on S. S was the software of choice for many statisticians at the time but it didn't run on the computers in the labs they used for teaching. So Ihaka and Gentleman decided to write their own software. R was consciously designed to blur the distinction between users and programmers, a feature that later contributed greatly to it's success. They first made their software available in 1993 but it was another key decision, taken in 1995, that put R on the path to become the most widely used statistical software: they decided to make the source code available. The decision to make R an open-source software opened the door for others to contribute and the "R Core Team" was formed. In 2000, the team felt ready to make the software more widely available and R version 1.0.0 was released.

One factor that contributed to R's growth is the Comprehensive R Archive Network (CRAN, https://www.r-project.org), which has all the essential information and software available for download from a single place. The basic

functionality of R comes with the "base installation" but what really makes R useful is that it has a lot of contributed packages that can be downloaded and installed as needed. This system enables the entire R community to contribute to software development and the number of packages is growing exponentially. As I write this in early 2021, there were >17,000 packages on CRAN.

Writing R packages has become relatively easy and you will come across R packages in many places. The R packages on CRAN have been reviewed and a functionality check. This makes them more reliable as the packages have to clear a relatively high bar. It is easier to publish an R package on GitHub and you will find a lot of useful packages there but they may need to be used with a bit of additional care.

Read the paper by Thieme (2018, Significance 15: 14-18; pdf on Vula) if you want to know a bit more about the story behind R.

### 2.1.2 What is R?

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or in print, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

Read the this web page and explore the CRAN.

### 2.1.3 Downloads

We are going to use two separate programs that need to be downloaded and installed separately, R and RStudio.

Download R from CRAN. Click on 'CRAN' and you will see a list of mirrors. Each of those should offer the same version of R and all packages. The idea is to choose one close to you (e.g. there is a mirror in South Africa) but the cloud option (first one from top) or the master site in Vienna, Austria are also good choices. If you have any issues with downloading packages (I've never had any issues downloading R itself), try any of these. When you click on a mirror, download the precompiled version for your system (Linux, Mac OS or Windows). Depending on your system, there are further choices. E.g. if you follow the version for Windows, click on 'base' on the next screen and then 'Download R 4.0.4 for Windows', and you will get an executable file that you can install on your computer (it's fairly large: 85MB! If data costs are an issue and you are using Windows, you can download the 'R-4.0.4-win.exe' file from the Resources -> 'R and RStudio executables' folder on the course Vula site). You can also download contributed packages from this site but don't worry about this right now.

With R installed and running, you have the full functionality of R, as we will see in a moment. Technically, you could start writing your code and do the data preparation and analysis and everything else R can do.

This is fine but the R user interface is relatively basic. This is where RStudio comes in.

One aspect of RStudio is that it is a fancy interface to R. But as we will see,

RStudio is also an Integrated Development Environment that has tools that help with debugging, documentation of your analysis (via R Markdown) and workspace management in general.

Download RStudio from the RStudio web page. Go to downloads and choose the RStudio Desktop Free version. Click on 'Download' and you will again have to choose the appropriate link for your system. E.g. for Windows, you download an executable file (150MB; If data costs are an issue and you are using Windows, you can download the 'RStudio-1.4.1106.exe' file from the Resources -> 'R and RStudio executables' folder on the course Vula site).

Make sure you install R before installing RStudio.

## 2.2 First steps with R

Now you should be set up to start using R. The only way to become proficient in R is to use it – a lot. From here onward, we expect you to **try out all the code** shown in these notes. Enter the code into R, run it and see what happens. Modify it and don't be shy to break it. Then fix it again. It is critical that you engage with the code to the point that you fully understand what it is doing.

This section helps you get started with R. If you are already familiar with R, you can read through this section quickly but do make sure that you are comfortable with everything covered here as it forms the basis for what follows.

Now let's start using R.

### 2.2.1 The R Graphical User Interface

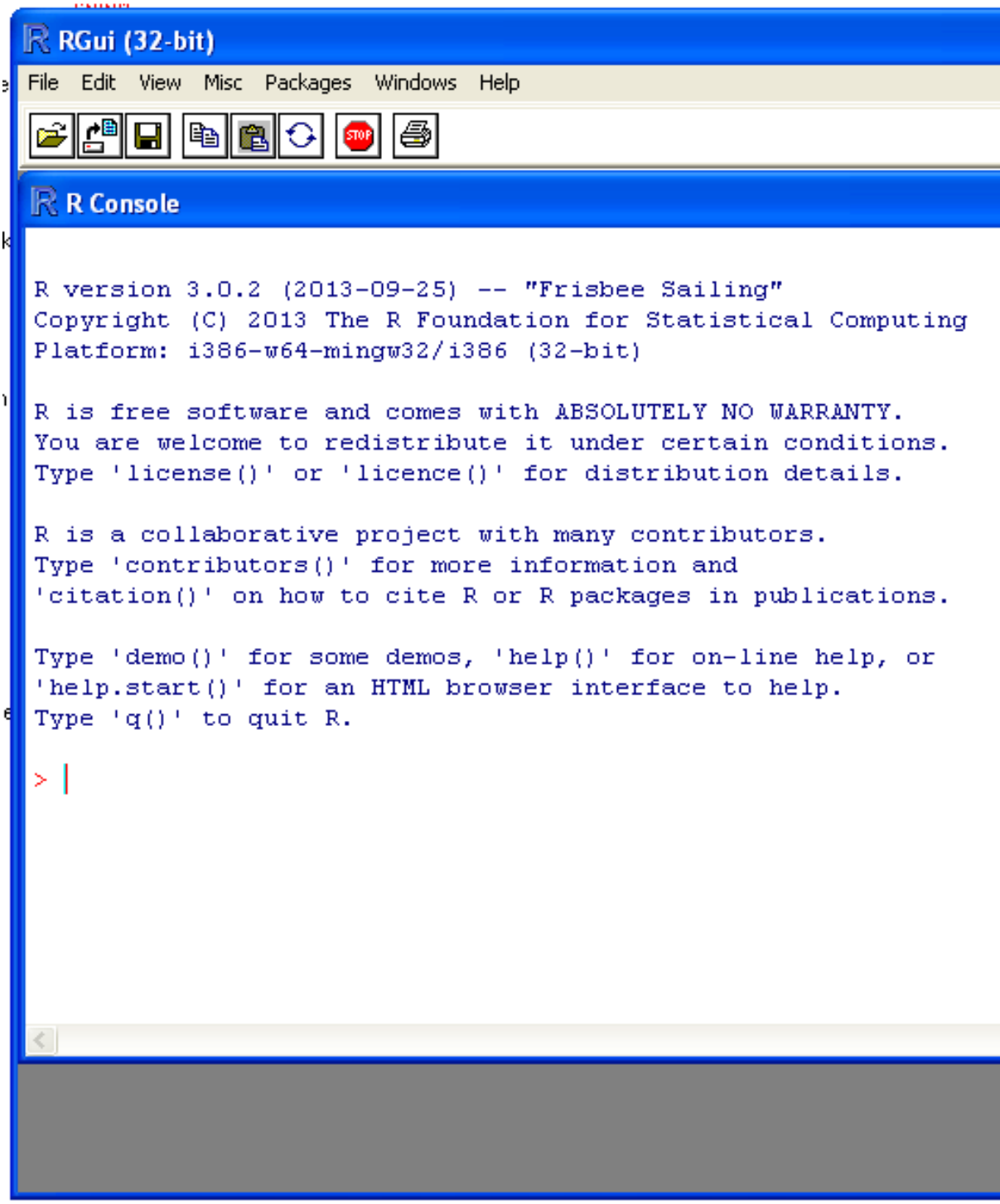
---

1. Let's start by making a new folder in a convenient location on your computer. Give this folder an informative name – perhaps "MySTA5075Zanalyses".

2. Once you've done this, double-click the R shortcut on your desktop:

---

This opens R and presents a window called the R console with some information about the version of R you are using etc.:


RGui (32-bit)
File Edit View Misc Packages Windows Help
R Console
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-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.
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.
>

The R console is one interface you could use to communicate with R. We are going to use a different interface in a moment but let's look at the console first just to get a feeling for how R works. R expects commands from us. These need to be entered after the prompt (the red '>'). To see how it works let's try some basic arithmetics. Type the following command and hit enter:

```
5+2
```

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

You've asked R to calculate 5 plus 2 and it returns the answer, 7, in the next line in blue colour. A new prompt appears and R is ready for the next command.

Hopefully, you will see that `5+2` above is in a grey box, although the box might print out quite light and difficult to distinguish from the background. You will also notice that the font changes to something like `this`. This type setting represents code, which you must type into R, unless otherwise stated. The output will usually follow, so that you can check that you are getting the right response from R. The output will be preceded by a ## sign, like the `7` above. The number [1] between the square brackets is just an index that we don't have to worry about for now.

Now you can type the following command and hit enter:

```
sqrt(2)
```

```
## [1] 1.414214
```

You've asked R to calculate the square root of 2. `sqrt()` is what we call an R function. R has many built-in functions and you may also create new, customized ones. Functions take in arguments (in the previous example the number 2 would be an argument) and output some result (in the previous example the square root of 2 is the output). We'll see plenty of functions and learn how to write our own ones.

Then, we have "operators". Operators also give instructions to R but they are less complex than functions and don't need arguments. For example, we have the common arithmetic operators:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Exponentiation: ^

You already know these. Let's try a new one. Type in:

```
1:10
```

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

The `:` operator creates a sequence of numbers; in this case from 1 to 10. This will be very useful for indexing in coming labs; for example, if you wanted to select the first 10 rows of your data.

OK, great, so R is a (very complicated) calculator. Let's make things a bit more interesting. Apart from calculating things, R can also store things that, in general, we call "objects". There are many types of different objects but for now we will look at only a few basic ones. A very simple object might be just a number. For example, type in the following command:

```
x <- 2
```

Nothing happened? We know that the code was executed because a new prompt appeared under our code and we didn't get any error messages from R. We have just created a object `x` and we have assigned the value 2 to it, using the `<-` operator. Even though R doesn't say anything, now our object `x` is stored in R´s "memory" and we can work with it as if it was a variable with value 2. If we type in the following command:

```
x + 3
```

```
## [1] 5
```

We obtain the value 5, which is just 2 + 3. In fact, we can just retrieve the value of the variable `x` by typing:

```
x
```

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

Now, we can take this further and play around with different variables. For example, type in the following code (press enter after each line):

```
my.income <- 15000
my.expenses <- 12000
my.savings <- my.income - my.expenses
my.savings
```

```
## [1] 3000
```

All three variables: `my.income`, `my.expenses` and `my.savings` are stored in R´s memory and we can call them at any time by writing their names.

```
my.income
```

```
## [1] 15000
```

But we must be very careful, if we assign a different value to any of these variables, R will overwrite them without any warning! Type the following lines and press enter after each one of them:

```
my.expenses <- 15000
my.savings <- my.income - my.expenses
my.savings
```

```
## [1] 0
```

Our savings are gone! We have overwritten the object `my.expenses` and R didn't say anything. Well, you get the idea; for R, variables are just variables and you can assign any value to them at any time. Be careful when picking names for your objects and try to keep track of them. This will be easier in RStudio, which is a different interface we use to communicate with R. We'll see that in a minute.

Also, writing code in the R console is not very effective. Instead of feeding R commands one-by-one, we can type up a sequence of commands that we'll pass on to R. This is called an R script and it usually contains all the commands that we require to conduct some analysis. Writing clean and organized scripts is extremely important, as we will see shortly. R has an in-built script editor but RStudio makes it easier to write and run code in an organized way.

In principle, you could do most of what we do in this course directly in R. However, we will assume that you run R via RStudio from here on. Do remember, though, that you are still working with R even though it may not be obvious after you move to RStudio. Don't make the mistake to refer to RStudio when you actually mean R. E.g. when you list the statistical software packages you are familiar with, you would list R, not RStudio!

We are now moving to this more user-friendly way of communicating with R. Close R and when it asks you whether you want to save the work space, say 'no'.

### 2.2.2 R Scripts

In this course, we will use R through the RStudio interface. RStudio is really just an interface that makes controlling R easier.

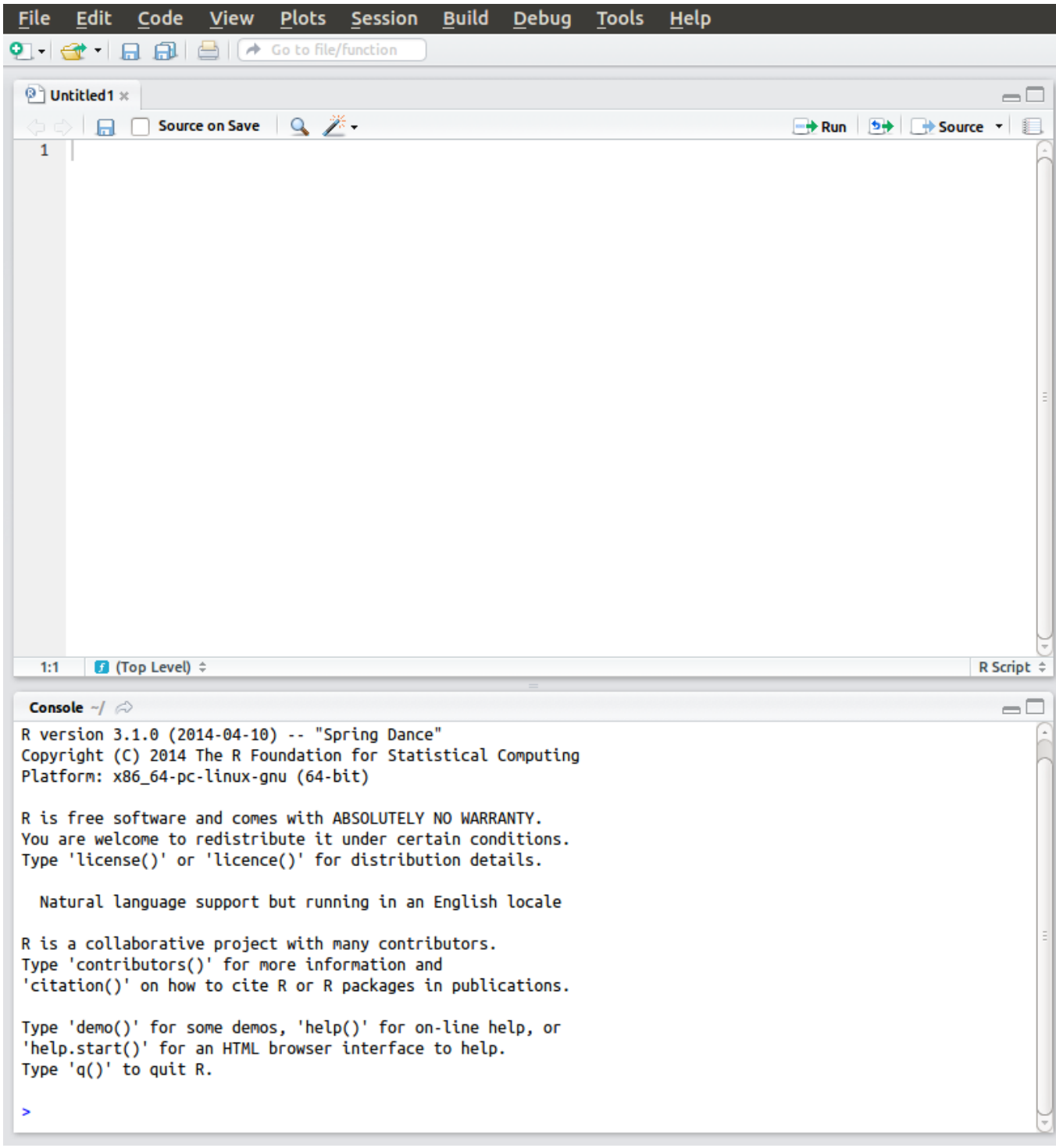
---

1. Double-click this icon on your desktop to open RStudio:

2. RStudio opens. Go to File → New File and click on 'R script'.

---

You should now see something like this:


File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Untitled1
Source on Save
Run
Source
1
1:1
(Top Level)
R Script
Console ~/
R version 3.1.0 (2014-04-10) -- "Spring Dance"
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.
>

The RStudio window is divided into four sections (if you only see three sections, go to File → New File → R Script). The bottom left section should look familiar to you. It is the R console that we just looked at, with the information on the R version and the command prompt. The top left window currently looks just blank. This is the script editor and we are going to use it next.

### 2.2.3 Starting an R script

An R script is a collection of R commands and we are going to type these commands into the script editor, i.e. the top left window in RStudio. You will see that writing something here and pressing enter doesn't run any code; it just jumps to the next line, like in any other text editor. That's exactly what the script is for. We can write several lines of code and run them all at once when we are ready. But before we let you loose on R commands, we introduce you to another important aspect of scripts: annotation (comments). Annotation contains information for you (not for R), to remind you what a particular R script does. Any information you don't want R to read and process should be preceded by a #. When R sees a #, it ignores the rest of the line and does not attempt to do anything with it.

It is good practice to start every script with some annotation – information that helps you remember what the script does. In the script editor (top left window), type something like this (don't worry about running the code just yet):

```
# Amazing R User (your name)
# 1 March (today's date)
# This script contains some commands I learned in STA5075Z
```

You see there is no output for this code *chunk*. That is because even if we run the code, R wouldn't do anything with it, because of the # signs. It will interpret this as normal text, not commands.

Now save this script to the folder you created earlier ("MySTA5075Zanalyses" or whatever you called it) using File → Save. Call it *day1script.R* (or something sensible). Avoid using special characters (#, $, %, &, *, +, etc) or spaces in the name. This might cause trouble down the line. The file is just a plain text file but giving it the extension '.R' will tell your operating system to open this file in RStudio in the future.

### 2.2.4 Setting a working directory

Next, we need to be able to tell R where to look for our data and where to save any files. The location where files are saved on a computer is called a *path*. When you want to either save a file or access an existing file, R will look in the path that is set as the working directory. To find out where R is currently

looking, we can use the function `getwd()`. Now you can type the following lines of code (still don't run the code, just type as if it was normal text):

```
# Find out what directory R is working with
getwd()
```

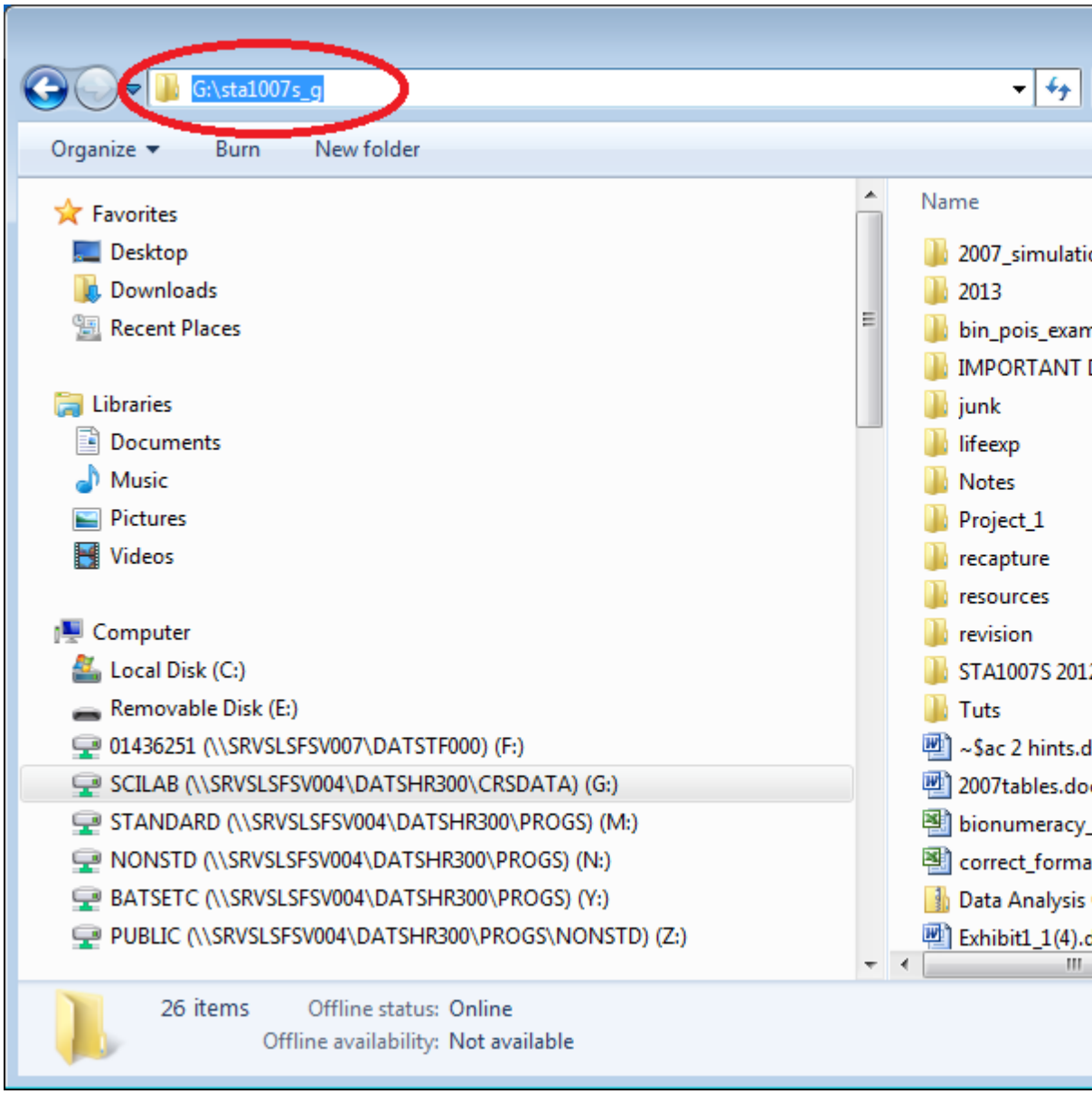
Everyone might get a slightly different output from this function. The name of the function `getwd` stands for *get working directory*. This is the *path* on your computer where R is currently expecting to get things from. It is usually not the location where your files are actually stored!

Luckily, we can change it easily with the function `setwd()`, for *set working directory*. Inside the round brackets, you need to give R the path to your folder. There are easy ways to get the path without having to type it. Mistyping the path is one of the most common (and most frustrating) problems when you start using R. So how do you find the path to a folder or file on your computer?

---

1. Open Windows Explorer and navigate to the folder you created at the beginning of the lab "MySTA5075Zanalysis" or similar.
2. Windows Explorer will display the path in the title bar when you click on the left corner of this bar (see figure below).
3. You can highlight the path in the address bar with your mouse and copy it with `ctrl-c`.

---




In Windows, the path starts with a drive letter and the folders are then separated by backward slashes (e.g. C:\user\mySTA1007analyses). R conforms to the more general convention (shared e.g. with Mac and Unix operating systems) of separating the folders with a forward slash (e.g. C:/user/mySTA5075Zanalyses). So, when you copy the path from Windows Explorer, you will need to change the backslashes (\) to forward slashes (/).

You can do this now and type the following command in your R script. Sub-

stitute the three dashes --- by the correct path to your working directory and enclose your path in double quotes:

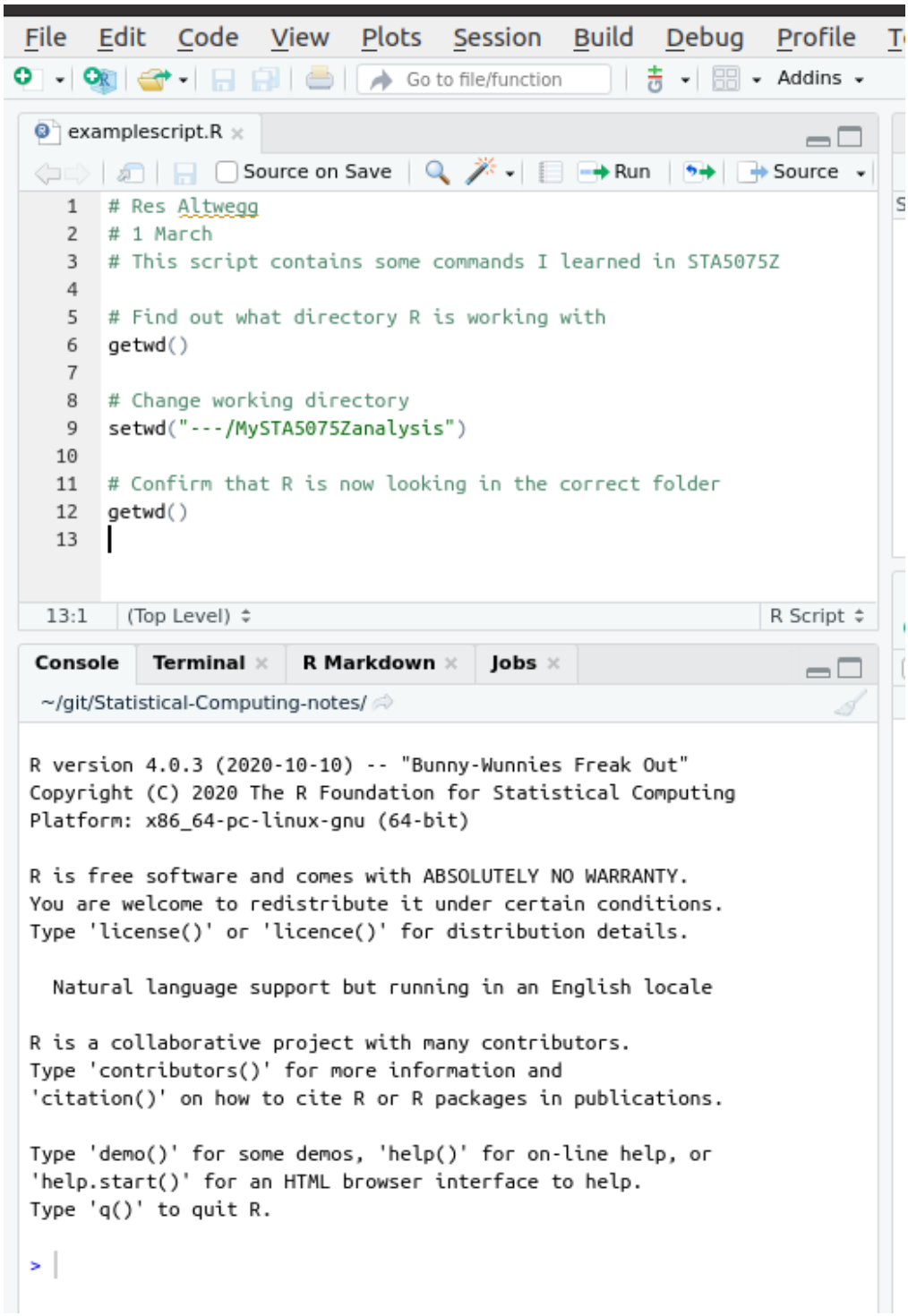
```
# Change working directory to folder in which you want to store your analysis
setwd("---/MySTA5075Zanalysis")
```

You can then again use:

```
# Confirm that R is now looking in the correct folder
getwd()
```

to verify that R is now looking in the correct place.

