Simulation and Modeling to Understand the Change

Alejandro Martínez-Mingo & Manuele Leonelli

2022-01-20

Contents

Preface

These are lecture notes for the module *Simulation and Modelling to Understand Change* given in the School of Human Sciences and Technology at IE University, Madrid, Spain. Knowledge of basic elements of R programming as well as probability and statistics is assumed.

Chapter 1

Introduction

The first introductory chapter gives an overview of simulation, what it is, what it can be used for, as well as some examples.

1.1 What is simulation

A *simulation* is an imitation of the dynamics of a real-world process or system over time. Although simulation could potentially still be done "by hand", nowadays it almost always implicitly requires the use of a computer to create an artificial history of a system to draw inferences about its characteristics and workings.

The behavior of the system is studied by constructing a *simulation model*, which usually takes the form of a set of assumptions about the workings of the system. Once developed, a simulation model can be used for a variety of tasks, including:

- Investigate the behaviour of the system under a wide array of scenarios. This is also often referred to as "what-if" analyses;
- Changes to the system can be simulated before implementation to predict their impact in real-world;
- During the design stage of a system, meaning while it is being built, simulation can be used to guide its construction.

Computer simulation has been used in a variety of domains, including manifacturing, health care, transport system, defense and management science, among many others.

1.1.1 A simple simulation model

Suppose we decided to open a donut shop and are unsure about how many employees to hire to sell donuts to costumers. The operations of our little shop is the real-world system whose behavior we want to understand. Given that the shop is not operating yet, only a simulation model can provide us with insights.

We could of course devise models of different complexities, but for now suppose that we are happy with a simple model where we have the following elements:

- costumers that arrive at our shop at a particular rate;
- employees (of a number to be given as input) that take a specific time to serve costumers.

Implicitly, we are completely disregarding the number of donuts available in our shop and assuming that we have an infinite availability of these. Of course, in a more complex simulation model we may want to also include this element to give a more realistic description of the system.

1.1.2 Why simulate?

An alternative approach to computer simulation is direct experimentation. In the bagel shop setting, we could wait for the shop to open and observe its workings by having a different number of employees on different days. Considered against real experimentation, simulation has the following advantages:

- It is *cheaper* to implement and does not require a disruption of the real-world system;
- It is *faster* to implement and time can be compressed or expanded to allow for a speed-up or a slow-down of the system of interest;
- It can be *replicated* multiple times and the workings of the systems can be observed a large number of times;
- It is *safe* since it does not require an actual disruption of the system;
- It is *ethical* and *legal* since it can implement changes in policies that would be unethical or illegal to do in real-world.

Another alternative is to use a mathematical model representing the system. However, it is often infeasible, if not impossible, to come up with an exact mathematical model which can faithfully represent the system under study.

1.2 Types of simulations

Before starting the construction of a simulation model, we need to decide upon the principal characteristics of that model. There are various choices to be made, which depend upon the system we are trying to understand.

1.2.1 Stochastic vs deterministic simulations

A model is *deterministic* if its behavior is entirely predictable. Given a set of inputs, the model will result in a unique set of outputs. A model is *stochastic* if it has random variables as inputs, and consequently also its outputs are random.

Consider the donut shop example. In a deterministic model we would for instance assume that a new customer arrives every 5 minutes and an employee takes 2 minutes to serve a customer. In a stochastic model we would on the other hand assume that the arrival times and the serving time follows some random variables: for instance, normal distributions with some mean and variance parameters.

In this course we will only consider stochastic simulation, but for illustration we consider now an example of a deterministic simulation.

A social media influencer decides to open a new page and her target is to reach 10k followers in 10 weeks. Given her past experience, she assumes that each week she will get 1.5k new followers that had never followed the page and of her current followers she believes 10% will stop following the page each week. However, 20% of those that the left the page in the past will join again each week. Will she reach her target?

To answer this question we can construct a deterministic simulation. Let F_t the number of followers at week t and U_t the number of users that are unfollowing the profile at week t. Then

$$F_t = F_{t-1} + 1500 - L_t + R_t, \qquad U_t = U_{t-1} + L_t - R_t$$

where $L_t = 0.1 \cdot F_{t-1}$ is the number of unfollowers from time $t-1$ to time t, and $R_t = 0.2 \cdot U_{t-1}$ is the number of users that follow the page back from time $t-1$ to time t.

To compute the number of followers after ten weeks we can use the R code below. It does not matter if you do not understand it now, we will review R coding in the next chapters.

```
Ft <- Ut <- Lt <- Rt <- rep(0,11)
for (i in 2:11){
  Lt[i] <- 0.1*Ft[i-1]
  Rt[i] <- 0.2*Ut[i-1]
```

Table 1.1: Dataframe ‘result‘ from the social media deterministic simulation

Followers	Total.Unfollowers	Weekly.Unfollowers	Weekly.Returns
0.000	0.000	0.000	0.0000
1500.000	0.000	0.000	0.0000
2850.000	150.000	150.000	0.0000
4095.000	405.000	405.000	30.0000
5266.500	733.500	733.500	81.0000
6386.550	1113.450	1113.450	146.7000
7470.585	1529.415	1529.415	222.6900
8529.409	1970.591	1970.591	305.8830
9570.587	2429.413	2429.413	394.1181
10599.411	2900.589	2900.589	485.8827
11619.587	3380.413	3380.413	580.1179

```
  Ut[i] <- Ut[i-1] + Lt[i] - Rt[i]
  Ft[i] <- Ft[i-1] + 1500 - Lt[i] + Rt[i]
}
result <- data.frame("Followers" = Ft, "Total Unfollowers" = Ut,
           "Weekly Unfollowers" = Ut, "Weekly Returns" = Rt)
```

The dataframe `result` is reported in Table 1.1, showing that she will be able to hit her target of 10k followers since she will have 11619 followers. If we run again the simulation we will obtain the exact same results: there is no stochasticity/uncertainty about the outcome.

The above application could be transformed into a stochastic simulation by allowing the rate at which she gets new followers, unfollowers etc. to be random variables of which we do not know the exact value.

1.2.2 Static vs dynamic simulations

Simulation models that represent the system at a particular point in time only are called *static*. This type of simulations are often called as *Monte Carlo simulations* and will be the focus of later chapters.

Dynamic simulation models represent systems as they evolve over time. The simulation of the donut shop during its working hours is an example of a dynamic model.

1.2.3 Discrete vs continuous simulations

Dynamic simulations can be further categorized into discrete or continuous.

Discrete simulation models are such that the variables of interest change only at a discrete set of points in time. The number of people queuing in the donut shop is an example of a discrete simulation. The number of customers changes only when a new customer arrives or when a customer has been served. Figure 1.1 gives an illustration of the discrete nature of the number of customers queuing in the donut shop.

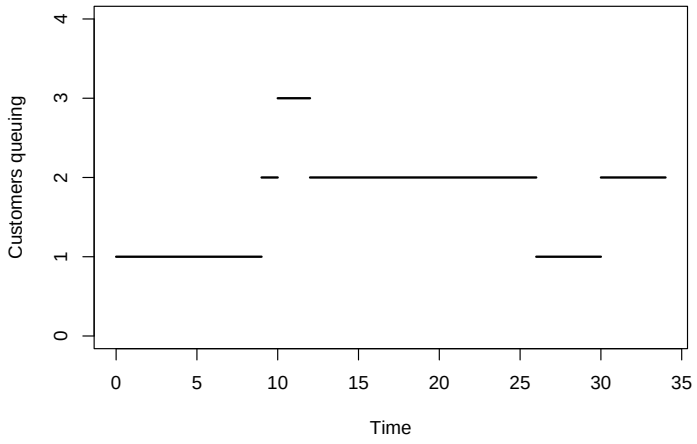


Figure 1.1: Example of a discrete dynamic simulation

Figure 1.1 further illustrates that for specific period of times the system does not change state, that is the number of customers queuing remains constant. It is therefore useless to inspect the system during those times where nothing changes. This prompts the way in which time is usually handled in dynamic discrete simulations, using the so-called *next-event technique*. The model is only examined and updated when the system is due to change. These changes are usually called *events*. Looking at Figure 1.1 at time zero there is an event: a customer arrives; at time nine another customer arrives; at time ten another customer arrives; at time twelve a customer is served; and so on. All these are examples of events.

Continuous simulation models are such that the variables of interest change continuously over time. Suppose for instance a simulation model for a car journey was created where the interest is on the speed of the car throughout the journey. Then this would be a continuous simulation model. Figure 1.2 gives an illustration of this.

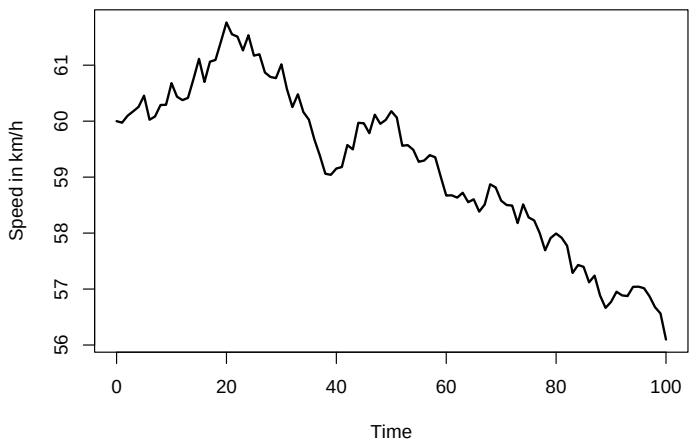


Figure 1.2: Example of a discrete dynamic simulation

In later chapters we will focus on discrete simulations, which are usually called *discrete-event simulation*.

1.3 Elements of a simulation model

We next introduce some terminology which we will need in the following.

1.3.1 Objects of the model

There are two types of objects a simulation model is often made of:

- *Entities*: individual elements of the system that are being simulated and whose behavior is being explicitly tracked. Each entity can be individually identified;
- *Resources*: also individual elements of the system but they are not modelled individually. They are treated as countable items whose behavior is not tracked.

Whether an element should be treated as an entity or as a resource is something that the modeller must decide and depends on the purpose of the simulation. Consider our simple donut shop. Clients will be most likely be resources since

we are not really interested in what each of them do. Employees may either be considered as entities or resources: in the former case we want to track the amount of time each of them are working; in the latter the model would only be able to output an overview of how busy overall the employees are.

1.3.2 Organization of entities and resources

- *Attributes*: properties of objects (that is entities and resources). This is often used to control the behavior of the object. In our donut shop an attribute may be the state of an employee: whether she is busy or available. In a more comprehensive simulation, an attribute might be the type of donut a customer will buy (for instance, chocolate, vanilla or jam).
- *State*: collection of variables necessary to describe the system at any time point. In our donut shop, in the simplest case the necessary variables are number of customers queuing and number of busy employees. This fully characterizes the system.
- *List*: collection of entites or resources ordered in some logical fashion. For instance, the customers waiting in our shop may be ordered in the so-called "fist-come, first-served" scheme, that is customers will be served in the order they arrived in the shop.

1.3.3 Operations of the objects

During a simulation study, entities and resources will cooperate and therefore change state. The following terminology describe this as well as the flow of time:

- *Event*: instant of time where the state of the system changes. In the donut shop suppose that there are currently two customers being served. An event is when a customer has finished being served: the number of busy employees decreases by one and there is one less customer queuing.
- *Activity*: a time period of specified length which is known when it begins (although its length may be random). The time an employee takes to serve a customer is an example of an activity: this may be specified in terms of a random distribution.
- *Delay*: duration of time of unspecified length, which is not known until it ends. This is not specified by the modeller ahead of time but is determined by the conditions of the system. Very often this is one of the desired output of a simulation. For instance, a delay is the waiting time of a customer in the queue of our donut shop.
- *Clock*: variable representing simulated time.

1.4 The donut shop example

Let's consider in more details the donut shop example and let's construct and implement our first simulation model. At this stage, you should not worry about the implementation details. These will be formalized in more details in later chapters.

Let's make some assumptions:

- the queue in the shop is possibly infinite: whenever a customer arrives she will stay in the queue independent of how many customers are already queuing and she will wait until she is served.
- customers are served on a first-come, first-served basis.
- there are two employees. On average they take the same time to serve a customer. Whenever an employee is free, a customer is allocated to that employee. If both employees are free, either of the two starts serving a customer.

The components of the simulation model are the following:

- **System state**: $N_C(t)$ number of customers waiting to be served at time t; $N_E(t)$ number of employees busy at time t.
- **Resources**: customers and employees;
- **Events**: arrival of a customer; service completion by an employee.
- **Activities**: time between a customer arrival and the next; service time by an employee.
- **Delay**: customers' waiting time in the queue until an employee is available.

From an abstract point of view we have now defined all components of our simulation model. Before implementing, we need to choose the length of the activities. This is usually done using common sense, intuition or historical data. Suppose for instance that the time between the arrival of customers is modeled as an Exponential distribution with parameter 1/3 (that is on average a customer arrives every three minutes) and the service time is modeled as a continuous Uniform distribution between 1 and 5 (on average a service takes three minutes).

With this information we can now implement the workings of our donut shop. It does not matter the specific code itself, we will learn about it in later chapters. At this stage it is only important to notice that we use the `simmer` package together with the functionalities of `magrittr`. We simulate our donut shop for two hours.

```
library(simmer)
library(magrittr)
set.seed(2021)

env <-  simmer("donut shop")

customer <- trajectory("customer") %>% seize("employee", 1) %>%
  timeout(function() runif(1,1,5)) %>% release("employee", 1)

env %>%
  add_resource("employee", 2) %>%
  add_generator("customer", customer, function() rexp(1,1/3))

env %>%
  run(until=120)
```

The above code creates a simulation of the donut shop for two hours. Next we report some graphical summaries that describe how the system worked.

```
library(simmer.plot)
library(gridExtra)
p1 <- plot(get_mon_resources(env), metric = "usage", items = "server",step = T)
p2 <- plot(get_mon_arrivals(env), metric = "waiting_time")

grid.arrange(p1,p2,ncol=2)
```

The left plot in Figure 1.3 reports the number of busy employees busy throughout the simulation. We can observe that often no employees were busy, but sometimes both of them are busy. The right plot in Figure 1.3 reports the waiting time of customers throughout the simulation. Most often customers do not wait in our shop and the largest waiting time is of about four minutes.

Some observations:

- this is the result of a single simulation where inputs are random and described by a random variable (for instance, Poisson and Uniform). If we were to run the simulation again we would observe different results.
- given that we have built the simulation model, it is straightforward to change some of the inputs and observe the results under different conditions. For instance, we could investigate what would happen if we had only one employee. We could also investigate the use of different input parameters for the customer arrival times and the service times.

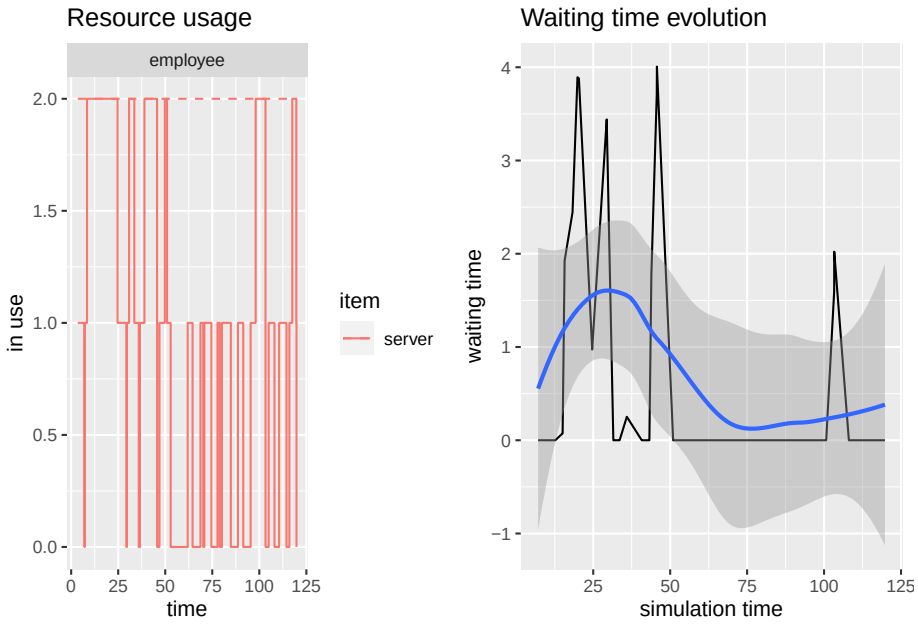


Figure 1.3: Graphical summaries from the simulation of the donut shop

1.5 Simulating a little health center

Consider now a slightly more complex example where we want to simulate the workings of a little health center. Patients arrive at the health center and are first visited by a nurse. Once they are visited by the nurse they have an actual consultation with a doctor. Once they are finished with the doctor, they meet the administrative staff to schedule a follow-up appointment.

We make the following assumptions:

- as before we assume queues to be infinite and that patients do not leave the health center until they are served by the administrative staff;
- at all steps patients are visited using a first-come, first-served basis
- the health center has one nurse, two doctors and one administrative staff. The two doctors take on average the same time to visit a patient.

The components of the simulation model are the following:

- **System state**:
  - $Q_N(t)$: number of patients queuing to see the nurse;

  - $Q_D(t)$: number of patients queing to see a doctor;
  - $Q_A(t)$: number of patients queuing to see the staff;
  - $N_N(t)$: number of nurses available to visit patients;
  - $N_D(t)$: number of doctors available to visit patients;
  - $N_A(t)$: number of administrative staff available to visit patients.

- **Resources**: patients, nurses, doctors and administrative staff;
- **Events**: arrival of a patient, completion of nurse's visit, completation of doctor's visit, completion of administrative staff's visit.
- **Activities**: time between the arrival of a patient and the next, visit's times of nurses, doctors and admin staff.
- **Delay**: customers' waiting time for nurses, doctors and administrative staff

We further assume the following activities:

- Nurse visit times follow a Normal distribution with mean 15 and variance 1;
- Doctor visit times follow a Normal distribution with mean 20 and variance 1;
- Administrative staff visit times follow a Normal distribution with mean 5 and variance 1;
- Time between the arrival of patients is modeled as a Normal with mean 10 and variance 4.

The model above can be implemented using the following code (we run the simulation for four hours). Again do not worry about it now!

```
set.seed(2021)
env <- simmer("HealthCenter")

patient <- trajectory("patients' path") %>%
  seize("nurse", 1) %>%
  timeout(function() rnorm(1, 15)) %>%
  release("nurse", 1) %>%
  seize("doctor", 1) %>%
  timeout(function() rnorm(1, 20)) %>%
  release("doctor", 1) %>%
  seize("administration", 1) %>%
  timeout(function() rnorm(1, 5)) %>%
```

```
  release("administration", 1)

env %>%
  add_resource("nurse", 1) %>%
  add_resource("doctor", 2) %>%
  add_resource("administration", 1) %>%
  add_generator("patient", patient, function() rnorm(1, 10, 2))

env %>% run(240)
```

Let's look at some summary statistics.

```
plot(get_mon_resources(env), metric = "utilization")
```

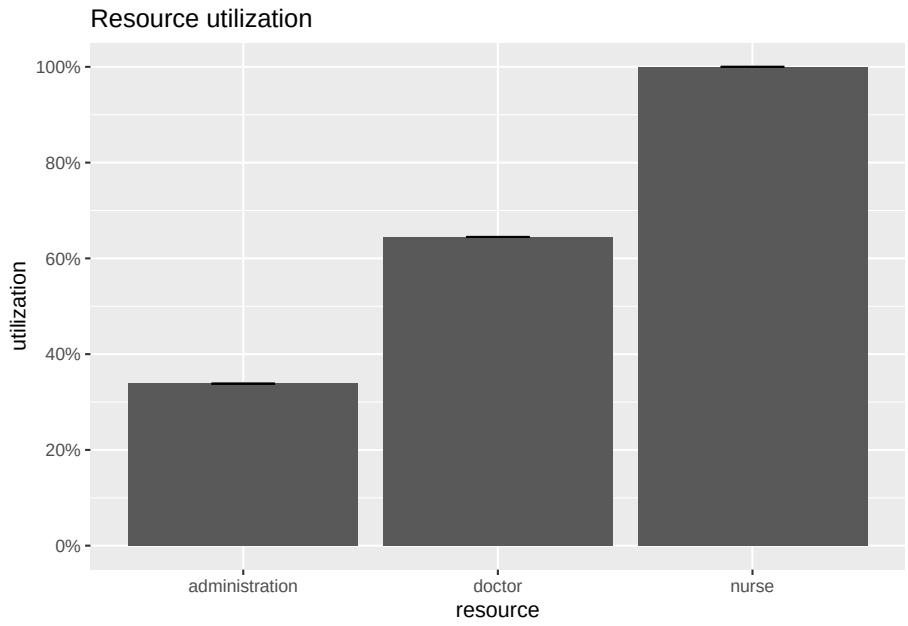


Figure 1.4: Utilization of the resources in the health center

Figure 1.4 shows the utilization of the different resources in the system. Nurses are most busy, doctors are overall fairly available, whilst the administration is more than half of the time available.

```
plot(get_mon_resources(env), metric = "usage", item = "server")
```

Figure 1.5 confirms this. We see that the usage of nurses is almost 1, whilst for

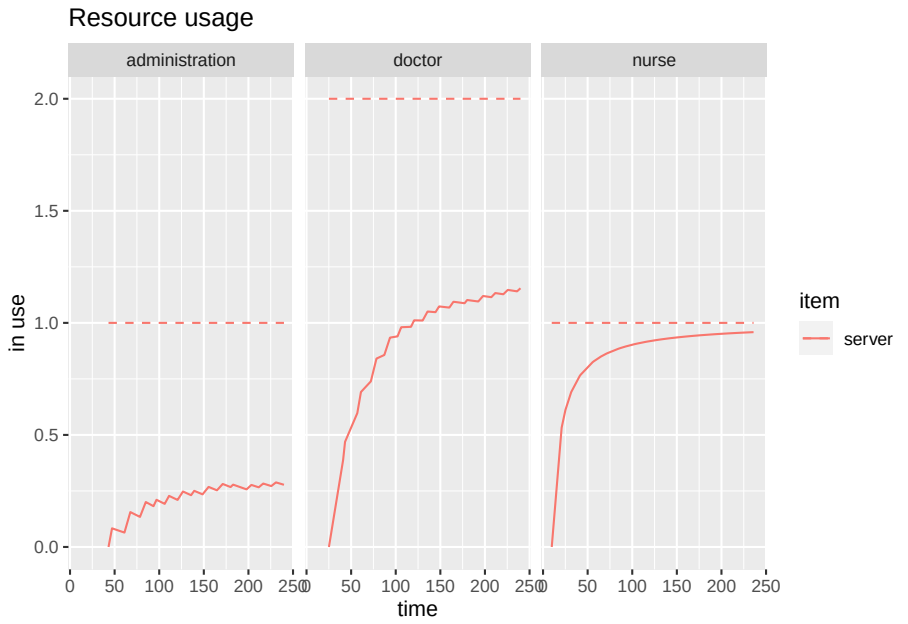


Figure 1.5: Usage of the resources in the health center

doctors and administrative staff we are below the number of doctors and staff available.

```
plot(get_mon_arrivals(env), metric = "flow_time")
```

Last Figure 1.6 reports the average time spent by patients in the health center. We can see that as the simulation clock increases, patients spend more time in the health center. From the previous plots, we can deduce that in general patients wait for the nurse, who has been busy all the time during the simulation.