Your script should now look something like this:


File Edit Code View Plots Session Build Debug Profile
Go to file/function
Addins
examplescript.R
Source on Save
Run
Source
1 # Res Altwegg
2 # 1 March
3 # This script contains some commands I learned in STA5075Z
4
5 # Find out what directory R is working with
6 getwd()
7
8 # Change working directory
9 setwd("---/MySTA5075Zanalysis")
10
11 # Confirm that R is now looking in the correct folder
12 getwd()
13
13:1 (Top Level) R Script
Console Terminal R Markdown Jobs
~/git/Statistical-Computing-notes/
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 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.
>

Make sure you save the script file frequently. Together with the raw data, this is the most important file in your R-life.

Alternatively, you can also use the RStudio drop down menus to set the working directory. To do this click on the "Session" option and then select "Set Working Directory" followed by "Choose Directory" and then browse to the location that you want.

We are shortly going to look at R projects where RStudio remembers the correct working directory. Ultimately, when you are working as a data scientist, it is important that your code is portable to other computers and then it will be better to use a method that does not rely on 'hardcoded' setting of the working directory – on a different computer, the path will be different. However, it is important to know how to check and set the working directory. R not finding your files is one of the most frequent sources of problems when you are new to R.

### 2.2.5 Running your R script

OK, so now we've created a script, but we still need to run it! In the console window (the bottom left hand one), you can see the prompt ('>' and a cursor flashing after it). Now go back to the script window (top left). Select the first few lines of your script (the ones starting with #) with your mouse. This should highlight them. Then click on the 'Run' button just above the top right corner of the script window. Now look at the console window again. RStudio has sent the highlighted lines of code to the R console, which returned the text to you and effectively ignored the message. We know this because R did not return any errors and is showing the prompt again. Since these lines contained only annotation, this is what we had expected. R is ready for the next commands.

Now select the line of the script with the `getwd()` command, run it and look at the console. You've asked R to tell you where it is looking, and it returns the path that it currently uses. Now run the line with `setwd()` to change where R is looking. You can check that it has done this by running the `getwd()` again.

You are getting a feeling for how running scripts works in R.

### 2.2.6 RStudio projects

We can anticipate that dealing with working directories might cause some trouble. This is especially true when we are working on multiple computers and we want our scripts to work smoothly on all of them. So, what should we do when our scripts are in different directories on different computers? Or in that odd situation where we are running scripts from a flash drive? We can always change our working directory depending on the situation but even then, other links in

our script might be broken. It turns out that RStudio, once again, comes to the rescue. The solution to our problems in this case is RStudio projects.

The idea is simple, you create a project in the folder you are going to be working on and this folder will be your root directory from now on. This means that at the moment you open a project, R will immediately switch the working directory to this root directory. This also means that when you want to refer to any file, you can just ignore all the path up until the root directory. This makes your code much more compact and clear. And not only this, if you move your project folder to a different location or even to another computer, the root directory will still be the same as long as the project files are in that folder. This might sound a bit confusing, but it is actually quite simple.

---

1. Click on File → New Project. Alternatively, you can click on the Projects tab (see figure below) on the top right corner of your screen → New Project.

2. Then click on New Project → Existing directory

3. Now browse to the folder you created at the beginning of the lab, "MySTA5075Zanalyses" or something similar.

4. Once you are in that folder, click "Create project".

---

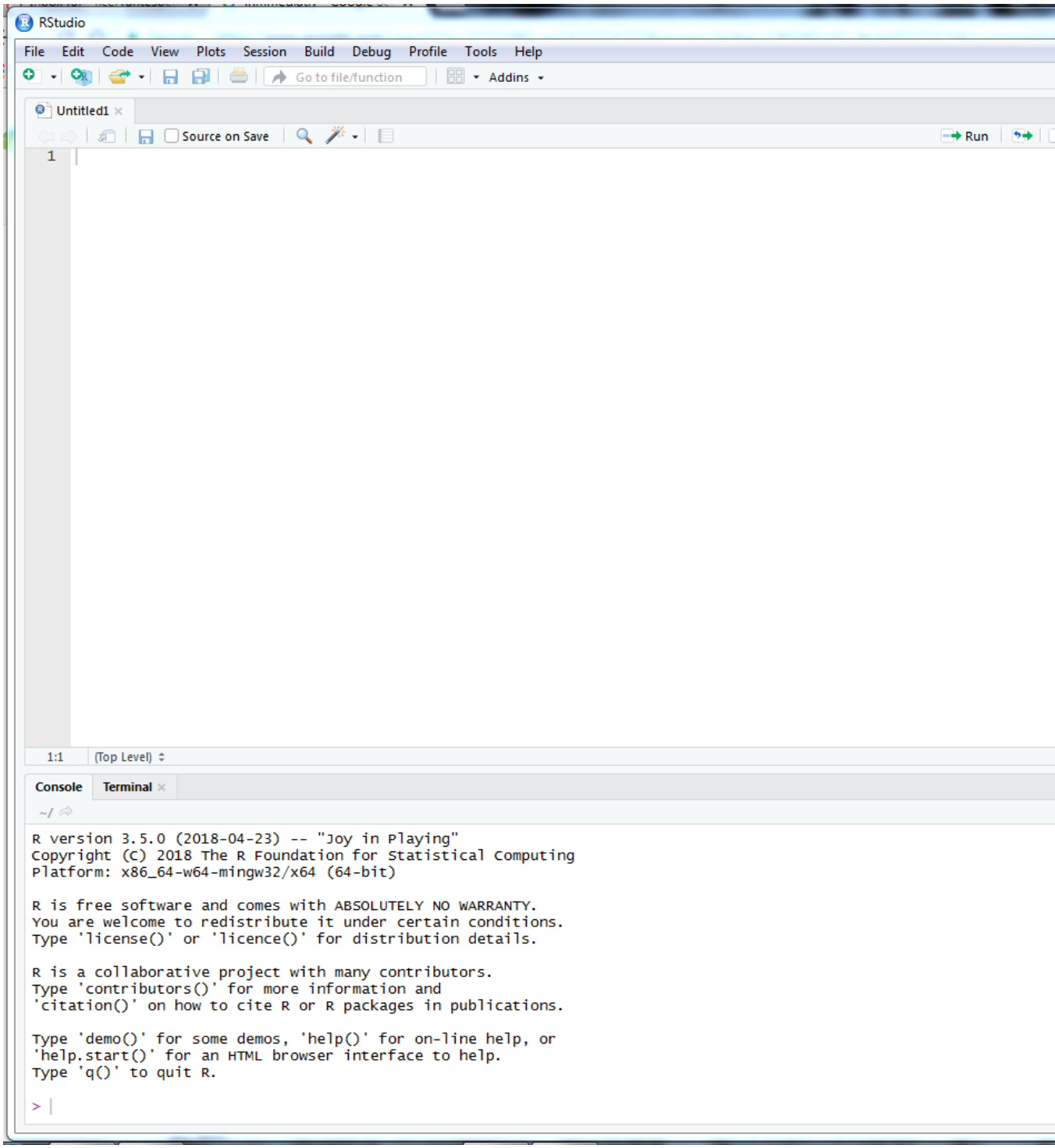
Now, if you go into your "MySTA5075Zanalyses" folder you will find a new R Project file. This is all you need. From now on, whenever you want to work on STA5075Z pracs you can open your project by:

---

1. Click on File → Open Project… Alternatively, you can click on the Projects tab (see figure below) on the top right corner of your screen → Open Project…

2. Browse to your project folder and double click in the R Project file.

---


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Untitled1
Source on Save
Run
1
1:1
(Top Level)
Console
Terminal
~/
R version 3.5.0 (2018-04-23) -- "Joy in Playing"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (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.
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.
>

R will automatically load your project as you left it last time and it will make the project folder the root directory. This means that R will first look for any data in this folder and will save any files or objects in this destination as well. You may copy and paste the project folder anywhere on your computer or even on a different computer and R will will make the new destination the root directory.

### 2.2.7 R packages

All R functions live in packages. For example, the `getwd()` function we met earlier is part of the `base` package that comes with the basic installation. To use this function, all we had to do is type its name. The same is true for any function that is part of a package that is considered standard.

Some packages have to be loaded first before you can use the functions contained in them. Perhaps somewhat confusingly, you load a package using the command `library()`. E.g. let's say we want to use the function `lme`. This function is part of the `nlme` package that comes with the base installation but is not automatically loaded. So we first have to load the package (note use of quotation marks!):

```
library("nlme")  # load package 'nlme'
?lme  # use one of the functions in that package
```

And then there are the >17,000 contributed packages on CRAN plus lots of packages available from other sources. If you need a function that is in a package that doesn't form part of the base installation, you first have to download and install that package. You have several options to install a package that is on CRAN. One option is to go to the 'Packages' tab in the lower right-hand panel of RStudio. Clicking on that tab gives you a list of packages that are currently installed. If you can't find the package you are looking for, click on the 'Install' button and RStudio will prompt you to select a mirror and type in the package name. Make sure you have the box 'Install dependencies' ticked; this will also install all the packages that are needed by functions in the package you want to install.

You can also use commands. E.g. to download and install package `MASS`, you can run the following command:

```
install.packages("MASS")
```

You only need to do this step once! From now on, the package `MASS` is available on your computer. However, each time you re-start R, you have to load the package as we saw earlier:

```
library("MASS")  # load package 'MASS'
?glm.nb  # use one of the functions in that package
```

## 2.3 Resources

When you start working with R, the learning curve is steep. So here are some useful references. Look up quickly now and remember to consult these when you need more detailed information on any aspect of R.

- Type `help.start()` into your console. This will bring up a list of manuals and other useful information.
- 'Getting Started with R - FAQ.pdf' (in the Resources / Reading folder on Vula) has a brief compilation of useful information.
- The Quick-R web site presents a lot of useful information in an easily accessible way.
- Roger Peng's R Programming for Data Science is an excellent online book.
- Some programming tutorials.
  - http://cran.r-project.org/doc/manuals/R-intro.html provides examples to work through.
  - Lumley's course http://faculty.washington.edu/tlumley/Rcourse/ is also good.
  - http://zoonek2.free.fr/UNIX/48_R/02.html more than you need now but may be useful later.
- The RStudio IDE cheat sheet is extremely handy to have ready for quick reference
- RStudio has a number of Cheat Sheets on various aspects of R and RStudio here. You can also find them in RStudio under 'Help → Cheatsheets'.
- As a data scientist working in R, make sure you develop a good coding style from the start. Here is a useful style guide.
- The 'tidyverse' is a set of R packages that follow a particular approach to data manipulation, visualisation and analysis. This book is a great introduction. Even though we are going to stick mostly to base R in this module, you will sooner or later be exposed (and hopefully get into) the tidyverse.

## 2.4 Working with R – some basics

### 2.4.1 Object orientation

R is an object-oriented language. It stores data, functions, and output from data analysis (as well as everything else) as objects. Things are assigned to and stored in objects using the <- operator. For example:

```
x <- 4 # this creates a numeric vector of length 1, holding the number 4
y <- "Hello" # this creates a character vector
x
```

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

```
y
```

```
## [1] "Hello"
```

You can also use = as assignment operator. It is often used because it is shorter than <- but using the latter is better style because it allows you to reserve = for those cases where it means 'equal to'.

A list of all objects in the current session (local level environment) can be obtained with:

```
ls()
```

Each object has attributes that depend on it's class and R functions can do different things to objects belonging to different classes.

```
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "character"
```

### 2.4.2 Data types (types of values)

So let's first look at the classes of data that R uses:

- The class 'numeric' is reserved for numbers. Real numbers are stored in double precision format and have class 'double' (which is identical to 'numeric'). Other number formats are 'integer' and 'complex'.

```
is.integer(2)           # double
```

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

```
is.integer(2L)          # integer
```

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

- Character stings (e.g. text) is stored as class 'character'. For statistical analyses, it is often necessary to convert a character vector to class 'factor'. A factor variable is a character variable that can only assume a given number of values. These are called levels. E.g. a variable of country names could have levels "South Africa", "Namibia", "Botswana", etc.
- Values can also be of class 'logical' (FALSE, TRUE, F, T).
- Dates are a particular type of numeric variable and can sometimes need a bit of wrangling to get right. Have a look at the overview on date values on QuickR.

These are the main data types you will encounter.

### 2.4.3 Variables/objects and the assignment operator <-

Out of these values, vectors, arrays, lists can be built (data objects).

Run the following code and explore what it does.

```
x1 <- 49
sqrt(x1)

x2 <- "South Africa"
x2

x3 <- x1 == 5       # does x1 equal 5?
x3

x4 <- date()
```

- Of what data type are the above?

We saw a bit earlier that the function `class(x)` returns the class of object `x`. R has the very useful function `str()`. Try it out on the objects you just created above. What does it do?

### 2.4.4 Constants

R has a number of built-in constants. Run the following code and explore what it does.

```
pi

FALSE               # logical constants

val1 <- TRUE
val1

val2 <- F
```

### 2.4.5 Functions

So we've had a look at R objects holding different types of information. The other main type of object in R contain functions. Functions take *arguments*. E.g. what is $\sqrt{9}$?

```
sqrt(9)
```

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

You can write you own functions in R. So, for example, let's write a function that adds 1 to the value supplied as argument:

```
x.plus1 <- function(x) {
    return(x + 1)
}
x.plus1(3)
```

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

Note that we use round brackets around the *arguments* and curly brackets around the function instructions. Brackets of different types are used in specific situations in R and it is important to use the correct type of brackets.

### 2.4.6 Miscellaneous

A few more things you need to know about R:

- R is very case sensitive and this can easily trip you up. E.g. SAM, sam, Sam are three entirely different things to R.
- names of R objects must start with a letter or '.', can contain any number, letter, '.', '_', e.g. model.residuals, model2, x, X, x2, house.number.bedrooms are all valid names.
- NA: missing value (not available)
- NaN: not a number (arithmetically undefined)

### 2.4.7 Help

R has built-in help files for all functions and these help files are all organised in the same way: they give a brief description of what the function does, then some information on how it is used and what kind of arguments it accepts. Then there is some more detail and finally at the end some examples. The helpfiles may look a little hard to read at first but once you get used to their format, they are really helpful.

If you get stuck, the help files should always be your first port of call.

Have a look at the help files for the `rnorm` function and the `&` operator:

```
?rnorm
?"&"
```

### 2.4.8 Comment your code!

And just to remind you: always comment your code liberally. Your code needs to be readable both by machines and humans. While the former is obvious, it is easy to neglect the latter. Your comments need to be detailed enough so that someone else (or you, in a few months time when you prepare for the exam!) can understand what your code is doing without having to pick through individual code bits. E.g:

```
## generate 1000 values from N(0,1)
x <- rnorm(1000)        # note round brackets
```

At some point, you may have more comments than code and we are going to look at another way of combining text and code shortly, with R Markdown.

## 2.5 R Markdown

R Markdown is a fantastic way to integrate data analysis with report writing. R Markdown allows you to embed R code and all the output it generates into a text environment. Results such as tables and figures are created each time you run the code. It is therefore easy to update a report should the data change. Writing a report with R Markdown also makes the data workflow transparent and reproducible since you can examine to code to see exactly what was done. R Markdown also allows you to produce your report in different formats, such as pdf, html, Word, etc. You can use it to make slides, a text report, blogs, websites, dashboards and many more things. You can use Latex commands, for example to insert equations.

To get started, have a look at the RStudio Markdown guide and click on 'Get started'.

Again, the best way to learn is by doing and we therefore would like you to **create your own handbook for this course** using R Markdown. To help you get started, I have uploaded an R Markdown file – SC1_basics_vectors.Rmd – to the Resources → RMarkdown folder on the course Vula site. This file will create a set of slides (also uploaded on Vula for your reference: SC1_basics_vectors.pdf) that represent the content of these notes. You should be able to reproduce the pdf from the Rmd file and the figure files also provided in the Vula folder.

Keep the R Markdown cheatsheet handy.

Experiment with the **output:** I used beamer_presentation but try html_document, pdf_document, word_document … Then personalise the document with your own R code, output, comments and annotations. Get a feeling for the basics but don't spend too much time on this yet. You will learn more as we go.

For details, consult: Yihui Xie et al. *R Markdown: The Definitive Guide.*

## 2.6 Data and calculations in R

### 2.6.1 Basic Calculations

We have already seen how R can do basic calculations. Here are a few more examples.

```
2 + 3
2 - 3
2^3
sqrt(3)
```

```
log(10); log10(100)
2 - 3 * 3
pi
exp(2)
```

## 2.7 Data Structures: Vectors

$$x_1 = (3 \;\; 5 \;\; 6 \;\; 7 \;\; 1)$$

In statistics: measurements on a variable are typically stored in a vector, e.g. here $x_1$ could be age of 5 children.

There are different ways to create a vector of numbers in R. We first use the concatenate (`c()`) function to read in the values above. Then, we look at a few ways of generating sequences of numbers.

```
x1 <- c(3, 5, 6, 7, 1)   # c for combine/concatenate
x2 <- 1:10

## sequence from 0.5 to 2.5, step size 0.5
x3 <- seq(0.5, 2.5, 0.5)
x3b <- seq(from = 0.5, to = 2.5, by = 0.5)   # equivalent

## sequence from 0.5 to 2.5, length 100
x4 <- seq(0.5, 2.5, length = 100)
x5 <- rep(0, times = 4)

## Repeat each of 1 and 2 three times. Note that the first
## argument has to be a single object, hence the use of
## c() to build a vector.
x6 <- rep(c(1, 2), each = 3)
```

Note the way in which R prints vectors to the screen, e.g.:

```
x1
```

```
## [1] 3 5 6 7 1
```

Then examine each of the vectors you created above using the following functions:

```
str(x1)       # summary of data type, length, structure
class(x1)
typeof(x1)
is.numeric(x1)
is.logical(x1)
is.integer(x1)
```

### 2.7.1 Calculations with vectors

R can do calculations on vectors. This is one of the most powerful features of R but it can also have unexpected side-effects if you are not careful. Explore what each of the following lines of code does, exactly!

```
x1 + x1               # elementwise
x1^2
x1 == 5               # returns logical vector

x1 + x2               # recycles shorter vector

length(x1)
sum(x1)
max(x1)
min(x1)
prod(x1)
mean(x1)
```

### 2.7.2 Indexing/subsetting a vector

We often need to work with subsets of data and so it is important to be able to extract parts of vectors. This is where indexing comes in. Each element in a vector is in one particular position and this position is labeled by the index. We can subset vectors by indicating which positions in the vector we want to extract, using square brackets.

```
x1[1]         # first element of vector
x1[-1]        # all but first element
x1[1:3]       # first 3 elements
x1[c(4, 5)]   # 4th and 5th

## which elements = 5, returns indices/positions
ind <- which(x1 == 5); ind
x1[ind]
```

```
x1[x1 > 5]   # only keep elements > 5

## change values
x1[3] <- 4; x1
```

For some of the lines in the code above, it helps to just run part of the line. E.g. check what you get when you type `c(4, 5)` into your console. Is it the same as `4:5`? What are you going to get if you write `x1[4:5]`. What happens if you run `which(x1 == 5)` or `x1 > 5`? R allows you to carry out several sequential calculations in a single line of code. When trying to understand such code, the trick is to 'work from inside out', i.e. first run bits of code that are inside brackets surrounded by other commands.

## 2.8 Prac 1

Open *SC1 basics vectors.Rmd* in RStudio (this file is in the Resources → RMarkdown folder on Vula). Go to Prac 1. We deliberately give you the solutions here so you can check that your calculations are correct. The goal of this prac is to write the R code that does these calculations. We want to see your code, however: submit it via the Vula Assignment 'Prac 1' where you can upload your R script file (containing just the R code you used to solve the problems below).

1. Calculate log(i + 1) for i = 0 to 100. The average of these values should = 3.647074.
2. Generate 10000 random values as follows:

```
set.seed(20230130)   # set starting point for random number generator

## generate 10000 values from an exponential distribution, mean = 1
y <- rexp(10000)
```

3. Find the largest number and its position. (Answer: 9.131717, 9293)
4. How many values are > 2 (absolute and %)? (Answer: 1333, 13.33%)
5. y2: select every 2nd element of y, starting from 1st.
6. y3: replace values > 3 in y2 with 3. The average of these values should = 0.9413817.
7. Find the variance of y3 using vector operations. Check with `var()`.

As a reminder, the variance of a sample of numbers $y$ of length $n$ is:

$$var(y) = \frac{\sum(y_i - \bar{y})^2}{n-1}$$

# Chapter 3

# Matrices and data frames

In this session, we will learn how to work with rectangular data in R. With rectangular data, we mean data that are stored in a two-dimensional array, i.e. we have rows and columns. In R, rectangular data are mainly organised in two basic classes:

- **Matrices** are rectangular data sets that only contain numeric data. We can do calculations on these using **matrix algebra**.
- **Data frames** are rectangular data sets where different columns can have different data types. Typically, each row represents one observation and the columns hold the values for the different variables that were measured during that observation.

## 3.1 Matrices

In statistics, we often work with matrices. One example is in regression analysis.

As a reminder, in a simple linear regression, we have measured two things on each observation and we want to predict the value of one measure from the observed value of the other. For example, the R data set `cars` contains observations on the speed of cars and how long it took them to break (distance).

```
data(cars)
head(cars) # print the first few observations
```

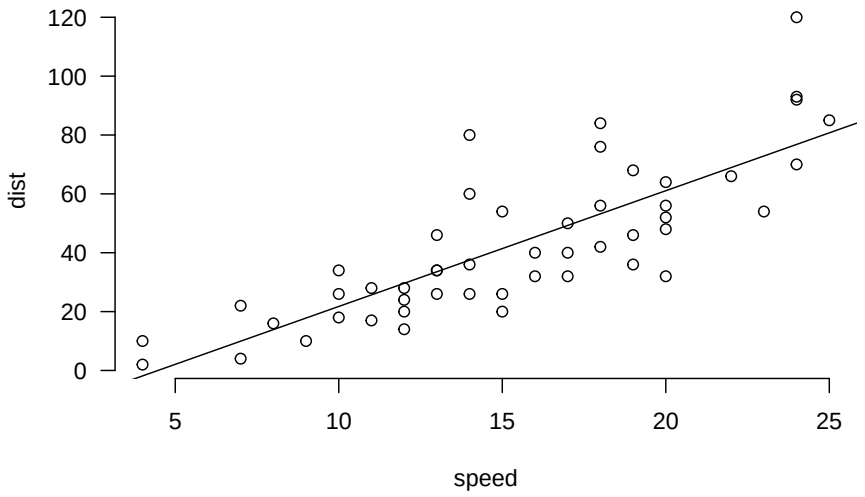
```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

```
## 4      7    22
## 5      8    16
## 6      9    10
```

Type `?cars` into your R console to get more info about this data set.

We want to predict breaking distance from speed. We call distance the response (or dependent) variable and speed the explanatory (or independent) variable.

Here is a scatterplot of the two variables, with the fitted regression line added:




The line in the plot above is the best-fitting regression line. It shows the distance we predict for all values of speed. The fitted regression model is:

$$y_i = \beta_0 + \beta_1 x_i + e_i, \qquad e_i \sim N(0, \sigma^2)$$

where $y_i$ is the distance measured in the $i^{th}$ observation and $x_i$ is the speed measured for that observation. The $e_i$ are the errors or residuals, i.e. the difference between the predicted value and the actual observed value for each distance measurement. The residuals are assumed to follow a normal distribution with a mean of 0 and variance $\sigma^2$. The model has two parameters: the intercept ($\beta_0$) and the slope ($\beta_1$). How do we know what values these parameters should have for the model to provide the best fit to the data?

You might have guessed it: this is where calculations on matrices come in. How so?

So, for example, for the first observation, we have (compare to the box where the first few rows of the data set are printed):

$$2 = \beta_0 + \beta_1 4 + e_1$$

For the second observation, we have:

$$10 = \beta_0 + \beta_1 4 + e_2$$

and so on…

You can see that we actually have a set of equations and thinking back to your linear algebra classes, you will realise that we can write the regression model in matrix notation:

$$Y = X\beta + \varepsilon$$

where $Y$, $X$, $\beta$ and $\varepsilon$ are as follows:

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \cdot \\ \cdot \\ \cdot \\ Y_n \end{bmatrix}; \quad \mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & x_n \end{bmatrix}; \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}; \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \cdot \\ \cdot \\ \cdot \\ \varepsilon_n \end{bmatrix}$$

In the $X$-matrix, also called *design matrix*, we have one column for every explanatory variable. The first column of 1's is for the intercept, the second column is the value of the explanatory variable (speed in this case) for observations 1 to $n$.

We can apply some clever calculations to these matrices to obtain the estimates for $\beta$. We will do these calculations as a prac a bit later today, after we have learned how to work with matrices in R.

The aim of this little detour was 1) to remind you about regressions, and 2) to motivate you to learn about matrices in R. So here goes…

### 3.1.1 Creating Matrices in R

Before we can do any calculation on matrices, we first need to know how to create matrices in R. There are a number of ways to do that:

1. Convert a vector into an n x m matrix. The following code will create a $4 \times 4$ matrix. Note that the matrix is created column wise, i.e. column one is filled up first, etc.. You can change this by specifying `byrow = T`.

```
numbers <- 1:16
m1 <- matrix(numbers, nrow = 4)
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

2. With `cbind()` or `rbind()`: binds vectors column- or row-wise

```
numbers1 <- 1:4
numbers2 <- 11:14
m2 <- cbind(numbers1, numbers2)  # column-bind
m2
```

```
##      numbers1 numbers2
## [1,]        1       11
## [2,]        2       12
## [3,]        3       13
## [4,]        4       14
```

```
m3 <- rbind(numbers1, numbers2)  # row-bind
m3
```

```
##          [,1] [,2] [,3] [,4]
## numbers1    1    2    3    4
## numbers2   11   12   13   14
```

3. A diagonal matrix is a matrix that has non-zero elements only in the main diagonal, and the identity matrix $I$ is the special case where all elements in the main diagonal are 1 and all other elements are 0:

```
diag(c(0.1, 0.2, 0.7))
```

```
##      [,1] [,2] [,3]
## [1,]  0.1  0.0  0.0
## [2,]  0.0  0.2  0.0
## [3,]  0.0  0.0  0.7
```

```
diag(4)          # 4 by 4 identity matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

### 3.1.2 Calculations with matrices

Let's look at how to do the common matrix calculations in R. If you are a bit rusty on matrix algebra, read through this excellent brief overview (the pdf is also in the Resources → Reading folder on Vula).

Transposing a matrix means interchanging rows and columns. The **transpose** of a matrix $X$ is denoted $X'$ or $X^T$:

```
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

```
t(m1)                # transpose of m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

We often need to know what the **dimensions** of a matrix are. The code below returns the number of rows and the number of columns of matrix `m1`. Try this code with a matrix that isn't square, i.e. the number of rows is different from the number of columns!

```
dim(m1)              # 4 by 4 matrix
```

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

We often need to multiply matrices. See what happens if we just use the regular multiplication operator *:

```
m1 * m1          # element-wise multiplication
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   25   81  169
## [2,]    4   36  100  196
## [3,]    9   49  121  225
## [4,]   16   64  144  256
```

```
m1 * 2           # multiplication with a number
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2   10   18   26
## [2,]    4   12   20   28
## [3,]    6   14   22   30
## [4,]    8   16   24   32
```

R multiplies the matrices element by element (or if we use a single number instead of one of the matrices, it multiplies each element by that number following the general R logic that the shorter vector is recycled to match the length of the longer one).

Regular **matrix multiplication** $XY$ works like this:

```
v1 <- rep(1, times = 4)
v1 %*% m1         # left multiplication with a vector
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   26   42   58
```

```
m1 %x% v1         # right multiplication with a vector
```

```
##       [,1] [,2] [,3] [,4]
##  [1,]    1    5    9   13
##  [2,]    1    5    9   13
##  [3,]    1    5    9   13
##  [4,]    1    5    9   13
##  [5,]    2    6   10   14
##  [6,]    2    6   10   14
##  [7,]    2    6   10   14
##  [8,]    2    6   10   14
```

```
##  [9,]    3    7   11   15
## [10,]    3    7   11   15
## [11,]    3    7   11   15
## [12,]    3    7   11   15
## [13,]    4    8   12   16
## [14,]    4    8   12   16
## [15,]    4    8   12   16
## [16,]    4    8   12   16
```

```
m1 %*% m1          # matrix multiplication
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   90  202  314  426
## [2,]  100  228  356  484
## [3,]  110  254  398  542
## [4,]  120  280  440  600
```

Remember that $XY \neq YX$ in general and if you want to multiply two matrices $A$ and $B$, the number of columns in $A$ needs to be equal to the number of rows in $B$. If these things sound unfamiliar, brush up on your matrix algebra by reading the overview mentioned above!

The **inverse** of an $n \times n$ matrix $X$ is a matrix $X^{-1}$ that gives the identity matrix $I$ when multiplied by $X$.

$$XX^{-1} = X^{-1}X = I$$

```
m1 <- matrix(1:4,nrow = 2); m1
```

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

```
solve(m1)          # inverse of m1
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

```
solve(m1) %*% m1
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Why are we using the syntax 'solve'? Matrices are closely related to systems of linear equations. Look at the two equations

$$x_1 + 3x_2 = 7$$

$$2x_1 + 4x_2 = 10$$

We can write this system of equations in matrix form:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 10 \end{bmatrix}$$

i.e. $Ax = b$

We want to solve for $x$. Since $A^{-1}A = I$ and $Ix = x$, we can multiply both sides with $A^{-1}$ and obtain:

$$x = A^{-1}b = \begin{bmatrix} -2 & 1.5 \\ 1 & -0.5 \end{bmatrix} \begin{bmatrix} 7 \\ 10 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Check that I have calculated $A^{-1}$ correctly (using R)!

In R, these calculations are:

```
A <- matrix(c(1, 2, 3, 4), ncol = 2)
b <- c(7, 10)
x <- solve(A) %*% b
x
```

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

Finally, R has easy functions to calculate means and sums on rows and columns of matrices `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()`, e.g.:

```
rowSums(m1)
```

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

```
colMeans(m1)
```

```
## [1] 1.5 3.5
```

### 3.1.3 Subsetting/indexing matrices

We saw earlier how to subset vectors using square brackets `[]`. Subsetting matrices is similar except that we now have two dimensions, rows and columns, so we need two sets of indices. In R, rows always come first and then columns and the indices are separated by a comma. A blank returns all elements of the corresponding dimension.

```
m4 <- matrix(1:12, nrow = 3); m4
```

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

```
m4[2, 3]    # element at row 2, column 3
```

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

```
m4[2, ]     # row 2
```

```
## [1]  2  5  8 11
```

```
m4[, 3]     # column 3
```

```
## [1] 7 8 9
```

```
m4[-1, ]     # all except row 1
```

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

## 3.2 Prac 2

There are 2 trees in the middle of the Kalahari. On each tree birds of unknown species are sitting and feeling very hot. A bird from the first tree says to those on the second tree: "Hi – if one of you come to our tree then there will be the same number of us on each tree". "Yeah, right", says a bird from the second

tree, "but if one of you comes to our tree, then we will be twice as many on our tree as on yours".[1]

**Question:** How many birds are on each tree? More specifically:

- Write up two equations with two unknowns.
- Solve these equations using the methods you have learned above.
- Simply finding the solution by trial–and–error is considered cheating.

## 3.3 Logical operators in R

We have already met some R operators. An important set of operators are the logical operators:

`!` not

`&` and

`|` or

`==` is equal to (a test)

```
y <- c(1, 2, 3, NA, NA, 4, 5)

is.na(y)
```

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

```
!is.na(y)
```

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

```
y > 1 & y < 2
```

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

```
y > 1 | y < 2
```

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

[1] adapted from https://www.math.uh.edu/~jmorgan/Math6397/day13/LinearAlgebraR-Handout.pdf

You can see that logical operators generally return `TRUE` or `FALSE`, the two values of the data type 'logical'.

Logical operators are very useful for subsetting. The following works:

```
y[!is.na(y)] # pick all non-missing elements of y
```

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

Internally, `TRUE` is stored as 1 and `FALSE` as 0. This is quite handy. E.g. how many missing values does the vector `y` have?

```
sum(is.na(y))
```

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

## 3.4 Data frames

Data frames are the main way data is stored in R and it is in this format that most R functions for statistical analyses expect the data to be in. Data frames are like spreadsheet; they have rows and columns. Usually, the columns have labels (column names). The best way to set up a data frame is generally so that each row is an observation (case), each column is a variable (*tidy data*: read Hadley Wickham's excellent short paper on data format and tidy data). Data frames are similar to matrices, but columns can be of different data types.

Here is an example data frame that comes with R:

```
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Another example, from Gapminder

```
library(gapminder)
str(gapminder)
```

```
## tibble [1,704 x 6] (S3: tbl_df/tbl/data.frame)
##  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
##  $ pop      : int [1:1704] 8425333 9240934 10267083 11537966 13079460 14880372 1288
##  $ gdpPercap: num [1:1704] 779 821 853 836 740 ...
```

```
head(gapminder)
```

```
## # A tibble: 6 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
## 5 Afghanistan Asia       1972    36.1 13079460      740.
## 6 Afghanistan Asia       1977    38.4 14880372      786.
```

### 3.4.1 Character vectors

Before we get deeper into data frames, let's have a quick look at character vectors since columns in data frames often consist of these.

Character values need to be supplied to R in quotation marks. For example:

```
filename <- "Rintro.R"    # character vector of length 1
```

In data frames, it is common to have grouping variables in this format, e.g. if your data come from different sites:

```
site <- c("CT", "JHB", "JHB", "DB")
class(site)
```

```
## [1] "character"
```

When we use this type of variable in statistical analyses, we call it a categorical variable and such variables usually need to be in class `factor`. The difference between the classes `character` and `factor` is that the latter has a given (usually small) number of levels.

```
site <- as.factor(site)
class(site)
```

```
## [1] "factor"
```

```
levels(site)
```

```
## [1] "CT"  "DB"  "JHB"
```

Here is another useful way of generating a character variable, for example if you need a vector of labels for the treatments of an experiment:

```
lett <- letters; letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "
```

```
treat <- rep(LETTERS[1:3], each = 3); treat
```

```
## [1] "A" "A" "A" "B" "B" "B" "C" "C" "C"
```

### 3.4.2 Extracting values from data frames

Extracting values from data frames works in a similar way as we saw with matrices. We have rows and columns, and can access specific sets of values using indices:

```
gapminder[1:10,1]  # the first 10 values of the first column
```

```
## # A tibble: 10 x 1
##    country
##    <fct>
##  1 Afghanistan
##  2 Afghanistan
##  3 Afghanistan
##  4 Afghanistan
##  5 Afghanistan
##  6 Afghanistan
##  7 Afghanistan
##  8 Afghanistan
##  9 Afghanistan
## 10 Afghanistan
```

```
gapminder[1,]      # the first row
```

```
## # A tibble: 1 x 6
##   country     continent  year lifeExp     pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8 8425333      779.
```

Since columns in data frames often have names, we can use these to access specific variables (the code below produces a lot of output, so I'm not going to print it here in the notes). Variables can be accessed with the $ sign or square brackets:

```
names(gapminder)
gapminder$country  # only the country column, now a vector
gapminder["country"]
```

Often, we need a subset of rows:

```
gapminder[gapminder$country == "South Africa", ]   # only SA rows
```

```
## # A tibble: 12 x 6
##    country      continent  year lifeExp      pop gdpPercap
##    <fct>        <fct>     <int>   <dbl>    <int>     <dbl>
##  1 South Africa Africa     1952    45.0 14264935     4725.
##  2 South Africa Africa     1957    48.0 16151549     5487.
##  3 South Africa Africa     1962    50.0 18356657     5769.
##  4 South Africa Africa     1967    51.9 20997321     7114.
##  5 South Africa Africa     1972    53.7 23935810     7766.
##  6 South Africa Africa     1977    55.5 27129932     8029.
##  7 South Africa Africa     1982    58.2 31140029     8568.
##  8 South Africa Africa     1987    60.8 35933379     7826.
##  9 South Africa Africa     1992    61.9 39964159     7225.
## 10 South Africa Africa     1997    60.2 42835005     7479.
## 11 South Africa Africa     2002    53.4 44433622     7711.
## 12 South Africa Africa     2007    49.3 43997828     9270.
```

```
gapminder[gapminder$year==2007,]                    # only 2007 values
```

```
## # A tibble: 142 x 6
##    country     continent  year lifeExp       pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>     <int>     <dbl>
##  1 Afghanistan Asia       2007    43.8  31889923      975.
```

```
##  2 Albania     Europe     2007    76.4   3600523     5937.
##  3 Algeria     Africa     2007    72.3  33333216     6223.
##  4 Angola      Africa     2007    42.7  12420476     4797.
##  5 Argentina   Americas   2007    75.3  40301927    12779.
##  6 Australia   Oceania    2007    81.2  20434176    34435.
##  7 Austria     Europe     2007    79.8   8199783    36126.
##  8 Bahrain     Asia       2007    75.6    708573    29796.
##  9 Bangladesh  Asia       2007    64.1 150448339     1391.
## 10 Belgium     Europe     2007    79.4  10392226    33693.
## # ... with 132 more rows
```

### 3.4.3 Some basic statistics

**Frequency Tables** are very useful to get a quick summary for categorical variables:

```
table(continent)
```

```
## Error in table(continent): object 'continent' not found
```

Oops, this doesn't work. R doesn't 'see' the variables inside data frames directly. You need to tell R that this variable is part of the `gapminder` object. Here are two different ways that work:

```
table(gapminder$continent)
```

```
##
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
```

```
with(gapminder, table(continent))
```

```
## continent
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
```

There is another way you will come across: using the `attach` function. It can save you some typing but also comes with **severe risks**. One risk is that you end up with several versions of the same variable and accidentally end up using the wrong version in your further analysis without necessarily noticing. We therefore *recommend that you avoid attach*! If you must use it, make sure you detach the data frame again as soon as you are done with whatever you wanted to attach the data frame for!

```
## avoid attach!
attach(gapminder)
table(continent)
```

```
## continent
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
```

```
detach(gapminder)
```

### 3.4.4 Summary Statistics

We have already seen that the `str()` function can be used to get a quick summary of what is in a data frame. Another useful summary is provided by the `summary()` function:

```
summary(gapminder)
```

```
##         country        continent        year         lifeExp          pop
##  Afghanistan:  12   Africa  :624   Min.   :1952   Min.   :23.60   Min.   :6.001e+04
##  Albania    :  12   Americas:300   1st Qu.:1966   1st Qu.:48.20   1st Qu.:2.794e+06
##  Algeria    :  12   Asia    :396   Median :1980   Median :60.71   Median :7.024e+06
##  Angola     :  12   Europe  :360   Mean   :1980   Mean   :59.47   Mean   :2.960e+07
##  Argentina  :  12   Oceania : 24   3rd Qu.:1993   3rd Qu.:70.85   3rd Qu.:1.959e+07
##  Australia  :  12                  Max.   :2007   Max.   :82.60   Max.   :1.319e+09
##  (Other)    :1632
```

This gives us a frequency table for all categorical variables (if that doesn't happen, it may be that the variable is not of class `factor` and the class needs to be changed) and a 5-number summary plus mean for numerical variables.

Numerical variables behave just like numerical vectors:

```
mean(gapminder$lifeExp)
```

```
## [1] 59.47444
```

```
summary(gapminder$gdpPercap)  # 5-number summary
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    241.2   1202.1   3531.8   7215.3   9325.5 113523.1
```

Try out these functions: `sd`, `var`, `min`, `range`, `quantile`. What do they do?

## 3.5 Types of R data objects

Everything in R is an object and we have seen a number of different objects. For our purposes, R objects can be grouped into three main categories: data objects, functions and other objects, such as output from a regression model. This section was mostly about data objects.

**Data objects** include:

- vectors: we have seen that these can hold different data types like numeric, character, etc.. All elements in a single vector need to be of the same type, though.
- matrices: these are two-dimensional data objects holding numeric values
- arrays: these are similar to matrices except that they can have more than two dimensions
- data frames: two-dimensional (flat) data types; the columns can hold different data types
- lists: a list is a data object that can contain elements of different type, e.g. numbers, strings, etc. Elements in a list can also be vectors, matrices or even lists (lists of lists). So this is a very flexible data type.

## 3.6 Prac 3

1. Find all rows in airquality that have missing values.
2. Find mean, sd, min, max for each of temperature and ozone level.
3. For **linear regression**, parameter estimates can be found as follows.

$$\hat{\beta} = (X'X)^{-1}X'Y$$

Here, $Y$ is the response variable, and $X$ is the design matrix.

The `cars` data (an R data set) contains two variables: speed and distance to stop. Fit a simple linear regression model to these data, i.e. find the $\hat{\beta}$ estimates, using the equation above, and matrix calcuations in R.

4. Check that you get the same $\hat{\beta}$ estimates as when fitting the linear regression model using `lm()` in R.

```
m1 <- lm(dist ~ speed, data = cars)
summary(m1)
```

5. Mean life expectancy in South Africa? 53.99317.

# Chapter 4

# Simple graphics

In this section, we look at simple graphics in R. We wil mostly use the graphic capabilities that come with the R base installation because they are relatively easy to use and customize. The tidyverse graphics, especially the `ggplot2` package, are becoming ever more popular. These are very powerful graphics functions and they produce visually pleasing graphs using the tidyverse language philosophy. However, we feel that the `ggplot2` graphs can quickly become a bit tedious if you are not happy with the defaults and want to customise your graphs. We therefore mostly go with the basic plotting functions in this section as we feel they are easier to get started on, especially if you are new to R.

## 4.1 Hans Rosling and Gapminder

We will use data from the Gapminder project quite a bit in this session. Watch the **TED talk by Hans Rosling:** it goes for being one of the best stats talks and is a great example of effective graphics.

We will use the `gapminder` R package for our examples but here is the link for downloading a lot more **data** from the Gapminder project.

The Gapminder project also has **online visualization tools**.

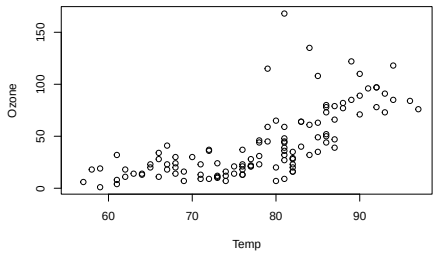
## 4.2 Simple graphics in R:

But first, we are going to look at some simple graphs using the 'airquality' data set that comes with R.

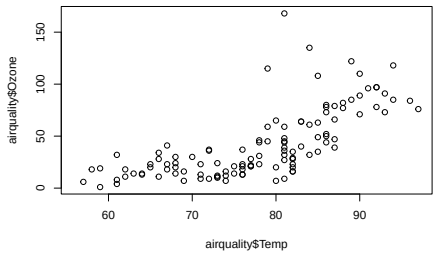
### 4.2.1 Scatter plot

The **scatter plot** is one of the simplest and most useful plots to visualize the relationship between two continuous numerical variables. Think about which variable should be plotted along the horizontal (x) axis and which one should be plotted along the vertical (y) axis. If there is a sense that one variable is dependent on the other (either causally or because you are interested in showing how one variable changes in relation to the other), then the dependent variable should be plotted on the y axis.

```
plot(Ozone ~ Temp, data = airquality)
```




```
plot(airquality$Temp, airquality$Ozone)
```



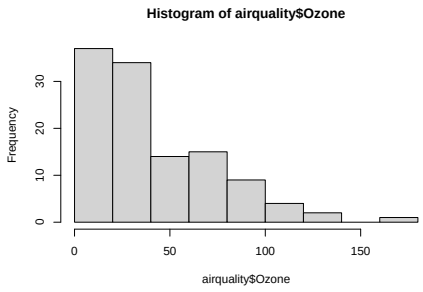

Notice that we used the formula `Ozone ~ Temp` and the `data` statement in the first call to `plot()`. This syntax is used by a lot of R functions (you saw it with regression analysis using `lm()`, for example) and where it works, we recommend using this syntax. The second call to function `plot` is actually using a different `plot` function with a different syntax: we first give the x variable and then the y variable, as separate arguments. This is using the `plot()` function from the `base` package whereas the former used the `plot()` function from the `graphics` package, which is the default for scatter plots. The two functions do exactly the same thing here but the `plot()` function from the `base` package does not have

a `data` argument and we therefore have to give to use the $ notation. Have a look at the help files `?plot`.

### 4.2.2 Histogram

**Histograms** are used to visualise the distribution of numerical variables. The basic idea is that we divide the range of observed values into equal intervals and then count the number of observations that fall into each interval. Then we draw bars whose heights correspond to these numbers. The `hist` function does not have a `data` argument.

```
hist(airquality$Ozone)
```



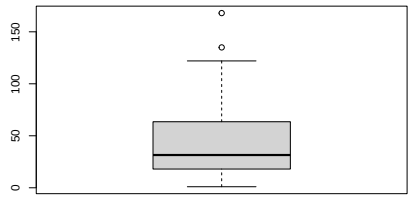

### 4.2.3 Boxplot

A boxplot is a visual representations of the 5-number summary for a numerical variable. It is a summary of the distribution showing the range of 'typical' values (those falling between the first and third quartile) and how far the extremes extend in both directions. It also shows outliers.

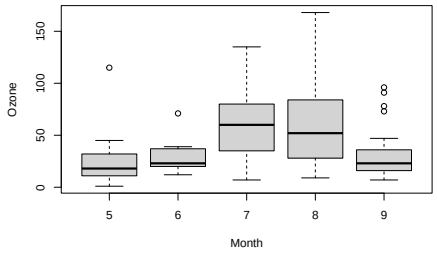
For a single sample, we directly supply the vector of numeric values. If this is a variable in a data frame, we need to give the name of the data frame.

Side-by-side boxplots are also really good to compare the spread and location of several samples (e.g. is ozone generally higher and more variable in certain months?). In this case, we can use the forumla notation together with the `data` statement.

```
boxplot(airquality$Ozone)
```




```
boxplot(Ozone ~ Month, data = airquality)
```




Check the help file `?boxplot` for the two usages. While you are looking at the help file, find out what definition of outlier the `boxplot()` function uses by default and how you could change that (hint: look at the 'range' argument).

## 4.3 Customizing your graphs: layout and fine-tuning

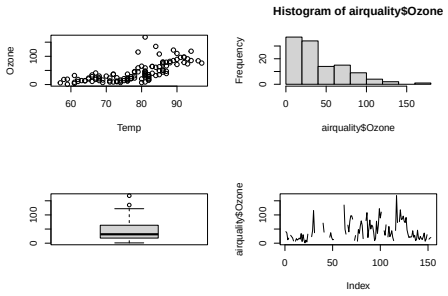
You can see that base R has some plotting functions that allow you to produce some simple plots easily. Often, however, the default plots don't look exactly the way you want them to look like, e.g. they might not look nice enough for a report, etc. Fortunately, these plots are relatively easy to customize.

### 4.3.1 Multiple panels in a single graph

Often, you need a plot that consists of multiple panels. One quick way to do that is by splitting up the graphics window into a matrix of multiple rows and columns as needed. Then, you can call the plotting functions as needed and until all the panels are filled.

We split the window by changing the graphics parameters using the `par()` command. This changes the graphic parameters for the rest of the session, i.e. your graphics window will be split until you change this setting again. Very conveniently, the `par()` function returns the old parameter settings before changing them. I like to save those in an object I called `op` here (for 'old parameters'; name it whatever makes sense to you). Then, I can re-set the graphics parameters very easily when I'm done, with `par(op)`.

```
## split window into 2x2 matrix
op <- par(mfrow = c(2, 2))

plot(Ozone ~ Temp, data = airquality)
hist(airquality$Ozone)
boxplot(airquality$Ozone)
plot(airquality$Ozone, type = "l")   # type = line
```




```
par(op)
```

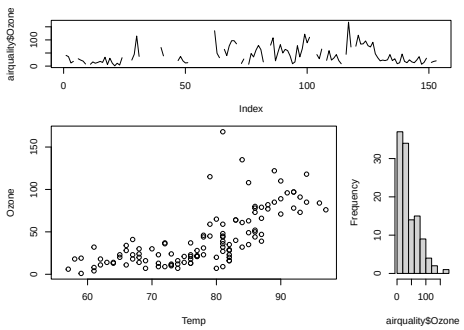
R fills the panels row by row, going left to right. In this example, we set up the window for four panels and created four plots. If we had created a fifth plot, R would have cleared the window and started in the top left corner again.

The fourth plot we created is a time-series plot. This is what the `plot()` function does by default if you supply a single vector of numeric variables. We had to use the `type = 'l'` argument to get a line graph, though. See what happens if you omit this argument. And check out what other plotting options the `type` argument gives you (`?plot`, you want the generic X-Y plotting function).