1.6 What's next

The previous examples should have given you an idea of what a simulation model is and what you will be able to implement by the end of the course. However, it will take some time before we get to actually simulate systems. There are various skills that you will need to learn or revise before being able to implement simulation in R yourself. Specifically:

- first we will review the basics of R programming;
- we will then review basic elements of probability and statistics;

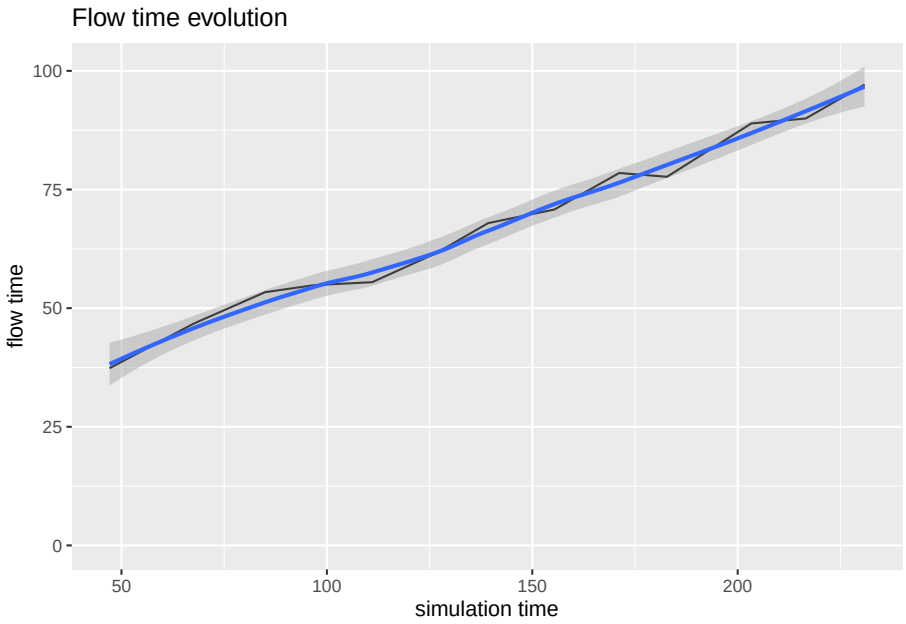


Figure 1.6: Time spent in the health center

- we will discuss how randomness is implemented in programming languages and in R;
- at this stage you will be able to implement your first simple simulations. In particular we will start with static simulation, also called *Monte Carlo* simulation
- we will then look at dynamic simulations as in the previous examples.

Chapter 2

R programming

R is a programming language most commonly used within the statistical and machine learning community. This chapter will review some of the elements of R programming that will be used in later chapters. Do not expect this chapter to be exhaustive or self-contained. It is intended to give a quick refresh of R for users that have at least some experience with this programming language. There are many topics and concepts which are fundamental but will not be reviewed in this chapter. However, you should aim to master the topics included in this chapter since they will appear again later on in these notes. There are many other resources if you want to have a more in-depth look into R programming.

- The books of Hadley Wickham are surely a great starting point and are all available here.
- If you are unsure on how to do something with R, Google it!!! The community of R users is so wide that surely someone else has already asked your same question.
- The R help is extremely useful and comprehensive. If you want to know more about a function, suppose it is called function, you can type `?function`.

2.1 Why R?

As mentioned in the previous chapter, simulation is very often applied in many areas, for instance management science and engineering. Often a simulation is carried out using an Excel spreadsheet or using a specialised software whose only purpose is creating simulations. Historically, R has not been at the forefront of the implementation of simulation models, in particular of discrete-event simulations. Only recently, R packages implementing discrete-event simulation have

appeared, most importantly the `simmer` R package that you will learn using in later chapters.

These notes are intended to provide a unique view of simulation with specific implementation in the R programming language. Some of the strenght of R are:

- it is free, open-source and available in all major operating systems;
- the community of R users is huge, with many forums, sites and resources that give you practical support in developing your own code;
- a massive set of add-on packages to increase the capabilities of the basic R environment;
- functions to perform state-of-the-art statistical and machine-learning methods. Researchers sometimes create an associated R package to any article they publish so for others to use their methods;
- the integrated development environment RStudio provides a user-friendly environment to make the R programming experience more pleasing;
- powerful communication tools to create documents and presentations embedding R code and R output. As a matter of fact this very book is created in R!!!!

2.2 Section Bibliography

- Crawley, M. J. (2012). The R book. John Wiley & Sons.

2.3 R basics

So let's get started with R programming!

2.3.1 Introduction to R

R is an *Open Source*, powerful, flexible and extensible statistical language. It is used by many companies (Google, Microsoft, Facebook, BBVA, etc…) and universities by Statisticians and Data Scientists in software development. Unlike traditional spreadsheets, in R programming sentences are written instead of the classic formulas. It is necessary to know the structure of the data. Prototypes can be made with a few lines of code.

2.3.2 R History

R is an implementation of the statistical language S (combined with the programming language Scheme). S was developed in the AT&T labs by John Chambers in the late 1970s. The two main implementations of S are:

- R
- S+ (S-PLUS)

There are usually several releases a year (usually the most important in April):

- 3.1.0 (Spring Dance) 10/04/2014
- 3.2.0 (Full of Ingredients) 16/04/2015
- 3.5.0 (Joy in Playing) 23/04/2018
- 4.0.0 (Bunny-Wunnies Freak Out) 24/04/2020
- 4.1.0 (Camp Pontanezen) 18/05/2021

2.3.3 R Advantages

R is a great software for solving data analysis problems. There are many packages for data processing, statistical modelling, data mining and graphics. There is a community of users creating packages called the R project.

R is very useful for making graphs, analyzing data and obtaining statistical models with data that fit in the RAM memory of the PC. There are limitations, from a memory point of view, with large volumes of data. It is very common to use another resources to prepare the data:

- Small or medium volumes: Python, Julia, Perl…
- Large Volumes: Spark, Hadoop, Pig, Hive…

2.3.4 What do we mean by R?

By R we usually mean:

- The programming language.
- The interpreter who executes the code written in R.
- The graphics generation system of R.
- The R programming IDE, or also known as RStudio (includes the R interpreter, graphics system, package manager and user interface).

2.3.5 Console Mode

To open the **R console**, run from the command line (Terminal in Mac):

```
$>R
```

The console opens, which allows you to write commands interactively. Each of these commands is called **expressions**. The **R interpreter** reads these expressions and responds with the result or an error message. The command interface will store the steps followed when analyzing the data.

The `history()` command displays the history of commands entered during the **R session**. Names of variables, packages, directories, etc. are auto-completed using **tabulator**. If the name of a function is written in the console, its code is displayed. For example: `history`

```
history
```

```
## function (max.show = 25, reverse = FALSE, pattern, ...)
## {
##     file1 <- tempfile("Rrawhist")
##     savehistory(file1)
##     rawhist <- readLines(file1)
##     unlink(file1)
##     if (!missing(pattern))
##         rawhist <- unique(grep(pattern, rawhist, value = TRUE,
##             ...))
##     nlines <- length(rawhist)
##     if (nlines) {
##         inds <- max(1, nlines - max.show):nlines
##         if (reverse)
##             inds <- rev(inds)
##     }
##     else inds <- integer()
##     file2 <- tempfile("hist")
##     writeLines(rawhist[inds], file2)
##     file.show(file2, title = "R History", delete.file = TRUE)
## }
## <bytecode: 0x55d282ecf170>
## <environment: namespace:utils>
```

2.3.6 Getting help in R

The simplest way to get help in R is to click on the Help button on the toolbar of the RGui window (this stands for R's Graphic User Interface).

However, if you know the name of the function you want help with, you just type a question mark `?` at the command line prompt followed by the name of the function. So to get help on `read.table`, just type:

```
?read.table
```

Sometimes you cannot remember the precise name of the function, but you know the subject on which you want help (e.g. data input in this case). Use the help.search function (without a question mark) with your query in double quotes like this:

```
help.search("read tables")
```

Other useful functions are `find` and `apropos`. The `find` function tells you what package something is in:

```
find("mean")
```

```
## [1] "package:base"
```

while `apropos` returns a character vector giving the names of all objects in the search list that match your (potentially partial) enquiry:

```
apropos("lm")
```

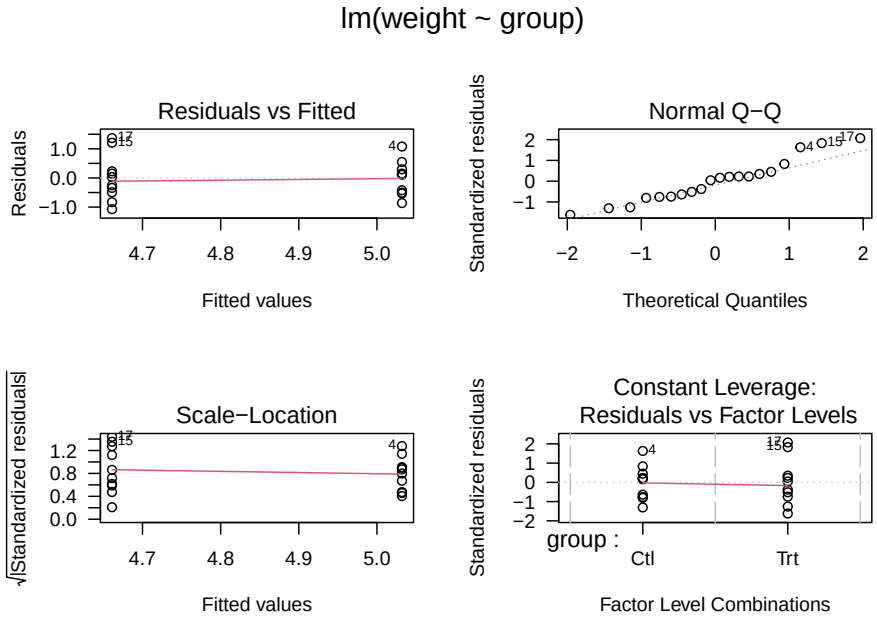
```
##  [1] ".colMeans"       ".lm.fit"         "colMeans"        "confint.lm"
##  [5] "contr.helmert"   "dummy.coef.lm"   "glm"             "glm.control"
##  [9] "glm.fit"         "KalmanForecast"  "KalmanLike"      "KalmanRun"
## [13] "KalmanSmooth"    "kappa.lm"        "lm"              "lm.fit"
## [17] "lm.influence"    "lm.wfit"         "model.matrix.lm" "nlm"
## [21] "nlminb"          "predict.glm"     "predict.lm"      "residuals.glm"
## [25] "residuals.lm"    "summary.glm"     "summary.lm"
```

To see a worked example just type the function name (e.g. linear models, `lm`) and you will see the printed and graphical output produced by the `lm` function:

```
example(lm)
```

```
##
## lm> require(graphics)
##
## lm> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## lm> ## Page 9: Plant Weight Data.
```

```
## lm> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
##
## lm> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
##
## lm> group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
##
## lm> weight <- c(ctl, trt)
##
## lm> lm.D9 <- lm(weight ~ group)
##
## lm> lm.D90 <- lm(weight ~ group - 1) # omitting intercept
##
## lm> ## No test:
## lm> ##D anova(lm.D9)
## lm> ##D summary(lm.D90)
## lm> ## End(No test)
## lm> opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
##
## lm> plot(lm.D9, las = 1)      # Residuals, Fitted, ...
```




```
##
## lm> par(opar)
##
## lm> ## Don't show:
```

```
## lm> ## model frame :
## lm> stopifnot(identical(lm(weight ~ group, method = "model.frame"),
## lm+                     model.frame(lm.D9)))
##
## lm> ## End(Don't show)
## lm> ### less simple examples in "See Also" above
## lm>
## lm>
## lm>
```

Demonstrations of R functions can be useful for seeing the range of things that R can do. Here are some for you to try:

```
#demo(persp)
#demo(graphics)
#demo(Hershey)
#demo(plotmath)
```

2.3.7 Packages in R

Finding your way around the contributed packages can be tricky, simply because there are so many of them, and the name of the package is not always as indicative of its function as you might hope. There is no comprehensive cross-referenced index, but there is a very helpful feature called 'Task Views' on CRAN, which explains the packages available under a limited number of usefully descriptive headings.

Click here to see the 'Task Views'

2.3.8 Built-in R libraries

To use one of the built-in libraries, simply type the library function with the name of the library in brackets. Thus, to load the `dplyr` library type:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked from 'package:gridExtra':
##
##     combine
```

```
## The following object is masked from 'package:simmer':
##
##     select

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

2.3.9 Contents of Packages

It is easy to use the `help` function to discover the contents of library packages. Here is how you find out about the contents of the `dplyr` library:

```
library(help=dplyr)
```

Then, to find out how to use, say, mutate (`mutate`), just type:

```
?mutate
```

2.3.10 Installing Packages

The base package does not contain some of the libraries referred to in this course, but downloading these is very simple. Before you start, you should check whether you need to "Run as administrator" before you can install packages (right click on the R icon to find this).

Run the R program, then from the command line use the install.packages function to download the libraries you want. For example, to install the `ggplot2` package type this:

```
#install.packages("ggplot2")
```

2.3.11 Command line versus scripts

When writing functions and other multi-line sections of input you will find it useful to use a text editor rather than execute everything directly at the command line.

Currently, most users prefer to use an IDE rather than executable text files. The most famous IDE for using R is Rstudio.

2.3.12 RStudio

Programming IDE to develop projects in R: https://www.rstudio.com/

There are two versions:

- RStudio Desktop
- RStudio Server (RStudio Desktop interface in web version)

Both versions have open source (free) and commercial (with support included) versions.

Allows the complete management of a software project:

- Console R
- File management
- Help
- Package management (installation, update, etc.)
- Review of command history

2.3.13 Working Directory

As we have mentioned, R is a programming language that allows us to perform certain actions through an IDE installed in our computer.

In many cases we will need to store data or code sets to use them later. We may also need to read a data set from an external format or even write it. To do all these things, we need to know where we are on the computer, in other words, which folder we are currently in.

We will call this location the working directory. We are going to place there all the resources we need to work with R.

We will use the function setwd() to indicate our location to the R session we are working at.

Example:

```
setwd("C://User/Desktop/My_Working_Directory")
```

2.3.14 Exercise: Set up your Working Directory

Try to start getting familiar with Rstudio and to set your working directory in a folder that is suitable for the rest of the course.

Remember, within this folder you can create sub-folders for each session in which you can include all the necessary material.

2.4 Expressions, Objects and Symbols

The R code is composed by expressions. Some examples of expressions:

- Assignments
- Conditional sentences
- Arithmetic operations
- ...

Expressions are made up of objects and functions. Each expression is separated from another by a new line or semicolon (;).

The R code manipulates objects. Some examples of objects:

- Vectors
- Lists
- Functions
- ...

Formally the variable names in R are called symbols. Thus, we assign the object to a symbol of the current environment. The environment is formed by the set of symbols in a certain context.

2.4.1 Some examples to start with

We can use R for many things, but there are certain basics that need to be learned. When we enter a code in the R console, this code will provide us with an output. Let's look at an example: If we enter a number or a word, R will return the same number or word.

```
23
```

```
## [1] 23
```

```
"Baby Yoda"
```

```
## [1] "Baby Yoda"
```

Each of these codes we have generated is interpreted by R, sent to our computer in a low level language, and returned in an understandable format for us.

Each code can be understood as an object (not stored) that R is interpreting for us. If we want to store that objects in our computer RAM to use them later in the same R session we need to create an assignment (we assign something to a symbol). Example:

```
my_number = 23
my_character = "Baby Yoda"
```

So we can use them later. For example, let's just print them:

```
print(my_number)
```

```
## [1] 23
```

```
print(my_character)
```

```
## [1] "Baby Yoda"
```

Notice that to assign something to a symbol we can use `=` or `<-`:

```
my_number = 23
my_character = "Baby Yoda"
```

```
print(my_number)
```

```
## [1] 23
```

```
print(my_character)
```

```
## [1] "Baby Yoda"
```

2.5 R as a Calculator

The screen prompt `>` is an invitation to put R to work. You can use the Rstudio command line as a calculator, like this:

```
log(42/7.3)
```

```
## [1] 1.749795
```

Each line can have at most 8192 characters, but if you want to see a lengthy instruction or a complicated expression on the screen, you can continue it on one or more further lines simply by ending the line at a place where the line is obviously incomplete (e.g. with a trailing comma, operator, or with more left parentheses than right parentheses, implying that more right parentheses will follow).

When continuation is expected, the prompt changes from $>$ to $+$:

```
5+6+3+6+4+2+4+8+
3+2+7
```

```
## [1] 50
```

Note that the + continuation prompt does not carry out arithmetic plus. If you have made a mistake, and you want to get rid of the + prompt and return to the > prompt, then press the Esc key and use the Up arrow to edit the last (incomplete) line.

Two or more expressions can be placed on a single line so long as they are separated by semi-colons:

```
2+3; 5*7; 3-7
```

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

```
## [1] 35
```

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

For very big numbers or very small numbers R uses the following scheme (called exponents):

- `1.2e3`: means 1200 because the e3 means 'move the decimal point 3 places to the right'
- `1.2e-2`: means 0.012 because the e-2 means 'move the decimal point 2 places to the left'
- `3.9+4.5i`: is a complex number with real (3.9) and imaginary (4.5) parts, and i is the square root of –1.

2.5.1 Complex numbers in R

Complex numbers consist of a real part and an imaginary part, which is identified by lower-case i like this:

```
z = 3.5-8i
```

The elementary trigonometric, logarithmic, exponential, square root and hyperbolic functions are all implemented for complex values. The following are the special R functions that you can use with com- plex numbers:

- Determine the real part:

```
Re(z)
```

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

- Determine the imaginary part:

```
Im(z)
```

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

- Calculate the modulus (if x is the real part and y is the imaginary part, then the modulus is $\sqrt{(x^2 + y^2)}$):

```
Mod(z)
```

```
## [1] 8.732125
```

- Calculate the argument (`Arg(x+ yi)= atan(y/x)`):

```
Arg(z)
```

```
## [1] -1.158386
```

- Work out the complex conjugate (change the sign of the imaginary part):

```
Conj(z)
```

```
## [1] 3.5+8i
```

- Check if the object is a complex number:

```
is.complex(z)
```

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

- Coerce a number into a complex number:

```
as.complex(3.8)
```

```
## [1] 3.8+0i
```

2.5.2 Rounding

Various sorts of rounding (rounding up, rounding down, rounding to the nearest integer) can be done easily. Take the number `5.7` as an example. The 'greatest integer less than' function is `floor`:

```
floor(5.7)
```

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

The 'next integer' function is ceiling:

```
ceiling(5.7)
```

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

You can round to the nearest integer by adding 0.5 to the number, then using floor. There is a built-in function for this, but we can easily write one of our own to introduce the notion of function writing. Call it `rounded`, then define it as a function like this:

```
rounded = function(x){floor(x+0.5)}
```

Now we can use the new function:

```
rounded(5.7)
```

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

```
rounded(5.4)
```

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

There is an R function called round that you can use by specifying 0 decimal places in the second argument:

```
round(5.7,0)
```

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

```
round(5.4,0)
```

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

```
round(-5.7,0)
```

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

2.5.3 Arithmetics

The screen prompt in R is a fully functional calculator. You can add and subtract using the obvious `+` and `-` symbols, while division is achieved with a forward slash `/` and multiplication is done by using an asterisk `*` like this:

```
7 + 3 - 5 * 2
```

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

Notice from this example that multiplication `(5 × 2)` is done *before* the additions and subtractions.

Powers (like squared or cube root) use the caret symbol `^` and are done before multiplication or division, as you can see from this example:

```
3^2 / 2
```

```
## [1] 4.5
```

All the mathematical functions you could ever want are here (see Table 2.1).

The `log` function gives logs to the base e (e = 2.718 282), for which the antilog function is `exp`:

```
log(10)
```

```
## [1] 2.302585
```

```
exp(1)
```

```
## [1] 2.718282
```

Logs to other bases are possible by providing the `log` function with a second argument which is the base of the logs you want to take. Suppose you want log to base 3 of 9:

```
log(9,3)
```

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

The trigonometric functions in R measure angles in radians. A circle is 2π radians, and this is 360° , so a right angle (90°) is $\pi/2$ radians. R knows the value of π as `pi`:

```
pi
```

```
## [1] 3.141593
```

```
sin(pi/2)
```

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

```
cos(pi/2)
```

```
## [1] 6.123234e-17
```

Notice that the cosine of a right angle does not come out as exactly zero, even though the sine came out as exactly 1. The `e-017` means 'times 10^{-17}'. While this is a very small number, it is clearly not exactly zero.

2.5.4 Modulo and integer quotients

Integer quotients and remainders are obtained using the notation `%/%` (percent, divide, percent) and `%%` (percent, percent) respectively. Suppose we want to know the integer part of a division: say, how many 13s are there in 119:

```
119 %/% 13
```

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

Now suppose we wanted to know the remainder (what is left over when 119 is divided by 13): in maths this is known as **modulo**:

```
119 %% 13
```

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

Modulo is very useful for testing whether numbers are odd or even: odd numbers have modulo 2 value 1 and even numbers have modulo 2 value 0:

```
9 %% 2
```

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

```
8 %% 2
```

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

Likewise, you use modulo to test if one number is an exact multiple of some other number. For instance, to find out whether 15 421 is a multiple of 7 (which it is), then ask:

```
15421 %% 7
```

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

2.5.5 Operators

R uses the following operator tokens:

- `+ - * / %/% %% ^`: arithmetic (plus, minus, times, divide, integer quotient, modulo, power)
- `>= < <= == !=`: relational (greater than, greater than or equals, less than, less than or equals, equals, not equals)
- `! & |`: logical (not, and, or)
- `~`: model formulae ('is modelled as a function of')
- `= ->`: assignment (gets)

- `$`: list indexing (the 'element name' operator)
- `::` create a sequence

Several of these operators have different meaning inside model formulae. Thus * indicates the main effects plus interaction (rather than multiplication), : indicates the interaction between two variables (rather than generate a sequence) and ^ means all interactions up to the indicated power (rather than raise to the power). You will learn more about these ideas in further sessions.

2.6 R Basic Data types

In the previous examples we worked with numbers, but variables could be assigned other types of information. There are four basic types:

- *Integers*: integer numbers. If you type an integer in R, as before 3 or 4, it will usually be stored as a double unless explicitly defined;
- *Doubles*: real numbers;
- *Logicals* or *Booleans*: corresponding to `TRUE` and `FALSE`, also abbreviated as `T` and `F` respectively;
- *Characters*: strings of text surrounded by `"` (for example `"hi"`) or by `'` (for example 'by');
- *Dates*: date-time expressions in R.

2.6.1 Integers

Integer vectors exist so that data can be passed to C or Fortran code which expects them, and so that small integer data can be represented exactly and compactly. The range of integers is from $-2\,000\,000\,000$ to $+\,2\,000\,000\,000$ (`-2*10^9 to +2*10^9`, which R could portray as `-2e+09` to `2e+09`).

Be careful. Do not try to change the class of a vector by using the `integer` function. Here is a numeric vector of whole numbers that you want to convert into a vector of integers:

```
x = c(5,3,7,8)
is.integer(x)
```

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

```
is.numeric(x)
```

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

To coerce a numeric vector to be an integers vector we have to use the `as.integer` function like this:

```
x = c(5,3,7,8)
x = as.integer(x)
is.integer(x)
```

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

The integer function works as trunc when applied to real numbers, and removes the imaginary part when applied to complex numbers:

```
as.integer(5.7)
```

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

```
as.integer(5.7 -3i)
```

```
## Warning: imaginary parts discarded in coercion
```

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

2.6.2 Integer vs. Double

The two most common numeric classes used in R are integer and double (for double precision floating point numbers). R automatically converts between these two classes when needed for mathematical purposes. As a result, it's feasible to use R and perform analyses for years without specifying these differences.

By default, when you create a numeric vector using the `c()` function it will produce a vector of double precision numeric values. To create a vector of integers using `c()` you must specify explicity by placing an L directly after each number.

```
dbl_var = c(1, 2.5, 4.5)
dbl_var
```

```
## [1] 1.0 2.5 4.5
```

```
int_var = c(1L, 6L, 10L)
int_var
```

```
## [1]  1  6 10
```

To check whether a vector is made up of integer or double values:

```
typeof(dbl_var)
```

```
## [1] "double"
```

```
typeof(int_var)
```

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

By default, if you read in data that has no decimal points or you create numeric values using the `x = 1:10` method the numeric values will be coded as integer. If you want to change a double to an integer or vice versa you can specify one of the following:

```
# integers to doubles
as.double(int_var)
```

```
## [1]  1  6 10
```

```
# doubles to integers
as.integer(dbl_var)
```

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

2.6.3 Logical operators

A crucial part of computing involves asking questions about things. Is one thing bigger than other? Are two things the same size? Questions can be joined together using words like 'and' 'or', 'not'. Questions in R typically evaluate to `TRUE` or `FALSE` but there is the option of a 'maybe' (when the answer is not available, `NA`). In R, < means 'less than', > means 'greater than', and ! means 'not' (see Table 2.2).