### 4.3.2 using `layout`

The `layout()` function gives you more flexibility in how to arrange the different panels.

```
par(mar=c(4,4,1,1)) # reduce the figure margins a bit
# One figure in row 1 and two figures in row 2
# row 1 is 1/2 the height of row 2
# column 2 is 1/3 the width of the column 1
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE),
   widths = c(3, 1), heights = c(1, 2))
plot(airquality$Ozone, type = "l")   # type = line
plot(Ozone ~ Temp, data = airquality)
hist(airquality$Ozone, main = "")
```

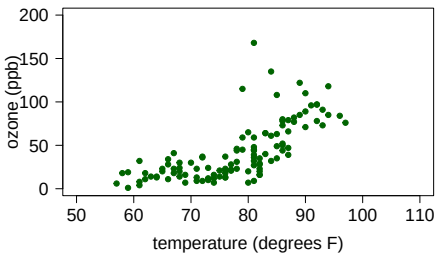



Look at the help file for `layout()` and try out a few different configurations for the plots above, to get a feeling for how this function works.

### 4.3.3 Graphics: fine-tuning

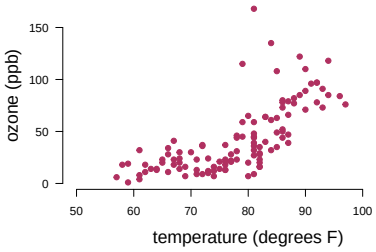
There are a lot of graphics parameters you can (and often might want to) change. Type `?par` into your R console to see what they are, and what options you have for each parameter (in the help file, scroll down to the section 'Graphical Parameters'. You usually don't want to change these parameters for the entire session but only a specific plot. That is done easily enough by supplying them as arguments. In the code below, I've changed some of the settings that often need changing.

```
plot(Ozone ~ Temp,
     pch = 19,          # plotting character
     las = 1,           # horizontal axis labels
     xlab = "temperature (degrees F)",   # x-axis label
     ylab = "ozone (ppb)",
     xlim = c(50, 110),  # limits of the x axis
     ylim = c(0, 200), # limits of the y axis
     cex.axis = 1.5,   # scaling factor for axis values
     cex.lab = 1.5,      # scaling factor for axis labels)
     col = "darkgreen",
     data=airquality)  # colour
```




To make the plot neater, you may want to draw the axes separately:

```
plot(Ozone ~ Temp,
     axes = F, # don't draw axes
     pch = 19,           # plotting character
     xlab = "temperature (degrees F)",   # x-axis label
     ylab = "ozone (ppb)",
     xlim = c(50, 110),  # limits of the x axis
     cex.axis = 1.5,    # scaling factor for axis values
     cex.lab = 1.5,       # scaling factor for axis labels)
     col = "maroon",
     data=airquality)  # colour

axis(side = 1, at = seq(50, 100, by = 10))
axis(side = 2, las = 1) # horizontal axis labels
```




Try out different settings for these graphical parameters, and try out other graphical parameters (e.g. change the margins of the figures). The Quick-R web site gives a useful overview and the reference card by Tom Short is also great as a quick reference.

Work on the plot until you are happy with what it looks like.

## 4.4 More on R Markdown

We covered the basics of R Markdown in section 2.5 hope you are making use of R Markdown to create your own annotated notes for this course. Basic R Markdown (.Rmd) for each section are in the Resources → RMarkdown folder on Vula.

Here some comments on using figures in R Markdown.

### 4.4.1 Chunk options

By now, you should be familiar with the more common chunk options. The R Markdown cheatsheet is a good quick reference. When working with graphs in R Markdown, there are a few things to think about:

- ```` ```{r} ```` R Markdown doesn't force you to name R code chunks and this syntax usually works just fine. If you don't supply a name, however, R Markdown will automatically name the unnamed chunks 'unnamed-chunk-something'. Graphs generated in a code chunk then inherit that name. So, for example, one of my figures is named 'unnamed-chunk-1-1.png'. This can cause some confusion, for example if you add more chunks, the name of this particular figure may change. It usually works all fine but to be safe, it is better to name the chunk that generate figures.
- ```` ```{r chunk1} ```` example of a named chunk. The figure would then be called 'chunk1-1.png' (for example). Obviously, while you are add it, it would be better to use a more informative chunk name instead of 'chunk1'!
- ```` ```{r chunk2, echo = FALSE} ```` when you want to display a figure in your report, you usually don't want to show the code; the option `echo = FALSE` prevents the code in that chunk from being printed.
- `echo = TRUE` if you want to show code in document, set `echo = TRUE`
- `eval = TRUE` the `eval` options determine whether the code in this chunk should be executed or not; for code that produced graphs, this normally needs to be set to `TRUE`, i.e. run the code.

Some chunk options are specifically for graphics:

- `fig.show = "hide"`: don't print figure; use this when you don't want the figure to be printed (yet). For example, you would use this if you want to insert the figures with LaTex commands instead of R code.
- `fig.cap = "Histogram of ozone values."` in a chunk that produces a figure, you can use this option to supply the figure caption.
- `fig.width = 4` with this option, you can control the width of the figure (in inches).

### 4.4.2 Inserting figures

A code chunk that produces a figure can be used to display that figure directly. However, it can be neater and give you more control (e.g. with numbering of figures, etc.) to first write the figure to file (using the option `fig.show = "hide"`) and then use separate code to insert the figure into your text later. Some options to do that:

**Option 1:**

You can use LaTeX code (outside of a code chunk), e.g.:

```
\includegraphics[width=\textwidth]{Figs/plots3-1.pdf}
```

This works particularly well for pdf documents and it works with a number of different graphics file formats. If you are planning to produce your document as a pdf, we find that also generating your figures as pdf works best. However, if you want to produce an html document, pdf figures don't work.

**Option 2:**

You can use the R function `include_graphics()` inside an R code chunk. This function is in the `knitr` R package, which needs to be installed. E.g. `knitr::include_graphics(Figs/plots3-1.png)`

This works with more different file types and output types than option 1 above.

Don't get confused by the similarity of the syntax. Option 1 is LaTex code. It accepts options in square brackets and arguments (the figure name and path) in curly brackets. Option 2 is R code and therefore expects the figure name and patch in round brackets. Options (e.g. figure width and caption) needs to be supplied as option in the code chunk header (see previous section).

Let's look at an example.

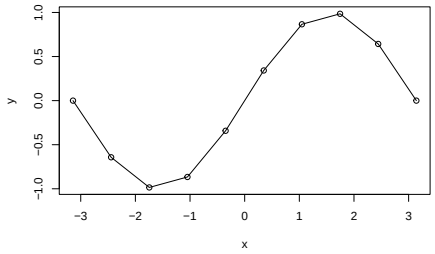
### 4.4.3 Plot a function

Let's say we can to plot the function $f(x) = sin(x)$ over the interval $-\pi \leq x \leq \pi$.

We first generate a few values that cover the interval over which we want to plot the function. This will be our x variable, or independent variable. Then, we calculate the value of the function at these points; this will be our y or response variable. We then plot the function values at these points and finally add a line connecting the points. Note that the x values are sorted from smallest to largest because of the way we generate them.

The options we used for this code chunk are "'`{r plots4, echo = TRUE, error = TRUE, out.width= '50%'}`.

```
x <- seq(-pi, pi, length = 10)   # vector of 10 values between -pi and pi
y <- sin(x)     # sin(x) for each x value
plot(y ~ x)
lines(x, y)   # adds a line connecting the points
```




## 4.5 Prac 4

Create a smooth version of the above sin function: increase the number of x-values at which f(x) is evaluated, e.g to 1000. Your final plot should:

- Plot the line directly, without points first. Increase line width, and change colour to red.
- Add the $cos(x)$ line to the same plot, in blue.
- Improve the general look of the figure.
- What would happen if the x's were a random uniform sample between $-\pi$ and $\pi$? Try.

## 4.6 Saving graphics

When you plot a figure, RStudio displays it in the lower right-hand panel. From there, you can save it in various formats using the 'Export' button. This may be fine for a quick job but is unsatisfactory in the long run. Manual steps make it harder to reproduce your workflow. In addition, when saving figures this way, you don't have much control over the size and resolution of your figures.

There is a much better way of saving figures: R has a number of graphics devices that allow you can save directly to pdf, svg, png and a whole lot of other formats. Check out the help file" `?Devices`.

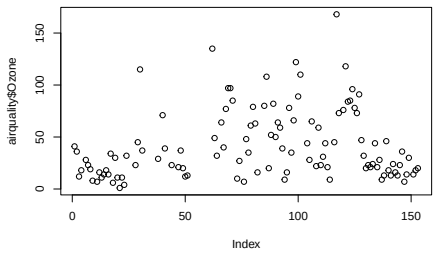
As an example, let's create a time-series plot of the ozone values in the 'airquality' data set and save it as a pdf file:

```
pdf("ozone.pdf", height = 5, width = 5)
plot(airquality$Ozone)
dev.off()
```

Note that we need to call the graphics device first (function `pdf()` in this case). We can control the height and width of the figure as you can see, and the function also accepts a number of other arguments (have a look at `?pdf`). Importantly, we need to close the graphics device with `dev.off()` once we are done plotting the figure.

As mentioned earlier, pdf works well with LaTeX. When producing figures for publications, you are often asked to supply them in .ps or .eps format and the `postscript()` function is helpful for that. For Word documents, the .svg format (function `svg()`) tends to work best, while I find .png (`png()`) a good option when working with html documents.

So here is the figure we just produced:




Imagine you come across this figure without any background information. Can you understand what it shows? Is there a better way to display these data? How could the figure be made more attractive, both in terms of how it conveys information and how visually pleasing it is?

## 4.7 Adding to plots: low-level plotting commands

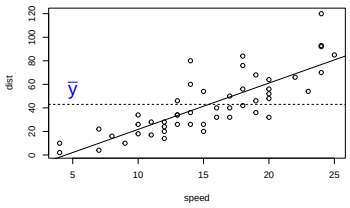
Plotting functions in base R are either high-level or low-level plotting functions. High-level plotting functions clear the graphics window, i.e. they plot a new figure (some functions can add to existing plots with the argument `add=TRUE`, though). Examples of high-level plotting functions we have met are `plot()`, `boxplot()`, `hist()`.

Low-level plotting functions, on the other hand, add to existing plots. We have already met one of these, the `lines()` function, which adds a line to an existing

plot. Other useful low-level plotting functions include `abline()`, `points()`, `segments()`, etc. The `text()` function adds text and the `axis()` function draws extra exes. Finally, `legend()` adds a legend.

Here is an example using several low-level plotting functions:

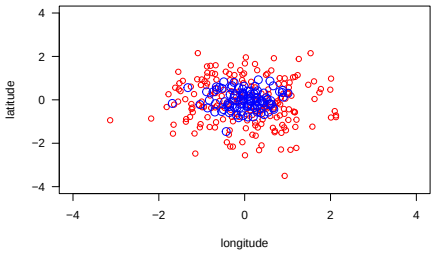
```
m1 <- lm(dist ~ speed, data = cars)
plot(dist ~ speed, data = cars)
abline(m1)        # regression line
ybar <- mean(cars$dist)
abline(h = ybar,   # horizontal line
       lty = 2)    # line type (dashed)
text(5, ybar, expression(bar(y)), pos = 3, col = "blue",
     cex = 2)  # x, y, text, position, colour, zoom factor
```




### 4.7.1 Adding Points

Often, we need to add extra points to a figure, e.g. a scatter plot. We can use the `points()` function to do that. Here is an example where we first draw an empty plot and then add two separate sets of points.

```
plot(-4:4, -4:4, type = "n",    # setting up coord. system
     xlab = "longitude", ylab = "latitude", las = 1)
points(rnorm(200), rnorm(200), col = "red")     # x, y
points(rnorm(100)/2, rnorm(100)/2, col = "blue", cex = 1.5)
```



The `rnorm()` function generates random values from a normal distribution. R has functions for working with a large number of statistical distributions. They always follow the same logic. Let's look at the `rnorm()` function a bit more closely by consulting the help file `?rnorm`. You can see that it comes with three related functions: the `dnorm()` function gives the density function for the normal distribution; the `pnorm()` function gives the distribution function and `qnorm()` gives the quantile function.

Have a look at the corresponding functions for the binomial distribution (`?rbinom`) and the Poisson distribution (`?rpois`). R has similar function for many more statistical distributions.

## 4.8 Prac 5

1. Generate 1000 values from a $N(1, 2)$ distribution and summarise these in a histogram. Change the histogram so that density and not frequency is shown on the y-axis. Calculate the true $N(1, 2)$ density over the domain of $N(1, 2)$ and plot this on top of the histogram using a thick red line.
2. Save different formats of this (see help for png `?png`). Copy all of these into a word document and comment on the differences. Which clearly works best, even when you really zoom in?

## 4.9 Visualising categorical variables

So far, we have mostly looked at how to visualise continuous numerical variables, e.g. using scatter plots to display the relationship between two numerical variables or histograms and boxplots to to examine the distribution of numerical variables.

Often, we also have categorical variables. Categorical variables are often used as factors in R, i.e. variables with a limited number of discrete levels.

### 4.9.1 More about Factors

Let's first look a bit more closely at factors in R. The variable `continent` in the Gapminder data set is a factor (with levels "Africa", "Americas", etc.)

```
library(gapminder)

conti <- gapminder$continent

str(conti)
```

```
##  Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
typeof(conti)    # factors are weird!
```

```
## [1] "integer"
```

```
class(conti)
```

```
## [1] "factor"
```

At first sight, factors behave strangely. We can see that the object `conti` is a factor but we are being told that it is of type "integer" and when we look at the structure, the first few values on the vector seem to be "3". What is going on?

```
levels(conti)
```

```
## [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
```

```
nlevels(conti)  # number of levels
```

```
## [1] 5
```

```
head(conti)
```

```
## [1] Asia Asia Asia Asia Asia Asia
## Levels: Africa Americas Asia Europe Oceania
```

As we know, factors have levels, i.e. the unique values the variable can take on. In this case, it is "Africa", "Americas", "Asia", "Europe" and "Oceania". These levels are internally stored as numbers, in alphabetical order. Asia is the third level. The first few values in the vector are all observations from Asia and are therefore stored as "3".

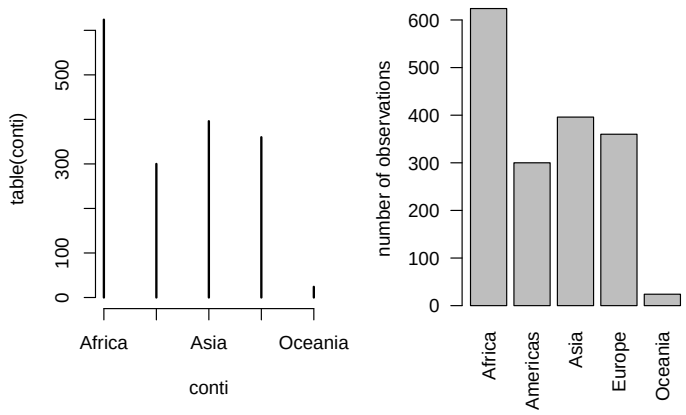
A useful summary for factors is the frequency table, which shows us how many observations we have in each level:

```
## conti
##   Africa Americas     Asia   Europe  Oceania
##      624      300      396      360       24
```

### 4.9.2 Visualising factors

One way to visualise factors is by plotting the frequency table, i.e. one bar per level with a length determined by the number of observations. This is similar to the idea of a histogram for continuous variables and also known as a bar plot.

```
par(mfrow = c(1, 2))
plot(table(conti))
barplot(table(conti), las = 2,
        ylab = "number of observations")
```

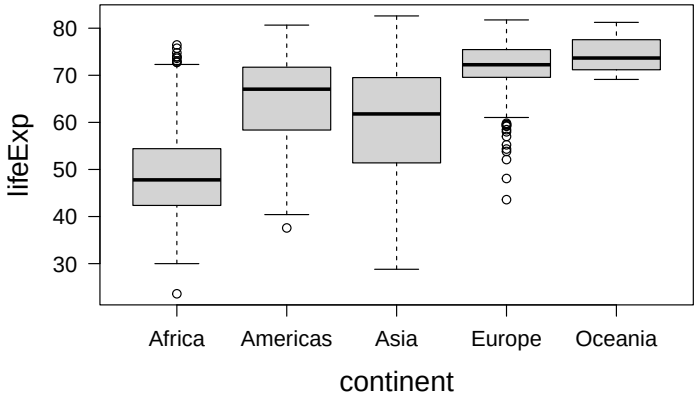



Barplots have a very low information/ink ratio! In this case, we have plotted an entire figure just to show five values. In a scientific report, one could consider conveying this information directly in the text.

### 4.9.3 Visualizing the relationship between numerical and factor variables

Factors are often grouping variables and we want to examine how the distribution of a numerical variable differs across the groups. Side-by-side boxplots are a good tool for that:

```
plot(lifeExp ~ continent, data = gapminder,
     cex.axis = 1.2, cex.lab = 1.5, las = 1)
```




## 4.10 Prac 6: Produce a beautiful plot in a document

For this prac, we want you to produce a high-quality plot that could be used in a document.

1. Take the Gapminder data and plot life expectancy against GDP, without attaching the data frame. Use colour to distinguish between data from different continents.
2. Does a log-transformation help to bring out the information more clearly?
3. Make sure the figure is effective and visually appealing by paying attention to axis labels and sizing. Add a legend.
4. Write a short R Markdown document that displays this figure. Generate the document in three different formats: html, Word and pdf.
5. Make sure the figure looks good in your documents; add a figure caption; improve size and placement.

Then, still working with the Gapminder data, address the following questions adding to the same R Markdown document. Check out how you can use R code directly in line with the text (as opposed to using code chunks): for example, if you want to report the mean of variable x, it would be something like "The mean of x was `` `r mean(x)` `` " in your R Markdown text section. This will calculate the mean and dsiplay it in the rendered document.

6. How many different countries occur in this data set?
7. How many African countries?

8. Which countries have the lowest and highest life expectancy, respectively? In which years? (There are several observations/years per country).

## 4.11 A glimpse of ggplot2

Some of you are probably itching to learn about ggplot2, so if you have extra time here is a glimpse of it. Have a look at this tutorial or this one.

The *Data Visualization with ggplot2* cheatsheet can be found under RStudio Help or here.

An excellent introduction into the tidyverse more generally can be found here.

### 4.11.1 ggplot2

Some example code and figures using the gapminder data.

```
library(ggplot2)
library(gapminder)
```

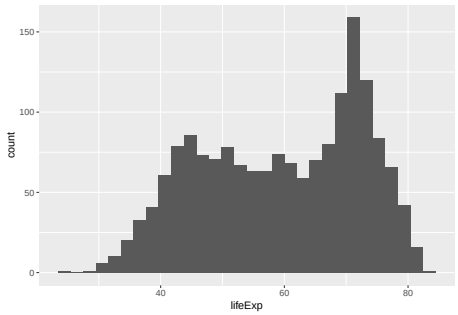
```
#declare data and x and y aesthetics, but no shapes yet
ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp))

## add layers
## it already knows what the x and y variables are from
## the ggplot part

ggplot(data = gapminder, aes(x = gdpPercap, y = lifeExp)) +
  scale_x_log10() +   # log x axis
  geom_point(aes(color = continent)) + # colour by continent
  geom_smooth()
```

```
## define data and x for univariate plots
f1 <- ggplot(gapminder, aes(x = lifeExp))
```

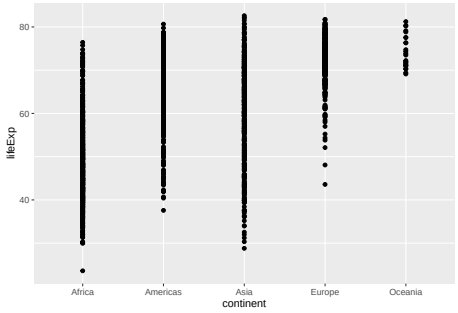
`ggplot` output is an object. So one can add to it.

```
## histogram
f1 + geom_histogram()
```

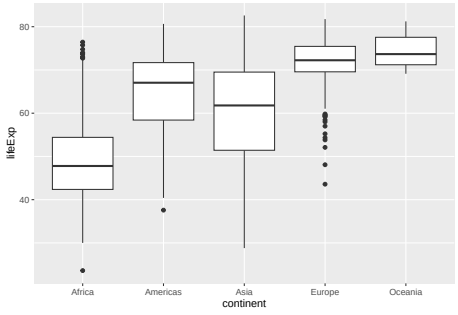
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## define data, x and y, x is a factor
tp <- ggplot(gapminder, aes(x = continent, y = lifeExp))

## scatter plot and boxplot by continent
tp + geom_point()
```




```
tp + geom_boxplot(aes(group = continent))
```



# Chapter 5

# Reading and writing data

Getting external data into R is a critical step in any data analysis work flow. And it is one of the trickier steps because data come in all sorts of formats and layouts. As data scientists, you will often have to deal with data sets that were created by people who have little understanding of (or don't care about) data work flows and the consequences some of their decisions might have for the analyst. Often, though, there are very good reasons for why people have captured their data in a particular format even if it is perhaps not the easiest format for analysis. In any case, we need to have a tool set ready that helps us deal with a broad range of possible formats effectively.

In this section, we will learn how to read external data from the following sources into R:

- .txt files
- .csv files
- .xls, .xlsx
- data base
- website

## 5.1 Problems you will/could encounter

Here is a shortlist of common problems one often encounters and can cause a lot of frustration:

- **Can't find file:** if R doesn't find the file you want to read, it is normally either because R is looking in the wrong place or because the file name is not spelled correctly. To check the former, use `getwd()` to see what path your current R session is using. If that looks correct, you can use

the `dir()` function to get a list of the objects in the target folder. If you see the file in that list, make sure the file name is spelled 100% correctly in your code. Remember that R is case sensitive. There might also be hidden spaces in the file name, etc. The safest is to copy the file name and paste into the script.

- **Number of elements in row x is wrong:** this often has to do with R trying to use the wrong column separators. E.g. you might think you are dealing with a comma-separated (,) file but in reality, the separators are semi-colons (;). If possible, open the file in a plain text editor and have a look. Avoid touching the file with Excel or any other software that could try to impose its own formatting and make things worse.
- **Converted variables to wrong type:** R only allows one data type per column (since these are vectors). So if your file includes a non-numeric character (including spaces, which are hard to find…) in a column that is supposed to contain only numbers, it will be read in as character.
- **0 or missing value?** It is often unclear whether missing data represent zeros or truly missing values. E.g. if you are looking at bird counts and there are no pied crows reported, is it because there were no pied crows (and so the count should be zero) or because the observer didn't bother to count pied crows (in which case it is a missing value). This information needs to be obtained from the metadata or a person familiar with the data collection protocol.
- ***and many others…***

To get to the bottom of these problems, it is often tempting (and sometimes necessary) to make changes to the input data file. Before you do that, however, make sure that you never touch the original copy of the data file. Make a copy and work on that file until you have figured out what is wrong. If at all possible, solve any data problems in your code, not by changing the input data file, so that your data work flow is completely reproducible. If the input data file needs to be changed, keep the changes to a minimum and carefully document all the changes you make to the working copy (and be sure there is a copy of the original file that will stay untouched).

## 5.2 Writing data to a file

Before we look at how to get data into R, we briefly cover the related topic of getting data out of R. This is typically easier because it is more under our control.

In the following code chunk, we are getting some data objects ready to work with: a matrix that we call 'X', a vector called 'y' and the gapminder data set.

```
X <- matrix(rnorm(10000), nrow = 100)
y <- 3:50

library(gapminder)
data(gapminder)
```

We start by writing the data frame to a text file.

```
write.table(gapminder, file = "f1.txt")
write.csv(gapminder, file = "f1.csv")
```

The `write.table()` function uses a space as separator by default, whereas the `write.csv()` function uses commas. For some strange reasons, the Excel convention for CSV files sometimes is to use semi-colons as separator. There is also a function called `write.csv2()` with defaults that better match that version of Excel.

Open these two files in Notepad or any other plain text editor.

### 5.2.1 Dumping output in a file

We can use the `sink()` function to divert output to a file (instead of the console). In the example below, the command `rnorm(10)` produces 10 randomly drawn values from the standard normal distribution but instead of printing them to the console as would normally happen, the numbers end up in a text file called "Routput.txt".

```
sink("Routput.txt")
rnorm(10)
sink()     # switch back to console
```

### 5.2.2 Saving your R data objects: Save/Load Workspace

We can save objects to an ".RData" file using the `save()` function. `save.image()` saves the entire workspace to a file called ".RData" by default (this can be changed).

```
save(X, y, file = "mystuff.RData")  # saving objects X and y

save.image()  # saves current workspace into .RData
```

We can now clear the workspace and check that everything is indeed removed. The `ls()` function lists all objects in the workspace and if it doesn't return anything, the workspace is empty.

```
rm(list = ls()); ls()
```

```
## character(0)
```

Then load the objects back into the workspace:

```
load("mystuff.RData")   # load into workspace
ls()
```

```
## [1] "X" "y"
```

We can see that we now have `X` and `y` in our workspace again, i.e. the same R objects as before.

## 5.3 Reading text files

Data are often saved as text files. The general function for reading such files is the `read.table()` function. Look at the help file `?read.text` to see what arguments the function accepts. There are a lot of options to play around with and you'll definitely need to reach into this 'trick box' sooner or later. E.g. the are options to skip lines; what to do with blank lines; what the commenting character is; and many more.

Often, ".txt" files use a tabulator as separator and include a header row. So let's try with those options:

```
df1 <- read.table("f1.txt", header = T, sep = '\t')

## check if it has done the right thing
dim(df1)
```

```
## [1] 1704    1
```

```
names(df1)
```

```
## [1] "country.continent.year.lifeExp.pop.gdpPercap"
```

```
head(df1)
```

```
##          country.continent.year.lifeExp.pop.gdpPercap
## 1  1 Afghanistan Asia 1952 28.801 8425333 779.4453145
## 2  2 Afghanistan Asia 1957 30.332 9240934 820.8530296
## 3   3 Afghanistan Asia 1962 31.997 10267083 853.10071
## 4  4 Afghanistan Asia 1967 34.02 11537966 836.1971382
## 5 5 Afghanistan Asia 1972 36.088 13079460 739.9811058
## 6   6 Afghanistan Asia 1977 38.438 14880372 786.11336
```

- sep = "\t": tab delimited
- header = TRUE: file contains a header line (with variable names), and values are separated with spaces

Does this look right? No! The first indication of trouble comes from the dimensions. The gapminder data set has 1704 rows and 6 columns. But the `df1` object only has one column. From the other checks we ran above, we can see that all data have been amalgamated into a single column.

This is a common problem and is usually caused by by getting the separator wrong.

Let's try again. At this stage, we'd try to find out what the separator is, e.g. by having a look at the file in a text editor. Alternatively, we can give it a try using the default option, which is that the values separated by any sort of space.

```
df1 <- read.table("f1.txt", header = T, sep = "")

## check if it has done the right thing
names(df1)
```

```
## [1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

```
dim(df1)
```

```
## [1] 1704    6
```

```
head(df1);tail(df1)
```

```
##       country continent year lifeExp      pop gdpPercap
## 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 Afghanistan      Asia 1977  38.438 14880372  786.1134
```

```
##       country continent year lifeExp      pop gdpPercap
## 1699 Zimbabwe    Africa 1982  60.363  7636524  788.8550
## 1700 Zimbabwe    Africa 1987  62.351  9216418  706.1573
## 1701 Zimbabwe    Africa 1992  60.377 10704340  693.4208
## 1702 Zimbabwe    Africa 1997  46.809 11404948  792.4500
## 1703 Zimbabwe    Africa 2002  39.989 11926563  672.0386
## 1704 Zimbabwe    Africa 2007  43.487 12311143  469.7093
```

Ok, this time we are in luck and all seems fine.

Let's check that the data types are also read correctly (e.g. number columns as numeric, etc.):

```
str(df1)
```

```
## 'data.frame':    1704 obs. of  6 variables:
##  $ country  : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
##  $ continent: chr  "Asia" "Asia" "Asia" "Asia" ...
##  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
##  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
##  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 138
##  $ gdpPercap: num  779 821 853 836 740 ...
```

```
summary(df1)
```

```
##    country           continent              year         lifeExp           pop
##  Length:1704        Length:1704        Min.   :1952   Min.   :23.60   Min.   :6.001
##  Class :character   Class :character   1st Qu.:1966   1st Qu.:48.20   1st Qu.:2.794
##  Mode  :character   Mode  :character   Median :1980   Median :60.71   Median :7.024
##                                        Mean   :1980   Mean   :59.47   Mean   :2.960
##                                        3rd Qu.:1993   3rd Qu.:70.85   3rd Qu.:1.959
##                                        Max.   :2007   Max.   :82.60   Max.   :1.319
```

Looks all good. We can see that the variables 'country' and 'continent' are of class `character` and need to remember to turn them into factors (using the `as.factor()` function) before we use them in statistical analyses. Alternatively, we could have set the option `stringsAsFactors = TRUE` when reading in the data.

A very common data format is the comma-delimited text file with '.csv' extension. While we could easily read these files using the `read.table()` function, R has a very convenient function to read csv files, `read.csv()`. It has defaults that often work well, for example it assumes that the column names are given in the first row and that commas are used as delimiters. These and other options can be changed, of course (see `?read.csv`).

```
df1 <- read.csv("f1.csv")

## check if it has done the right thing
dim(df1)
```

```
## [1] 1704    7
```

```
names(df1)
```

```
## [1] "X"         "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

```
head(df1)
```

```
##   X     country continent year lifeExp      pop gdpPercap
## 1 1 Afghanistan      Asia 1952  28.801  8425333  779.4453
## 2 2 Afghanistan      Asia 1957  30.332  9240934  820.8530
## 3 3 Afghanistan      Asia 1962  31.997 10267083  853.1007
## 4 4 Afghanistan      Asia 1967  34.020 11537966  836.1971
## 5 5 Afghanistan      Asia 1972  36.088 13079460  739.9811
## 6 6 Afghanistan      Asia 1977  38.438 14880372  786.1134
```

One consideration is that some country conventions use a dot as decimal separator and others use comma. The decimal separator can be specified as an argument; it is a dot "." by default in the `read.table()` and `read.csv` functions. Now, in some country settings (apparently mostly those who use decimal commas) Excel has decided to use semi-colons as column separators when writing '.csv' files, perhaps to reduce confusion with the decimal separator. So a common problem is that you think you are looking at a comma-delimited file when in reality, the delimiters are semi-colons ";". The function `read.csv2()` often works well in these situations. It assumes commas for the decimal separators and that the data are separated by semi-colons, by default.

**To recap:** it is absolutely critical to check that R has read in the data correctly because things often go wrong at this stage. Some things to check are that the dimensions are correct (number of rows and columns as expected?); have the column names been read correctly?; do the first few and the last few rows look like they should (sometimes the data-reading process derails halfway through the file and the rest gets jumbled); are all variables in the format they should be?

## 5.4 Importing .xls, .xlsx data

Even though it is a bad idea to store data in Excel, many people still do it and as a data scientist, you will often have to wrangle with Excel files. One

problem with proprietary formats is that the owner (Microsoft in this case) can and will change the format arbitrarily, causing all sorts of problems for the users. Also, the way Excel spreadsheets are set up seems to encourage bad data organisation. You will probably encounter column headers that take up several rows; arbitrary empty rows and cells with non-data content; rows and columns that look empty but turn out not to be… you name it.

These issues sometimes make it hard to keep your work flow fully automated and reproducible. Nevertheless, try to avoid manual steps as much as possible. Luckily, there are several R packages that have functions to interact with the different versions and flavours of Excel. Often, it is worth trying out a few options before settling on the one that works best for your specific case.

So here are some of the options for getting Excel data into R:

1. In Excel copy data to clipboard, in R type `df <- read.table("clipboard", header = TRUE)`. We do not recommend this approach in general because it is not reproducible. E.g. how can you make sure all data were really copied?
2. In Excel save .xls or .xlsx file as a tab delimited .txt or .csv file. Then, in R use one of

```
df <- read.table("file.txt", header = TRUE, sep = "\t")
df <- read.table("file.txt", header = TRUE, sep = ",")
df <- read.csv("file.csv")
```

This approach also involves a manual step and it therefore not the best option.

3. Use the R package `openxlsx` (or `xlsx`)

`openxlsx` and `xlsx` are two R packages with similar functionality. The former has the advantage of not depending on Java.

```
library(openxlsx) # functionality of 'xlsx' package
                  # but does not depend on Java

df <- read.xlsx("CO2.xlsx", sheet = 1)
head(df); tail(df)
```

```
##   Year Concentration
## 1 1000        277.00
## 2 1001        277.01
## 3 1002        277.02
## 4 1003        277.03
## 5 1004        277.04
## 6 1005        277.05
```

```
##      Year Concentration
## 1008 2007        383.77
## 1009 2008        385.59
## 1010 2009        387.37
## 1011 2010        389.85
## 1012 2011        391.62
## 1013 2012        393.81
```

```
dim(df)
```

```
## [1] 1013    2
```

```
detach(package:openxlsx)
```

This often works well with '.xlsx' files but can't handle the older '.xls' format.

4. Use the R package `XLConnect`

```
library(XLConnect)
df <- readWorksheetFromFile("CO2.xlsx", sheet = 1)
head(df)
dim(df)
plot(df)
```

You need to have Java installed on your computer to be able to use the `XLConnect` package.

5. Use the R package `readxl` (from the Tidyverse)

This has become my favourite way of reading Excel files. The function `read_excel()` works both on .xls and .xlsx files and doesn't depend on external packages (such as Java). It also has a lot of useful options that make it quite flexible, e.g. if you want to extract a specific part of a spreadsheet.

```
library(readxl)

df <- read_excel( "CO2.xlsx", sheet = 1)
head(df); tail(df)
dim(df)
```

A brief overview of this package can be found here.

## 5.5 Data from data base (SQLite)

R can connect to many data base types. Here is a simple example using SQLite:

```
library(DBI)
library(RSQLite)

mydb <- dbConnect(RSQLite::SQLite(), "my-db.sqlite")
dbListTables(mydb)

## Queries
dbGetQuery(mydb, 'SELECT * FROM airquality')

dbDisconnect(mydb)
```

Read the vignette for the `RSQLite` package for a brief introduction:

```
vignette("RSQLite")
```

Other useful R packages are `dplyr` and `odbc`.

Reading in data bases or querying data bases with R is a large topic and we are only scratching the surface here. Here is a good short introduction to database queries with R. And here is a good introduction to SQL databases and R.

## 5.6 Data from web

Increasingly, one needs to download data from the internet. Obtaining data from the internet with R can take many forms, from downloading text-based data files that are stored on the web to web scraping and much more. Again, here we just scratch the surface and only look at a quick example of the former.

The code below downloads records of earthquakes collected by the US Geological Survey:

```
URL <- "http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month.csv"
Earthquake_30Days <- read.table(URL, sep = ",", header = T)
names(Earthquake_30Days)
head(Earthquake_30Days)
```

## 5.7 Some additional points about reading data into R

### 5.7.1 Including path name (files in a different working directory)

Sometimes, you want to read data files that are stored in a different directory than your working directory. You can supply the entire file path instead of just the file name to the function you use to read your data. Having static path names in your code can cause problems if you want to run your analysis on a different computer, though. It is better to have a more flexible way of specifying the path.

For example, you might have all your data in a subfolder of your working directory, called 'Data'. Below are two options: 1) use the `getwd()` function to get the path to your current working directory and then paste this together with the rest of the path using the `paste0()` function; and 2) use './' to give the path relative to your current working directory.

- . is for current working directory
- `paste0()` pastes 2 (or more) character strings into 1 string, without space

```
getwd()   # what is my working directory
filePath <- "~/whateveryourcurrentdirectoryis"
df2 <- readWorksheetFromFile(paste0(filePath, "/Data/CO2b.xlsx"), sheet = 1)
head(df2)
dim(df2)

## relative to current working directory
df1 <- readWorksheetFromFile("./Data/CO2b.xlsx", sheet = 1)
head(df1)
dim(df1)
```

The `readWorksheetFromFile()` function is from the `XLConnect` package. You could use any of the other functions we met earlier.

### 5.7.2 Variable names

Sometimes you need to specify, or rename variables

```
air <- read.table("air.txt")
head(air)
```

```
vars <- names(airquality)    # or type in by hand
vars <- c("Ozone", "Solar.R", "Wind", "Temp", "Month", "Day")

names(air) <- vars
head(air)
```

### 5.7.3 Common data import problems

To end this section, here a quick recap on common data import problems. As a general tip: use R to sort out any problems rather than changing the data file (and make sure you always have a copy of the original data that you won't change). The advantage is that you will have code to document EXACTLY what you did. As a result, you will be able to reproduce your analysis at a later point.

1. R uses type conversions to try to guess data type of variable/column, This can sometimes go wrong. You can use `as.is`, or supply the data type with `colClasses` in the read functions.
2. Special characters/separators (;, spaces) are prone to cause problems. Often, just looking at the data file in a plain text editor (e.g. Notepad) helps diagnosing the problem. Don't use Excel to open text-based data files as it tends to try to impose its own formatting.
3. Files with blanks instead of missing values can cause problems. In numeric variables, R uses 'NA' for missing values.

Here is the R reference manual for importing and exporting data, for your reference.

## 5.8 Prac 7

First check that you get all of the above to run, and update with your own comments.

Read the following data into R. Each time check that R has done the right thing. Leave the data as original as possible.

1. counts.xlsx (on Vula, Resources → data)
2. Tortoise data.xls, sheet 'Tortoise measurements' (on Vula)
3. Globular clusters: http://www.physics.mcmaster.ca/~harris/GCS_table.txt

4. Large-scale climatic index (MEI): https://www.esrl.noaa.gov/psd/enso/mei/data/meiv2.data
5. Voice Data from Singing the Vowel 'ooh': http://www.statsci.org/data/general/ooh.txt
6. Air quality data: air2.dat (on Vula)
7. 2D Pollen counts: 8.Red-cross-feb-aug-89-starts 20 Feb-1989.xls (on Vula)
8. wader counts.xls (on Vula)

# Chapter 6

# Programming in R

Often, you need to repeat the same calculations many times. Carrying out repetitive tasks is something computers are particularly good at. All we need is a way to program them for the job. In this section, we give an introduction to programming in R.

The basic features we will cover are:

- *Loops* are an intuitive tool for telling the computer to repeat a task. There are different types of loop: you can ask the computer to carry out the same task 1) a given number of times, 2) while a certain condition is true, or 3) until a certain condition is met.
- *Conditional statements* are needed if you want the computer to do different things depending on some condition.
- *Functions* can make your code neater and more efficient. They are also used for tasks that need to be carried out repeatedly.
- *Vectorization* is an efficient way of applying the same calculation to each element of a vector.

## 6.1 `for` loops

The `for` loop is used when you want R to carry out some instructions a given number of times. The basic syntax is:

```
for(i in 1:n) {
  ## instructions
}
```

Let's look at the above code carefully. There are two parts: first, a part that controls how many times the loop is run (`for(i in 1:n)`) and, second, the

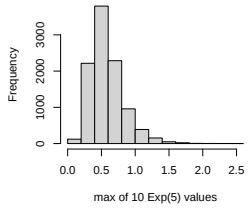
body of the loop that holds the code you want R to carry out at each iteration. The body of the loop is enclosed in curly brackets `{}`. It can contain pretty much any R code you want.

Clearly, the part that controls the loop is the new thing here. If you think about it, we need to tell R how many times the loop should be run and we need to keep track of which iteration we are at at any time. So these are the two things we need, in order to set up a loop. Let's read the control part 'from inside out'. We have `1:n` and we know that this creates a vector with the numbers $1, 2, \cdots, n$, i.e. $n$ numbers. Then, we have the so-called running variable $i$, which will take on each value in the vector `i:n` in succession. With this code, we are telling R to run through the loop $n$ times and $i$ keeps track of which iteration we are at, i.e. iteration 1, iteration 2, etc., until iteration $n$, at which point the loop stops.

Here is an example:

What is the distribution of the maximum of 10 $Exp(5)$ values?

```
nsim <- 10000      # number of simulations
emax <- numeric(nsim)     # empty vector to store values
                          # generated in for loop
for(i in 1:nsim) {
  vals <- rexp(10, rate = 5)
  emax[i] <- max(vals)
}

hist(emax, main = "", xlab = "max of 10 Exp(5) values")
```




Let's again start by looking at the body of the loop, enclosed byt the curly brackets. The first line generates 10 random draws from the exponential distribution and stores them in the vector `vals`. The second line calculates the maximum of these values, so the result is a single value.

The control part `for(i in 1:nsim)` works exactly like what we discussed a bit earlier; it sets up a loop with a running variable `i` that will take on the values 1 to `nsim`. Before we start the loop, we have to set the number of iterations we want. In this case, we call this number `nsim`. Then, we typically need to have

a way to store the result of the calculations carried out in the body of the loop. As we saw, each iteration of the loop produces a single value as output and we therefore set up an empty numeric vector of length `nsim` and call it `emax`. The only remaining piece now is to realise that we can use the running variable to fill the different positions in the output vector `emax`, through the index `[i]`: the first iteration (when $i = 1$) through the loop produces the first value and will be stored in the first slot of `emax`, i.e. it will be `emax[1]`. At the second iteration, $i = 2$ and the result will be stored as `emax[2]`, etc, until we reach the last iteration with $i = 10000$ after which the loop stops.

## 6.2 Functions

One of the most powerful features of R is that it allows us to make our own functions quite easily, using the command `function()`. We place the code that we want the function to carry out into curly brackets `{}` following the `function()` command. This will tell the function what we expect it to do. Inside the round brackets `()` we tell R what arguments (inputs) the function will need to perform its task.

```
myfunction <- function(arguments) {
  # instructions
  return(object)   # returns exactly one object
}
```

The function is usually assigned to an object that we name. For example, here is a simple function that calculates the mean of a set of values:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

```
# create a function to calculate the mean of a vector
calcMean <- function(my_vector){
    n <- length(my_vector)
    my_mean <- sum(my_vector)/n
    return(my_mean)
}
```

This function is called `calcMean` and takes one argument called `my_vector`. These two names are arbitrary and you can pick whatever you want. It is usually recommended to give functions "action" names, like "calculate_mean" or abbreviated "calcMean". Then, we must try to avoid using names that R already uses for other functions like "mean". The function would still work, but it might cause you problems down the line.

The function works like this:

1. You give the function `calcMean()` an input: `my_vector`.
2. Then, the function `calcMean()` starts executing the code inside the curly brackets and calculates the number of elements inside the object `my_vector`, using the function `length()` and creates an object `n` with the result.
3. Next, it creates the object `my_mean` by summing through the elements in `my_vector` and dividing this sum by the number of elements in `my_vector`, which was stored in the object `n`.
4. Finally, the function returns the value stored in the object `my_vector`. We specify what the output of our function is with the function `return()`. Actually, the function will output the last value it calculated if we don't specify a `return()` statement, but it is good to do it while we are learning.

Now, let's see this in action. Create an arbitrary vector of five values:

```
# Create a vector of 5 arbitrary numbers
x <- c(3.215, 0.561, 0.714, 1.643, 1.227)
```

And now, we will use the function `calcMean()` to calculate the mean of these five values:

```
# Use the function I've created to calculate the mean of the vector x
calcMean(my_vector = x)
```

```
## [1] 1.472
```

And that is your answer! We can check that it is indeed the correct mean by using the built-in R function `mean()` and compare the results.

```
# Confirm that the function calcMean gives the correct answer
mean(x)
```

```
## [1] 1.472
```

These results should agree… otherwise, you will need to review your `calcMean()` function. Notice that all those objects that the function creates and uses, like: `n` or `my_mean`, don't show up on your working environment. This is because R creates those objects, uses them internally and then discards them. This is one of the advantages of using functions to run series of commands; your working environment doesn't get too cluttered with intermediate objects.

Now, we are going to add a second argument to the function `calcMean()` that specifies how many decimal places should be printed out:

```
# Modify the function calcMean to be able to specify the number of decimal places in the output
calcMean <- function(my_vector, decimals = 2){
    n <- length(my_vector)
    my_mean <- sum(my_vector)/n
    round_mean <- round(my_mean, digits = decimals)
    return(round_mean)
}
```

Note the use of the `round()` command that rounds off the calculated number to the specified number of decimal places. Also note that the first argument (called `my_vector`) does not have any default values specified, whereas the 2nd argument (called `decimals`) has a default value of 2 specified. This means that if we run the function and do not specify anything for the 2nd argument, the default of 2 decimal places will be used.

Now, we will use this new version of the function `calcMean()` to calculate the mean of the vector `x`, first with 2 and then with 1 decimal places:

```
# Calculate the mean of x with 2 decimal places
calcMean(my_vector = x, decimals = 2)
```

```
## [1] 1.47
```

```
# Calculate the mean of x with 1 decimal
calcMean(my_vector = x, decimals = 1)
```

```
## [1] 1.5
```

## 6.3 Prac 8: empirical cumulative distribution function

1. Use a `for` loop to calculate cumulative probabilities from the counts produced by the `hist()` function below.

```
x <- rnorm(1000)

x.hist <- hist(x)
str(x.hist)
x.hist$counts

for( ... ) {
  cdf <- ...
}
```

2. Plot this empirical cumulative distribution function. Use a step function.
3. Add `lines(ecdf(x), col = "red")` to compare.
4. Make this into a function

```
?cdf   # good, does not already exist

cdf <- function(arguments) {
  # instructions
  return(object)   # returns exactly one object
}
```

and apply the function to the vector `x`.

```
cdf(x)
lines(ecdf(x), col = "red", lwd = 2)
```

5. Test your function with other data.

```
uuu <- runif(50000)
cdf(uuu)
```

## 6.4 Conditional expressions

With conditional expressions, you ask R to do different things based on which condition is met. A condition is any expression that evaluates to TRUE or FALSE. Here are some examples: `i < 20.5`, `difference > 0.005`, `x == 8`, `all(x > 0)`

### 6.4.1 `if` statements

In R, conditional expressions use an `if` statement in two ways. 1) You can use an `if` statement alone, i.e. tell R to do something if a condition is met and skip this task if it is not met. And 2), you can use the `if` statement together with an `else` statement to tell R to do one task if the condition is met and another task if it is not met.

```
if (condition) {
  ## operations
}
```

| Operator | Meaning |
|---|---|
| > | greater than |
| < | less than |
| <= | less than or equal to |
| == | equality |
| != | non-equality |
| & | elementwise and |
| \| | elementwise or |

```
if (condition) {
  ## operations
  } else {    # !! this line is important
  ## other operations
}
```

Here are some often used logical operators and what they mean:

### 6.4.2 The `ifelse()` function

The `ifelse()` function is a very useful short version of the `if` - `else` statement. It works on vectors, element-wise.

```
out <- ifelse(condition, what to return when TRUE,
                         what to return when FALSE)
```

For example, we generate 10 random draws from a standard normal distribution and then turn all negative numbers into zeros:

```
x <- rnorm(10)
y <- ifelse(x < 0, 0, x)
cbind(x, y)
```

```
##                x         y
##  [1,] -0.1999056 0.0000000
##  [2,]  0.6819671 0.6819671
##  [3,] -0.9545714 0.0000000
##  [4,]  0.4753786 0.4753786
##  [5,]  0.2137588 0.2137588
##  [6,]  0.9034104 0.9034104
##  [7,] -0.7320182 0.0000000
##  [8,] -0.8868060 0.0000000
##  [9,]  1.5502405 1.5502405
## [10,]  1.1173386 1.1173386
```

## 6.5 Prac 9

Generate 100 values from a Poisson distribution with mean 2. Now count the number of zeroes using loops and conditional expressions.

This exercise is good for practicing loops but, actually, a much better way to do the same job is:

```
num.zer <- length(x[x == 0])
```

You can use this to check your answer.

## 6.6 Conditional loops

With conditional loops, we can get R to keep carrying out operations 1) as long as a condition is true, or 2) until a condition is true.

With the `while` statement, R cycles through the loop as long as the specified condition is met.

```
while (condition) {
  # operations
}
```

With the `repeat` statement, R circles through the loop until the specified condition is met.

```
repeat {
  # operations
  if (condition) break
}
```

When working with conditional loops, there is always a risk that one sends R on an infinite loop. So make sure that the condition eventually turns false if using a `while` loop or that the condition eventually is true if using a `repeat` loop (if R happens to get stuck in an infinite loop, you can stop it by pressing on the red 'Stop' button in the top right corner of the 'Console' panel in RStudio; pressing 'Esc' also usually works).

Here is an example:

```
x <- 0
repeat {
  x <- x + 1
```

```
  print(x)
  if (x > 20) break
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
```

```
print("I'm done")
```

```
## [1] "I'm done"
```

If you replace `break` with `stop("x > 20")` in the code above, the loop will stop with an error message. Try it out.

## 6.7 Prac 10

Use a **while loop**. Simulate a random walk until $|\text{rw}| > 2$.

$$y_i = y_{i-1} + e_i \qquad e_i \sim N(0, 0.05^2)$$

Plot this while 'walking'.

## 6.8 Vectorization

Vectorization means applying a calculation to each element of a vector. Many R operations are vectorised as we have already seen. For example element-wise summing of two vectors:

```
a <- b <- 1:5
c <- a + b
c
```

```
## [1]  2  4  6  8 10
```

This probably looks obvious to you by now and you can think of many other cases of vectorization we have already met.

But to drive the point home, without vectorization, we would have to do this:

```
a <- b <- 1:5
c <- numeric(length(a))
for(i in 1:length(a)) {
  c[i] = a[i] + b[i]
}
c
```

```
## [1]  2  4  6  8 10
```

That would make coding in R a lot more work!

There are other opportunities for vectorising calculations and they may not always be so obvious. Often when we consider using a loop, it is actually possible to vectorise the calculations. Vectorised calculations are more efficient (go faster - sometimes by a lot!) and so it's worth using vectorised operations when possible. Here, we are going to look at three useful functions for vectorising calculations: `apply()`, `lapply()`, `sapply()`

- **apply**: apply a function over array margins
- **lapply**: apply a function over a list or vector
- **tapply**: apply a function over a ragged array

There are more functions related to the ones above: `sapply()`, `mapply()`, `rapply()`; and also `sweep()` and `aggregate()` … we leave it to you to figure out what they do.