You can use `T` for TRUE and `F` for FALSE, but you should be aware that `T` and `F` might have been allocated as variables. So this is obvious:

```
TRUE == FALSE
```

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

```
T == F
```

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

This, however, is not so obviours:

```
T = 0
T == FALSE
```

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

```
F = 1
TRUE == F
```

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

But now, of course, `T` is not equal to `F`:

```
T != F
```

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

To be sure, always write TRUE and FALSE in full, and never use T or F as variable names.

Maybe you noticed in the last code chunk that, in R, TRUE is coded as `1` and FALSE as `0`:

```
TRUE == 1
```

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

```
FALSE == 0
```

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

Let's remove the T and F variables to avoid future errors:

```
rm(list = c(T,F))
```

```
## Warning in rm(list = c(T, F)): object '0' not found
```

```
## Warning in rm(list = c(T, F)): object '1' not found
```

2.6.4 Real numbers equality

You need to be careful in programming when you want to test whether or not two computed numbers are equal. R will assume that you mean 'exactly equal', and what that means depends upon machine precision. Most numbers are rounded to an accuracy of 53 binary digits.

Typically therefore, two floating point numbers will not reliably be equal unless they were computed by the same algorithm, and not always even then. You can see this by squaring the square root of 2: surely these values are the same?

```
x = sqrt(2)
x * x == 2
```

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

In fact, they are not the same. We can see by how much the two values differ by subtraction:

```
x * x - 2
```

```
## [1] 4.440892e-16
```

So how do we test for equality of real numbers? The best advice is not to do it. Try instead to use the alternatives 'less than' with 'greater than or equal to', or conversely 'greater than' with 'less than or equal to'. Then you will not go wrong. Sometimes, however, you really do want to test for equality. In those circumstances, do not use double equals to test for equality, but employ the `all.equal` function instead.

```
x = 0.3 - 0.2
y = 0.1
x == y
```

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

```
all.equal(x,y)
```

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

WARNING: Do not use `all.equal` directly in `if` expressions. Either use `isTRUE(all.equal( ....))`

2.6.5 Logical arithmetic

Arithmetic involving logical expressions is very useful in programming and in selection of variables. The key thing to understand is that logical expressions evaluate to either true or false (represented in R by `TRUE` or `FALSE`), and that R can coerce `TRUE` or `FALSE` into numerical values: 1 for `TRUE` and 0 for `FALSE`. Suppose that x is a sequence from 0 to 6 like this:

```
x = 0:6
```

Now we can ask questions about the contents of the vector called x. Is x less than 4?

```
x < 4
```

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

The answer is yes for the first four values (0, 1, 2 and 3) and no for the last three (4, 5 and 6).

Two important logical functions are `all` and `any`. They check an entire vector but return a single logical value: `TRUE` or `FALSE`. Are all the x values bigger than 0?

```
all(x>0)
```

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

No. The first x value is a zero. Are any of the x values negative?

```
any(x<0)
```

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

No. The smallest x value is a zero.

We can use the answers of logical functions in arithmetic. We can count the true values of (x<4), using sum:

```
sum(x<4)
```

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

We can multiply (x<4) by other vectors:

```
(x<4) * runif(7)
```

```
## [1] 0.3638331 0.9908843 0.4828548 0.2473120 0.0000000 0.0000000 0.0000000
```

2.6.6 Characters

In R, character strings are defined by double quotation marks:

```
a = "abc"
b = "123"
```

Numbers can be coerced to characters (as in b above), but non-numeric characters cannot be coerced to numbers:

```
as.numeric(a)
```

```
## Warning: NAs introduced by coercion
```

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

```
as.numeric(b)
```

```
## [1] 123
```

One of the initially confusing things about character strings is the distinction between the length of a character object (a vector), and the numbers of characters (nchar) in the strings that comprise that object. An example should make the distinction clear:

```
pets = c("cat","dog","gerbil","terrapin")
```

Here, `pets` is a vector comprising four character strings:

```
length(pets)
```

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

and the individual character strings have 3, 3, 6 and 8 characters, respectively:

```
nchar(pets)
```

```
## [1] 3 3 6 8
```

When first defined, character strings are not factors:

```
class(pets)
```

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

```
is.factor(pets)
```

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

We have to coerce this variable to convert it into a factor

```
pets = as.factor(pets)
is.factor(pets)
```

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

2.6.7 Characters: letters vectors

There are built-in vectors in R that contain the 26 letters of the alphabet in lower case (letters) and in upper case (LETTERS):

```
letters
```

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

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

To discover which number in the alphabet the letter `n` is, you can use the `which` function like this:

```
which(letters == "n")
```

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

For the purposes of printing you might want to suppress the quotes that appear around character strings by default. The function to do this is called noquote:

```
noquote(letters)
```

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

2.6.8 Characters: Pasting strings together

You can amalgamate individual strings into vectors of character information:

```
c(a,b)
```

```
## [1] "abc" "123"
```

This shows that the concatenation produces a vector of two strings. It does not convert two 3-character strings into one 6-character string. The R function to do that is `paste`:

```
paste(a,b,sep = "")
```

```
## [1] "abc123"
```

The third argument, `sep=""`, means that the two character strings are to be pasted together without any separator between them: the default for `paste` is to insert a single blank space, like this:

```
paste(a,b)
```

```
## [1] "abc 123"
```

Notice that you do not lose blanks that are within character strings when you use the `sep=""` option in `paste`.

```
paste(a,b," a longer phrase containing blanks",sep="")
```

```
## [1] "abc123 a longer phrase containing blanks"
```

If one of the arguments to `paste` is a vector, each of the elements of the vector is pasted to the specified character string to produce an object of the same length as the vector:

```
d = c(a,b,"new")
e = paste(d, "a longer phrase containing blanks")
e
```

```
## [1] "abc a longer phrase containing blanks"
## [2] "123 a longer phrase containing blanks"
## [3] "new a longer phrase containing blanks"
```

2.6.9 Characters: Extracting parts of strings

We being by defining a phrase:

```
phrase = "the quick brown fox jumps over the lazy dog"
```

The function called `substr` is used to extract substrings of a specified number of characters from within a character string. In the next example we are extracting the first 20 characters from `phrase` object:

```
substr(phrase, 1, 20)
```

```
## [1] "the quick brown fox "
```

The second argument in `substr` is the number of the character at which extraction is to begin (in this case the first), and the third argument is the number of the character at which extraction is to end (in this case, the 20th).

2.6.10 Characters: Counting things within strings

Counting the total number of characters in a string could not be simpler; just use the `nchar` function directly, like this:

```
nchar(phrase)
```

```
## [1] 43
```

So there are 43 characters including the blanks between the words. To count the numbers of separate individual characters (including blanks) you need to split up the character string into individual characters (43 of them), using `strsplit` like this:

```
strsplit(phrase, split = "")
```

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

The `split = ""` argument is for determine the character we are going to use to split the entire object. If we use a blank space instead we can separate the string in all the different words:

```
strsplit(phrase, split = " ")
```

```
## [[1]]
## [1] "the"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog"
```

The table function can then be used for counting the number of occurrences of each of the characters:

```
table(strsplit(phrase, split = ""))
```

```
##
##   a b c d e f g h i j k l m n o p q r s t u v w x y z
## 8 1 1 1 1 3 1 1 2 1 1 1 1 1 1 4 1 1 2 1 2 2 1 1 1 1 1
```

This demonstrates that all of the letters of the alphabet were used at least once within our phrase, and that there were eight blanks within the string called phrase. This suggests a way of counting the number of words in a phrase, given that this will always be one more than the number of blanks (so long as there are no leading or trailing blanks in the string):

```
nwords = 1+table(strsplit(phrase, split = ""))[1]
nwords
```

```
##
## 9
```

Another solution is to `sum` all the elements in the table splitting the phrase with a blank space:

```
sum(table(strsplit(phrase, split = " ")))
```

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

2.6.11 Characters: Upper- and lower-case text

It is easy to switch between upper and lower cases using the `toupper` and `tolower` functions:

```
toupper(phrase)
```

```
## [1] "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
```

```
tolower(toupper(phrase))
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

2.6.12 Dates and times

The measurement of time is highly idiosyncratic. Successive years start on different days of the week. There are months with different numbers of days. Leap years have an extra day in February. Americans and Britons put the day and the month in different places: 3/4/2006 is March 4 for the former and April 3 for the latter.

All these things mean that working with dates and times is extremely complicated. Fortunately, R has a robust system for dealing with this complexity.

To see how R handles dates and times, have a look at Sys.time():

```
Sys.time()
```

```
## [1] "2022-01-20 13:06:47 CET"
```

This description of date and time is strictly hierarchical from left to right: the longest time scale (years) comes first, then month, then day, separated by hyphens, then there is a blank space, followed by the time, with hours first (using the 24-hour clock), then minutes, then seconds, separated by colons. Finally, there is a character string explaining the time zone (CET stands for Central Europe Time)

This representation of the date and time as a character string is user-friendly and familiar, but it is no good for calculations. For that, we need a single numeric representation of the combined date and time. The convention in R is to base this on seconds (the smallest time scale that is accommodated in Sys.time)

The baseline for expressing today's date and time in seconds is 1 January 1970:

```
as.numeric(Sys.time())
```

```
## [1] 1642680407
```

This is fine for plotting time series graphs, but it is not much good for computing monthly means (e.g. is the mean for June significantly different from the July mean?) or daily means (e.g. is the Monday mean significantly different from the Friday mean?).

To answer questions like these we have to be able to access a broad set of categorical variables associated with the date: the year, the month, the day of the week, and so forth. To accommodate this, R uses the POSIX system for representing times and dates:

```
class(Sys.time())
```

```
## [1] "POSIXct" "POSIXt"
```

You can think of the class `POSIXct`, with suffix 'ct', as continuous time (i.e. a number of seconds), and `POSIXlt`, with suffix 'lt', as list time (i.e. a list of all the various categorical descriptions of the time, including day of the week and so forth). It is hard to remember these acronyms, but it is well worth making the effort. Naturally, you can easily convert to one representation to the other:

```
time.list = as.POSIXlt(Sys.time())
unlist(time.list)
```

```
##                sec                min               hour               mday
## "47.0734965801239"                "6"               "13"               "20"
##                mon               year               wday               yday
```

```
##                "0"              "122"                "4"               "19"
##              isdst               zone             gmtoff
##                "0"              "CET"             "3600"
```

Here you see the nine components of the list. The time is represented by the number of seconds (`sec`), minutes (`min`) and hours (on the 24-hour clock). Next comes the day of the month (`mday`, starting from 1), then the month of the year (mon, starting at January = 0), then the year (starting at 0 = 1900). The day of the week (`wday`) is coded from Sunday = 0 to Saturday = 6. The day within the year (`yday`) is coded from 0 = January 1. Finally, there is a logical variable `isdst` which asks whether daylight saving time is in operation (0 = FALSE in this case). The ones you are most likely to use include `year` (to get yearly mean values), `mon` (to get monthly means) and `wday` (to get means for the different days of the week).

2.6.13 Dates and times: Reading time data from files

It is most likely that your data files contain dates in Excel format, for example 03/09/2014 (a character string showing month/day/year separated by slashes).

```
df = read.csv(file = "Data/boulder-precip.csv")
attach(df)
head(df)
```

```
##    ID    DATE PRECIP TEMP
## 1 756 8/21/13    0.1   55
## 2 757 8/26/13    0.1   25
## 3 758 8/27/13    0.1   NA
## 4 759  9/1/13    0.0 -999
## 5 760  9/9/13    0.1   15
## 6 761 9/10/13    1.0   25
```

To convert a factor or a character string into a POSIXlt object, we employ an important function called 'strip time', written `strptime`.

2.6.14 Dates and times: The `strptime` function

To convert a factor or a character string into dates using the strptime function, we provide a format statement enclosed in double quotes to tell R exactly what to expect, in what order, and separated by what kind of symbol. For our present example we have day (as two digits), then slash, then month (as two digits), then slash, then year (without the century, making two digits).

```
Rdate = strptime(DATE, "%m/%d/%y")
class(Rdate)
```

```
## [1] "POSIXlt" "POSIXt"
```

It is always a good idea at this stage to add the R-formatted date to your dataframe:

```
df = data.frame(df, Rdate)
head(df)
```

```
##    ID    DATE PRECIP TEMP      Rdate
## 1 756 8/21/13    0.1   55 2013-08-21
## 2 757 8/26/13    0.1   25 2013-08-26
## 3 758 8/27/13    0.1   NA 2013-08-27
## 4 759  9/1/13    0.0 -999 2013-09-01
## 5 760  9/9/13    0.1   15 2013-09-09
## 6 761 9/10/13    1.0   25 2013-09-10
```

Now, at last, we can do things with the date information. We might want the mean value of `PRECIP` for each day of the week. The name of this object is Rdate$wday:

```
tapply(PRECIP,Rdate$wday,mean)
```

```
##         0         1         2         3         4         5         6
## 0.5000000 0.2250000 0.3666667 1.2000000 9.8000000 0.8000000 0.1000000
```

It is hard to remember all the format codes for strip time, but they are roughly mnemonic and they are always preceded by a percent symbol. Here is the full list of format components:

- `%a`: Abbreviated weekday name
- `%A`: Full weekday name
- `%b`: Abbreviated month name
- `%B`: Full month name
- `%c`: Date and time, locale-specific
- `%d`: Day of the month as decimal number (01–31)
- `%H`: Hours as decimal number (00–23) on the 24-hour clock
- `%I`: Hours as decimal number (01–12) on the 12-hour clock
- `%j`: Day of year as decimal number (0–366)
- `%m`: Month as decimal number (0–11)

- %M: Minute as decimal number (00–59)
- %p: AM/PM indicator in the locale
- %S: Second as decimal number (00–61, allowing for two 'leap seconds')
- %U: Week of the year (00–53) using the first Sunday as day 1 of week 1
- %w: Weekday as decimal number (0–6, Sunday is 0)
- %W: Week of the year (00–53) using the first Monday as day 1 of week 1
- %x: Date, locale-specific
- %X: Time, locale-specific
- %Y: Year with century
- %y: Year without century
- %Z: Time zone as a character string (output only)

There is a useful function called weekdays (note the plural) for turning the day number into the appropriate name:

```
y = strptime("01/02/2020", format="%d/%m/%Y")
weekdays(y)
```

```
## [1] "Saturday"
```

which is converted from:

```
y$wday
```

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

because the days of the week are numbered from Sunday = 0.

Here is another kind of date, with years in two-digit form (%y), and the months as abbreviated names (%b) using no separators:

```
other.dates = c("1jan99", "2jan05", "31mar04", "30jul05")
strptime(other.dates, "%d%b%y")
```

```
## [1] "1999-01-01 CET"  "2005-01-02 CET"  "2004-03-31 CEST" "2005-07-30 CEST"
```

Here is yet another possibility with year, then month in full, then week of the year, then day of the week abbreviated, all separated by a single blank space:

```
yet.another.date = c("2016 January 2 Mon","2017 February 6 Fri","2018 March 10 Tue")
strptime(yet.another.date,"%Y %B %W %a")
```

```
## [1] "2016-01-11 CET" "2017-02-10 CET" "2018-03-06 CET"
```

2.6.15 Dates and times: Summary

The key thing to understand is the difference between the two representations of dates and times in R. They have unfortunately non-memorable names.

- POSIXlt gives a list containing separate vectors for the year, month, day of the week, day within the year, and suchlike. It is very useful as a categorical explanatory variable (e.g. to get monthly means from data gathered over many years using date$mon).
- POSIXct gives a vector containing the date and time expressed as a continuous variable that you can use in regression models (it is the number of seconds since the beginning of 1970).

2.6.16 Testing and coercing

The concepts of membership and coercion may be unfamiliar. Membership relates to the class of an object in R. Coercion changes the class of an object. For instance, a logical variable has class `logical` and mode logical. This is how we create the variable:

```
lv = c(TRUE,FALSE,TRUE)
```

We can assess its membership by asking if it is a logical variable using the `is.logical` function:

```
is.logical(lv)
```

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

It is not a factor, and so it does not have levels:

```
levels(lv)
```

```
## NULL
```

But we can coerce it be a two-level factor like this:

```
fv = as.factor(lv)
fv
```

```
## [1] TRUE  FALSE TRUE
## Levels: FALSE TRUE
```

We can coerce a logical variable to be numeric: `TRUE` evaluates to 1 and `FALSE` evaluates to zero, like this:

```
nv = as.numeric(lv)
nv
```

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

In general, the expression `as(object, value)` is the way to coerce an object to a particular class. Membership functions ask `is.something` and coercion functions say `as.something`.

Objects have a type, and you can test the type of an object using an is.type function (Table 2.3). For instance, mathematical functions expect numeric input and text-processing functions expect character input. Some types of objects can be coerced into other types. A familiar type of coercion occurs when we interpret the TRUE and FALSE of logical variables as numeric 1 and 0, respectively. Factor levels can be coerced to numbers. Numbers can be coerced into characters, but non-numeric characters cannot be coerced into numbers.

2.6.17 Missing values, infinity and things that are not numbers

Calculations can lead to answers that are plus infinity, represented in R by `Inf`, or minus infinity, which is represented as `-Inf`:

```
3/0
```

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

```
-12/0
```

```
## [1] -Inf
```

Calculations involving infinity can be evaluated: for instance,

```
exp(-Inf)
```

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

```
0/Inf
```

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

Other calculations, however, lead to quantities that are not numbers. These are represented in R by NaN ('not a number'). Here are some of the classic cases:

```
0/0
```

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

```
Inf-Inf
```

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

```
Inf/Inf
```

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

You need to understand clearly the distinction between `NaN` and `NA` (this stands for 'not available' and is the missing-value symbol in R).

The function `is.nan` is provided to check specifically for `NaN`, and `is.na` also returns `TRUE` for `NaN`. Coercing `NaN` to logical or integer type gives an `NA` of the appropriate type. There are built-in tests to check whether a number is finite or infinite:

Missing values in dataframes are a real source of irritation, because they affect the way that model-fitting functions operate and they can greatly reduce the power of the modelling that we would like to do.

You may want to discover which values in a vector are missing. Here is a simple case:

```
y = c(4,NA,7)
```

The missing value question should evaluate to `FALSE TRUE FALSE`. There are two ways of looking for missing values that you might think should work, but do not. These involve treating `NA` as if it was a piece of text and using double equals (==) to test for it. So this does not work:

```
y == NA
```

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

because it turns all the values into `NA` (definitively not what you intended). This does not work either:

```
y == "NA"
```

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

It correctly reports that the numbers are not character strings, but it returns `NA` for the missing value itself, rather than `TRUE` as required. This is how you do it properly:

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

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

To produce a vector with the `NA` stripped out, use subscripts with the not `!` operator like this:

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

```
## [1] 4 7
```

Some functions do not work with their default settings when there are missing values in the data, and `mean` is a classic example of this:

```
x = c(1:8, NA)
mean(x)
```

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

In order to calculate the mean of the non-missing values, you need to specify that the NA are to be removed, using the `na.rm=TRUE` argument:

```
mean(x, na.rm = T)
```

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

2.7 R Basic Data Structures

2.7.1 Vectors

A vector is a variable with one or more values of the same type (atomic one dimensional arrays). For instance, the numbers of peas in six pods were 4, 7, 6, 5, 6 and 7. The vector called peas is one object of `length = 6`. In this case, the class of the object is numeric. The easiest way to create a vector in R is to concatenate (link together) the six values using the concatenate function, `c`, like this:

```
peas = c(4, 7, 6, 5, 6, 7)
```

We can ask all sorts of questions about the vector called peas. For instance, what type of vector is it?

```
class(peas)
```

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

How big is the vector?

```
length(peas)
```

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

The great advantage of a vector-based language is that it is very simple to ask quite involved questions that involve all of the values in the vector. These vector functions are often self-explanatory:

```
mean(peas)
```

```
## [1] 5.833333
```

```
max(peas)
```

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

```
min(peas)
```

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

Another way to create a vector is to input data from the keyboard using the function called `scan`:

```
#peas = scan()
```

The prompt appears `1:` which means type in the first number of peas (4) then press the return key, then the prompt `2:` appears (you type in 7) and so on. When you have typed in all six values, and the prompt `7:` has appeared, you just press the return key to tell R that the vector is now complete. R replies by telling you how many items it has read:

```
Read 6 items
```

As we explained, vectors are single-dimensional arrays. The array indexes range from 1 to the vector length, `length(v)`. Vectors are also known as atomic vectors.

All elements of the vector are of the same basic type:

- logical
- integer
- double
- character
- complex

It has a fixed size that is fixed in its creation. The simplest way to create a vector is by using the combination function `c(v1, v2,...)`. To name the elements of a vector we use the function `names(v)`. Here you have some different ways to create vectors in R:

- Using the `vector` function

```
# Logical Vector
v1 = vector(mode = 'logical', length = 4)
v1
```

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

```
# Integer vector
v2 = vector(mode = 'integer', length = 4)
v2
```

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

- Using the "type" function

```
# Numeric vector
v3 = numeric(4)
v3
```

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

```
# Character vector
v4 = character(4)
v4
```

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

2.7.2 Sequences

An important way of creating vectors is to generate a sequence of numbers. The simplest sequences are in steps of 1, and the colon operator is the simplest way of generating such sequences. All you do is specify the first and last values separated by a colon. Here is a sequence from 0 up to 10:

```
0:10
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

Here is a sequence from 15 down to 5:

```
15:5
```

```
##  [1] 15 14 13 12 11 10  9  8  7  6  5
```

To generate a sequence in steps other than 1, you use the seq function. There are various forms of this, of which the simplest has three arguments: `from`, `to`, `by` (the initial value, the final value and the increment). If the initial value is smaller than the final value, the increment should be positive, like this:

```
seq(0, 1.5, 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5
```

If the initial value is larger than the final value, the increment should be negative, like this:

```
seq(6, 4, -0.2)
```

```
##  [1] 6.0 5.8 5.6 5.4 5.2 5.0 4.8 4.6 4.4 4.2 4.0
```

In many cases, you want to generate a sequence to match an existing vector in length. Rather than having to figure out the increment that will get from the initial to the final value and produce a vector of exactly the appropriate length, R provides the `along` and `length` options. Suppose you have a vector of population sizes:

```
N = c(55,76,92,103,84,88,121,91,65,77,99)
```

You need to plot this against a sequence that starts at 0.04 in steps of 0.01:

```
seq(from=0.04,by=0.01,length=11)
```

```
##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

But this requires you to figure out the length of `N`. A simpler method is to use the `along` argument and specify the vector, `N`, whose length has to be matched:

```
seq(0.04,by=0.01,along=N)
```

```
##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

Alternatively, you can get R to work out the increment (0.01 in this example), by specifying the start and the end values (`from` and `to`), and the name of the vector (`N`) whose length has to be matched:

```
seq(from=0.04,to=0.14,along=N)
```

```
##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

If you want a vector made up of sequences of unequal lengths, then use the sequence function. Suppose that most of the five sequences you want to string together are from 1 to 4, but the second one is 1 to 3 and the last one is 1 to 5, then:

```
sequence(nvec = c(4,3,4,4,4,5), from = 1, by = 1)
```

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

2.7.2.1 Generating repeats

You will often want to generate repeats of numbers or characters, for which the function is `rep`. The object that is named in the first argument is repeated a number of times as specified in the second argument. At its simplest, we would generate five 9s like this:

```
rep(9,5)
```

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

You can see the issues involved by a comparison of these three increasingly complicated uses of the `rep` function:

```
rep(1:4, times = 2)
```

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

In the simplest case, the entire first argument is repeated (i.e. the sequence 1 to 4 appears twice).