### 6.8.1 apply

Here is what the R help file (`?apply`) says about the `apply()` function: *"Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix."*

One of the arguments the `apply()` function needs is `MARGIN`: with this argument, we tell the function over which dimension of the array we want to vectorise. For a two-dimensional array, it's either rows (1), columns (2) or both (1:2). "both" means we "apply the function to each individual value."

We can ask `apply()` to use any (sensible) R function, including functions that we have written ourselves.

So here is how this works:

```
## create a matrix of 10 rows x 2 columns
M <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)
M
```

```
##       [,1] [,2]
##  [1,]    1   11
##  [2,]    2   12
##  [3,]    3   13
##  [4,]    4   14
##  [5,]    5   15
##  [6,]    6   16
##  [7,]    7   17
##  [8,]    8   18
##  [9,]    9   19
## [10,]   10   20
```

```
apply(M, 1, mean)  # row means
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

```
apply(M, 2, mean)    # column means
```

```
## [1]  5.5 15.5
```

```
apply(M, 1:2, function(x) x/2) # divide all values by 2
```

```
##       [,1] [,2]
##  [1,]  0.5  5.5
```

```
##  [2,]  1.0  6.0
##  [3,]  1.5  6.5
##  [4,]  2.0  7.0
##  [5,]  2.5  7.5
##  [6,]  3.0  8.0
##  [7,]  3.5  8.5
##  [8,]  4.0  9.0
##  [9,]  4.5  9.5
## [10,]  5.0 10.0
```

```
## how many values > 10 in each column
apply(M, 2, function(x) length(x[x > 10]))
```

```
## [1]  0 10
```

```
## and applied to a data frame:
apply(airquality, 2, mean)
```

```
##     Ozone   Solar.R      Wind      Temp     Month       Day
##        NA        NA  9.957516 77.882353  6.993464 15.803922
```

Why do we get missing values with the last command? What exactly is this doing?

### 6.8.2 lapply

The help file for the `lapply()` function says: *apply function to every element of a vector or list*

This works in a similar way as the `apply()` function except that it works on vectors or lists (rather than arrays) and returns a list.

```
l <- list(a = 1:10, b = rnorm(5))
l
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $b
## [1]  0.1006505 -0.5174168 -1.4465813  0.3884361  0.5346034
```

```
# the mean of the values in each element
lapply(l, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] -0.1880616
```

```
# the sum of the values in each element
lapply(l, sum)
```

```
## $a
## [1] 55
##
## $b
## [1] -0.9403081
```

### 6.8.3 tapply

The `tapply()` function follows a similar logic as the `apply()` and `lapply()` functions but the 't' hints at the connection with the `table()` function: it groups data into subsets and then applies the function to each subset.

```
tapply(vector, groups, function)
```

As an example, we use it to calculate the mean life expectancy by continent from the Gapminder data:

```
library(gapminder)
with(gapminder, tapply(lifeExp, continent, mean))
```

```
##   Africa Americas     Asia   Europe  Oceania
## 48.86533 64.65874 60.06490 71.90369 74.32621
```

## 6.9 Prac 11

1. Generate a vector `u` with 20 random values from a uniform distribution. Now create a new categorical variable `u.cat` with values depending on the values in `u`: 'low' if $u_i \leq 0.3$, 'medium' if $0.3 < u_i \leq 0.6$ and 'high' if $u_i > 0.6$.

2. How can you vectorize (avoid for loops) the earlier exponential maximum example?

3. Create a list object with 4 matrices:

```
first = matrix(38:67, 3)
second = matrix(56:91, 3)
third = matrix(82:144, 3)
fourth = matrix(46:95, 5)
listobj = list(first, second, third, fourth)
```

Extract the second column from each of the matrices.

4. Create an array (10x10x10). Fill it up with random numbers. Calculate the trace (i.e. the sum of the diagonal elements) of every matrix (10 matrices each in dimensions 1, 2 and 3).

# Chapter 7

# Programming solutions to problems in R

Well done on making it through week 1! This week we shift gears to focus on using programming in R to solve problems. We cover a variety of challenging problems with the level of difficulty increasing as the week progresses. Most of the learning this week will happen as we try to solve problems. Therefore, the notes for this week will mostly consist of problem examples and practicals, with brief discussions of key statistical concepts and links for further reading on those concepts.

At the outset, it is important to consider how one should approach solving a problem using programming.

## 7.1 Guidelines to using programming to solve problems

- **Understand your problem**
  - Do you clearly understand what are the objectives of solving the problem?
  - *'A problem well put is half solved'* - John Dewey
- COLLABORATE! Asking for help from your team mates is a good and NORMAL part of learning. For those of you who find a particular problem 'easy' - take the initiative to help your team mates. The old adage 'You only understand something as well as you can explain it to someone else' applies here - if you can arrive at the solution quickly, then consider the exercise one of testing how well you can help one of your team mates.

'Well' doesn't mean that you simply share your solution - it means that you help your team mate to think through the problem. Remember, most often there are multiple ways to solve a problem!

- Don't be impatient - both with your team mates and with the problem at hand. Only use Google once you have made at least three attempts at the problem yourself & you have sought help from your team mates.
  - Conversely, don't wait to start programming until you think you have definitely solved the problem. Once you have an idea, code it and give it a try.
- Set up a step by step procedure of what you want to do
  - Write it down/develop a mindmap
  - Can you explain how you are going to solve the problem to one of your team mates without creating any confusion?
  - Discuss alternative ways to approach the problem. Almost invariably, you/your team mate will recognise a way to improve the approach via this discussion
- Think about what arguments you will require
  - What is the logic behind your approach?
  - What programming constructs will best serve that logic - loops/decision control/writing your own function etc.
- What outputs will be stored?
  - How many objects will you need to create to most efficiently store the outputs
- Code up each of the bits and test that each of them works. Do not move on to the next bit until a section works.
- Struggling is part of the process!!! Do not be afraid to make a mistake
- Comment your code *liberally*. You should be able to look back in 5 years time and easily understand what each part of your code is supposed to do.

**Important note:** Unless otherwise specified, please solve problems by writing your own code from *first principles*. This means that, other than functions available in the packages that come with the R installation, e.g. `base`, `stats` and `utils`, you don't use already-existing functions in other packages i.e. you write these functions yourself!

Lets get right into it!

| Symbol | Number of tiles | Payout |
|---|---|---|
| hearts | 7 | 2 |
| diamonds | 5 | 2.5 |
| spades | 3 | 8 |
| clubs | 6 | 4 |
| joker | 9 | 2.5 |
| ca | 20 | 0 |

## 7.2 Slot machine example

(Note that in this example we demonstrate that there are different ways to solve the same problem. We also demonstrate how usually you iteratively build code, discarding what doesn't work until you arrive at the solution.

A slot machine has a single reel - think of this as a circular wheel that can turn - on which one of five different symbols can appear. These symbols are hearts, diamonds, spades, clubs and a joker.

People bet R2 at a time in order to play the game and are paid out according to the symbol that appears after spinning the machine. The reel has 50 tiles and the number of tiles that occurs on the reel for the different types is shown in the Table below.

```
library(knitr)
#notice how to make a table in Rmarkdown
tile <- c("hearts", "diamonds", "spades", "clubs", "joker", "ca")
n_tiles <- c(7, 5, 3, 6, 9, 20)
win <- c(2, 2.5, 8, 4, 2.5, 0)
reel <- cbind(tile, n_tiles, win)
colnames(reel) <- c("Symbol","Number of tiles", "Payout")
kbl(reel, align = c("l","c", "c"))  %>% kable_styling(bootstrap_options = 'striped')
```

### 7.2.1 Replicate the game

1. Make the reel.
2. Spin the reel and record the outcome.
3. What does the player win.
4. Record the pay out.

#### 7.2.1.1 One way to make the reel

```
#let "h" = hearts, "d" = diamonds, …, "ca" = Castle
reel2 <- c(rep("h",7), rep("d",5), rep("s",3), rep("c",6), rep("j",9), rep("ca",20))
reel2
```

```
##  [1] "h"  "h"  "h"  "h"  "h"  "h"  "h"  "d"  "d"  "d"  "d"  "d"  "s"  "s"  "s"  "c"
## [31] "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca" "ca"
```

#### 7.2.1.2 Another way to make the reel

```
#let 1 = hearts, 2 = diamonds, …, 6 = Castle
reel3 <- c(rep(1,7), rep(2,5), rep(3,3), rep(4,6), rep(5,9), rep(6,20))
reel3
```

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

#### 7.2.1.3 Spin the reel and record the outcome.

Choose a random number from 1 to 50. The number indicates what position on the reel has been selected.

```
rn <- runif(1, min = 1, max = 50) #this does not work!!! Why?

spin <- reel[ rn ] #this does not (really) work!!! Why?
```

```
#?sample #look at the base package version, the dplyr package also has a version
#randomly select an element from within reel
spin <- sample(reel, size=1)
```

```
reel <- 1:6 #the different categories
probs <- c( 7, 5, 3, 6, 9, 20  )/50

spin <- sample(reel, size = 1, prob = probs)
```

#### 7.2.1.4 What does the player win.

```
if (spin==1){payout<-2
}else
if (spin==2)
{payout<-2.5
}else
if (spin==3)
{payout<-8
}else
if (spin==4)
{payout<-4
}else
if (spin==5)
{payout<-2.5
}else
{ payout<-0
}
```

#### 7.2.1.5 dplyr function *case_when()* for an alternative to the above

```
#install.packages('dplyr')
library(dplyr)
spin<- sample(reel, size=1)
spin

payout <- case_when(
  spin == 1 ~ 2,
  spin == 2 ~ 2.5,
  spin == 3 ~ 8,
  spin == 4 ~ 4,
  spin == 5 ~ 2.5,
  spin == 6 ~ 0
)
payout
```

```
reel2 <- 1:6

#probabilities of different symbols on reel
probs <- c( 7, 5, 3, 6, 9, 20  )/50
winnings <- c(2, 2.5, 8, 4, 2.5, 0)

spin <- sample(reel, size=1, prob = probs)
payout <- winnings[spin]
payout
```

```
## [1] 0
```

Note that you are encouraged to write your own functions in solving problems. Write a function for any operation that has inputs that you need to vary and which you intend to repeat multiple times.

## 7.3 Prac 12

### 7.3.1 Part 1

What is probability of getting the different winning symbols?

1.1 Spin the reel once.
1.2 Note the symbol.
1.3 Repeat 1 and 2 above many times (10000).
1.4 Take note of how many times each of the symbols appear.
1.5 Divide by 10000. This is an estimate of the probability.

### 7.3.2 Part 2

Calculate the average pay out if you spin the reel once. - i.e. one 'play' of the game is to spin the reel once, and you play the game 10000 times.

2.1 Spin the reel once.
2.2 Note the symbol.
2.3 Note the payout – store it.
2.4 Repeat 1, 2 and 3 above many times (10000).
2.5 You should now have 10 000 payout values.
2.6 Calculate the mean of these values. This is an estimate of the average. payout if you play the game once. - Compare it to the true value!

### 7.3.3 Part 3

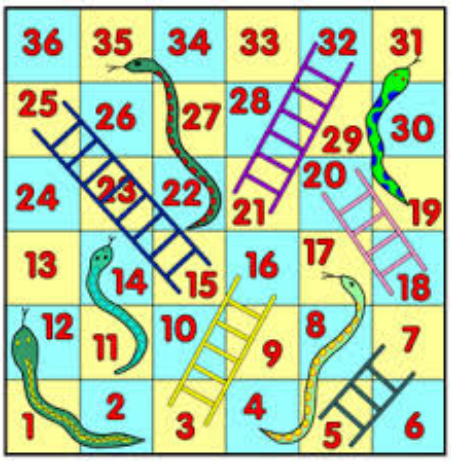
Calculate the average total pay out if you spin the reel 10/100/1000 times - i.e. one 'play' of the game is to spin the reel 10/100/1000 times, and you play the game 10000 times.

3.1 Graphically display the average total pay out after 10/100/1000 plays of the game by means of histograms and box and whisker plots.
3.2 Describe the distribution of the average total pay out pay out for each case by calculating relevant summary statistics.

Submit all parts in one Rmarkdown file, along with the html output of the markdown file.

## 7.4 Prac 13

Who ever thought you would play Snakes & Ladders again? The figure below is a simplified depiction of a Snakes & Ladders board which has 36 playing blocks.




**The rules of the game are as follows:**
- A player starts the game at the block number 1.
- The player throws a six-sided die once (1 dice) and moves to a new position depending on the outcome of the throw. e.g. if the player is situated at block number 2 and throws a six he/she would move to block number 8.
- If the player lands at the foot of a ladder he/she then moves to the top of the ladder. e.g. if the player lands on block number 3 he/she would move to block number 16.
- If a player lands on a block containing a snake head he/she moves down to the bottom of the snake. e.g. if the player lands on block number 35 he/she would move to block number 22.
- The game ends when a player lands exactly on block number 36. For example, if a player is situated at block number 33 and throws a 5 the player will remain at block number 33 since he/she has to throw a 3 to finish the game.

Write an R program in order to simulate the above game and provide a histogram of the empirical distribution of how many rounds it will take before the player reaches block number 36. Use this as an opportunity to practice formulating the problem BEFORE coding, and to collaborate with your team mates. Submit your program in an Rmarkdown file, along with the html output of the markdown file.

# Chapter 8

# Monte Carlo Simulation tutorial

## 8.1 Overview

1. For a brief history of Monte Carlo simulation see nhere

2. *What is Monte Carlo Simulation?*

    - See here for an introduction.
    - It is a simulation-based inference method used when you cannot find a deterministic solution to a system or problem. It relies heavily on conducting experiments based on random numbers i.e. drawing many random samples from some underlying data generating process. Practically, it is often used as a way of assessing risk.

3. How does it work?

   - Identify mathematical model (e.g. probability distribution) of the activity/process/system you want to explore
   - Define the parameters of the model
   - Generate many realizations (i.e. simulate data) from the data generating process
   - Each time calculate parameter(s) of interest
   - Analyse the output e.g. get an empirical distribution for parameter values

4. Advantages

   - You can predict outcomes of your activity/process/system without having to conduct hundreds of real-world experiments or collect thousands of data points!
   - You can estimate parameters of distributions where these can't be obtained from analytical or numerical methods
   - It provides fantastic framework for important statistical concepts of sampling and inference
     - What are the chances of seeing data like this again, if I repeat the same experiment?
     - Sampling distributions of test statistics
   - It is a comprehensive 'What-if' analysis approach, avoiding three pitfalls of traditional What-if analyses, which:
     - Don't take into account the probability of a scenario occurring
     - Only consider a small number of potential values of the parameters/variables
     - Don't take into account combinations of values of parameters/variables

5. Important Components

   - Random number generation - How is this done?
     - You sample a value from the a U(0,1) distribution. This value serves as a probability that you feed into the inverse of the CDF of the distribution of interest, which generates a random value from that distribution.
     - See here for an explanation
   - A couple of practical questions for you to consider:
     - What code (i.e. constructs/functions) to use to increase the accuracy (speed, efficiency) of Monte Carlo? This can become important as both the sample size and number of repetitions become very large.
     - How you, as an aspiring Data Scientist, would interpret and apply the results of a MC exercise! One must always remember that a solution to a problem is only as good as your explanation of it! This touches on what is referred to as 'Soft' skills: "The softer underbelly of the Data Scientist - knowing where to tickle is an art, not an exact science"

6. A few examples of the application of Monte Carlo simulation

   - Determine median level of smog exposure in Beijing
   - Mean time between infection and first appearance of symptoms of coronavirus
   - Can you think of any real-world examples???

## 8.2 Integration example

Suppose you wanted to calculate the following integral:

$$\int x5e^{-5x}dx.$$

How would you do it? Recall that here the integral is equal to the expectation of an exponential random variable i.e. $X \sim \text{Exp}(\lambda = 5)$ such that

$$E(X) = \int x5e^{-5x}dx.$$

(i.e. the long-run average = the mean of the variable)

Well here it is easy. You use simple integration rules or see that the integral is of the form:

$$\int xf(x)dx.$$

where $f(x)$ is an exponential distribution with parameter 5, in which case the integral simplifies to 0.2.

An alternate method is to randomly sample (simulate) from the exponential distribution and get $x_1, \ldots, x_n$ and approximate the integral as

$$\int x5e^{-5x}dx \approx \frac{1}{n}\sum_{i=1}^{n} x_i.$$

In fact we can approximate any one dimensional integral as:

$$\int h(x)f(x)dx \approx \frac{1}{n}\sum_{i=1}^{n} h(x_i).$$

## 8.3 How do we sample from a distribution?

See here for readings on the following ways:

1. Probability integral transform (1.2.1 SC notes)
2. Accept reject methods (1.3 SC notes)
3. Gibbs sampling (1.4 SC notes)
4. Many other methods

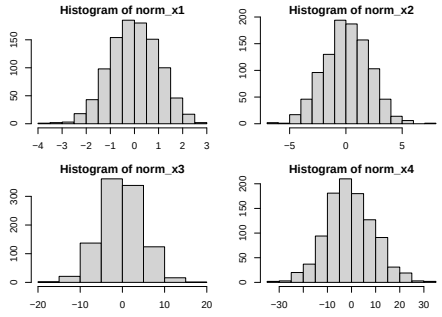
SC notes = Statistical Computing notes by Stewart and Erni (2013)

For now we don't focus on the theory, but see how it can be done in R.

## 8.4 Sampling from the normal distribution

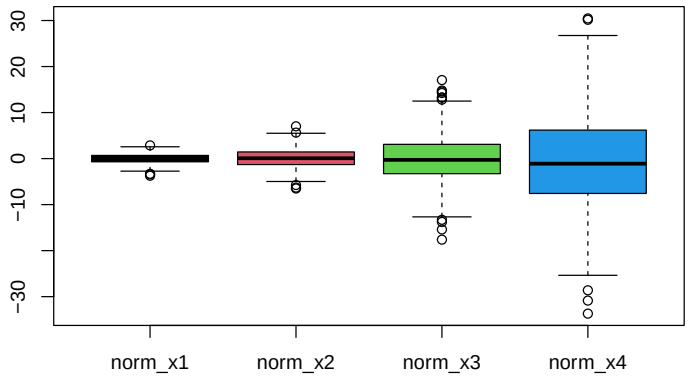
A simple example of generating random samples from a distribution:

```
nsamp = 1000    #number samples drawn

# four normal distributions with mean 0 and different standard deviations
norm_x1 = rnorm(nsamp)
norm_x2 = rnorm(nsamp, mean=0, sd =2)
norm_x3 = rnorm(nsamp, mean=0, sd =5)
norm_x4 = rnorm(nsamp, mean=0, sd =10)

#plot all four distributions together
par(mfrow=c(2,2), mar=c(3,3,1,1))
hist(norm_x1)
hist(norm_x2)
hist(norm_x3)
hist(norm_x4)
```




Plot boxplots of all four distributions together:

```
norm_samples <- cbind(norm_x1, norm_x2, norm_x3, norm_x4)
boxplot(norm_samples, col=1:4)
```

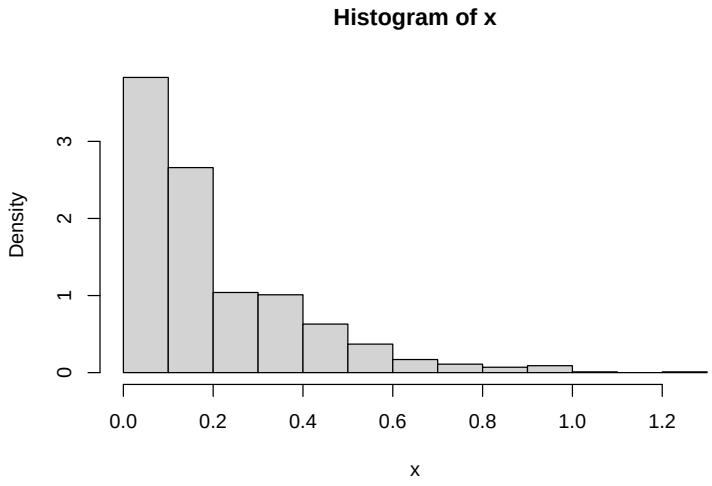



R has distribution functions d, p, q and r etc. for most statistical distributions e.g. `dgamma`, `pgamma`, `qgamma`, `rgamma`.

## 8.5 Sampling from the Exponential distribution

Using in-built distribution function from R:

```
nsamp = 1000   #number samples drawn
x <- rexp(nsamp,5)  #sample from Exp(rate=5)
hist(x, prob=TRUE)  #plot histogram
```




```
mean(x)  #mean of samples
```

```
## [1] 0.2013328
```

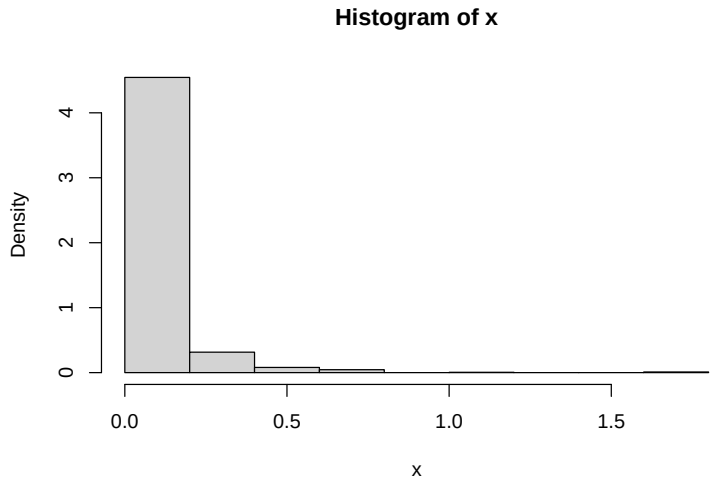
**Challenge**: See if you can code the above without using `rexp`

### 8.5.1 Exercise

Calculate $E(X^2)$. i.e. $\int x^2 f(x) dx$ where $X$ is an exponential distribution with parameter $\lambda = 5$.

#### 8.5.1.1 Solution using `rexp` and Monte Carlo simulation

```
nsamp <- 1000   #number samples drawn
x <- rexp(nsamp,5)^2  #Note the square!!!
hist(x, prob=TRUE)  #plot histogram
```




```
mean(x)  #mean of samples
```

```
## [1] 0.06707851
```

#### 8.5.1.2 Solution using `integrate`

```
hx <- function(x){ (x^2)*5*exp(-5*x) }

integrate(hx, lower=0, upper=Inf)
```

```
## 0.08 with absolute error < 8.2e-06
```

Now lets randomly draw 5 observations from an exponential distribution. (i.e. we would have a data set which contains 5 observations)

```
mean(rexp(5, 5))
```

```
## [1] 0.3303642
```

What if we use had 10 observations?

```
mean(rexp(10, 5))
```

```
## [1] 0.1257283
```

What if we use 1000 observations?

```
mean(rexp(1000, 5))
```

```
## [1] 0.2044278
```

What do yo notice, if anything?

### 8.5.2 Exercise

**For this exercise, please attempt each step before running the solution code**

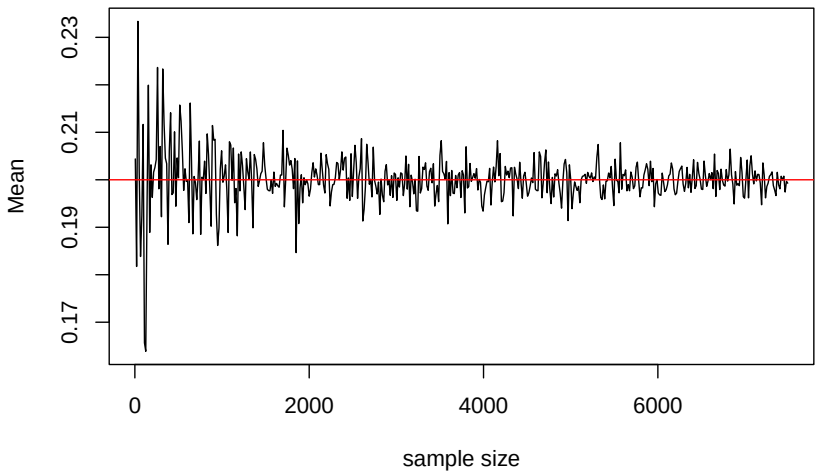
1. Lets consider investigating data sets of different sizes.
2. Randomly draw a sample from the exponential distribution for each of the data sizes.
3. Calculate the sample mean for each of the simulated data sets!
4. Plot the samples means.

Attempt to do this using loops!!!

Thereafter attempt this using apply, sapply, ..., etc!!!