```
rep(1:4, each=2)
```

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

You often want each element of the sequence to be repeated, and this is accomplished with the each argument.

```
rep(1:4, each = 2, times = 3)
```

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

Finally, you might want each number repeated and the whole series repeated a certain number of times (here three times).

When each element of the series is to be repeated a different number of times, then the second argument must be a vector of the same length as the vector comprising the first argument (length 4 in this example). So if we want one 1, two 2s, three 3s and four 4s we would write:

```
rep(1:4, 1:4)
```

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

In a more complicated case, there is a different but irregular repeat of each of the elements of the first argument. Suppose that we need four 1s, one 2, four 3s and two 4s. Then we use the concatenation function `c` to create a vector of length 4 `c(4,1,4,2)` which will act as the second argument to the `rep` function:

```
rep(1:4, c(4,1,4,2))
```

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

Here is the most complex case with character data rather than numbers: each element of the series is repeated an irregular number of times:

```
rep(c("cat", "dog", "gerbil", "goldfish", "rat"), c(2,3,2,1,3))
```

```
##  [1] "cat"      "cat"      "dog"      "dog"      "dog"      "gerbil"
##  [7] "gerbil"   "goldfish" "rat"      "rat"      "rat"
```

This is the most general, and also the most useful form of the `rep` function.

2.7.2.2 Generating Factor Levels

The function gl ('generate levels') is useful when you want to encode long vectors of factor levels. The syntax for the three arguments is: 'up to', 'with repeats of', 'to total length'. Here is the simplest case where we want factor levels up to 4 with repeats of 3 repeated only once (i.e. to total length 12):

```
gl(4,3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4
```

Here is the function when we want that whole pattern repeated twice:

```
gl(4,3,24)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4
```

If you want text for the factor levels, rather than numbers, use labels like this:

```
Temp = gl(2, 2, 24, labels = c("Low", "High"))
Soft = gl(3, 8, 24, labels = c("Hard","Medium","Soft"))
M.user = gl(2, 4, 24, labels = c("N", "Y"))
Brand = gl(2, 1, 24, labels = c("X", "M"))
data.frame(Temp,Soft,M.user,Brand)
```

```
##    Temp   Soft M.user Brand
## 1   Low   Hard      N     X
## 2   Low   Hard      N     M
## 3  High   Hard      N     X
## 4  High   Hard      N     M
## 5   Low   Hard      Y     X
## 6   Low   Hard      Y     M
## 7  High   Hard      Y     X
## 8  High   Hard      Y     M
## 9   Low Medium      N     X
## 10  Low Medium      N     M
## 11 High Medium      N     X
## 12 High Medium      N     M
## 13  Low Medium      Y     X
## 14  Low Medium      Y     M
## 15 High Medium      Y     X
## 16 High Medium      Y     M
## 17  Low   Soft      N     X
## 18  Low   Soft      N     M
## 19 High   Soft      N     X
## 20 High   Soft      N     M
## 21  Low   Soft      Y     X
## 22  Low   Soft      Y     M
## 23 High   Soft      Y     X
## 24 High   Soft      Y     M
```

2.7.3 Vector and Subscripts

You will often want to use some but not all of the contents of a vector. To do this, you need to master the use of subscripts (or indices as they are also known). In R, subscripts involve the use of square brackets `[]`. Our vector called `peas` shows the numbers of peas in six pods:

```
peas = c(4, 7, 6, 5, 6, 7)
```

The first element of `peas` is 4, the second 7, and so on. The elements are indexed left to right, 1 to 6. It could not be more straightforward. If we want to extract the fourth element of peas (which you can see is a 5) then this is what we do:

```
peas[4]
```

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

If we want to extract several values (say the 2nd, 3rd and 6th) we use a vector to specify the pods we want as subscripts, either in two stages like this:

```
pods = c(2,3,6)
peas[pods]
```

```
## [1] 7 6 7
```

or in a single step, like this:

```
peas[c(2,3,6)]
```

```
## [1] 7 6 7
```

You can drop values from a vector by using negative subscripts. Here are all but the first values of peas:

```
peas[-1]
```

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

Here are all but the last (note the use of the `length` function to decide what is last):

```
peas[-length(peas)]
```

```
## [1] 4 7 6 5 6
```

We can use sequences of numbers to extract values from a vector. Here are the first three values of `peas`:

```
peas[1:3]
```

```
## [1] 4 7 6
```

Here are the even-numbered values of `peas`:

```
peas[seq(2,length(peas), 2)]
```

```
## [1] 7 5 7
```

or alternatively:

```
peas[1:length(peas) %% 2 == 0]
```

```
## [1] 7 5 7
```

Using the modulo function %% on the sequence 1 to 6 to extract the even numbers 2, 4 and 6. Finally, we can assign some value to the elements between some specific indices

```
peas[4:5] = 0
peas
```

```
## [1] 4 7 6 0 0 7
```

2.7.3.1 Classes of vector

The vector called `peas` contained numbers: in the jargon, it is of class numeric. R allows vectors of six types, so long as all of the elements in one vector belong to the same class. The classes are logical, integer, real, complex, string (or character) or raw. You will use numeric, logical and character variables all the time. Engineers and mathematicians will use complex numbers. But you could go a whole career without ever needing to use integer or raw.

2.7.3.2 Naming elements within vectors

It is often useful to have the values in a vector labelled in some way. For instance, if our data are counts of 0, 1, 2, . . . occurrences in a vector called `counts`,

```
counts = c(25,12,7,4,6,2,1,0,2)
```

so that there were 25 zeros, 12 ones and so on, it would be useful to name each of the counts with the relevant number 0 to 8:

```
names(counts) = 0:8
```

Now when we inspect the vector called counts we see both the names and the frequencies:

```
counts
```

```
##  0  1  2  3  4  5  6  7  8
## 25 12  7  4  6  2  1  0  2
```

Or even access to some element using its name:

```
counts["0"]
```

```
##  0
## 25
```

2.7.3.3 Working with logical subscripts

Take the example of a vector containing the 11 numbers 0 to 10:

```
x = 0:10
```

There are two quite different kinds of things we might want to do with this. We might want to add up the values of the elements:

```
sum(x)
```

```
## [1] 55
```

Alternatively, we might want to count the elements that passed some logical criterion. Suppose we wanted to know how many of the values were less than 5:

```
sum(x<5)
```

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

You see the distinction. We use the vector function `sum` in both cases. But `sum(x)` adds up the values of the x's and `sum(x<5)` counts up the number of cases that pass the logical condition 'x is less than 5'. This works because of coercion. Logical `TRUE` has been coerced to numeric 1 and logical `FALSE` has been coerced to numeric 0.

How do you add up the values of just some of the elements of x? We specify a logical condition, but we do not want to count the number of cases that pass the condition, we want to add up all the values of the cases that pass.

Note that when we counted the number of cases, the counting was applied to the entire vector, using `sum(x<5)`. To find the sum of the values of x that are less than 5, we write:

```
sum(x[x<5])
```

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

Let us look at this in more detail. The logical condition `x<5` is either true or false:

```
x<5
```

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

You can imagine false as being numeric 0 and true as being numeric 1. Then the vector of subscripts `[x<5]` is five 1s followed by six 0s:

```
as.numeric(x<5)
```

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

Now imagine multiplying the values of x by the values of the logical vector

```
x*(x<5)
```

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

When the function `sum` is applied, it gives us the answer we want: the sum of the values of the numbers $0 + 1 + 2 + 3 + 4 = 10$.

```
sum(x*(x<5))
```

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

This produces the same answer as `sum(x[x<5])`, but is rather less elegant. There are many other ways of indexing elements of a vector using logical operators. We have a vector with values between 0 and 100:

```
v = 0:100
```

We can select all the elements over 30:

```
v[v>30]
```

```
##  [1]  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49
## [20]  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
## [39]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87
## [58]  88  89  90  91  92  93  94  95  96  97  98  99 100
```

even, all those over 30 and under or equal 50:

```
v[v > 30 & v <= 50]
```

```
##  [1] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Finally, we can use a specific set of elements to select those from the vector:

```
v[ v %in% c(10, 20, 30)]
```

```
## [1] 10 20 30
```

2.7.4 Vector Operations

When arithmetic operations are performed between two vectors, R returns another vector with the results of the element by element operation. Boolean operations are also possible. Most functions and operations are "vectorized".

```
avector = c(1,2,3)
bvector = c(4,5,6)
```

If we sum up the two vectors the result will be a new vector with the sum of the elements with the same index:

```
avector + bvector
```

```
## [1] 5 7 9
```

If the vectors are not the same size, R repeats the smallest of them as many times as necessary.

```
avector + 1
```

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

Vectors multiplication will follow the same rule, as the multiplication by an scalar:

```
avector * bvector
```

```
## [1]  4 10 18
```

```
avector*2
```

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

If we want to multiplicate vectors as one-dimensional matrices (dot product) we need to use a different sintax:

```
avector %*% bvector
```

```
##      [,1]
## [1,]   32
```

In this case we obtained an scalar because the inner product was computed, but we can transpose one of the vectors to obtain the outer product using the `t()` function:

```
avector %*% t(bvector)
```

```
##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    8   10   12
## [3,]   12   15   18
```

2.7.5 Vector Functions

One of R's great strengths is its ability to evaluate functions over entire vectors, thereby avoiding the need for loops and subscripts. The most important vector functions are listed in Table 2.4.

Here is a numeric vector:

```
y = c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
```

Some vector functions produce a single number:

```
mean(y)
```

```
## [1] 6.333333
```

Others produce two numbers:

```
range(y)
```

```
## [1]  2 11
```

here showing that the minimum was 2 and the maximum was 11. Other functions produce several numbers:

```
fivenum(y)
```

```
## [1]  2.0  4.0  6.0  8.5 11.0
```

This is Tukey's famous five-number summary: the minimum, the lower hinge, the median, the upper hinge and the maximum.

Perhaps the single most useful vector function in R is `table`. You need to see it in action to appreciate just how good it is. Here is a huge vector called `counts` containing 10 000 random integers between 0 and 10 from a uniform distribution

```
counts = sample(1:10, size = 10000, replace = TRUE)
```

Here is a look at the first 30 values in counts:

```
counts[1:30]
```

```
##  [1] 10  7  9  7  3  4  7  1  5  7  1  3  4  1  6  5  7  5 10  5  6  8  2  3  3
## [26]  7  1  8  2  5
```

The question is this: how many zeros are there in the whole vector of 10 000 numbers, how many 1s, and so on right up to the largest value within counts? A formidable task for you or me, but for R it is just:

```
table(counts)
```

```
## counts
##    1    2    3    4    5    6    7    8    9   10
## 1017  986  989  987  974  978  997 1022 1053  997
```

2.7.5.1 tapply function

One of the most important functions in all of R is tapply. It does not sound like much from the name, but you will use it time and again for calculating means, variances, sample sizes, minima and maxima. With weather data, for instance, we might want the 12 monthly mean temperatures rather than the whole-year average. We have a response variable, temperature, and a categorical explanatory variable, month:

```
temperature = read.csv("./Data/city_temperature.csv")
attach(temperature)
names(temperature)
```

```
## [1] "Region"         "Country"        "State"          "City"
## [5] "Month"          "Day"            "Year"           "AvgTemperature"
```

```
tapply(AvgTemperature, Month, mean)
```

```
##        1        2        3        4        5        6        7        8
## 30.90471 33.50187 38.67561 46.13972 52.81063 59.37346 63.68533 63.12633
##        9       10       11       12
## 55.79678 47.70037 38.83678 30.47562
```

It is easy to apply other functions in the same way: here are the monthly variances

```
tapply(AvgTemperature,Month,var)
```

```
##         1         2         3         4         5         6         7         8
##  694.4614  633.2653  697.8510  767.8669  973.1084 1042.9352  980.9215  948.5103
##         9        10        11        12
##  882.3228  812.8217  785.9120  979.5516
```

and the monthly max

```
tapply(AvgTemperature,Month,max)
```

```
##     1     2     3     4     5     6     7     8     9    10    11    12
##  66.5  68.6  74.8  81.1  88.7 102.5 100.4 101.0  91.4  83.3  74.5  70.0
```

If R does not have a built in function to do what you want, then you can easily write your own. Here, for instance, is a function to calculate the standard error of each mean (these are called anonymous functions in R, because they are unnamed):

```
tapply(AvgTemperature, Month,  function(x) sqrt(var(x)/length(x)))
```

```
##         1         2         3         4         5         6         7         8
## 0.1446289 0.1446299 0.1449792 0.1545970 0.1730663 0.1835656 0.1751918 0.1723541
##         9        10        11        12
## 0.1689770 0.1595505 0.1594831 0.1746138
```

The `tapply` function is very flexible. It can produce multi-dimensional tables simply by replacing the one categorical variable (`Month`) by a `list` of categorical variables. Here are the monthly means calculated separately for each year, as specified by `list(Year,Month)`. The variable you name first in the list (`Year`) will appear as the row of the results table and the second will appear as the columns (`Month`):

```
tapply(AvgTemperature,list(Year,Month),mean)[,1:6]
```

```
##             1        2        3        4        5        6
## 200        NA       NA       NA       NA       NA       NA
## 201        NA       NA       NA       NA       NA       NA
## 1995 22.59082 31.34159 31.27190 37.80881 44.95441 50.55163
## 1996 18.08208 19.10475 25.95685 35.57652 43.73778 50.99244
## 1997 25.79462 30.60540 33.23219 39.33719 46.48344 53.38289
## 1998 27.00767 29.15087 32.80416 40.43363 48.75534 54.22356
## 1999 26.07477 30.85468 35.46308 42.24148 47.33011 54.67496
## 2000 25.05871 32.17456 35.57957 46.41452 53.99290 59.49000
## 2001 33.14029 33.28294 40.46552 43.84822 52.21448 54.50970
## 2002 30.88344 36.58143 34.60559 43.68704 54.10151 38.15141
## 2003 28.42867 26.63484 36.12358 40.95430 48.39011 56.53770
## 2004 27.87397 31.66127 37.06129 44.72202 52.05897 62.11159
## 2005 35.04232 31.47177 38.67312 48.97246 55.90238 63.44032
## 2006 29.23203 32.16905 38.62158 47.16979 53.53997 60.69301
## 2007 40.09750 38.35009 43.29540 52.09333 57.68266 65.78025
## 2008 37.41471 40.13305 42.38426 50.03049 58.44178 64.35772
## 2009 33.20831 35.33643 42.16637 47.20308 59.01427 64.22975
## 2010 25.41574 30.35200 37.11865 45.17927 51.46735 60.83972
## 2011 32.77954 33.30758 40.97986 50.51585 57.07600 62.27951
## 2012 35.67784 30.45354 45.51511 49.80105 58.81749 60.94851
## 2013 34.17175 35.80811 37.82493 49.38892 57.52337 64.08892
## 2014 37.04133 35.72973 46.33086 52.17991 57.58596 64.26360
## 2015 36.87864 36.98958 43.73880 50.28946 57.90881 64.77072
## 2016 34.73095 41.22544 38.58570 50.50559 53.44534 65.47865
## 2017 30.96312 39.02403 46.69494 48.59640 58.65650 66.35387
## 2018 34.86553 32.96776 39.32351 53.77184 61.45348 65.32789
## 2019 30.31927 40.28722 45.52725 46.30789 34.89703 66.84974
## 2020 38.31691 42.69720 44.17044 50.36505 53.66778       NA
```

The subscripts [,1:6] simply restrict the output to the first six months.

There is just one thing about `tapply` that might confuse you. If you try to apply a function that has built-in protection against missing values, then `tapply` may not do what you want, producing `NA` instead of the numerical answer. This is most likely to happen with the mean function because its default is to produce NA when there are one or more missing values. The remedy is to provide an extra argument to `tapply`, specifying that you want to see the average of the non-missing values. Use `na.rm=TRUE` to remove the missing values like this:

```
tapply(AvgTemperature,Year,mean,na.rm=TRUE)
```

```
##       200       201      1995      1996      1997      1998      1999      2000
## -99.00000 -99.00000  38.63447  36.53759  41.29181  40.93638  42.25044  46.42167
##      2001      2002      2003      2004      2005      2006      2007      2008
##  44.92223  43.83141  43.50428  47.62057  48.85940  48.09095  50.78713  51.06084
##      2009      2010      2011      2012      2013      2014      2015      2016
##  49.92104  47.09856  49.96520  49.11578  51.17017  52.33811  51.77581  50.39242
##      2017      2018      2019      2020
##  51.68254  50.20742  49.60530  44.80030
```

You might want to trim some of the extreme values before calculating the mean (the arithmetic mean is famously sensitive to outliers). The `trim` option allows you to specify the fraction of the data (between 0 and 0.5) that you want to be omitted from the left- and right-hand tails of the sorted vector of values before computing the mean of the central values:

```
tapply(AvgTemperature,Year,mean,trim=0.2)
```

```
##       200       201      1995      1996      1997      1998      1999      2000
## -99.00000 -99.00000  49.73343  48.49846  50.11555  50.42651  51.01058  52.25806
##      2001      2002      2003      2004      2005      2006      2007      2008
##  51.33122  51.84361  50.82469  51.57898  51.90048  52.28212  52.73041  52.23925
##      2009      2010      2011      2012      2013      2014      2015      2016
##  52.62449  50.94259  52.62774  51.74824  51.57276  53.23475  52.13158  52.07910
##      2017      2018      2019      2020
##  52.29157  53.30457  52.35184  44.95409
```

2.7.5.2 `aggregate` function

Suppose that we have two response variables (`y` and `z`) and two explanatory variables (`x` and `w`) that we might want to use to summarize functions like mean or variance of `y` and/or `z`. The `aggregate` function has a formula method which allows elegant summaries of four kinds:

- one to one: `aggregate(y ~ x, mean)`
- one to many: `aggregate(y ~ x + w, mean)`
- many to one: `aggregate(cbind(y,z) ~ x, mean)`
- many to many: `aggregate(cbind(y,z) ~ x + w, mean)`

Here is an example using a dataframe with many categorical and continuous variables:

```
df = read.csv("./Data/bank.csv", sep = ";", stringsAsFactors = TRUE)
head(df)
```

```
##   age         job marital         education default housing    loan   contact
## 1  30 blue-collar married          basic.9y      no     yes      no  cellular
## 2  39    services  single       high.school      no      no      no telephone
## 3  25    services married       high.school      no     yes      no telephone
## 4  38    services married          basic.9y      no unknown unknown telephone
## 5  47      admin. married university.degree      no     yes      no  cellular
## 6  32    services  single university.degree      no      no      no  cellular
##   month day_of_week duration campaign pdays previous    poutcome emp.var.rate
## 1   may         fri      487        2   999        0 nonexistent         -1.8
## 2   may         fri      346        4   999        0 nonexistent          1.1
## 3   jun         wed      227        1   999        0 nonexistent          1.4
## 4   jun         fri       17        3   999        0 nonexistent          1.4
## 5   nov         mon       58        1   999        0 nonexistent         -0.1
## 6   sep         thu      128        3   999        2     failure         -1.1
##   cons.price.idx cons.conf.idx euribor3m nr.employed  y
## 1         92.893         -46.2     1.313      5099.1 no
## 2         93.994         -36.4     4.855      5191.0 no
## 3         94.465         -41.8     4.962      5228.1 no
## 4         94.465         -41.8     4.959      5228.1 no
## 5         93.200         -42.0     4.191      5195.8 no
## 6         94.199         -37.5     0.884      4963.6 no
```

Here is one-to-one use of `aggregate` to find mean the `cons.price.idx` in the different `marital` samples:

```
aggregate(cons.price.idx ~ marital, df, mean)
```

```
##    marital cons.price.idx
## 1 divorced       93.60055
## 2  married       93.59904
## 3   single       93.52997
## 4  unknown       93.53755
```

Here is a one-to-many use to look at the interaction between `marital` and `contact`:

```
aggregate(cons.price.idx ~ marital + contact, df, mean)
```

```
##    marital   contact cons.price.idx
## 1 divorced  cellular       93.34316
## 2  married  cellular       93.32591
## 3   single  cellular       93.34077
## 4  unknown  cellular       93.27671
## 5 divorced telephone       94.04743
## 6  married telephone       94.03531
## 7   single telephone       93.99391
## 8  unknown telephone       93.99400
```

Finally, here is a many-to-many use to find mean `euribor3m` as well as mean `cons.price.idx` for the interaction between `marital` and `contact`:

```
aggregate(cbind(cons.price.idx, euribor3m) ~ marital + contact, df, mean)
```

```
##    marital   contact cons.price.idx euribor3m
## 1 divorced  cellular       93.34316  3.155212
## 2  married  cellular       93.32591  3.206862
## 3   single  cellular       93.34077  2.936846
## 4  unknown  cellular       93.27671  3.282143
## 5 divorced telephone       94.04743  4.665497
## 6  married telephone       94.03531  4.561857
## 7   single telephone       93.99391  4.372266
## 8  unknown telephone       93.99400  4.858000
```

2.7.5.3 Parallel minima and maxima

Here are three vectors of the same length, `x`, `y` and `z`. The parallel minimum function, `pmin`, finds the minimum from any one of the three variables for each subscript, and produces a vector as its result (of length equal to the longest of `x`, `y`, or `z`):

```
x = runif(6)
x
```

```
## [1] 0.8711427 0.4030111 0.5606412 0.2872073 0.2260351 0.1647041
```

```
y = runif(6)
y
```

```
## [1] 0.9675614 0.2442290 0.9152520 0.2996409 0.5876295 0.9767194
```

```
z = runif(6)
z
```

```
## [1] 0.6375336 0.9140610 0.3177856 0.4496507 0.6497485 0.2029685
```

```
pmin(x,y,z)
```

```
## [1] 0.6375336 0.2442290 0.3177856 0.2872073 0.2260351 0.1647041
```

2.7.5.4 Summary by groups with tapply

The vector function `tapply` is one of the most important and useful vector functions to master. The 't' stands for 'table' and the idea is to apply a function to produce a table from the values in the vector, based on one or more grouping variables (often the grouping is by factor levels). This sounds much more complicated than it really is:

```
daphnia = read.table("./Data/daphnia.txt", header = TRUE)
attach(daphnia)
names(daphnia)
```

```
## [1] "Growth.rate" "Water"       "Detergent"   "Daphnia"
```

The response variable is `Growth.rate` and the other three variables are factors. Suppose we want the mean growth rate for each detergent:

```
tapply(Growth.rate, Detergent, mean)
```

```
##   BrandA   BrandB   BrandC   BrandD
## 3.884832 4.010044 3.954512 3.558231
```

This produces a table with four entries, one for each level of the factor called `Detergent`.

To produce a two-dimensional table we put the two grouping variables in a list. Here we calculate the median growth rate for water type and daphnia clone:

```
tapply(Growth.rate,list(Water,Daphnia),median)
```

```
##        Clone1   Clone2   Clone3
## Tyne 2.874053 3.908644 4.618288
## Wear 2.590373 5.532726 4.302642
```

The first variable in the list creates the rows of the table and the second the columns.

2.7.5.5 Addresses within vectors

There is an important function called `which` for finding addresses within vectors. The vector y looks like this:

```
y = c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
```

Suppose we wanted to know which elements of `y` contained values bigger than 5. We type:

```
which(y>5)
```

```
## [1]  1  4  5  6  7  8 11 13 15
```

Notice that the answer to this enquiry is a set of subscripts. We do not use subscripts inside the `which` function itself. The function is applied to the whole array. To see the values of y that are larger than 5, we just type:

```
y[y>5]
```

```
## [1]  8  7  6  6  8  9  9 10 11
```

2.7.5.6 Finding closest values

Finding the value in a vector that is closest to a specified value is straightforward using which. The vector `x` contains 1000 random numbers from a normal distribution with mean = 100 and standard deviation = 10:

```
set.seed(2020)
x = rnorm(1000, 100, 10)
```

Here, we want to find the value of `x` that is closest to 108.0. The logic is to work out the difference between 108 and each of the 1000 random numbers, then find which of these differences is the smallest. This is what the R code looks like:

```
which(abs(x-108) == min(abs(x-108)))
```

```
## [1] 626
```

The closest value to 108.0 is in location 626 within `x`. But just how close to 108.0 is this 626 value? We use 626 as a subscript on `x` to find this out:

```
x[626]
```

```
## [1] 108.0139
```

2.7.5.7 Sorting, Ranking and Ordering

These three related concepts are important, and one of them (order) is difficult to understand at the beginning. Let us take a simple example:

```
set.seed(2020)
houses_prices = runif(10, 150, 500)
houses_prices
```

```
##  [1] 376.4160 287.9790 366.4756 316.9119 197.6340 173.5845 195.2034 287.5913
##  [9] 150.9039 367.0721
```

We apply the three different functions to the vector called `houses_price`:

```
ranks = rank(houses_prices)
ranks
```

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

```
sorted = sort(houses_prices)
sorted
```

```
##  [1] 150.9039 173.5845 195.2034 197.6340 287.5913 287.9790 316.9119 366.4756
##  [9] 367.0721 376.4160
```

```
ordered = order(houses_prices)
ordered
```

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

Let's create a data frame to observe the data easily:

```
view = data.frame(houses_prices, ranks, sorted, ordered)
view
```

```
##    houses_prices ranks   sorted ordered
## 1       376.4160    10 150.9039       9
## 2       287.9790     6 173.5845       6
## 3       366.4756     8 195.2034       7
## 4       316.9119     7 197.6340       5
## 5       197.6340     4 287.5913       8
## 6       173.5845     2 287.9790       2
## 7       195.2034     3 316.9119       4
## 8       287.5913     5 366.4756       3
## 9       150.9039     1 367.0721      10
## 10      367.0721     9 376.4160       1
```

RANK: The prices themselves are in no particular sequence. The `ranks` column contains the value that is the rank of the particular data point (value of `Price`), where 1 is assigned to the lowest data point and `length(Price)` – here 10 – is assigned to the highest data point.

SORT: The sorted vector is very straightforward. It contains the values of `Price` sorted into ascending order. If you want to sort into descending order, use the reverse order function `rev` like this:

```
rev(sort(houses_prices))
```

```
##  [1] 376.4160 367.0721 366.4756 316.9119 287.9790 287.5913 197.6340 195.2034
##  [9] 173.5845 150.9039
```

ORDER: This is the most important of the three functions, and the hardest to understand. The numbers in this column are subscripts between 1 and 10. The order function returns an integer vector containing the permutation that will sort the input into ascending order.

Using `order` with subscripts is a much safer option than using `sort`, because with `sort` the values of the response variable and the explanatory variables

could be uncoupled with potentially disastrous results if this is not realized at the time that modelling was carried out.

Imagine we have another variable that is the location of the house represented by a letter:

```
set.seed(2020)
location = sample(LETTERS, 10)
location
```

```
##  [1] "L" "W" "V" "A" "Q" "D" "J" "F" "X" "M"
```

We can use the `order` function to organize the location of the houses like this:

```
location[order(houses_prices)]
```

```
##  [1] "X" "D" "J" "Q" "F" "W" "A" "V" "M" "L"
```

2.7.5.8 `unique` and `duplicated` functions

The difference is best seen with a simple example. Here is a vector of names:

```
names = c("Williams","Jones","Smith","Williams","Jones","Williams")
```

We can see how many times each name appears using table:We can see how many times each name appears using `table`:

```
table(names)
```

```
## names
##    Jones    Smith Williams
##        2        1        3
```

It is clear that the vector contains just three different names. The function called `unique` extracts these three unique names, creating a vector of length 3, unsorted, in the order in which the names are encountered in the vector:

```
unique(names)
```

```
## [1] "Williams" "Jones"    "Smith"
```

In contrast, the function called `duplicated` produces a vector, of the same length as the vector of names, containing the logical values either `FALSE` or `TRUE`, depending upon whether or not that name has appeared already (reading from the left). You need to see this in action to understand what is happening, and why it might be useful:

```
duplicated(names)
```

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

The first three names are not duplicated (`FALSE`), but the last three are all duplicated (`TRUE`). We can mimic the `unique` function by using this vector as subscripts like this:

```
names[!duplicated(names)]
```

```
## [1] "Williams" "Jones"    "Smith"
```

There you have it: if you want a shortened vector, containing only the unique values in names, then use `unique`, but if you want a vector of the same length as names then use `duplicated`.

2.7.6 Sets

There are three essential functions for manipulating sets. The principles are easy to see if we work with an example of two sets:

```
setA = c("a", "b", "c", "d", "e")
setB = c("d", "e", "f", "g")
```

The **union** of two sets is everything in the two sets taken together, but counting elements only once that are common to both sets:

```
union(setA, setB)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g"
```

The **intersection** of two sets is the material that they have in common:

```
intersect(setA, setB)
```

```
## [1] "d" "e"
```

Note, however, that the **difference** between two sets is order-dependent. It is the material that is in the first named set, that is not in the second named set. Thus `setdiff(A,B)` gives a different answer than `setdiff(B,A)`. For our example:

```
setdiff(setA, setB)
```

```
## [1] "a" "b" "c"
```

```
setdiff(setB, setA)
```

```
## [1] "f" "g"
```

There is also a built-in function `setequal` for testing if two sets are equal:

```
setequal(setA, setB)
```

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

You can use `%in%` for comparing sets. The result is a logical vector whose length matches the vector on the left:

```
setA %in% setB
```

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

```
setB %in% setA
```

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

2.7.7 Factors

Conceptually, factors are variables in R which take on a limited number of different values; such variables are often refered to as categorical variables. One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors ensures that the modeling functions will treat such data correctly.

Factors in R are stored as a vector of integer values with a corresponding set of character values to use when the factor is displayed. The `factor` function is used to create a factor. The only required argument to `factor` is a vector of

values which will be returned as a vector of factor values. Both numeric and character variables can be made into factors, but the factor levels will always be character values. You can see the possible levels for a factor through the `levels` command.

To change the order in which the levels will be displayed from their default sorted order, the `levels=` argument can be given a vector of all the possible values of the variable in the order you desire. If the ordering should also be used when performing comparisons, use the optional `ordered=TRUE` argument. In this case, the factor is known as an ordered factor.

Reminder:

- **Categorical variable**: one that can take a limited number of possible values (categories). Examples of categorical variables: gender of a person, nationality…
  - Categorical nominal variable: one that does not have a pre-established order.
  - Categorical ordinal variable: that which has an established order.
- **Continuous variable**: one that can take an infinite number of possible values. Examples of continuous variables: weight of a person, height of a person…

The different values that the variable can take are called levels, **factor levels**.

Why use factors?

- They allow to establish a different order than the alphabetical one.
- Many R models/packages use them

Factors are categorical variables that have a fixed number of levels. A simple example of a factor might be a variable called gender with two levels: 'female' and 'male'. If you had three females and two males, you could create the factor like this:

```
gender = factor(c("female", "male", "female", "male", "female"))
class(gender)
```

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

```
mode(gender)
```

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

More often, you will create a dataframe by reading your data from a file using read.table. When we did this in the previous version of R (R3), the strings in our data table were automatically converted into factors. This was useful as long as our data table was well structured and clean, but it could cause problems in some cases. Let's see an example of a well-structured data table:

```
data = read.table(file = "./Data/GenderPurchase.csv", header = TRUE, sep = ",")
is.factor(data$Gender)
```

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

As we can see, in this version of R (R4), the strings are coded as default character type. We can solve this by adding a parameter to our command as seen below:

```
data = read.table(file = "./Data/GenderPurchase.csv", header = TRUE, sep = ",", stringsAsFactors
is.factor(data$Gender)
```

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

Let's see what happens with a data table that is not well structured in case of reading the strings as factors:

```
data = read.table(file = "./Data/titanic3.csv", header = TRUE, sep = ",", stringsAsFactors = TRUE
str(data)
```

```
## 'data.frame':    1309 obs. of  14 variables:
##  $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
##  $ name     : Factor w/ 1307 levels "Abbing, Mr. Anthony",..: 22 24 25 26 27 31 46 47 51 55 ..
##  $ sex      : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 2 ...
##  $ age      : num  29 0.917 2 30 25 ...
##  $ sibsp    : int  0 1 1 1 1 0 1 0 2 0 ...
##  $ parch    : int  0 2 2 2 2 0 0 0 0 0 ...
##  $ ticket   : Factor w/ 929 levels "110152","110413",..: 188 50 50 50 50 125 93 16 77 826 ...
##  $ fare     : num  211 152 152 152 152 ...
##  $ cabin    : Factor w/ 187 levels "","A10","A11",..: 45 81 81 81 81 151 147 17 63 1 ...
##  $ embarked : Factor w/ 4 levels "","C","Q","S": 4 4 4 4 4 4 4 4 4 2 ...
##  $ boat     : Factor w/ 28 levels "","1","10","11",..: 13 4 1 1 1 14 3 1 28 1 ...
##  $ body     : int  NA NA NA 135 NA NA NA NA NA 22 ...
##  $ home.dest: Factor w/ 370 levels "","?Havana, Cuba",..: 310 232 232 232 232 238 163 25 23 23
```

We can find ourselves with variables that are completely useless and very difficult to manage, with as many levels of the factors as cases we have in the data table.

In these cases it is much more comfortable to read the strings as characters and work on those variables to extract useful information from them as we will see later.

We will use the daphnia dataset to perform some examples with factors:

```
daphnia = read.table("./Data/daphnia.txt", header = TRUE, stringsAsFactors = TRUE)
attach(daphnia)
```

```
## The following objects are masked from daphnia (pos = 3):
##
##     Daphnia, Detergent, Growth.rate, Water
```

There are some important functions for dealing with factors. You will often want to check that a variable is a factor (especially if the factor levels are numbers rather than characters):

```
is.factor(Water)
```

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

To discover the names of the factor levels, we use the levels function:

```
levels(Detergent)
```

```
## [1] "BrandA" "BrandB" "BrandC" "BrandD"
```

To discover the number of levels of a factor, we use the nlevels function:

```
nlevels(Detergent)
```

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

The same result is achieved by applying the length function to the levels of a factor:

```
length(levels(Detergent))
```

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

By default, factor levels are treated in alphabetical order. If you want to change this (as you might, for instance, in ordering the bars of a bar chart) then this is straightforward: just type the factor levels in the order that you want them to be used, and provide this vector as the second argument to the factor function.

Suppose we have an experiment with three factor levels in a variable called treatment, and we want them to appear in this order: 'nothing', 'single' dose and 'double' dose. We shall need to override R's natural tendency to order them 'double', 'nothing', 'single':

```
set.seed(2020)
treatment = as.factor(sample(c("double", "nothing", "single"), 100, replace = TRUE))
response = sample(c(0,1), 100, replace = TRUE)
tapply(response, treatment, mean)
```

```
##    double   nothing    single
## 0.4166667 0.4871795 0.4000000
```

This is achieved using the factor function like this:

```
treatment = factor(treatment,levels=c("nothing","single","double"))
```

Now we get the order we want:

```
tapply(response,treatment,mean)
```

```
##   nothing    single    double
## 0.4871795 0.4000000 0.4166667
```

Only == and != can be used for factors. Note, also, that a factor can only be compared to another factor with an identical set of levels (not necessarily in the same ordering) or to a character vector. For example, you cannot ask quantitative questions about factor levels, like > or <=, even if these levels are numeric.

To turn factor levels into numbers (integers) use the unclass function like this:

```
as.vector(unclass(treatment))
```

```
##   [1] 2 1 3 3 1 1 3 3 1 1 1 3 1 2 1 3 1 1 1 2 1 2 1 3 2 1 1 1 2 1 2 2 3 3 3 1 2
##  [38] 3 3 1 2 3 3 3 2 3 3 2 1 1 1 3 2 1 3 1 2 3 3 1 3 1 2 1 2 1 1 3 2 1 2 3 1 2
##  [75] 1 1 1 3 3 3 2 1 1 3 3 1 3 3 1 3 2 3 2 3 3 2 2 2 1 3
```

2.7.8 Matrices and Arrays

An array is a multi-dimensional object. The dimensions of an array are specified by its `dim` attribute, which gives the maximal indices in each dimension. So for a three-dimensional array consisting of 24 numbers in a sequence 1:24, with dimensions $2 \times 4 \times 3$, we write:

```
y = 1:24
dim(y) = c(2,4,3)
y
```

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

This produces three two-dimensional tables, because the third dimension is 3. This is what happens when you change the dimensions:

```
dim(y) = c(3,2,4)
y
```

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

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

Now we have four two-dimensional tables, each of three rows and two columns.

A matrix is a two-dimensional array containing numbers. A dataframe is a two-dimensional list containing (potentially a mix of) numbers, text or logical variables in different columns.

When there are two subscripts [5,3] to an object like a matrix or a dataframe, the first subscript refers to the row number (5 in this example; the rows are defined as margin number 1) and the second subscript refers to the column number (3 in this example; the columns are margin number 2).

There is an important and powerful convention in R, such that when a subscript appears as a blank it is understood to mean 'all of'. Thus:

- [,4] means all rows in column 4 of an object;
- [2,] means all columns in row 2 of an object.

2.7.9 Matrices

There are several ways of making a matrix. You can create one directly like this:

```
X = matrix(c(1,0,0,0,1,0,0,0,1),nrow=3)
X
```

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

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

where, by default, the numbers are entered column-wise.

The class and attributes of `X` indicate that it is a matrix of three rows and three columns (these are its `dim` attributes):

```
class(X)
```

```
## [1] "matrix" "array"
```

```
attributes(X)
```

```
## $dim
## [1] 3 3
```

In the next example, the data in the vector appear row-wise, so we indicate this with `byrow=T`:

```
vector = c(1,2,3,4,4,3,2,1)
V = matrix(vector,byrow=T,nrow=2)
V
```

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

Another way to convert a vector into a matrix is by providing the vector object with two dimensions (rows and columns) using the dim function like this:

```
dim(vector) = c(4,2)
is.matrix(vector)
```

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

We need to be careful, however, because we have made no allowance at this stage for the fact that the data were entered row-wise into vector:

```
vector
```

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

The matrix we want is the transpose, `t`, of this matrix:

```
t(vector)
```

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

2.7.9.1 Naming the rows and columns of matrices

At first, matrices have numbers naming their rows and columns (see above). Here is a 4×5 matrix of random integers from a Poisson distribution with mean 1.5:

```
X = matrix(rpois(20,1.5),nrow=4)
X
```

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

Suppose that the rows refer to four different trials and we want to label the rows 'Trial.1' etc. We employ the function `rownames` to do this. We could use the `paste` function but here we take advantage of the `prefix` option:

```
rownames(X) = rownames(X,do.NULL=FALSE,prefix="Trial.")
X
```

```
##         [,1] [,2] [,3] [,4] [,5]
## Trial.1    0    2    4    0    1
## Trial.2    1    6    2    2    2
## Trial.3    0    2    3    1    1
## Trial.4    1    2    1    1    1
```

For the columns we want to supply a vector of different names for the five drugs involved in the trial, and use this to specify the `colnames(X)`:

```
drug.names = c("aspirin", "paracetamol", "nurofen", "hedex", "placebo")
colnames(X) = drug.names
X
```

```
##         aspirin paracetamol nurofen hedex placebo
## Trial.1       0           2       4     0       1
## Trial.2       1           6       2     2       2
## Trial.3       0           2       3     1       1
## Trial.4       1           2       1     1       1
```

2.7.9.2 Matrices Functions

Using R, we can operate with matrices easily, since by default this type of data structures are designed for matrix algebra.

Let's create a matrix to see some examples:

```
M1 = matrix(1:9, byrow = T, nrow = 3)
M1
```

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

We can know the size of the matrix using the `dim` function:

```
dim(M1)
```

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

We can also create diagonal matrices using the `diag` function:

```
diag(5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

Or even create a diagonal matrix using a defined vector:

```
diag(c(1,2,3))
```

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

We can calculate a matrix transpose:

```
t(M1)
```

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

And the trace of a matrix easily:

```
sum(diag(M1))
```

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

To calculate the determinant we just need to use the `det` function:

```
det(M1)
```

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

In case the determinant is not 0 we can calculate the inverse of that matrix using the `solve` function. Let's see what happens in this case:

```
#solve(M1)
```

```
Error in solve.default(M1) : system is computationally singular: reciprocal condition number = 2.
```

We can create a random matrix to check this function:

```
set.seed(2020)
M2 = matrix(sample(1:100, size = 9), byrow = T, nrow = 3)
det(M2)
```

```
## [1] -345436
```

And now apply the `solve` function:

```
solve(M2)
```

```
##               [,1]         [,2]         [,3]
## [1,] -0.0111048067  0.016060862 -0.003925474
## [2,]  0.0149550134 -0.003381234 -0.005662409
## [3,] -0.0001418497 -0.004226543  0.016894591
```

Finally, we can perform a matrix decomposition using the `eigen` function in R, obtaining the eigenvectors and the eigenvalues of that decomposition,

```
eigen(M1)
```

```
## eigen() decomposition
## $values
## [1]  1.611684e+01 -1.116844e+00 -5.700691e-16
##
## $vectors
##            [,1]       [,2]       [,3]
## [1,] -0.4645473 -0.8829060  0.4082483
## [2,] -0.5707955 -0.2395204 -0.8164966
## [3,] -0.6770438  0.4038651  0.4082483
```

or even a Singular Value Decomposition using the `svd` function:

```
s = svd(M1)
s
```

```
## $d
## [1] 1.684810e+01 1.068370e+00 5.543107e-16
##
## $u
##            [,1]        [,2]       [,3]
## [1,] -0.4796712  0.77669099  0.4082483
## [2,] -0.5723678  0.07568647 -0.8164966
```

```
## [3,] -0.6650644 -0.62531805  0.4082483
##
## $v
##             [,1]       [,2]       [,3]
## [1,] -0.2148372 -0.8872307  0.4082483
## [2,] -0.5205874 -0.2496440 -0.8164966
## [3,] -0.8263375  0.3879428  0.4082483
```

We can check that R makes a successful decomposition:

```
s$u %*% diag(s$d) %*% t(s$v)
```

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

2.7.9.3 Matrices Operations

We can perform arithmetic operations with matrices (additions, subtractions, multiplications…) in a similar way as we have done with vectors.

```
M1 = matrix(1:9, byrow = TRUE, nrow = 3)
M2 = matrix(11:19, byrow = TRUE, nrow = 3)

M3 = M1 + M2
M3
```

```
##      [,1] [,2] [,3]
## [1,]   12   14   16
## [2,]   18   20   22
## [3,]   24   26   28
```

```
M1 + 2
```

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

In the case of matrices multiplication we must remember that two matrices can be multiplied only when the number of columns in the first equals the number of rows in the second

```
M1 = matrix(1:6, byrow = TRUE, nrow = 3, ncol = 2)
M1
```

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

```
M2 = matrix(10:15, byrow = TRUE, nrow = 2, ncol = 3)
M2
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
```

```
M3 = M1 %*% M2
M3
```

```
##      [,1] [,2] [,3]
## [1,]   36   39   42
## [2,]   82   89   96
## [3,]  128  139  150
```

```
M4 = M2 %*% M1
M4
```

```
##      [,1] [,2]
## [1,]  103  136
## [2,]  130  172
```

We can also apply some other R built-in functions to a matrix. For example:

- `rowSums` will sum up all the elements for each row
- `colSums` will do the same by columns
- `rowMeans` will calculate the mean of each row (is the same for columns with `colMeans`)
- `summary` will show some statistics about the matrix by columns

```
summary(M4)
```

```
##        V1              V2
##  Min.   :103.0   Min.   :136
##  1st Qu.:109.8   1st Qu.:145
##  Median :116.5   Median :154
##  Mean   :116.5   Mean   :154
##  3rd Qu.:123.2   3rd Qu.:163
##  Max.   :130.0   Max.   :172
```

2.7.9.4 Matrices Manipulation

To add columns to a matrix you use the function `cbind(m1, m2, ...)`, which joins matrices and/or vectors per column.

```
M5 = cbind(M1, M3)
M5
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2   36   39   42
## [2,]    3    4   82   89   96
## [3,]    5    6  128  139  150
```

We can add just a new column:

```
M5 = cbind(M5, c(1,2,3))
M5
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2   36   39   42    1
## [2,]    3    4   82   89   96    2
## [3,]    5    6  128  139  150    3
```

To add rows to a matrix you use the `rbind(m1, m2, ...)` function, which joins arrays and/or vectors per row.

```
M5 = rbind(M2, M3)
M5
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   36   39   42
## [4,]   82   89   96
## [5,]  128  139  150
```

And we can add an extra row using the same function:

```
M5 = rbind(M5, c(1,2,3))
M5
```

```
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   36   39   42
## [4,]   82   89   96
## [5,]  128  139  150
## [6,]    1    2    3
```

2.7.9.5 Indexing Matrices

As with vectors, we will use the square brackets `[ ]` to index matrices. In the particular case of matrices we will use two integers: one for the row and one for the column `[row, column]`.

To select all the elements of a row or a column, it is enough not to include any number before or after the comma, respectively. For example: `matrix[row, ]`; `matrix[, col]`.

```
M1 = matrix(1:9, byrow = T, nrow = 3)
M1
```

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

Let's see some examples:

```
# We select the first two rows
M1[1:2,]
```

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

```
# We select the last column
M1[, ncol(M1)]
```

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

As we did with vectors, we can use vectors to perform selections inside a matrix,

```
# Select the first and last columns
M1[, c(1,ncol(M1))]
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

and assign values to a specific position

```
M1[1,1] = 0
M1
```

```
##      [,1] [,2] [,3]
## [1,]    0    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Finally, we can use logical expressions to select elements inside a matrix:

```
M1[M1 < 4]
```

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

```
M1[M1 > 3 & M1 < 8]
```

```
## [1] 4 5 6 7
```

2.7.10 Lists

Lists are extremely important objects in R. You will have heard of the problems of 'comparing apples and oranges' or how two things are 'as different as chalk and cheese'. You can think of lists as a way of getting around these problems. Here are four completely different objects: a numeric vector, a logical vector, a vector of character strings and a vector of complex numbers:

```
apples = c(4,4.5,4.2,5.1,3.9)
oranges = c(TRUE, TRUE, FALSE)
chalk = c("limestone", "marl","oolite", "CaC03")
cheese = c(3.2-4.5i,12.8+2.2i)
```

We cannot bundle them together into a dataframe, because the vectors are of different lengths. However, despite their differences, however, we can bundle them together in a single list called items:

```
items = list(apples,oranges,chalk,cheese)
items
```

```
## [[1]]
## [1] 4.0 4.5 4.2 5.1 3.9
##
## [[2]]
## [1]  TRUE  TRUE FALSE
##
## [[3]]
## [1] "limestone" "marl"      "oolite"    "CaCO3"
##
## [[4]]
## [1]  3.2-4.5i 12.8+2.2i
```

Subscripts on vectors, matrices, arrays and dataframes have one set of square brackets [6], [3,4] or [2,3,2,1], but subscripts on lists have double square brackets [[2]] or [[i,j]]. If we want to extract the chalk from the list, we use subscript [[3]]:

```
items[[3]]
```

```
## [1] "limestone" "marl"      "oolite"    "CaCO3"
```

If we want to extract the third element within chalk (oolite) then we use single subscripts after the double subscripts like this:

```
items[[3]][3]
```

```
## [1] "oolite"
```

There is another indexing convention in R which is used to extract named components from lists using the element names operator $. This is known as 'indexing tagged lists'. For this to work, the elements of the list must have names. At the moment our list called `items` has no names:

```
names(items)
```

```
## NULL
```

You can give names to the elements of a list in the function that creates the list by using the equals sign like this:

```
items = list(first=apples,second=oranges,third=chalk,fourth=cheese)
```

Now you can extract elements of the list by name

```
items$fourth
```

```
## [1]  3.2-4.5i 12.8+2.2i
```

2.7.10.1 lapply function

We can ask a variety of questions about our new list object:

```
class(items)
```

```
## [1] "list"
```

```
is.numeric(items)
```

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

```
is.list(items)
```

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

```
length(items)
```

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

Note that the length of a list is the number of items in the list, not the lengths of the individual vectors within the list.

The function lapply applies a specified function to each of the elements of a list in turn (without the need for specifying a loop, and not requiring us to know how many elements there are in the list). A useful function to apply to lists is the length function; this asks how many elements comprise each component of the list. Technically we want to know the length of each of the vectors making up the list:

```
lapply(items, length)
```

```
## $first
## [1] 5
##
## $second
## [1] 3
##
## $third
## [1] 4
##
## $fourth
## [1] 2
```

This shows that items consists of four vectors, and shows that there were 5 elements in the first vector, 3 in the second 4 in the third and 2 in the fourth. But 5 of what, and 3 of what? To find out, we apply the function class to the list:

```
lapply(items, class)
```

```
## $first
## [1] "numeric"
##
## $second
## [1] "logical"
##
## $third
## [1] "character"
##
## $fourth
## [1] "complex"
```

So the answer is there were 5 numbers in the first vector, 3 logical variables in the second, 4 character strings in the third vector and 2 complex numbers in the fourth.

Applying numeric functions to lists will only work for objects of class `numeric` or `complex`, or objects (like logical values) that can be coerced into numbers. Here is what happens when we use `lapply` to apply the function `mean` to `items`:

```
lapply(items, mean)
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```
## $first
## [1] 4.34
##
## $second
## [1] 0.6666667
##
## $third
## [1] NA
##
## $fourth
## [1] 8-1.15i
```

Finally, the most useful overview of the contents of a list is obtained with str, the structure function:

```
str(items)
```

```
## List of 4
##  $ first : num [1:5] 4 4.5 4.2 5.1 3.9
##  $ second: logi [1:3] TRUE TRUE FALSE
##  $ third : chr [1:4] "limestone" "marl" "oolite" "CaC03"
##  $ fourth: cplx [1:2] 3.2-4.5i 12.8+2.2i
```

2.7.11 Data Frames

Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length.

2.7.11.1 Check if a variable is a data frame or not

We can check if a variable is a data frame or not using the `class()` function.