```
n_sizes<-seq(from=5, by=15, length.out=500)
```

Please attempt the exercise yourself before you continue reading. You should get something similar to the figure below:




#### 8.5.2.1 Solution:

1. Lets consider using a number of data sizes:

```
n_sizes <- seq(from=5, by=15, length.out=500)
```

2. Randomly draw a sample from the exponential distribution for each of the data sizes:

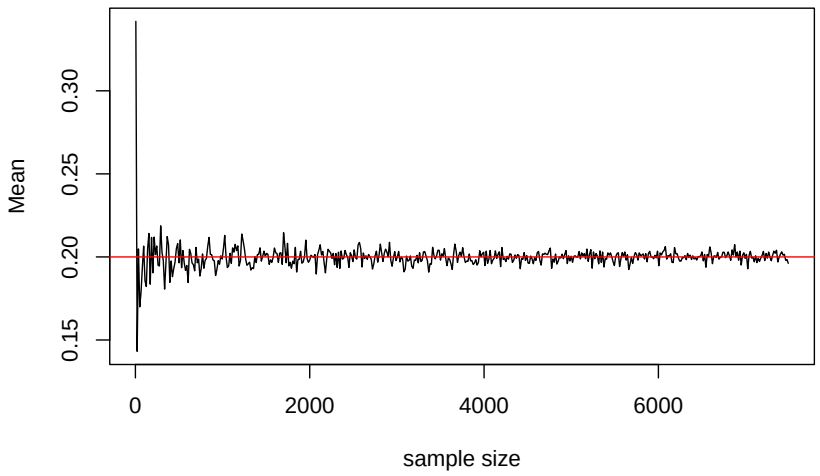
```
samples_exp <- sapply(n_sizes, FUN=rexp, rate=5)
```

3. Calculate the sample mean for each of the data sets:

```
mean_samples <- lapply(samples_exp, mean)
```

4. Plot the sample means:

```
plot(n_sizes, unlist(mean_samples), type="l",
      xlab="sample size", ylab="Mean")

abline(h=0.2, col="red")
```




It seems that as the sample size increases, the estimate of the mean gets closer and closer to 0.2.

### 8.5.3 Exercise: The sampling distribution of the mean of an exponential distribution

**Again, please attempt the exercise before running the solution code**

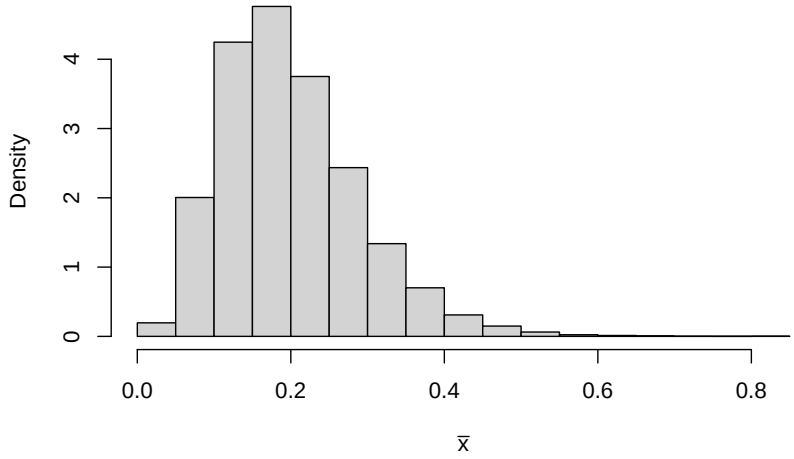
Write code to obtain the sampling distribution of the sample mean of exponential data. Assume that size of the data set is 5. Note that **the sample mean is a random variable and thus it has a sampling distribution**! Can you explain why?

So do the following:

- Sample from an exponential distribution. Generate a data set with 5 observations in it.
- Calculate the mean of the data set.
- Do this 5000 times.
- Store all the mean values.
- Plot the histogram of the means.

- The histogram is sampling distribution of the sample mean of exponential data when with a sample size of 5.

You should get something this:




Write some code to investigate what happens as the size of the data set increases! Consider writing a function to do this. Comment on your results.

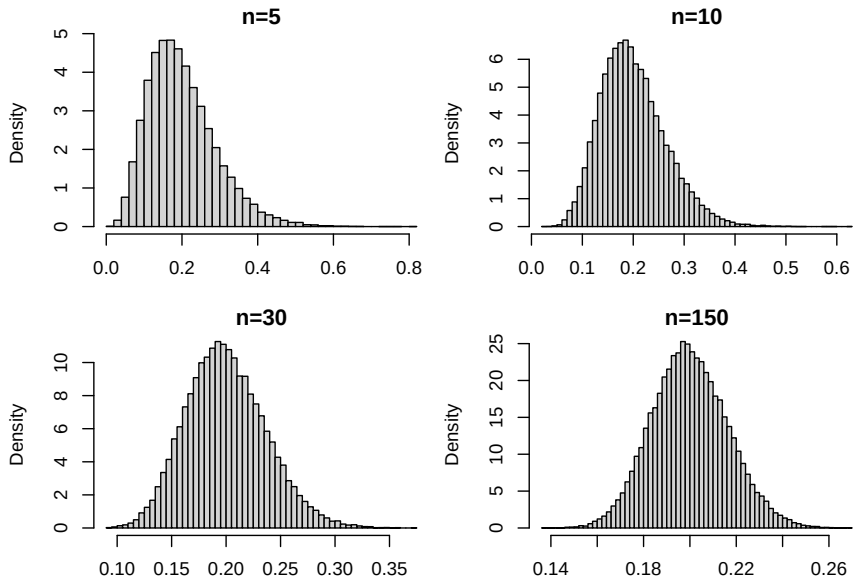
Recall that a function is defined as follows:

```
function_name <- function(args){

  expressions are placed here

    return something
}
```

Use the following data set sizes:

```
nsize = c(5, 10, 30, 150)
```

You should get something similar to this:




## 8.6 Random Sums

We have a sequence $X_1$, $X_2$, ..., with $X_1 = x_0$ known, but

$$N \sim U(1, 20)$$

(a random integer between 1 and 20. (end points are included.))

$$X_i \sim |Normal(X_{i-1}, 1)| \quad i = 2, ...$$

$$S = \sum_{i=1}^{N} X_i$$

We are interested in $E(S), Std(S), Pr(S > 10)$. Let $x_0 = 1$

We can sample from $U(1, 20)$ as defined above as

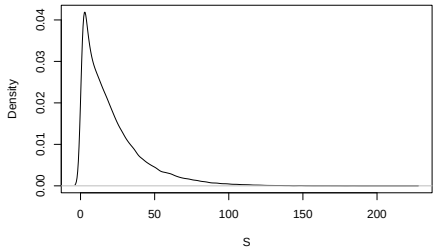
```
sample(1:20, 1)
```

Now write a function to:

- Plot the density of S

- Find E(S)
- Find sd(S)
- Find Prob(S>10)

Answers:




```
## E(S) =  21.46379
```

```
## sd(S) =  21.05414
```

```
## Prob(S>10) =  0.63359
```

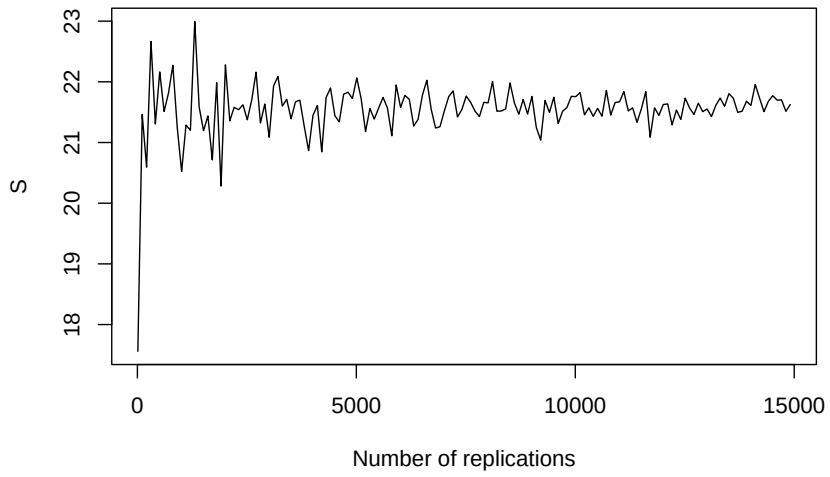
```
sample_S<-function(x0){

  X <- NULL
  X[1] = x0 #the first element in the sequence of X
  N<-sample(1:20, size=1)

  if (N==1){
    S<-X[1]
  }else
  {for i in 2:N{ X[i]<- abs(rnorm(1, mean=X[i-1]))
  }
    S<-sum(X)
  }#endif
  return(S)
}

nsamps = 100000
S <- replicate(nsamps, sample_S(x0=1))
plot(density(S), xlab="S", main="") #produce a density plot!
```

### 8.6.1 How many samples are required before the results are stable?

Rerun your code but now allow the number of replications to vary. Use a number of values and plot your solutions. Consider

```
#the number of replications
n_sizes <- c(seq(from=10, to=15000, by=100))
```

If the code takes too long, use a smaller number of potential sample sizes. You should end up with a plot similar to this:




## 8.7 Prac 14

### 8.7.1 Part 1: Some more sampling questions!

Assume that X is an exponential random variable with rate parameter 5. Now use Monte Carlo simulation to obtain the following quantities:

- $E(X^2) = \int x^2 f(x) dx$
- $\sigma^2 = E(X - \mu)^2 = E(X^2) - [E(X)]^2 = \int x^2 f(x) dx - \mu^2$ where $\mu = \int x f(x) dx$.

- Prob$(0.1 < X < 0.3)$

Recall: We can approximate any one dimensional integral as:

$$\int h(x)f(x)dx \approx \frac{1}{n}\sum_{i=1}^{n} h(x_i).$$

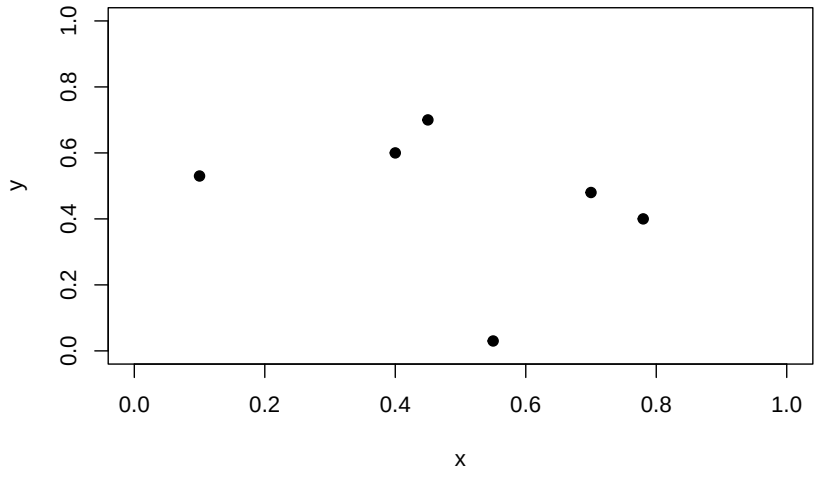
### 8.7.2 Part 2

**Hint: what do we usually measure when we are talking about points?**

#### 8.7.2.1 Part 2a: Distribution of points in a unit square

Consider the following set of points in the unit square. Decide whether or not they are randomly distributed in the square.

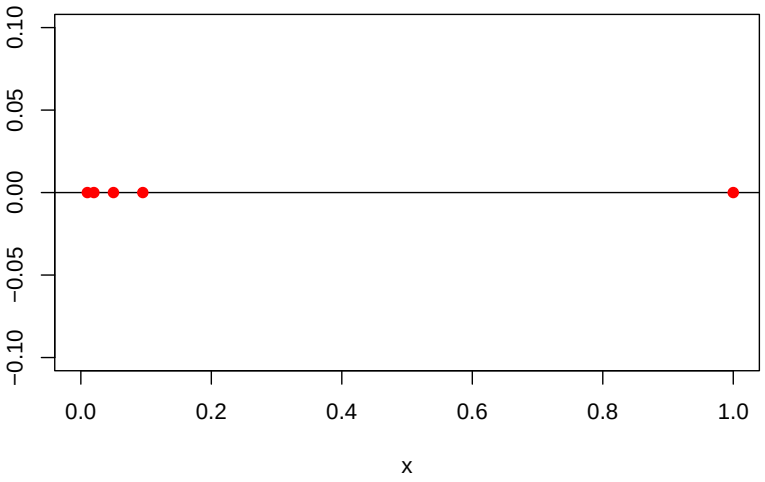
```
x<-c(0.1, .4, .45, .55, .7, .78 )
y<-c(.53, .6, .7, 0.03, .48, .4)
plot(x,y, xlim=c(0,1), ylim=c(0,1), pch=19)
```




#### 8.7.2.2 Part 2b: Distribution of points on a straight line

Are these points random between 0 and 1???

```
rand_x1<-c(.01, .02, .05, .095, 1)

xr<-seq(0,1, by=.01)
plot(xr, rep(0, 101), col="white",
     xlab="x", ylab="", ylim=c(-.1, .1))
abline(h=0)

points(rand_x1, rep(0,5), col="red", pch=19)
```




Submit your .Rmd and HTML files under the Assignments tab on Vula.

# Chapter 9

# Optimisation tutorial

## 9.1 What is optimisation?

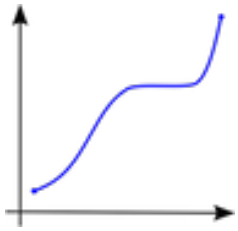
Simply put, optimisation is the process by which we find the optimal (minimum or maximum) of some objective function by varying the values of the parameters of the function, using some algorithm. The objective could be something like overall revenue of a company, in which case you would want to maximise the function that determines the revenue (i.e. the objective function). In machine learning, the objective is most often to minimise a loss function which is some measure of the difference between predicted outputs and 'true' outputs. Hence, many optimisation algorithms focus on minimisation of the objective function (aka 'loss' or 'cost' function).

## 9.2 Where do we use optimisation?

Everywhere in AI, Machine learning, Deep learning… basically all of Data Science! Hence, you will use optimisation very often when tackling problems as a data scientist. Two optimisation algorithms that you may have encountered before are least squares (e.g. regression) and maximum likelihood estimation. In machine learning, gradient descent is often referred to as the 'King' of optimisation methods. Other applications of optimisation involve problems where we try to minimize risk (e.g. Bayesian optimisation methods) or find the roots of functions (Newton's method).

There are plenty of examples of real-world applications! e.g. minimize cost of airline flight schedule. See here for some great examples of real-world machine learning optimisations.

## 9.3 Some Reminders



A few quick points to remember when dealing with optimisation problems:

- Maximizing $f(x)$ is the same as minimizing: $-f(x)$
- At an *interior* minimum/maximum: $f'(x) = 0$
- At an *interior* minimum $f''(x) > 0$
- If $h$ is strictly increasing, e.g. log, then min $f(x) =$ min $h(f(x))$
- Local vs global minimum/maximum: A local optimum is the optimal value of the objective function *within a region of its domain*, whilst the global optimum is the optimal value over the entire domain of the objective function.
- In >= 2 dimensions, the matrix of second derivatives is called the **Hessian** matrix. At the optimal (minimum) solution, it is positive definite (all eigenvalues $> 0$)
- Some optimisation algorithms (e.g. Newton's method) are very sensitive to the initial parameter values that you select, and it is often advised that you experiment with a number of different starting values to compare the optimal values you get. This is done to ensure that the algorithm is not converging at a local optimum instead of the global optimum.

## 9.4 Some Optimization Algorithms

### 9.4.1 Gradient Descent

See 1 & 2 for some good introductory reading.

The basic idea behind gradient descent is that you have some convex objective function for which you iteratively tweak the parameters until you reach the global minimum:

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0)$$

You start with an initial guess of your parameters, $\theta$, and define the step size $m$ (the amount by which you change the parameter values from iteration to iteration). The algorithm is then:

- While not at global minimum:
  - Find gradient at current point
  - New parameter vector $\theta_{i+1} = \theta_i - mf'(\theta_i)$
  - If $f'(\theta_i) > 0$ move right, $f'(\theta_i) < 0$ move left
- Continue until you have reached global minimum i.e. $f'(\theta_i) \approx 0$
- Do further research to grow your understanding!

### 9.4.2 Newton's Method

See 1 & 2 for some good introductory reading.

In Newton's method, the objective function is approximated using a Taylor series or a quadratic function:

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$

To find the minimum, you repeatedly find the gradient of the function at a point in its domain, set to zero, and solve for x:

$$x^{(t+1)} = x^{(t)} - \frac{f'(x^{(t)})}{f''(x^{(t)})}$$

The method terminates once you are sufficiently 'close' to the optimal value. But how do you measure how 'close' you need to be?

#### 9.4.2.1 Convergence Criteria, Stopping Rules

When should we stop?

- Even if $f'(x) \approx 0$, we can still have large changes in $x$ if $f()$ is very flat
- You should check that for small changes in $x$, $f'(x) \approx 0$ i.e. for the parameters of the objective function, you want:

$$|\theta^{(k)} - \theta^{(k-1)}| < \epsilon$$

where $\epsilon$ is a constant, tolerable imprecision … aka a **tolerance** level.

- You could also define a relative convergence criterion:

$$\frac{|\theta^{(k)} - \theta^{(k-1)}|}{|\theta^{(k)}|} < \epsilon$$

## 9.5 Optimization in R: optim() tutorial

Some of the optimisation functions in R include `optim`, `nlm`, `optimx`, `constrOptim` and others. In this tutorial we will focus on `optim`, the main arguments of which are:

`optim(par, fn, gr, method, control, hessian)`

see `?stats::optim` in R

- fn: function to be minimized
- par: initial parameter guess
- gr: gradient function; only needed for some methods
- method: defaults to "Nelder-Mead" (which does not use gradients)
- control: optional list of control settings (maximum number of iterations, tolerance for convergence)
- hessian: should the final Hessian be returned? default FALSE

Note that it is good practice to use `optimize()` for one-dimensional problems.

### 9.5.1 Simple linear regression example

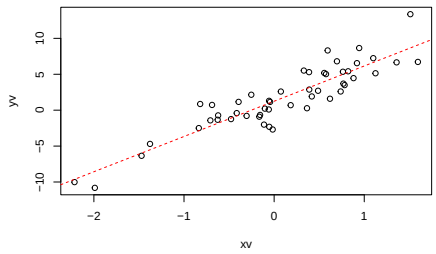
Lets simulate some data and perform a linear regression.

```
set.seed(1) #set the seed to make this example reproducible
xv <- rnorm(50) #generate some random values
yv <- 1 + 5*xv + rnorm(50)*2 #define an objective function

#perform ordinary least squares regression using lm() function from stats package:
ols <- lm(yv~xv)
ols
```

```
##
## Call:
## lm(formula = yv ~ xv)
##
## Coefficients:
## (Intercept)           xv
##       1.244        4.909
```

Plot the data and the best-fitted line from the least squares regression:

```
plot(xv, yv)
abline(ols, col="red", lty=2)
```




Now we create a function to numerically obtain the beta coefficients of a simple linear regression model:

```
regression_f <- function(betav,x,y){

  #x= explanatory variable
  #y = response variable
  #betav = c(intercept, slope)

  intercept <- betav[1]
  slope <- betav[2]

  fitted_values <- intercept + slope*x
  residuals <- y - fitted_values

  SSE <- sum(residuals^2) #a loss function

  return( SSE )
}
```

First attempt at a solution using `optim`. Run the code below with eval = T in the chunk options:

```
regression_f(c(1,10), x, y)

#you will get an error
optim(par=c(10,10), fn=regression_f)
```

Notice the error! This is because you have to pass the arguments 'x' and 'y' to 'optim'. In general, you must pass all the arguments of your objective function to `optim`. Let's do that and try again:

```
olsfit <- optim(par=c(10,10), fn=regression_f, x=xv, y=yv)
olsfit$par
```

```
## [1] 1.243281 4.909777
```

```
#compare to the results using lm()
ols
```

```
##
## Call:
## lm(formula = yv ~ xv)
##
## Coefficients:
## (Intercept)           xv
##       1.244        4.909
```

Now we obtain the correct results.

How about we restrict the intercept to be negative:

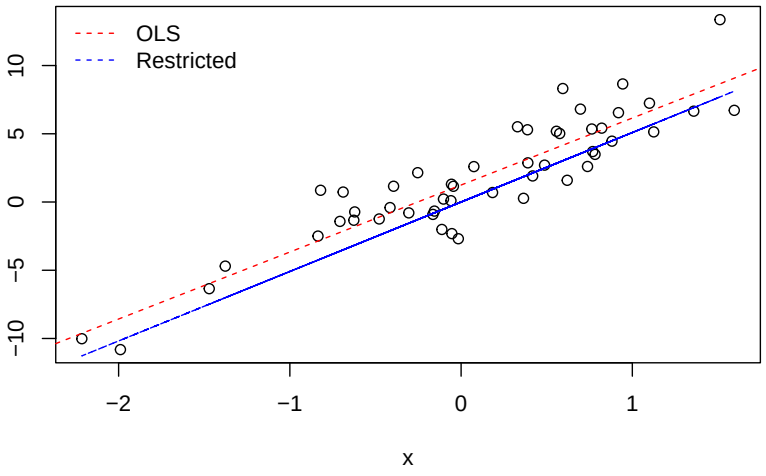
```
#no restriction set:
optim(par=c(10,10), fn=regression_f, x=xv, y=yv, method="L-BFGS-B")
```

```
## $par
## [1] 1.243803 4.908903
##
## $value
## [1] 183.6899
##
## $counts
## function gradient
##        7        7
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

```
#restricting the intercept:
fit2_res <- optim(par=c(10,10), fn=regression_f, x=xv, y=yv,
                  method="L-BFGS-B", lower=c(-Inf, -Inf), upper=c(0, Inf))

yfit_res <- fit2_res$par[1] + fit2_res$par[2]*xv
```

Plot the data and both model fits:

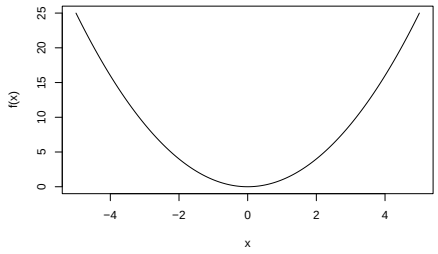
```
plot(xv, yv, xlab="x", ylab="")
abline(ols, col="red", lty=2)
lines(xv, yfit_res, col="blue", lty=2)
legend("topleft", legend = c("OLS", "Restricted"),
       col=c("red", "blue"), lty=rep(2,2), bty="n")
```




### 9.5.2 Non-linear optimization example

Consider minimizing the non-linear function $f(x) = x^2$. We start by generating some data and plotting the function:

```
xr <- seq(-5, 5, length=1000)
f1 <- function(x){x^2}
plot(xr, f1(xr), type="l", xlab="x", ylab="f(x)")
```




- We know that the solution is at $x = 0$.
- `optim` by default does minimization of a function.
- The default method used in `optim` is the 'Nelder-Mead' method, which is 'unreliable' when the function being minimized is one dimensional. However, in this case R does get the correct solution!

```
opt1 <- optim(par=5, fn=f1)
```

```
## Warning in optim(par = 5, fn = f1): one-dimensional optimization by Nelder-Mead is u
## use "Brent" or optimize() directly
```

```
opt1
```

```
## $par
## [1] 0
##
## $value
## [1] 0
##
## $counts
## function gradient
##       32       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Note the convergence message - 'opt1$par' contains the $x$ value R thinks the function is being minimized at.

Consider using 'method="Brent"' over the range $x \in [-5, 10]$. We see that R finds a solution at $x = 0$.

```
opt2 <- optim(par=5, fn=f1, method="Brent", lower = -5, upper=10)
opt2
```

```
## $par
## [1] 1.110223e-16
##
## $value
## [1] 1.232595e-32
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Take note what happens when you change the search area!

```
#Be careful about the range you are searching!
opt3 <- optim(par=5, fn=f1, method="Brent", lower = -5, upper=-1)
opt3
```

```
## $par
## [1] -1
##
## $value
## [1] 1
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We can also use R to minimize a two-dimensional function.

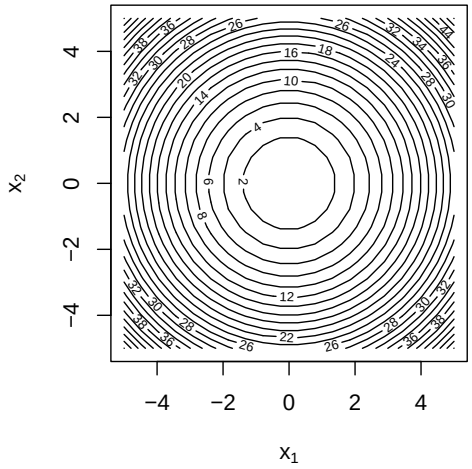
```
f2 <- function(x){
  #x = c(,) i.e. it must have two elements!!!
  x1 <- x[1]
  x2 <- x[2]

  x1^2 + x2^2
}
```

Lets plot the function. Notice that I redefine the 'f2' function. I use the `Vectorize` and the "'outer"" functions to produce the plots. Have a look at 'z'. Do you know how it was calculated?

```
f2_2 <- function(x1, x2){f2(c(x1,x2))}

f2_vec <- Vectorize(FUN=f2_2,vectorize.args = c("x1", "x2"))

x1_range <- seq(from = -5, to = 5, length=20)
x2_range <- seq(from = -5, to = 5, length=20)

z <- outer(X = x1_range, Y = x2_range, FUN= "f2_vec" )
```

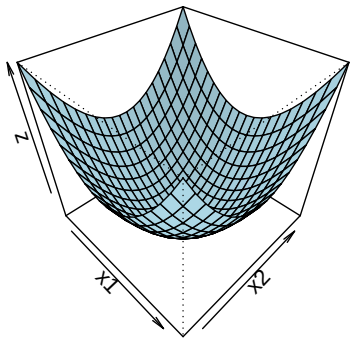
Plot a contour plot of the function:

```
par(mfrow=c(1,1), pty="s")

contour(x1_range, x2_range, z, nlevels=20, xlab=expression(x[1]), ylab=expression(x[2])
```




Now produce the three dimensional plot using the `persp` function from the base graphics package:

```
persp(x1_range, x2_range, z, theta=45, phi=40, xlab="x1",
      ylab= "x2", col = "lightblue", shade = 0.05)
```



Run the code below, changing eval = T in the chunk options. Notice the errors!