```
typeof(x)
```

```
## [1] "list"
```

```
class(x)
```

```
## [1] "data.frame"
```

In this example, x can be considered as a list of 3 components with each component having a two element vector.

2.7.11.2 Functions of data frame

Here you have some useful functions we can use to have a better understanding about the content we have inside our data frame:

- We can check the names of the variables using the `names()` function

```
names(x)
```

```
## [1] "SN"     "Age"    "Gender"
```

- We can know the number of columns and rows we have in our data frame (the `length()` function will return also the number of columns, it is taking the number of elements we are storing in the underlying list, which are the column vectors):

```
ncol(x)
```

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

```
nrow(x)
```

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

```
length(x)
```

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

- The `str()` and the `summary()` are the most useful functions to understand our data frames:

```
str(x)
```

```
## 'data.frame':    2 obs. of  3 variables:
##  $ SN    : int  1 2
##  $ Age   : num  21 15
##  $ Gender: chr  "Male" "Female"
```

```
summary(x)
```

```
##        SN            Age          Gender
##  Min.   :1.00   Min.   :15.0   Length:2
##  1st Qu.:1.25   1st Qu.:16.5   Class :character
##  Median :1.50   Median :18.0   Mode  :character
##  Mean   :1.50   Mean   :18.0
##  3rd Qu.:1.75   3rd Qu.:19.5
##  Max.   :2.00   Max.   :21.0
```

2.7.11.3 How to create a data frame

We can create a data frame using the `data.frame()` function. Below you can see how to create the data frame we are using for all the previous examples:

```
x = data.frame("SN" = 1:2, "Age" = c(21,15), "Gender" = c("Male","Female"))
```

Let's take another look to the data frame structure:

```
str(x)
```

```
## 'data.frame':    2 obs. of  3 variables:
##  $ SN    : int  1 2
##  $ Age   : num  21 15
##  $ Gender: chr  "Male" "Female"
```

Which is the type of the `Gender` variable? Do we need to change it? It is a character vector and, in order to use this variable in future models we will need to transform it into a factor. Remember that R3 automatically converts all the characters into factors, but we need to define those in R4.

```
x = data.frame("SN" = 1:2, "Age" = c(21,15), "Gender" = c("Male","Female"), stringsAsFactors = T)
str(x)
```

```
## 'data.frame':    2 obs. of  3 variables:
##  $ SN    : int  1 2
##  $ Age   : num  21 15
##  $ Gender: chr  "Male" "Female"
```

NOTE: Many data input functions of R like, `read.table()`, `read.csv()`, `read.delim()`, `read.fwf()` also read data into a data frame.

2.7.11.4 How to access components of a data frame

Components of data frame can be accessed like a list or like a matrix.

2.7.11.4.1 Accessing like a list We can use either `[`, `[[` or `$` operator to access columns of data frame.

Accessing with `[[` or `$` is similar. However, it differs for `[` in that, indexing with `[` will return us a data frame but the other two will reduce it into a vector.

```
x["Gender"]
```

```
##   Gender
## 1   Male
## 2 Female
```

```
x$Age
```

```
## [1] 21 15
```

```
x[["Gender"]]
```

```
## [1] "Male"   "Female"
```

```
x[2]
```

```
##   Age
## 1  21
## 2  15
```

```
x[[3]]
```

```
## [1] "Male"   "Female"
```

2.7.11.4.2 Accessing like a matrix Data frames can be accessed like a matrix by providing index for row and column.

To illustrate this, we use datasets already available in R. Datasets that are available can be listed with the command `library(help = "datasets")`.

We will use the `trees` dataset which contains `Girth`, `Height` and `Volume` for Black Cherry Trees.

A data frame can be examined using functions like `str()` and `head()`.

```
str(trees)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
head(trees, 3)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
```

We can see that trees is a data frame with 31 rows and 3 columns. We also display the first 3 rows of the data frame.

Now we proceed to access the data frame like a matrix.

```
trees[2:3,]    # select 2nd and 3rd row
```

```
##   Girth Height Volume
## 2   8.6     65   10.3
## 3   8.8     63   10.2
```

```
trees[trees$Height > 82,]    # selects rows with Height greater than 82
```

```
##    Girth Height Volume
## 6   10.8     83   19.7
## 17  12.9     85   33.8
## 18  13.3     86   27.4
## 31  20.6     87   77.0
```

2.7.11.5 How to modify a Data Frame

Data frames can be modified like we modified matrices through reassignment.

```
x[1,"Age"] = 20; x
```

```
##   SN Age Gender
## 1  1  20   Male
## 2  2  15 Female
```

2.7.11.5.1 Adding Components Rows can be added to a data frame using the `rbind()` function.

```
(rbind(x,list(1,16,"Female")))
```

```
##   SN Age Gender
## 1  1  20   Male
## 2  2  15 Female
## 3  1  16 Female
```

Similarly, we can add columns using `cbind()`.

```
(cbind(x,State=c("NY","FL")))
```

```
##   SN Age Gender State
## 1  1  20   Male    NY
## 2  2  15 Female    FL
```

Since data frames are implemented as list, we can also add new columns through simple list-like assignments.

```
x$State = c("NY","FL"); x
```

```
##   SN Age Gender State
## 1  1  20   Male    NY
## 2  2  15 Female    FL
```

2.7.11.5.2 Deleting Components Data frame columns can be deleted by assigning `NULL` to it.

```
x$State = NULL; x
```

```
##   SN Age Gender
## 1  1  20   Male
## 2  2  15 Female
```

Similarly, rows can be deleted through reassignments.

```
x = x[-1,]; x
```

```
##   SN Age Gender
## 2  2  15 Female
```

2.7.11.6 Exporting a data frame

As we have mentioned before, any data table we import in R will take the form of a data frame by default. This facilitates us enormously the work of data cleaning since there are many libraries that work on data frames and will help us in this process. We will see many of these resources in the "Data Wrangling" module.

For now, we only need some knowledge of importing and exporting data tables in R. Let's start with a built-in dataframe like `trees`.

```
head(trees, 3)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
```

This dataframe can be exported in different formats, but let's see only the two main ones that we will use in R, .csv and .txt. The basic function in R for exporting data is called `write.table()`, and it has a lot of parameters. My recommendation is that you take a look at the documentation of this function to familiarize yourself with it.

The main parameters of the `write.table()` function are

- `x`: will be the object we want to export, in this case a data frame
- `file`: is the name of the file to be exported. In this argument we can also enter a path to organize the file in our OS. The file will take the extension that we indicate in its name.
- `append`: we will use it if the file already exists and we want to add new information. By default it is `FALSE`.
- `quote`: by default it is also `FALSE`. If you set it to `TRUE` this parameter will return the factors and strings with quotes
- `sep`: in this parameter we can decide how to separate the columns of our file. The default is a comma.
- `dec`: this parameter will allow us to change the symbol of the decimals that by default is a dot.
- ...

Let's see an example. Next we are going to export the data frame `trees` as a .txt file.

```
write.table(x = trees, file = "./Data/trees.txt", quote = TRUE, sep = ";", dec = ",", row.names =
```

Although the function `write.table()` already provides a simple method to export data tables, we can also use the function `write.csv()`, which will facilitate the export of .csv files with a predetermined format to facilitate their subsequent use.

This function `write.csv()` or `write.csv2()` (a second version that has different default parameters) use as a base the `write.table()` function, so its behavior is practically the same.

Let's see the same example, but exporting the file in this case as a .csv:

```
write.csv(x = trees, file = "./Data/trees.csv")
write.csv2(x = trees, file = "./Data/trees2.csv")
```

2.7.11.7 Importing a data frame

When importing data tables the operation will be exactly the same. In this case, the basic function of R is `read.table()`, and the functions `read.csv()` and `read.csv2()` use the previous one to define other behaviors focused on reading .csv files.

The parameters of `read.table()` are the same as those of write.table() with some exceptions.

- `header`: is a logical parameter (TRUE/FALSE) that indicates if the file contains a header.
- `quote`: it is not a logical parameter anymore, but it allows us to decide how we want to code the quotes.
- `row.names` and `col.names`: they also existed in the previous function but in this case they make more sense. These parameters will take a vector the size of the number of rows or columns respectively, and set it as such in the data frame
- `na.strings`: logical vector that by default is FALSE, and assigns as NA's to all character type elements.
- `skip`: is a parameter that will skip the number of lines of the data table that we indicate with an integer.
- `stringsAsFactors`: logical vector that by default is FALSE and helps us to code the character type vectors as factors. In some cases it will be useful, but in others we prefer to have the versatility that character vectors offer.

Let's see how to import the files that we have previously exported with the `read.table()` function:

```
df = read.table(file = "./Data/trees.txt", header = TRUE, sep = ";", dec = ",",
                na.strings = FALSE, stringsAsFactors = TRUE)
str(df)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: int  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

In the case of the functions `read.csv()` and `read.csv2()` we can ensure that the format in which we have exported the files will be the same as that in which we should read it:

```
df1 = read.csv(file = "./Data/trees.csv")
str(df1)
```

```
## 'data.frame':    31 obs. of  4 variables:
##  $ X     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: int  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
df2 = read.csv2(file = "./Data/trees2.csv")
str(df2)
```

```
## 'data.frame':    31 obs. of  4 variables:
##  $ X     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: int  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

2.8 Control Flow

Flow control statements allow us to generate portions of code that perform certain actions given certain conditions. Let's differentiate between: Conditional sentences and loops

2.8.1 Conditional statements

Conditional statements allow us to perform actions based on specific conditions that we attribute to objects created in our R session. In this way, we can

generate huge chains of conditionals that result in multiple different actions. To work with conditional statements in R we will use the following keywords: `if`, `else if` and `else`.

The specific syntax to create conditional statements in R is the following

Let's see some code examples. In this first example R will perform an action (to print something) in case `x` is a negative number:

```
x = -3
if (x < 0) {
  print("x is a negative number")
}
```

```
## [1] "x is a negative number"
```

In a second example we are using the `if` and `else` statements to print if `x` is negative or positive

```
x = 5
if (x < 0) {
  print("x is a negative number")
} else {
  print("x is a positive number or zero")
}
```

```
## [1] "x is a positive number or zero"
```

In this third example we are using more conditions to print if `x` is negative, zero or positive.

```
x = -3
if (x < 0) {
  print("x is a negative number")
} else if (x == 0) {
  print("x is zero")
} else {
  print("x is a positive number")
}
```

```
## [1] "x is a negative number"
```

2.8.2 Loops

The classic, Fortran-like loop is available in R. The syntax is a little different, but the idea is identical; you request that an index (e.g. `i`), takes on a sequence of values, and that one or more lines of commands are executed as many times as there are different values of i. Here is a loop executed five times with the values of i from 1 to 5; we print the square of each value:

```
# This is the abbreviated way to write a for loop
for (i in 1:5) print(i^2)
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

For multiple lines of code, you use curly brackets `{}` to enclose material over which the loop is to work. Note that the 'hard return' (the Enter key) at the end of each command line is an essential part of the structure (you can replace the hard returns by semicolons if you like, but clarity is improved if you put each command on a separate line):

```
j = k = 0
for (i in 1:5) {
  j = j+1
  k = k+i*j
  print(i+j+k)
  }
```

```
## [1] 3
## [1] 9
## [1] 20
## [1] 38
## [1] 65
```

There are two types of loops:

- `while` loops: they perform a certain action iteratively until a stop condition is met
- `for` loops: they execute a certain action in an iterative way for each element of a vector (set of elements).

We must be very careful not to make an infinite loop. This will produce that our R session starts working without a stop condition and we will have to kill the process to finish it. To prevent this situation we can always use the `break` keyword in our loops.

Here you can see the specific syntax we use in R to create loops:

2.8.2.1 While loops examples

In this first example we are creating a `ctr` variable that starts in `1`. Inside our while loop we are going to print our variable and later add `1` in each iteration

```
ctr = 1
while (ctr <= 7) {
  print(paste("ctr is", ctr))
  ctr = ctr + 1
}
```

```
## [1] "ctr is 1"
## [1] "ctr is 2"
## [1] "ctr is 3"
## [1] "ctr is 4"
## [1] "ctr is 5"
## [1] "ctr is 6"
## [1] "ctr is 7"
```

We can apply the same logic to create a countdown to zero:

```
count = 10
while (count >= 0){
    print (count)
    count = count - 1
}
```

```
## [1] 10
## [1] 9
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

2.8.2.2 For loops example

As we said, `for` loops are going to execute certain action for each element in any data structure (normally a vector). In this example we are just printing each element we have inside the cities vector.

```
# Print each element in a vector
cities = c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro", "Cape Town")
for (i in cities) {
  print(i)
}
```

```
## [1] "New York"
## [1] "Paris"
## [1] "London"
## [1] "Tokyo"
## [1] "Rio de Janeiro"
## [1] "Cape Town"
```

2.8.2.3 Loops and conditional statements combination

Once we know the different flow control methods we can combine them to perform much more complex actions. Let's look at some examples.

We can use a conditional statement to break our loops. In this example we are printing our `ctr` variable as we did before, but we will break the loop if the module obtained dividing ctr by 5 is equal to 0.

```
# Using the break keyword with a condition
ctr = 1
while (ctr <= 7) {
  if (ctr %% 5 == 0)
    break
  print(paste("ctr is", ctr))
  ctr = ctr + 1
}
```

```
## [1] "ctr is 1"
## [1] "ctr is 2"
## [1] "ctr is 3"
## [1] "ctr is 4"
```

We can also introduce the `next` statement inside our loops. This will skip the current iteration, not performing all the actions we have below it inside the loop.

In this case we are printing all the cities with a number of characters different to 6.

```
# Print cities with 6 characters
cities = c("New York", "Paris", "London", "Tokyo", "Rio de Janeiro", "Cape Town")
for (city in cities) {
  if (nchar(city) == 6)
    next
  print(city)
}
```

```
## [1] "New York"
## [1] "Paris"
## [1] "Tokyo"
## [1] "Rio de Janeiro"
## [1] "Cape Town"
```

In this last example we are printing only the even numbers in the `numbers` list:

```
# Print even numbers
numbers = c(1:100)
for (number in numbers){
    if (number %% 2 == 0){
        print(number)
    }
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
## [1] 22
## [1] 24
## [1] 26
## [1] 28
## [1] 30
## [1] 32
## [1] 34
## [1] 36
```

```
## [1] 38
## [1] 40
## [1] 42
## [1] 44
## [1] 46
## [1] 48
## [1] 50
## [1] 52
## [1] 54
## [1] 56
## [1] 58
## [1] 60
## [1] 62
## [1] 64
## [1] 66
## [1] 68
## [1] 70
## [1] 72
## [1] 74
## [1] 76
## [1] 78
## [1] 80
## [1] 82
## [1] 84
## [1] 86
## [1] 88
## [1] 90
## [1] 92
## [1] 94
## [1] 96
## [1] 98
## [1] 100
```

2.8.2.4 Nested for loops

We can use a for loop within another for loop to iterate over two things at once (e.g., rows and columns of a matrix).

```
for(i in 1:5){
  for(j in 1:5){
    print(paste(i,j))
  }
}
```

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

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

2.8.2.5 The `ifelse` function

Sometimes you want to do one thing if a condition is true and a different thing if the condition is false (rather than do nothing, as in previous examples). The `ifelse` function allows you to do this for entire vectors without using for loops.

```
v = 0:10
ifelse(v>5, yes = T, no = F)
```

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

```
ifelse(v > 5, v*2, v)
```

```
##  [1]  0  1  2  3  4  5 12 14 16 18 20
```

`ifelse` returns a value with the same shape as `v` which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`.

2.8.3 Functions

Functions are R objects that evaluate a set of input arguments and return an output. This is the syntax we use in R to create functions:

Arguments are a set of variable names that will be used within the function

Arguments can be:

- Required.
- Optional (they are assigned a default value).
- Adjustable length (each of these parameters is accessed with ..1, ..2, etc.).

The function return is specified by calling the function `return(x)`. Sometimes it is not necessary, and the result of the last evaluated expression will be returned. It is recommended to use it for legibility.

The function below has a required parameter (a) and an optional parameter (b).

Parameter b, being optional, if it is not passed in the call, will take the value 1.

The function has two returns:

- When b equals zero (with return)
- When b is not equal to zero (without return)

```
z = 1
do_something = function(a, b = 1) {
    if (b == 0)
        return(0)
    z = z + a*b + a/b
    z
}

do_something(5)
```

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

```
do_something(4,4)
```

```
## [1] 18
```

2.8.3.1 Domain of variables in functions

When a function is invoked, a new memory frame is created for it.

Each memory frame is inside the call stack, the first frame corresponding to the global environment.

Each function call will create a local frame.

The names of the variables within a function are resolved in the following order:

- Local environment.
- Parent environment (functions defined within functions).
- ...
- Global environment.

Thus, a variable that is defined within a function is not available outside the function.

2.8.3.2 Replicating code with functions

As in any programming environment, the functions are very useful in order to replicate code by specifying certain parameters. Remember the countdown example? Below we have converted the same code into a function to decide the initial value of this process.

```
countdown = function(count = 5){
  while (count >= 0){
    print (count)
    count = count - 1
  }
}
```

By default, the countdown starts in 5,

```
countdown()
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

But we can change the `count` parameter, so it will start in the number we specify

```
countdown(3)
```

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

2.8.4 The apply Family

The `apply()` family pertains to the R base package and is populated with functions to manipulate slices of data from matrices, arrays, lists and dataframes in a repetitive way. These functions allow crossing the data in a number of ways and avoid explicit use of loop constructs. They act on an input list, matrix or array and apply a named function with one or several optional arguments.

The called function could be:

- An aggregating function, like for example the mean, or the sum (that return a number or scalar);
- Other transforming or subsetting functions; and
- Other vectorized functions, which yield more complex structures like lists, vectors, matrices, and arrays.

The `apply()` functions form the basis of more complex combinations and helps to perform operations with very few lines of code. More specifically, the family is made up of the `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()`, and `tapply()` functions.

But how and when should we use these?

Well, this depends on the structure of the data that you want to operate on and the format of the output that you need.

2.8.4.1 `apply()` Function

Let's start with the godfather of the family, `apply()`, which operates on arrays. For simplicity, we will use 2D arrays, which are also known as matrices.

The R base manual tells you that it's called as follows: `apply(X, MARGIN, FUN, ...)` where:

- `X` is an array or a matrix if the dimension of the array is 2;
- `MARGIN` is a variable defining how the function is applied: when `MARGIN=1`, it applies over rows, whereas with `MARGIN=2`, it works over columns. Note that when you use the construct `MARGIN=c(1,2)`, it applies to both rows and columns; and

- `FUN`, which is the function that you want to apply to the data. It can be any R function, including a User Defined Function (UDF).

Let's construct a 5 x 6 matrix and imagine you want to sum the values of each column.

```
# Construct a 5x6 matrix
X = matrix(rnorm(30), nrow=5, ncol=6)

# Sum the values of each column with `apply()`
apply(X, 2, sum)
```

```
## [1]  3.3068149  0.7576652 -1.7653971  4.3518894  2.3963091  2.3626304
```

Remember that in R, a matrix can be seen as a collection of line vectors when you cross the matrix from top to bottom (along the vertical line 1, which specifies the dimension or margin 1), or as a list of columns vectors, spanning the matrix left to right along the dimension or margin 2.

That means that the instruction you have just entered, depicted in figure 1, translates into: "apply the function 'sum' to the matrix X along margin 2 (by column), summing up the values of each column.

Note that, to avoid cluttering the picture, just one of the columns is highlighted.

```
# Construct a 5x6 matrix
X = matrix(rnorm(30), nrow=5, ncol=6)

# Sum the values of each column with `apply()`
apply(X, 2, sum)
```

```
## [1] -0.03206613 -1.80367548  1.05338074  0.33361302 -1.38118072  4.71834634
```

You end up with a line vector containing the sums of the values of each column.

The output of the above code, a line vector, would have also been given if you summed along the lines of the matrix. This is how R displays the result.

2.8.4.2 The `lapply()` Function

You want to apply a given function to every element of a list and obtain a list as a result. When you execute `?lapply`, you see that the syntax looks like the `apply()` function.

The difference is that:

1. It can be used for other objects like dataframes, lists or vectors; and
2. The output returned is a list (which explains the "l" in the function name), which has the same number of elements as the object passed to it.

To see how this works, create a few matrices and extract from each a given column.

This is a quite common operation performed on real data when making comparisons or aggregations from different dataframes.

Again, you start by specifying the object of interest, the list `Mylist.` You use the standard R selection operator `[` and then omit the first parameter (which therefore translates into "any", that's why you see the two commas).

Next, you specify the second parameter, which is 2: our margin is 'column'. So you extract the second column from all the matrices within the list.

In the right-hand side of figure 2, you can see an alternative extraction: this time you omit the first parameter, and you get the first row from each of the matrices. Our toy example, depicted in the last figure can be coded as:

```
# Create a list of matrices
A = matrix(1:9, nrow = 3)
B = matrix(4:15, nrow = 4)
C = cbind(c(8,9,10), c(8,9,10))

MyList = list(A,B,C)

# Extract the 2nd column from `MyList` with the selection operator `[` with `lapply()`
lapply(MyList,"[", , 2)
```

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

```
# Extract the 1st row from `MyList`
lapply(MyList,"[", 1, )
```

```
## [[1]]
## [1] 1 4 7
##
```

```
## [[2]]
## [1]  4  8 12
##
## [[3]]
## [1] 8 8
```

A few notes to the code above:

- The `[` notation is the select operator. Remember, for example, that to extract all the elements of the third line of B requires: `B[3,]`;
- The `[[ ]]` notation expresses the fact that we are dealing with lists: [[2]] means the second element of the list. This is also shown in the output given by R;
- The output is a list with as many elements as the element in the input; and
- Note that you could also have extracted a single element for each matrice, like this: `lapply(MyList,"[", 1, 2)`

2.8.4.3 The `sapply()` Function

The `sapply()` function works like `lapply()`, but it tries to simplify the output to the most elementary data structure that is possible. And indeed, `sapply()` is a 'wrapper' function for `lapply()`.

Essentially, `sapply()` calls `lapply()` on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If neither of the above simplifications can be performed then a list is returned

An example may help to understand this: let's say that you want to repeat the extraction operation of a single element as in the last example, but now take the first element of the second row (indexes 2 and 1) for each matrix.

Applying the `lapply()` function would give us a list unless you pass `simplify=FALSE` as a parameter to `sapply()`. Then, a list will be returned. See how it works in the code chunk below:

```
# Return a list with `lapply()`
lapply(MyList,"[", 2, 1 )
```

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

```
# Return a vector with `sapply()`
sapply(MyList,"[", 2, 1 )
```

```
## [1] 2 5 9
```

```
# Return a list with `sapply()`
sapply(MyList,"[", 2, 1, simplify=F)
```

```
## [1] 2 5 9
```

```
# Return a vector with `unlist()`
unlist(lapply(MyList,"[", 2, 1 ))
```

```
## [1] 2 5 9
```

2.8.5 `apply` Family Examples

Let's use the `mtcars` built-in dataset to see some examples:

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

In this datset we can appreciate some characteristics of different cars. Imagine we want to obtain the average of each feature. We can use the `apply` function for that purpose:

```
apply(mtcars, 2, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

We could also obtain the mean of each call in all the features, but it doesn't make sense at all. Let's generate a list of four items to provide an example:

```
data = list(item1 = 1:4,
            item2 = rnorm(10),
            item3 = rnorm(20,1),
            item4 = rnorm(100,5))
```

We can get the mean of each of the items we have in the list using the `lapply` function.

```
lapply(data, mean)
```

```
## $item1
## [1] 2.5
##
## $item2
## [1] 0.08497086
##
## $item3
## [1] 1.052272
##
## $item4
## [1] 4.877802
```

The last slide provides a simple example where each list item is simply a vector of numeric values. However, consider the case where you have a list that contains data frames and you would like to loop through each list item and perform a function to the data frame. In this case we can embed an `apply` function within an `lapply` function.

For example, the following creates a list for R's built in beaver data sets. The lapply function loops through each of the two list items and uses apply to calculate the mean of the columns in both list items.

```
beaver_data = list(beaver1 = beaver1,
                   beaver2 = beaver2)
```

```
lapply(beaver_data, function(x) apply(x, 2, mean))
```

```
## $beaver1
##          day         time         temp        activ
## 3.462018e+02 1.312018e+03 3.686219e+01 5.263158e-02
##
## $beaver2
##       day      time      temp     activ
##  307.1300 1446.2000   37.5967    0.6200
```

To illustrate the differences between `lapply` and `sapply` functions we can use the previous example using a list with the beaver data and compare the outputs:

```
lapply(beaver_data, function(x) round(apply(x, 2, mean), 2))
```

```
## $beaver1
##     day    time    temp   activ
##  346.20 1312.02   36.86    0.05
##
## $beaver2
##     day    time    temp   activ
##  307.13 1446.20   37.60    0.62
```

```
sapply(beaver_data, function(x) round(apply(x, 2, mean), 2))
```

```
##       beaver1 beaver2
## day    346.20  307.13
## time  1312.02 1446.20
## temp    36.86   37.60
## activ    0.05    0.62
```

Finally, I would like to show a `tapply` example. We didn't dig into this function because we had already seen it in previous sessions, but it also deserves it's own applied example.

Remember: `tapply()` is used to apply a function over subsets of a vector. It is primarily used when we have the following circumstances:

- A dataset that can be broken up into groups (via categorical variables - aka factors)

- We desire to break the dataset up into groups
- Within each group, we want to apply a function

The arguments to tapply() are as follows:

- `x` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed FUN
- `simplify`, should we simplify the result?

```
# syntax of tapply function
tapply(x, INDEX, FUN, ..., simplify = TRUE)
```

To provide an example we'll use the built in mtcars dataset and calculate the mean of the `mpg` variable grouped by the `cyl` variable.

```
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

Now let's say you want to calculate the mean for each column in the mtcars dataset grouped by the cylinder categorical variable. To do this you can embed the `tapply` function within the `apply` function.

```
apply(mtcars, 2, function(x) tapply(x, mtcars$cyl, mean))
```

```
##        mpg cyl     disp        hp     drat       wt     qsec        vs
## 4 26.66364   4 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909
## 6 19.74286   6 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286
## 8 15.10000   8 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000
##          am     gear     carb
## 4 0.7272727 4.090909 1.545455
## 6 0.4285714 3.857143 3.428571
## 8 0.1428571 3.285714 3.500000
```

2.9 The pipe operator

In practice we often have to call functions in a sequence. Suppose for example you have a vector of numbers. Of those numbers you would like to first compute the absolute value. Then you would like to compute the logarithm of those absolute values. Last you would like to compute the mean of those numbers. In standard R we can write this as

```
x <- -5:-1
mean(log(abs(x)))
```

```
## [1] 0.9574983
```

Such nested code where we apply multiple functions over the same line of code becomes cluttered and difficult to read.

For this reason the package `magrittr` introduces the so-called pipe operator `%>%` which makes the above code much more readable. Consider the same example using the pipe operator.

```
library(magrittr)
x <- -5:-1
x %>% abs() %>% log() %>% mean()
```

```
## [1] 0.9574983
```

The above code can be seen as follows: consider the vector `x` and apply the function `abs` over its entries. Then apply the function `log` over the resulting vector and last apply the function `mean`.

The code is equivalent to standard R but it is simpler to read. So sometimes it is preferrable to code using pipes instead of standard R syntax.

2.10 Plotting

R has great plotting capabilities. Details about plotting functions and a discussion of when different representations are most appropriate are beyond the scope of these notes. This is just to provide you with a list of functions:

- `barplot` creates a barplot: notice that you first need to construct a so-called contingency table using the function `table`.
- `hist` creates an histogram;
- `boxplot` creates a boxplot;
- `plot` creates a scatterplot;

There are many functions to customize such plots, and again details can be found in the references given. A package which is often used to create nice data visualization is `ggplot2`.

Chapter 3

Probability Basics

Our final aim is to be able to mimic real-world systems as close as possible. In most scenarios we will not know with certainty how things unfold. For instance, we will rarely know the times at which customers enter a shop or the time it will take an employee to complete a task. Let's think again at the donut shop example. The time it takes an employee to serve a costumer depends on the time it takes the customer to specify the order, the number and types of donuts requested, the type of payment etc. To an external observer all these possible causes of variation of serving times appear to be random and due to chance: they cannot be predicted with certainty.

For this reason we will in general assume a probabilistic model for the various components of a simulation. This chapter gives a review of possible models as well as their characteristics.

3.1 Discrete Random Variables

We start introducing discrete random variables. Here we will not enter in all the mathematical details and some concepts will be introduced only intuitively. However, some mathematical details will be given for some concepts.

In order to introduce discrete random variables, let's inspect each of the words:

- *variable*: this means that there is some process that takes some value. It is a synonym of function as you have studied in other mathematics classes.
- *random*: this means that the variable takes values according to some probability distribution.
- *discrete*: this refers to the possible values that the variable can take. In this case it is a countable (possibly infinite) set of values.

In general we denote a random variable as X and its possible values as $\mathbb{X} = \{x_1, x_2, x_3, ...\}$. The set $\mathbb{X}$ is called the sample space of X. In real-life we do not know which value in the set $\mathbb{X}$ the random variable will take.

Let's consider some examples.

- The number of donuts sold in a day in a shop is a discrete random variable which can take values $\{0, 1, 2, 3, ...\}$, that is the non-negative integers. In this case the number of possible values is infinite.
- The outcome of a COVID-19 test can be either positive or negative but in advance we do not know which one. So this can be denoted as a discrete random variable taking values in $\{negative, positive\}$. It is customary to denote the elements of the sample space $\mathbb{X}$ as numbers. For instance, we could let $negative = 0$ and $positive = 1$ and the sample space would be $\mathbb{X} = \{0, 1\}$.
- The number shown on the face of a dice once thrown is a discrete random variable taking values in $\mathbb{X} = \{1, 2, 3, 4, 5, 6\}$.

3.1.1 Probability Mass Function

The outcome of a discrete random variable is in general unknown, but we want to associate to each outcome, that is to each element of $\mathbb{X}$, a number describing its likelihood. Such a number is called a probability and it is in general denoted as P.

The *probability mass function* (or pmf) of a random variable X with sample space $\mathbb{X}$ is defined as

$$p(x) = P(X = x), \qquad \text{for all } x \in \mathbb{X}$$

So for any outcome $x \in \mathbb{X}$ the pmf describes the likelihood of that outcome happening.

Recall that pmfs must obey two conditions:

- $p(x) \geq 0$ for all $x \in \mathbb{X}$;
- $\sum_{x \in \mathbb{X}} p(x) = 1$.

So the pmf associated to each outcome is a non-negative number such that the sum of all these numbers is equal to one.

Let's consider an example at this stage. Suppose a biased dice is thrown such that the numbers 3 and 6 are twice as likely to appear than the other numbers. A pmf describing such a situation is the following:

x	1	2	3	4	5	6
$p(x)$	1/8	1/8	2/8	1/8	1/8	2/8

It is apparent that all numbers $p(x)$ are non-negative and that their sum is equal to 1: so $p(x)$ is a pmf. Figure 3.1 gives a graphical visualization of such a pmf.

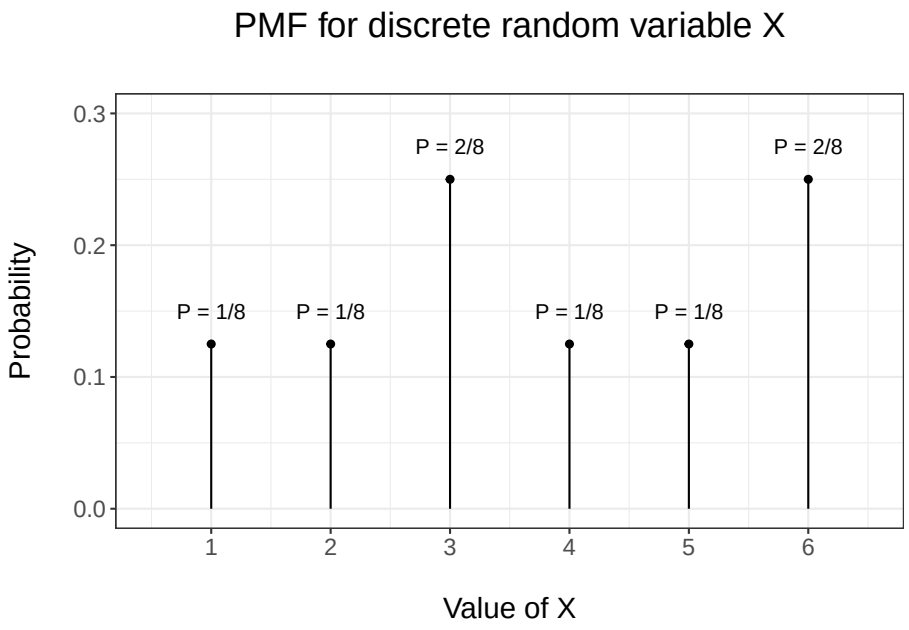


Figure 3.1: PMF for the biased dice example

3.1.2 Cumulative Distribution Function

Whilst you should have been already familiar with the concept of pmf, the next concept may appear to be new. However, you have actually used it multiple times when computing Normal probabilities with the tables.

We now define what is usually called the *cumulative distribution function* (or cdf) of a random variable X. The cdf of X at the point $x \in \mathbb{X}$ is

$$F(x) = P(X \leq x) = \sum_{y \leq x} p(y)$$

that is the probability that X is less or equal to x or equally the sum of the pmf of X for all values less than x.

Let's consider the dice example to illustrate the idea of cdf and consider the following values x:

- $x = 0$: we compute $F(0) = P(X \leq 0) = 0$ since X cannot take any values less or equal than zero;
- $x = 0.9$: we compute $F(0.9) = P(X \leq 0.9) = 0$ using the same reasoning as before;
- $x = 1$: we compute $F(1) = P(X \leq 1) = P(X = 1) = 1/8$ since X can take the value 1 with probability $1/8$;
- $x = 1.5$: we compute $F(1.5) = P(X \leq 1.5) = P(X = 1) = 1/8$ using the same reasoning as before;
- $x = 3.2$: we compute $F(3.2) = P(X \leq 3.2) = P(X = 1) + P(X = 2) + P(X = 3) = 1/8 + 1/8 + 2/8 = 0.5$ since X can take the values 1, 2 and 3 which are less than 3.2;

We can compute in a similar way the cdf for any value x. A graphical visualization of the resulting CDF is given in Figure 3.2.

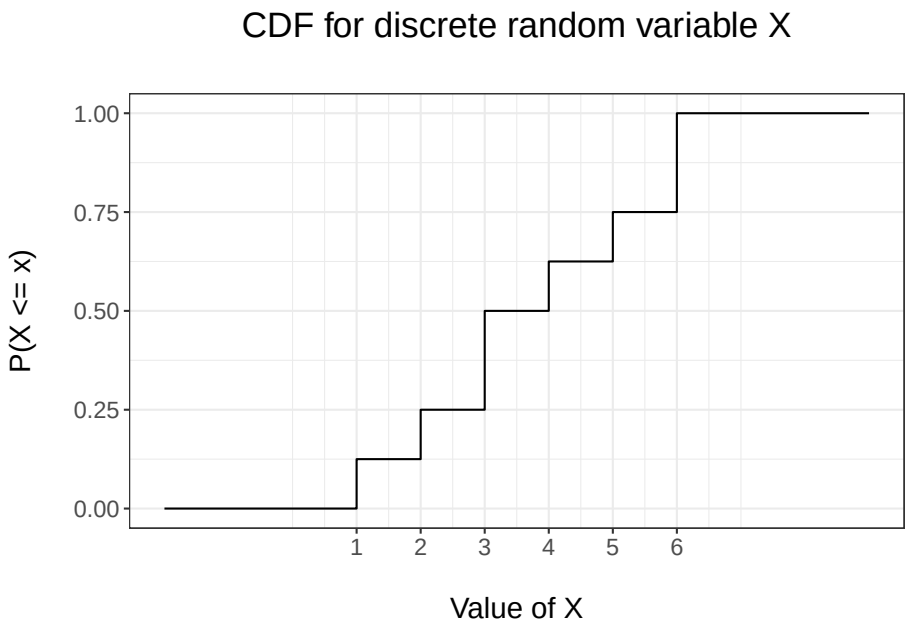


Figure 3.2: CDF for the biased dice example

The plot highlights some properties of CDFs which can proved hold in general for any discrete CDF:

- it is a step function which is also non-decreasing;

- on the left-hand-side it takes the value 0;
- on the right-hand-side it takes the value 1.

3.1.3 Summaries

The pmf and the cdf fully characterize a discrete random variable X. Often however we want to compress that information into a single number which still retains some aspect of the distribution of X.

The *expectation* or *mean* of a random variable X denoted as $E(X)$ is defined as

$$E(X) = \sum_{x \in \mathbb{X}} xp(x)$$

The expectation can be interpreted as the mean value of a large number of observations from the random variable X. Consider again the example of the biased dice. The expectation is

$$E(X) = 1 \cdot 1/8 + 2 \cdot 1/8 + 3 \cdot 2/8 + 4 \cdot 1/8 + 5 \cdot 1/8 + 6 \cdot 2/8 = 3.75$$

So if we were to throw the dice a large number of time, we would expect the average of the number shown to be 3.75.

The *median* of the random variable X denoted as $m(X)$ is defined as the value x such that $P(X \leq x)$ is larger or equal to 0.5 and $P(X \geq x)$ is larger or equal to 0.5. It is defined as the middle value of the distribution. For the dice example the median is the value 3: indeed $P(X \leq 3) = P(X = 1) + P(X = 2) + P(X = 3) = 0.5 \geq 0.5$ and $P(X \geq 3) = P(X = 3) + P(X = 4) + P(X = 5) + P(X = 6) = 0.75 \geq 0.5$.

The *mode* of the random variable X is the value x such that $p(x)$ is largest: it is the value of the random variable which is expected to happen most frequently. Notice that the mode may not be unique: in that case we say that the distribution of X is bimodal. The example of the biased dice as an instance of a bimodal distribution: the values 3 and 6 are the equally likely and they have the largest pmf.

The above three summaries are measures of *centrality*: they describe the central tendency of X. Next we consider measures of *variability*: such measures will quantify the spread or the variation of the possible values of X around the mean.

The *variance* of the discrete random variable X is the expectation of the squared difference between the random variable and its mean. Formally it is defined as

$$V(X) = E((X - E(X))^2) = \sum_{x \in \mathbb{X}} (x - E(X))^2 p(x)$$

In general we will not compute variance by hand. The following R code computes the variance of the random variable associated to the biased dice.

```
x <- 1:6  # outcomes of X
px <- c(1/8,1/8,2/8,1/8,1/8,2/8)  # pmf of X
Ex <- sum(x*px)  # Expectation of X
Vx <- sum((x-Ex)^2*px)  # Variance of X
Vx
```

```
## [1] 2.9375
```

The *standard deviation* of the discrete random variable X is the square root of $V(X)$.

3.2 Notable Discrete Variables

In the previous section we gave a generic definition of discrete random variables and discussed the conditions that a pmf must obey. We considered the example of a biased dice and constructed a pmf for that specific example.

There are situations that often happen in practice: for instance the case of experiments with binary outcomes. For such cases random variables with specific pmfs are given a name and their properties are well known and studied.

In this section we will consider three such distributions: Bernoulli, Binomial and Poisson.

3.2.1 Bernoulli Distribution

Consider an experiment or a real-world system where there can only be two outcomes:

- a toss of a coin: heads or tails;
- the result of a COVID test: positive or negative;
- the status of a machine: broken or working;

By default one outcome happens with some probability, that we denote as $\theta \in [0, 1]$ and the other with probability $1 - \theta$.

Such a situation is in general modeled using the so-called *Bernoulli distribution* with parameter θ. One outcome is associated to the number 1 (usually referred to as sucess) and the other is associated to the number 0 (usually referred to as failure). So $P(X = 1) = p(1) = \theta$ and $P(X = 0) = p(0) = 1 - \theta$.

The above pmf can be more coincisely written as

$$p(x) = \begin{cases} \theta^x(1-\theta)^{1-x}, & x = 0, 1 \\ 0, & \text{otherwise} \end{cases}$$

The mean and variance of the Bernoulli distribution can be easily computed as

$$E(X) = 0 \cdot (1-\theta) + 1 \cdot \theta = \theta,$$

and

$$V(X) = (0-\theta)^2(1-\theta) + (1-\theta)^2\theta = \theta^2(1-\theta) + (1-\theta)^2\theta = \cdots = \theta(1-\theta)$$

Figure 3.3 reports the pmf and the cdf of a Bernoulli random variable with parameter 0.3.

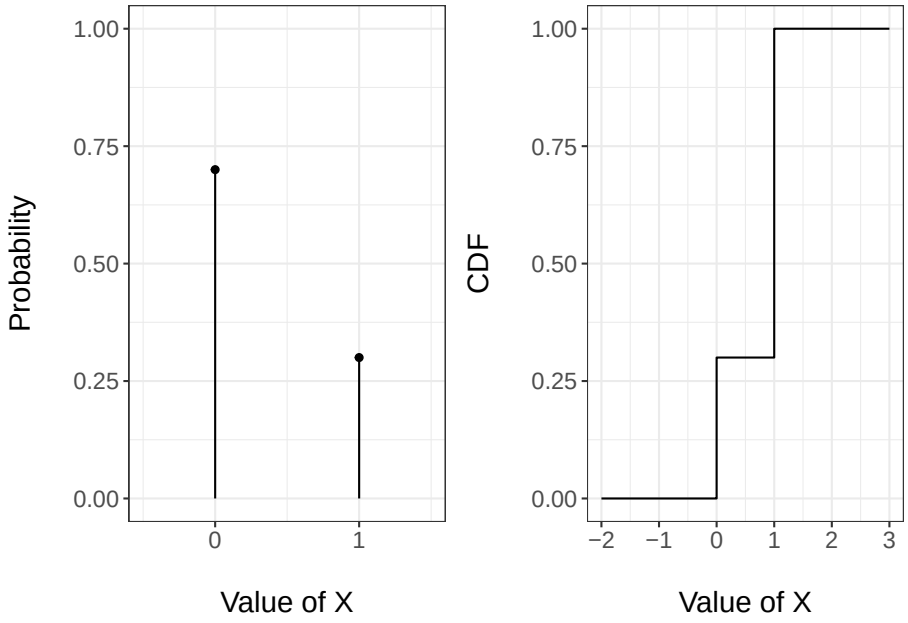


Figure 3.3: PMF (left) and CDF (right) of a Bernoulli random variable with parameter 0.3

3.2.2 Binomial Distribution

The Bernoulli random variable is actually a very special case of the so-called *Binomial* random variable. Consider experiments of the type discussed for Bernoullis: coin tosses, COVID tests etc. Now suppose that instead of having just one trial, each of these experiments are repeated multiple times. Consider the following assumptions:

- each experiment is repeated n times;
- each time there is a probability of success θ;
- the outcome of each experiment is independent of the others.

Let's think of tossing a coin n times. Then we would expect that the probability of showing heads is the same for all tosses and that the result of previous tosses does not affect others. So this situation appears to meet the above assumptions and can be modeled by what we call a Binomial random variable.

Formally, the random variable X is a Binomial random variable with parameters n and θ if it denotes the number of successes of n independent Bernoulli random variables, all with parameter θ.

The pmf of a Binomial random variable with parameters n and θ can be written as:

$$p(x) = \begin{cases} \binom{n}{x}\theta^x(1-\theta)^{n-x}, & x = 0, 1, \dots, n \\ 0, & \text{otherwise} \end{cases}$$

Let's try and understand the formula by looking term by term.

- if $X = x$ there are x successes and each success has probability θ - so θ^x counts the overall probability of successes
- if $X = x$ there are $n - x$ failures and each failure has probability $1 - \theta$ - so $(1 - \theta)^{n-x}$ counts the overall probability of failures
- failures and successes can appear according to many orders. To see this, suppose that $x = 1$: there is only one success out of n trials. This could have been the first attempt, the second attempt or the n-th attempt. The term $\binom{n}{x}$ counts all possible ways the outcome x could have happened.

The Bernoulli distribution can be seen as a special case of the Binomial where the parameter n is fixed to 1.

We will not show why this is the case but the expectation and the variance of the Binomial random variable with parameters n and θ can be derived as

$$E(X) = np, \qquad V(X) = np(1-p)$$

The formulae for the Bernoulli can be retrieved by setting $n = 1$.

Figure 3.4 shows the pmf of two Binomial distributions both with parameter $n = 10$ and with $\theta = 0.3$ (left) and $\theta = 0.8$. For the case $\theta = 0.3$ we can see that it is more likely that there are a small number of successes, whilst for $\theta = 0.8$ a large number of successes is more likely.