```
optim(par = 5 , fn = f2)

#gives an error
f2(5)

#the argument must be a vector of length 2
f2(c(2,7))
```

We get the incorrect solution using the default 'optim' call:

```
optim(par = c(5,5) , fn = f2)
```

```
## $par
## [1] 0.0001877005 0.0002589550
##
## $value
## [1] 1.022892e-07
##
## $counts
## function gradient
##       63       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

...but get the correct one using 'method="BFGS" ':

```
optim(par = c(5,5) , fn = f2, method= "BFGS")
```

```
## $par
## [1] -6.957677e-16 -6.957677e-16
##
## $value
## [1] 2.42044e-29
##
## $counts
## function gradient
##       11        3
##
```

```
## $convergence
## [1] 0
##
## $message
## NULL
```

We can set the maximum number of iterations 'optim' has in which to try converge on the optimal solution:

```
#it only has one chance!
optim(par = c(5,5) , fn = f2, method= "BFGS", control=list(maxit=1))
```

```
## $par
## [1] 3 3
##
## $value
## [1] 18
##
## $counts
## function gradient
##        3        2
##
## $convergence
## [1] 1
##
## $message
## NULL
```

```
#it can use up to 1000 iterations
optim(par = c(5,5) , fn = f2, method= "BFGS",control=list(maxit=1000))
```

```
## $par
## [1] -6.957677e-16 -6.957677e-16
##
## $value
## [1] 2.42044e-29
##
## $counts
## function gradient
##       11        3
##
## $convergence
## [1] 0
##
```

```
## $message
## NULL
```

Sometimes, things can converge really quickly:

```
for(i in 1:20){

  fit <- optim( par = c(5,5) , fn = f2, method= "BFGS",
       control=list(maxit=i))

  print(fit$par)
}
```

```
## [1] 3 3
## [1] 3 3
## [1] 5.435652e-12 5.435652e-12
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
## [1] -6.957677e-16 -6.957677e-16
```

The algorithm converges at the fourth iteration.

## 9.6 Maximum likelihood estimation

See 1 & 2 for some good introductory reading. Maximum likelihood estimation (MLE) is the process of using the observed data to estimate the value of $\theta$ by maximising the likelihood function. Essentially you iteratively change the values of the parameters of the objective function to maximise the probability of your observed data.

### 9.6.1 The 'likelihood'

The likelihood is a key concept in the realm of probability statistics. We will introduce via an example:

Assume we observe the outcome of many coin tosses. e.g HTTH…T. A model of each experiment is denoted as:

$$X_i \sim Bern(\theta)$$

where $\theta$ is

$$Pr(X_i = H) = \theta.$$

Here $X_i$ represents the outcome of the $i^{th}$ experiment. We assume that $\theta$ is constant for all experiments.

We now have the following joint probabilities for the following three cases:

$$Pr(X_1 = H) = \theta$$

$$Pr(X_1 = H, X_2 = T) = \theta(1-\theta)$$

$$Pr(X_1 = H, X_2 = T, X_3 = T) = \theta(1-\theta)(1-\theta)$$

Assume that we have now observed $n$ experiments, where each experiment is independent of the other. In this case the joint probability mass function is:

$$p(x_1, x_2, \dots, x_n) = p(x_1) \times p(x_2) \times p(x_n).$$

The above mass function is in fact a function of $\theta$ and is denoted as:

$$p(x_1, x_2, ..., x_n|\theta) = p(x_1|\theta) \times p(x_2|\theta) \times p(x_n|\theta) = \prod p(x_i|\theta)$$

This equation is termed the "likelihood function". In general,

$$p(x|\theta) = \theta^x(1-\theta)^{1-x}$$

where $x = 0$ or $x = 1$. The likelihood function in this example thus simplifies to

$$P(x_1, x_2, ..., x_n|\theta) = \prod_i \theta^{x_i}(1-\theta)^{1-x_i} = \theta^{\sum_i x_i}(1-\theta)^{n-\sum_i x_i}$$

The likelihood function is often denoted as $L(\boldsymbol{x}, \theta)$.

### 9.6.2 Maximum likelihood estimation tutorial

When conducting MLE in R, the objective function passed to your optimiser (e.g. `optim`) is the likelihood function. You thus need to code the likelihood function and then pass it to the optimiser. Here we code the likelihood function derived above and perform MLE using `optim`.

```
#generate some 'observed' data
x <- c(0,1,1,1,1,1,1,0,0,1,1,1,0,1,1)

#code a likelihood function
likelihood <- function(theta, x){
  sx <- sum(x)
  n <- length(x)

  (theta^sx)*( (1-theta)^(n-sx) )
}
#note that the likelihood function takes both the data and the parameters as its argum
```

Perform MLE by passing `likelihood` to `optim`:

```
#experiment with different starting values for theta
optim(0.5, fn = likelihood, x=x, control=list(fnscale=-1),
      method= "Brent", lower=0, upper=1)
```

```
## $par
## [1] 0.7333333
##
## $value
## [1] 0.0001667979
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Make sure you read the help file on the `optim` function! What does the argument `fnscale=-1` do?

Notice that the maximum likelihood estimate is:

$$\hat{\theta} = 0.7333333 = \frac{11}{15}$$

If we work with the logarithm of the likelihood function (the log-likelihood function) we will get the same result:

```
## $par
## [1] 0.7333333
##
## $value
## [1] -8.698728
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Notice we obtain the same parameter value. The reason for this is because the logarithm is a monotonically increasing function of its argument. This implies that maximising the log-likelihood is the same as maximising the likelihood function, resulting in the same optimal parameter value.

In practice, using the log-likelihood is most often preferred as it simplifies the mathematics involved in arriving at a final form of the likelihood. It removes the risk of the optimiser not converging because of the computer running out of floating point precision that results from having to calculate the product of a large number of small probabilities (as opposed to the sum of a large number of small probabilities).

### 9.6.3 Poisson distribution example

Run the following code with eval=T in the chunk options and see if you understand it. Why do you get an error?

```
#generate some count data (note that count data is known to be well-modelled by the Poisson distr
counts <- c(3, 1, 1, 3, 1, 4, 3, 2, 0, 5, 0, 4, 2)
#code the log-likelihood function of the Poisson distribution:
pois.ll <- function(lambda) {
```

```
  #log-likelihood function
  sum(dpois(y, lambda, log = TRUE))
}
optim(par = 5, fn = pois.ll, y=counts)
```

You need to pass the data and parameter to the log-likelihood and subsequent `optim` function

```
counts <- c(3, 1, 1, 3, 1, 4, 3, 2, 0, 5, 0, 4, 2)
pois.ll <- function(lambda, y) {
  sum(dpois(y, lambda, log = TRUE))
}

optim(par = 5, fn = pois.ll, y=counts)
```

```
## Warning in optim(par = 5, fn = pois.ll, y = counts): one-dimensional optimization by
## use "Brent" or optimize() directly
```

```
## $par
## [1] 9.046257e+74
##
## $value
## [1] -1.176013e+76
##
## $counts
## function gradient
##      502       NA
##
## $convergence
## [1] 1
##
## $message
## NULL
```

What is happening here? Take note of the convergence value and the value of `lambda`. Did the MLE find the global optimum?

Here we use the negative log-likelihood function:

```
pois.ll <- function(lambda, y) {
  -sum(dpois(y, lambda, log = TRUE))
}
optim(par = 5, fn = pois.ll, y=counts)
```

```
## Warning in optim(par = 5, fn = pois.ll, y = counts): one-dimensional optimization by Nelder-Me
## use "Brent" or optimize() directly
```

```
## $par
## [1] 2.230957
##
## $value
## [1] 23.63712
##
## $counts
## function gradient
##       30       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Can you explain why using the negative log-likelihood has resulted in the algorithm converging on the optimal solution?

Finally: Why does this not work?

```
optim(par = -5, fn = pois.ll, y=counts)
```

Why does this work?

```
optim(par = 5, fn = pois.ll, y=counts,
      method="L-BFGS-B",
      lower = 0.0001,
      upper=Inf)
```

The parameter `lambda` of the Poisson distribution must be positive!

## 9.7 Prac 15

- Note that the questions below are meant to be challenging!
- They are also designed to represent common situations that you face repeatedly throughout your career:
  - Problems that you have seen before, and have a good idea of how to tackle (Part 1a)

- Problems that you have seen before, but are not familiar with the proposed method (Parts 1b & c)
- Problems that, at first glance, you have no idea how to solve and little/no knowledge of the proposed method! (Part 2)

### 9.7.1 Maximum likelihood estimation

Assume these 13 observations are from a Poisson distribution, with rate parameter $\lambda$:

```
counts <- c(3, 1, 1, 3, 1, 4, 3, 2, 0, 5, 0, 4, 2)
```

The log-likelihood is

```
pois.ll <- function(lambda) {
  sum(dpois(y, lambda, log = TRUE))
}
```

a) Use `optim()` and `nlm()` (non-linear minimization) to find the MLE for $\lambda$.

b) Use Newton's method (from scratch) to find the MLE for $\lambda$ in the above problem.

$$loglik \propto \sum_i (x_i \log(\lambda) - \lambda)$$

c) Plot the function and check your answer.

Please submit your .Rmd and HTML files on the Assignments tab on Vula.

# Chapter 10

# More optimisation!

## 10.1 Prac 16

Today you will be tackling three more optimisation problems. If you have not done so already, please collaborate with your team mates in solving these problems.

Please submit your .Rmd and HTML files under the Assignment tab.

Recall: We can optimise a function using the `optim()` function:

```
optim(par, fn, gr, method, control, hessian)
```

- fn: the function to be optimised (minimise is the default)
- par: initial parameter(s) guess
- gr: gradient function; only needed for some methods
- method: what optimisation method to use (defaults to "Nelder-Mead" (which does not use gradients))
- control: optional list of control settings (maximum number of iterations, tolerance for convergence, fnscale = -1 for maximisation)
- hessian: should the final Hessian be returned? default FALSE

Use `optimize()` for one-dimensional problems

### 10.1.1 Part 1: Simple linear regression

We can undertake regression by using the method of least squares. This is done by minimising

$$A = \sum_{i=1}^{n} (y_i - a - \beta x_i)^2$$

where n is the number of observations, and we assume we have only one explanatory variable.

a) Download and import the the "tobacco.txt" set from Vula –> Resources –> data. The response variable `burn` measures the rate of cigarette burn in inches per 1000 seconds while the explanatory variables are percentages of total nitrogen, chlorine, potassium, phosphorous, calcium and magnesium. It is assumed that the true relationship between the burn rate and the explanatory variables is linear.

   - Explore the data, in particular the relationships between the response and each of the explanatory variables. Show some evidence of your exploratory data analysis and display the relationships on a single plot.
   - Build a correlation matrix that displays the correlations between the response and each of the explanatory variables.

Hint: you can use built-in R functions for the above. So do some exploration to find R packages with suitable functions.

b) Fit a simple linear regression model using `lm()` and print the model output. Choose the most pertinent explanatory variable to model the response variable, providing a motivation for your choice from the output in (a).

c) From first principles, create a function to fit a simple linear regression model, and find the least squares estimates using `optim()`. Compare the results to those found in (b).

d) It can easily be shown that the log-likelihood function in the above case is:

$$l = -0.5nlog(2\pi) - 0.5 * n * log(\sigma^2) - 0.5 * \frac{1}{\sigma^2} \sum_{i=1}^{n} (y_i - a - bxi)^2$$

Use `optim()` to maximize the above log-likelihood. Once again, assume that you only have one explanatory variable.

Take note that you now also have to estimate $\sigma$ and that it has to be positive. Read up on the use of `optim()` and how to set constraints on parameters. You might also consider transforming $\sigma$ to some other variable.

### 10.1.2 Part 2: Multiple Regression in R

Using the `tobacco` data set, let's fit a multiple regression model.

a) Using output from Part 1, fit a simple linear model using the most appropriate predictor. Take note of the correlation between the response variable and the fitted values of the regression. The square of this amount is known as the coefficient of determination ('Multiple R-squared' or simply $R^2$). Explain what this quantity represents and why a larger $R^2$ is preferred. Use the summary output and take note of the $R^2$ amount. Comment on the significance of the beta parameters and whether regression analysis should be undertaken on this data.

b) Add the remaining variables to the regression model one at a time. Add the variable with the next largest (absolute) correlation coefficient between with the response variable and so forth. Take note of the correlation coefficient between the fitted values of the response variable for the new regression equations and the response variable, the $R^2$ value and the significance of the beta parameters.

c) Fit the full model. i.e. the model that includes all of the explanatory variables. Comment on the estimation results.

d) Suggest a suitable final model/s. Explain your choice. Are all of the beta coefficients significant? Is the F statistic significant?

e) Once you have chosen a suitable model you now have to check whether or not the assumptions made at the start of the estimation phase are satisfied. i.e. Are the residuals normally distributed? What is mean of the residual series? Are the residuals homoscedastic? Are the residuals independent?

Plot the histogram of the estimated residuals. Do they look normally distributed? (don't always trust your eye). Use the `car` package and plot the QQ plot of the estimated residuals. (A QQ plot can be produced by first ordering the estimated residuals in ascending order and then using a standard normal to calculate the quantiles associated with the ordered residuals. If the residuals are normally distributed the plotted values should lie close to a straight line.)

f) Formal tests of normality can be undertaken by using the ks.test and the shapiro.test function. Use these functions to test whether the residuals are normally distributed.

g) Write a function to fit a multiple regression model from first principles. Use this function and `optim()` to find the MLE estimates for the coefficients of your chosen model in (e).

Table 10.1: Counts of flour beetles in all stages of development over 154 days.

| Days | Beetles |
|---|---|
| 0 | 2 |
| 8 | 47 |
| 28 | 192 |
| 41 | 256 |
| 63 | 768 |
| 79 | 896 |
| 97 | 1120 |
| 117 | 896 |
| 135 | 1184 |
| 154 | 1024 |

### 10.1.3 Part 3: Beetles population optimisation

The following table provides counts of a flour beetle (Tribolium confusum) population at various points in time. Beetles in all stages of development were counted, and the food supply was carefully controlled.

An elementary model for population growth is the logistic model given by:

$$\frac{dN}{dt} = rN(1 - \frac{N}{K})$$

where N is population size, t is time, r is a growth rate parameter, and K is a parameter that represents the population carrying capacity of the environment. The solution to this differential equation is given by:

$$N_t = f(t) = \frac{KN_0}{N_0 + (K - N_0)exp(-rt)}$$

where $N_t$ denotes the population side at time t.

a) Fit the logistic growth model to the flour beetle data using R's nlm function to minimise the sum of squared errors between model predictions and observed counts. Explain the important steps you have followed.

b) Doing your own research, provide approximate confidence intervals for the parameter estimates. Show the fitted model together with the observed data. Discuss your results briefly.

c) In many population modelling applications, an assumption of log-normality is adopted. The simplest assumption would be that the $logN_t$ are independent and normally distributed with mean $logf(t)$ and variance

$\sigma^2$. Find the MLEs under this assumption, using the Newton-Raphson method (again, do your own research). Provide standard errors for your parameter estimates, and an estimate of the correlation between them. Comment. Compare to the previous results.

Hints: hessian and jacobian matrices

# Chapter 11

# Parallel and High performance computing

## 11.1 Parallel computing in R

Take the time to read through these notes on Vula (in this order):

- Introduction to parallel computing
- Getting Started with doParallel and foreach

To perform parallel computing on a single machine, you need doParallel and foreach packages:

```
#install.packages("doParallel", dependencies=TRUE)
library(doParallel)
```

### 11.1.1 Foreach

A usual for loop

```
x <- for(i in 1:3){ sqrt(i) }
# does not return anything
```

A foreach loop

```
x <- foreach(i = 1:3) %do%{ sqrt(i) }
# returns an object (default is a list)

x <- foreach(i = 1:3, .combine="c") %do%{ sqrt(i) }
# returns a vector

x <- foreach(i = 1:3, .combine="cbind") %do%{ sqrt(i)^(1:3) }
# returns a matrix
```

### 11.1.2 doParallel

```
#how many cores are available?
detectCores()

# register a cluster (you always start with these commands)
cl <- makeCluster(3)
registerDoParallel(cl)

# FOREACH LOOP IN PARALLEL
x <- foreach(i = 1:3, .combine="c") %dopar% sqrt(i)
# each iteration is executed on a separate processor

x <- foreach(i = 1:100, .combine="c") %dopar% sqrt(i)
# will still use 3 processors

# remember to always stop the cluster!!!
stopCluster(cl)
```

#### 11.1.2.1 An example

```
myfunc <- function(){
  x <- 0
  for(i in 1:10^6){
    x <- x + runif(1)
  }
  return(x)
}

# Suppose we need to execute this function three times...

# SERIAL PROCESSING:
```

```
results <- rep(NA,3)
system.time(
  for(i in 1:3){ results[i] <- myfunc() }
)

# PARALLEL PROCESSING:
library(doParallel)

# register a cluster
cl <- makeCluster(3)
registerDoParallel(cl)

system.time(
  results <- foreach(i = 1:3, .combine="c") %dopar% { myfunc() }
)
# each iteration is run on a separate processor

stopCluster(cl)
```

#### 11.1.2.2 Iris data example

```
library(doParallel)

#how many cores are available
detectCores()
```

```
## [1] 16
```

```
#register cores for parallel processing:
cl <- makeCluster(2)
registerDoParallel(cl)
#double-check that you are using > 1 core!
getDoParWorkers()
```

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

```
#parallel processing:
x <- iris[which(iris[,5] != "setosa"),c(1,5)]
trials <- 10000
ptime <- system.time({
r <- foreach(icount(trials), .combine=cbind) %dopar% {
ind <- sample(100, 100, replace=TRUE)
```

```
result1 <- glm(x[ind,2] ~ x[ind,1], family=binomial(logit))
coefficients(result1)
}
})[3]
ptime
```

```
## elapsed
##   10.25
```

```
#serial processing:
stime <- system.time({
r <- foreach(icount(trials), .combine=cbind) %do% {
  ind <- sample(100, 100, replace=TRUE)
  result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
  coefficients(result1)
}
})[3]
stime
```

```
## elapsed
##   16.94
```

```
stopCluster(cl)
```

Although the above is really a 'toy' example, it serves as a 'proof of concept' that you should utilise parallel processing when your problem is suitable and you have the computing resources (i.e. multiple cores).

CPU processors with multiple cores (6+) will become the norm in ~5 years time, so parallel processing will become increasingly accessible. Currently I am typing these words on a 2020 M1 Macbook Air with and 8-core CPU, 7-core GPU and 16-core Neural engine. Just ten years ago most laptops were running on single-core CPUs!

## 11.2 High performance computing

What are your options if the models you need to run need more computing power than what your own PC/laptop can provide? The first question to answer is whether your models would benefit from the parallel processing power of a GPU, or whether you simply just need a few more cores to run the model more efficiently? There are several options to access more computing power, some of which may be:

- Buy your own dedicated GPU. The cost of these is coming down year-on-year, but you may still have to fork out significant 'cash dollar' to get one of the higher-end cards (~ R20k to R25k)
- Gain access to a high performance cluster while you are a student/lecturer at a University
- Rent a remote server with the specifications that you need (i.e. cloud computing)

### 11.2.1 UCT HPC

*TL;DR: If you know linux then this is a viable option for you while you are a student or employed by UCT.*

UCT has its own dedicated high performance cluster. The cluster contains several CPU partitions, each with a minimum of 1 node and and 24 cores per node. It contains one dedicated GPU partition with four nodes and a minimum of 12 cores per node.

To gain access to the cluster you must apply for an account. Note that you would only qualify for an account while you are studying at or employed by UCT. Once your account has been approved and created, you can begin to schedule jobs on the cluster.

Currently there isn't a simple GUI approach to scheduling jobs on the cluster. You need to have a decent knowledge of linux or be willing to invest the time to learn the necessary commands to transfer data to and from a node, schedule a job etc. One also needs to install some form of graphical application in order to view the progress of a model running on the cluster.

We used to request accounts for all students in this course and experiment with scheduling simple jobs on the cluster, however it became increasingly difficult to arrange this administratively. Hence, we do not cover this topic in any further detail in this course.

For further reading, please consult the documentation under Resources –> Notes

### 11.2.2 Cloud-based computing

*TL;DR: if you are prepared to pay for it, cloud-based computing enables a GUI-based HPC option where you have greater control over the running of your model. The learning curve is also more gradual than the UCT HPC.*

An alternative to the above is cloud-based computing, which is becoming increasingly ubiquitous and easier to access. There are multiple platforms and different instance types from which to select an instance with suitable specifications for your model. The average cost per hour of renting these remote instances is gradually decreasing (or relatively decreasing by staying constant

year-on-year). One such option is Amazon Web Services Elastic Compute Cloud (AWS EC2).

#### 11.2.2.1 Amazon Web Services EC2

AWS EC2 is a good option to consider when starting out with cloud computing. There is a free tier that you can use after creating an account - however the server in that tier is most likely less powerful than your local laptop. Still, one can use the free tier while you are becoming acquainted with AWS. Once you feel comfortable with setting up an AWS instance, you can then choose to pay for an instance that has the specifications you require.

Get started with Amazon EC2

AWS has various categories of instances, optimised for different objectives. Currently it has the following categories:

```
- General purpose
- Compute optimised
- GPU
- Memory optimised
- Storage optimised
- Machine learning
```

There are four pricing options: On-demand, Spot instances, Savings plans & Reserved instances. I highly recommend the spot instance option, as it can be up to 60% cheaper than the on-demand pricing.

AWS also has an education program, AWS Educate, via which you can gain valuable cloud computing knowledge. From time to time you can earn AWS credits by completing some online courses on AWS Educate. You can then use these credits when running models on AWS EC2.

Louis Aslett has created a range of RStudio AMIs, greatly simplifying the process of creating a suitable RStudio server instance on AWS. You can simply go to here, select an AWS region and then continue with the AWS instance set up. If you have a Dropbox account, then you can sync it to the instance, to allow seamless file transfer. There is a short video on today's lesson page demonstrating how this is done.

Once you have setup and launched your RStudio instance, you would install the required packages for your model, load them, load your code (by simply creating a new script file and copying and pasting your code from your pc to the new script file) and then run your model.

**NB**: once your model has finished running, and you have not synced a Dropbox account, you will need to download whatever code and data you need to keep before you Terminate your instance. *Remember to Terminate the instance once you are done!*

#### 11.2.2.2 Other cloud-based computing options:

- Microsoft Azure
- Google Cloud Platform
- Alibaba Cloud
- IBM
- Digital Ocean (currently has a 60-day free trial with $100 credit)

## 11.3 Prac 17

*Note: in this prac you cannot access help from your tutor. You can, however collaborate with your team mates! YAY!!!*

The `pima.csv` dataset is based on the Pima Indians Diabetes Database that contains information on several variables used to predict whether a subject has diabetes. See here for more information. You will be fitting a logistic regression model and experimenting with cross-validation to try and accurately predict the onset of diabetes using one or more of the explanatory variables.

If you are not familiar with either logistic regression, cross-validation or Area under the curve (AUC), conduct your own research on these topics. If you are - help your team mates! Such fun!

Use an appropriate package and function for the parallel processing questions. Also make sure that your analysis is reporducible by setting the random seed where necessary.

1) Download the `pima` data from Vula under Resources –> data and import it into R.

2) Explore the data, in particular the relationships (if any) between the explanatory variables.

3) Split the data into a training set, a validation set and a test set, with a [60:20:20] ratio.

4) Using only one explanatory variable, fit a logistic regression model to the training data, using the validation data to assess the predictive accuracy of the model.

5) Write code to fit all possible models with three explanatory variables, and compute the accuracy and AUC of each model on the validation set.

   (a) Run this code in series, recording the total time taken to fit all the models
   (b) Run this code in parallel, recording the total time taken to fit all the models

   (c) Compute the ratio of the time in (a) to the time in (b)
   (d) Use the best-fitting model to predict on the test dataset

6) Write code to perform 5-fold cross validation on all possible logistic regression models with 4 explanatory variables (70 models in total). Use an initial [80:20] split to separate out the test dataset, and then a [70:30] split on the remaining data for each of the 5-folds. Compute the accuracy and AUC for each model.

   (a) Run this code in series, recording the total time taken to fit all the models
   (b) Run this code in parallel, recording the total time taken to fit all the models
   (c) Compute the ratio of the time in (a) to the time in (b)
   (d) Use the best-fitting model on AUC to predict on the test dataset.

7) Write code to perform leave-one-out cross validation all possible logistic regression models of seven variables (8 models in total). Compute the accuracy and AUC for each model.

   (a) Run this code in series, recording the total time taken to fit all the models
   (b) Run this code in parallel, recording the total time taken to fit all the models
   (c) Compute the ratio of the time in (a) to the time in (b)
   (d) Use best-fitting model on AUC to predict on the test dataset

8) Create a table of the values obtained in 5c), 6c) & 7c) and the values obtained in 5d), 6d) & 7d). Discuss the results with respect to the impact (if any) of parallelisation on training time and explain which model one would choose for this dataset and classification task.

Please submit your .Rmd and HTML files under the Assignments tab on Vula.