R provides a straightforward implementation of the Binomial distribution through the functions `dbinom` for the pmf and `pbinom` for the cdf. They require three arguments:

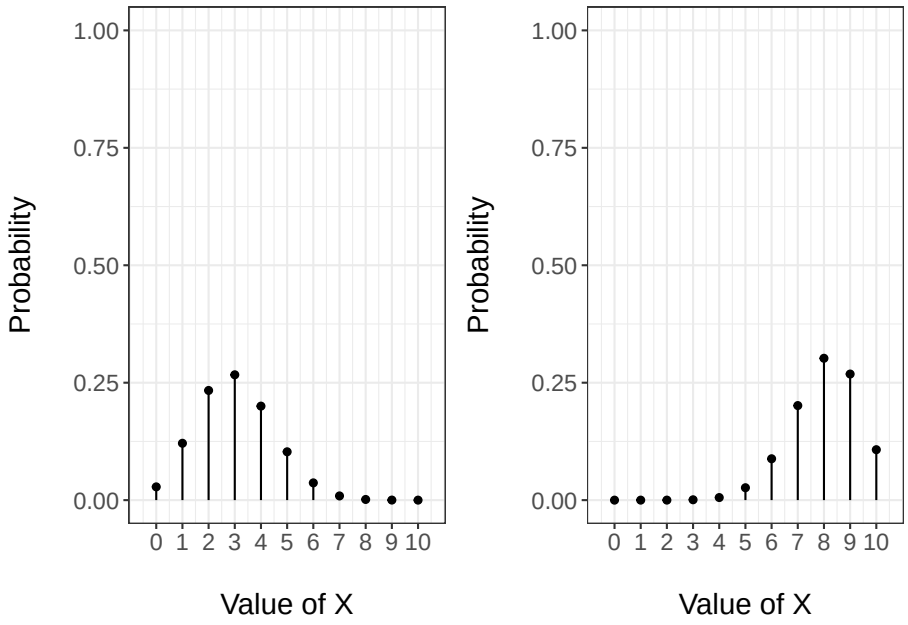


Figure 3.4: PMF of a Binomial random variable with parameters n = 10 and theta = 0.3 (left) and theta = 0.8 (right)

- first argument is the value at which to compute the pmf or the cdf;
- `size` is the parameter n of the Binomial;
- `prob` is the parameter θ of the Binomial.

So for instance

```
dbinom(3, size = 10, prob = 0.5)
```

```
## [1] 0.1171875
```

returns $P(X = 3) = p(3)$ for a Binomial random variable with parameter $n = 10$ and $\theta = 0.5$.

Similarly,

```
pbinom(8, size = 20, prob = 0.2)
```

```
## [1] 0.9900182
```

returns $P(X \leq 8) = F(8)$ for a Binomial random variable with parameter $n = 20$ and $\theta = 0.2$.

3.2.3 Poisson Distribution

The last class of discrete random variables we discuss is the so-called *Poisson* distribution. Whilst for Bernoulli and Binomial we had an interpretation of why the pmf took its specific form by associating it to independent binary experiments each with an equal probability of success, for the Poisson there is no such an interpretation.

A discrete random variable X has a Poisson distribution with parameter λ if its pmf is

$$p(x) = \begin{cases} \frac{e^{-\lambda}\lambda^x}{x!}, & x = 0, 1, 2, 3, ... \\ 0, & \text{otherwise} \end{cases}$$

where $\lambda > 0$.

So the sample space of a Poisson random variable is the set of all non-negative integers.

One important characteristic of the Poisson distribution is that its mean and variance are equal to the parameter λ, that is

$$E(X) = V(X) = \lambda.$$

Figure @ref{fig:poisson} gives an illustration of the form of the pmf of the Poisson distribution for two parameter choices: $\lambda = 1$ (left) and $\lambda = 4$ (right). The x-axis is shown until $x = 10$ but recall that the Poisson is defined over all non-negative integers. For the case $\lambda = 1$ we can see that the outcomes 0 and 1 have the largest probability - recall that $E(X) = 0$. For the case $\lambda = 4$ the outcomes $x = 2, 3, 4, 5$ have the largest probability.

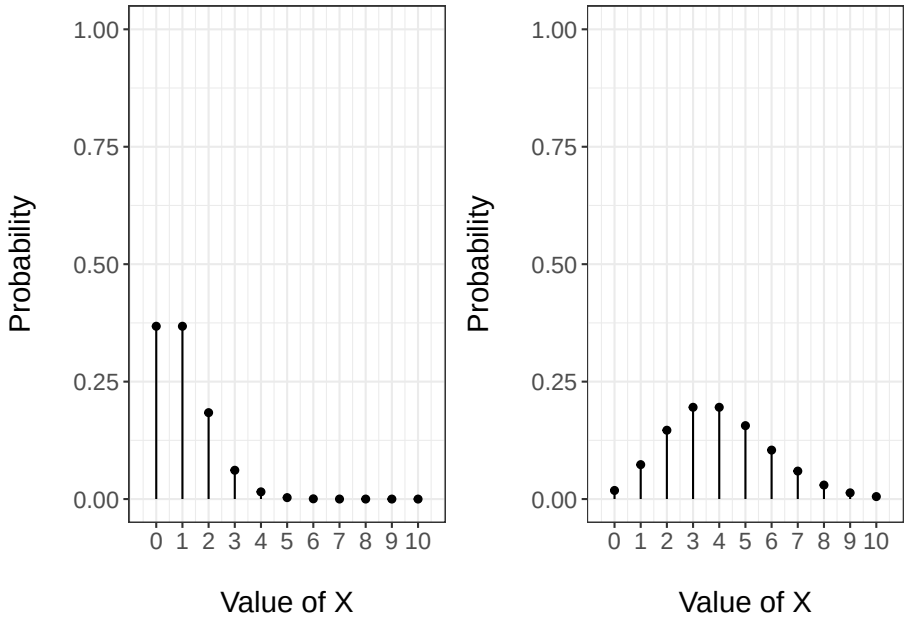


Figure 3.5: PMF of a Poisson random variable with parameter 1 (left) and 4 (right)

R provides a straightforward implementation of the Poisson distribution through the functions `dpois` for the pmf and `ppois` for the cdf. They require three arguments:

- first argument is the value at which to compute the pmf or the cdf;
- `lambda` is the parameter λ of the Poisson;

So for instance

```
dpois(3, lambda = 1)
```

```
## [1] 0.06131324
```

returns $P(X = 3) = p(3)$ for a Poisson random variable with parameter $\lambda = 1$.

Similarly,

```
ppois(8, lambda = 4)
```

```
## [1] 0.9786366
```

returns $P(X \leq 8) = F(8)$ for a Poisson random variable with parameter $\lambda = 4$.

3.2.4 Some Examples

We next consider two examples to see in practice the use of the Binomial and Poisson distributions.

3.2.4.1 Probability of Marriage

A recent survey indicated that 82% of single women aged 25 years old will be married in their lifetime. Compute

- the probability of at most 3 women will be married in a sample of 20;
- the probability of at least 90 women will be married in sample of 100;
- the probability of two or three women in a sample of 20 will never be married.

The above situation can be modeled by a Binomial random variable where the parameter n depends on the question and $\theta = 0.82$.

The first question requires us to compute $P(X \leq 3) = F(3)$ where X is Binomial with parameters $n = 20$ and $\theta = 0.82$. Using R

```
pbinom(3, size = 10, prob = 0.82)
```

```
## [1] 0.0004400767
```

The second question requires us to compute $P(X \geq 90)$ where X is a Binomial random variable with parameters $n = 100$ and $\theta = 0.82$. Notice that

$$P(X \geq 90) = 1 - P(X < 90) = 1 - P(X \leq 89) = 1 - F(89).$$

Using R

```
1 - pbinom(89, size = 100,  prob = 0.82)
```

```
## [1] 0.02003866
```

For the third question, notice that saying two women out of 20 will never be married is equal to 18 out of 20 will be married. Therefore we need to compute $P(X = 17)+P(X = 18) = p(17)+p(18)$ where X is a Binomial random variable with parameters $n = 20$ and $\theta = 0.82$. Using R

```
sum(dbinom(17:18, size = 20, prob = 0.82))
```

```
## [1] 0.4007631
```

3.2.4.2 The Bad Stuntman

A stuntman injures himself an average of three times a year. Use the Poisson probability formula to calculate the probability that he will be injured:

- 4 times a year
- Less than twice this year.
- More than three times this year.

The above situation can be modeled as a Poisson distribution X with parameter $\lambda = 3$.

The first question requires us to compute $P(X = 4)$ which using R can be computed as

```
dpois(4, lambda =3)
```

```
## [1] 0.1680314
```

The second question requires us to compute $P(X < 2) = P(X = 0) + P(X = 1) = F(1)$ which using R can be computed as

```
ppois(1,lambda=3)
```

```
## [1] 0.1991483
```

The third question requires us to compute $P(X > 3) = 1-P(X \leq 2) = 1-F(2)$ which using R can be computed as

```
1 - ppois(2, lambda = 3)
```

```
## [1] 0.5768099
```

3.3 Continuous Random Variables

Our attention now turns to continuous random variables. These are in general more technical and less intuitive than discrete ones. You should not worry about all the technical details, since these are in general not important, and focus on the interpretation.

A continuous random variable X is a random variable whose sample space $\mathbb{X}$ is an interval or a collection of intervals. In general $\mathbb{X}$ may coincide with the set of real numbers $\mathbb{R}$ or some subset of it. Examples of continuous random variables:

- the pressure of a tire of a car: it can be any positive real number;
- the current temperature in the city of Madrid: it can be any real number;
- the height of the students of Simulation and Modeling to understand change: it can be any real number.

Whilst for discrete random variables we considered summations over the elements of $\mathbb{X}$, i.e. $\sum_{x \in \mathbb{X}}$, for continuous random variables we need to consider integrals over appropriate intervals.

You should be more or less familiar with these from previous studies of calculus. But let's give an example. Consider the function $f(x) = x^2$ computing the squared of a number x. Suppose we are interested in this function between the values -1 and 1, which is plotted by the red line in Figure 3.6. Consider the so-called integral $\int_{-1}^{1} x^2 dx$: this coincides with the area delimited by the function and the x-axis. In Figure 3.6 the blue area is therefore equal to $\int_{-1}^{1} x^2 dx$.

```
ggplot(data = data.frame(x = seq(-1,1,0.01), y = seq(-1,1,0.01)^2), aes(x,y)) + geom_li
```

We will not be interested in computing integrals ourselves, so if you do not know/remember how to do it, there is no problem!

3.3.1 Probability Density Function

Discrete random variable are easy to work with in the sense that there exists a function, that we called probability mass function, such that $p(x) = P(X = x)$,

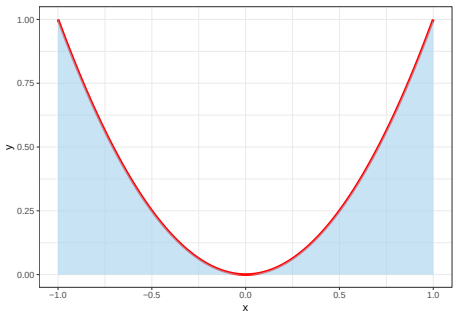


Figure 3.6: Plot of the squared function and the area under its curve

that is the value of that function in the point x is exactly the probability that $X = x$.

Therefore we may wonder if this is true for a continuous random variable too. Sadly, the answer is no and probabilities for continuous random variables are defined in a slightly more involved way.

Let X be a continuous random variable with sample space $\mathbb{X}$. The probability that X takes values in the interval $[a, b]$ is given by

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

where $f(x)$ is called the *probability density function* (pdf in short). Pdfs, just like pmfs must obey two conditions:

- $f(x) \geq 0$ for all $x \in \mathbb{X}$;
- $\int_{x \in \mathbb{X}} f(x)dx = 1$.

So in the discrete case the pmf is defined exactly as the probability. In the continuous case the pdf is the function such that its integral is the probability that random variable takes values in a specific interval.

As a consequence of this definition notice that for any specific value $x_0 \in \mathbb{X}$, $P(X = x_0) = 0$ since

$$\int_{x_0}^{x_0} f(x)dx = 0.$$

Let's consider an example. The waiting time of customers of a donuts shop is believed to be random and to follow a random variable whose pdf is

$$f(x) = \begin{cases} \frac{1}{4}e^{-x/4}, & x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The pdf is drawn in Figure 3.7 by the red line. One can see that $f(x) \geq 0$ for all $x \geq 0$ and one could also compute that it integrates to one.

Therefore the probability that the waiting time is between any two values (a, b) can be computed as

$$\int_a^b \frac{1}{4} e^{-x/4} dx.$$

In particular if we were interested in the probability that the waiting time is between two and five minutes, corresponding to the shaded area in Figure 3.7, we could compute it as

$$P(2 < X < 5) = \int_2^5 f(x)dx = \int_2^5 \frac{1}{4} e^{-x/4} dx = 0.32$$

```
data <- data.frame(x=seq(0.01,15,0.05),y=dexp(seq(0.01,15,0.05),1/4))
```

```
## Warning: Removed 240 rows containing missing values (position_stack).
```

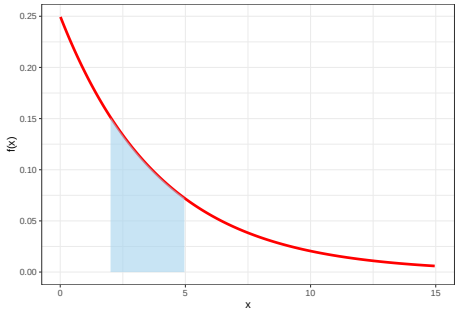


Figure 3.7: Probability density function for the waiting time in the donut shop example

Notice that since $P(X = x_0) = 0$ for any $x_0 \in \mathbb{X}$, we also have that

$$P(a \leq X \leq b) = P(a < X \leq b) = P(a \leq X < b) = P(a < X < b).$$

3.3.2 Cumulative Distribution Function

For a continuous random variable X the cumulative distribution function (cdf) is equally defined as

$$F(x) = P(X \leq x),$$

where now

$$P(X \leq x) = P(X < x) = \int_{-\infty}^{x} f(t)dt.$$

so the summation is substituted by an integral.

Let's consider again the donut shop example as an illustration. The cdf is defined as

$$F(x) = \int_{-\infty}^{x} f(t)dt = \int_{-\infty}^{x} \frac{1}{4}e^{-x/4}.$$

This integral can be solved and $F(x)$ can be calculated as

$$F(x) = 1 - e^{-x/4},$$

which is plotted in Figure 3.8.

```
x <- seq(-2,20,0.5)
```

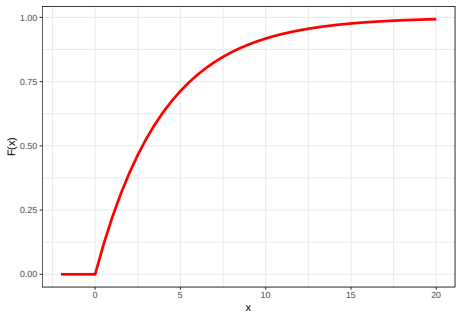


Figure 3.8: Cumulative distribution function for the waiting time at the donut shop

We can notice that the cdf has similar properties as in the discrete case: it is non-decreasing, on the left-hand side is zero and on the right-hand side tends to zero.

In the continuous case, one can prove that cdfs and pdfs are related as

$$f(x) = \frac{d}{dx}F(x).$$

3.3.3 Summaries

Just as for discrete random variables, we may want to summarize some features of a continuous random variable into a unique number. The same set of summaries exists for continuous random variables, which are almost exactly defined as in the discrete case (integrals are used instead of summations).

- *mean*: the mean of a continuous random variable X is defined as

$$E(X) = \int_{-\infty}^{+\infty} xf(x)dx$$

- *median*: the median of a continuous random variable X is defined as the value x such that $P(X \leq x) = 0.5$ or equally $F(x) = 0.5$.
- *mode*: the mode of a continuous random variable X is defined the value x such that $f(x)$ is largest.
- *variance*: the variance of a continuous random variable X is defined as

$$V(X) = \int_{-\infty}^{+\infty} (x - E(X))^2 f(x) dx$$

- *standard deviation*: the standard deviation of a continuous random variable X is defined as $\sqrt{V(X)}$.

3.4 Notable Continuous Distribution

As in the discrete case, there are some types of continuous random variables that are used frequently and therefore are given a name and their proprieties are well-studied.

3.4.1 Uniform Distribution

The first, and simplest, continuous random variable we study is the so-called (continuous) *uniform* distribution. We say that a random variable X is uniformly distributed on the interval $[a, b]$ if its pdf is

$$f(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b \\ 0, & \text{otherwise} \end{cases}$$

This is plotted in Figure 3.9 for choices of parameters $a = 2$ and $b = 6$

```
x <- c(seq(-1,2,0.01),seq(2,6,0.01),seq(6,9,0.01))
```

By looking at the pdf we see that it is a flat, constant line between the values a and b. This implies that the probability that X takes values between two values x_0 and x_1 only dependens on the length of the interval (x_0, x_1).

Its cdf can be derived as

$$F(x) = \begin{cases} 0, & x < a \\ \frac{x-a}{b-a}, & a \leq x \leq b \\ 1, & x > b \end{cases}$$

and this is plotted in Figure 3.10.

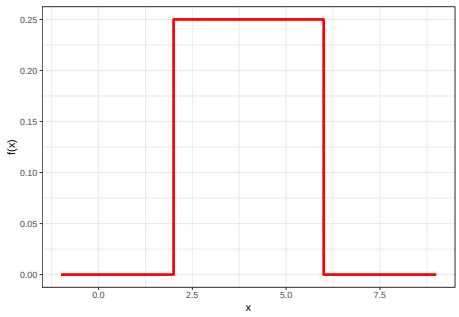


Figure 3.9: Probability density function for a uniform random variable with parameters a = 2 and b = 6

```
x <- c(seq(-1,2,0.01),seq(2,6,0.01),seq(6,9,0.01))
```

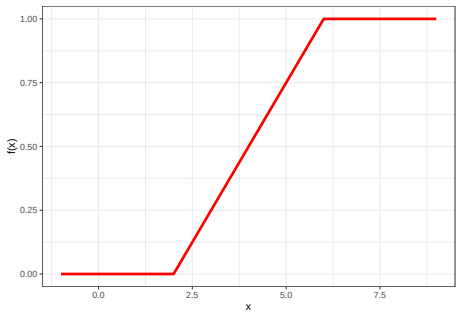


Figure 3.10: Cumulative distribution function for a uniform random variable with parameters a = 2 and b = 6

The mean and variance of a uniform can be derived as

$$E(X) = \frac{a+b}{2}, \qquad V(X) = \frac{(b-a)^2}{12}.$$

So the mean is equal to the middle point of the interval (a, b).

The uniform distribution will be fundamental in simulation. We will see that the starting point to simulate random numbers from any distribution will require the simulation of random numbers uniformly distributed between 0 and 1.

R provides an implementation of the uniform random variable with the functions `dunif` and `punif` whose details are as follows:

- the first argument is the value at which to compute the function;

- the second argument, `min`, is the parameter a, by default equal to zero;
- the third argument, `max`, is the parameter b, by default equal to one.

So for instance

```
dunif(5, min = 2, max = 6)
```

```
## [1] 0.25
```

computes the pdf at the point 5 of a uniform random variable with parameters $a = 2$ and $b = 6$.

Conversely,

```
punif(0.5)
```

```
## [1] 0.5
```

computes the cdf at the point 0.5 of a uniform random variable with parameters $a = 0$ and $b = 1$.

3.4.2 Exponential Distribution

The second class of continuous random variables we will study are the so-called *exponential* random variables. We have actually already seen such a random variable in the donut shop example. More generally, we say that a continuous random variable X is exponential with parameter $\lambda > 0$ if its pdf is

$$f(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Figure 3.11 reports the pdf of exponential random variables for various choices of the parameter λ.

```
values <- c(0.25,1,2.5)
```

Exponential random variables are very often used in dynamic simulations since they are very often used to model interarrival times in process: for instance the time between arrivals of customers at the donut shop.

Its cdf can be derived as

$$F(x) = \begin{cases} 0, & x < 0 \\ 1 - e^{-\lambda x}, & x \geq 0 \end{cases}$$

and is reported in Figure 3.12 for the same choices of parameters.

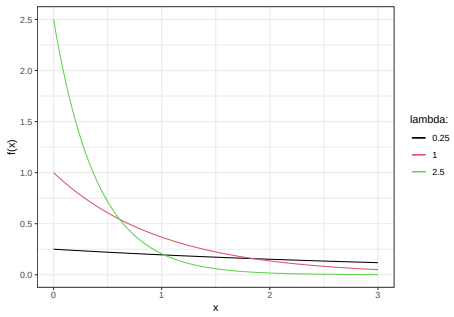


Figure 3.11: Probability density function for exponential random variables

```
values <- c(0.25,1,2.5)
```

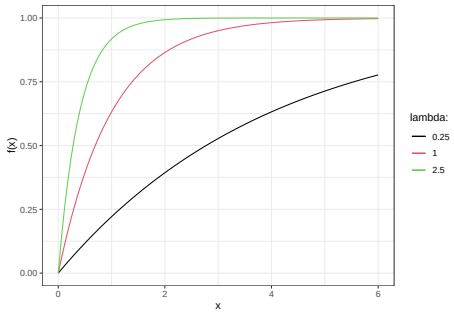


Figure 3.12: Cumulative distribution function for exponential random variables

The mean and the variance of exponential random variables can be computed as

$$E(X) = \frac{1}{\lambda}, \qquad V(X) = \frac{1}{\lambda^2}$$

R provides an implementation of the uniform random variable with the functions `dexp` and `pexp` whose details are as follows:

- the first argument is the value at which to compute the function;
- the second argument, `rate`, is the parameter λ, by default equal to one;

So for instance

```
dexp(2, rate = 3)
```

```
## [1] 0.007436257
```

computes the pdf at the point 2 of an exponential random variable with parameter $\lambda = 3$.

Conversely

```
pexp(4)
```

```
## [1] 0.9816844
```

computes the cdf at the point 4 of an exponential random variable with parameter $\lambda = 1$.

3.4.3 Normal Distribution

The last class of continuous random variables we consider is the so-called *Normal* or *Gaussian* random variable. They are the most used and well-known random variable in statistics and we will see why this is the case.

A continuous random variable X is said to have a Normal distribution with mean μ and variance σ^2 if its pdf is

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

Recall that

$$E(X) = \mu, \qquad V(X) = \sigma^2,$$

and so the parameters have a straightforward interpretation in terms of mean and variance.

Figure 3.13 shows the form of the pdf of the Normal distribution for various choices of the parameters. On the left we have Normal pdfs for $\sigma^2 = 1$ and various choices of μ: we can see that μ shifts the plot on the x-axis. On the right we have Normal pdfs for $\mu = 1$ and various choices of σ^2: we can see that all distributions are centered around the same value while they have a different spread/variability.

```
mu <- c(0,1,2)
```

The form of the Normal pdf is the well-known so-called bell-shaped function. We can notice some properties:

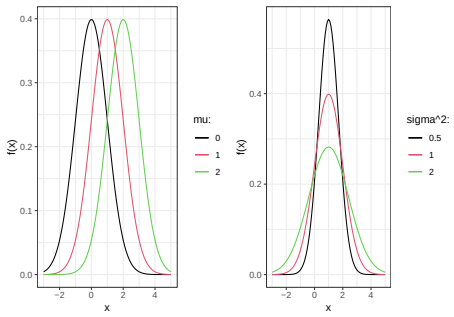


Figure 3.13: Probability density function for normal random variables

- it is symmetric around the mean: the function on the left-hand side and on the right-hand side of the mean is mirrored. This implies that the median is equal to the mean ;
- the maximum value of the pdf occurs at the mean. This implies that the mode is equal to the mean (and therefore also the median).

The cdf of the Normal random variable with parameters μ and σ^2 is

$$F(x) = P(X \leq x) = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) dx$$

The cdf of the Normal for various choices of parameters is reported in Figure 3.14.

```
mu <- c(0,1,2)
```

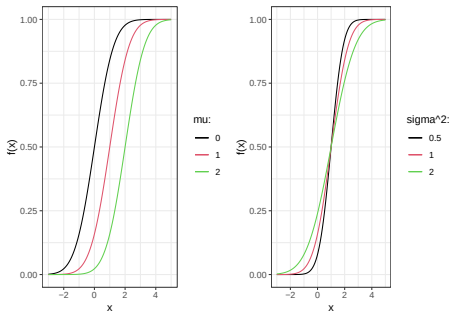


Figure 3.14: Cumulative distribution function for normal random variables

Unfortunately it is not possible to solve such an integral (as for example for the Uniform and the Exponential), and in general it is approximated using

some numerical techniques. This is surprising considering that the Normal distribution is so widely used!!!

However, notice that we would need to compute such an approximation for every possible value of (μ, σ^2), depending on the distribution we want to use. This is unfeasible to do in practice.

There is a trick here, that you must have used multiple times already. We can transform a Normal X with parameters μ and σ^2 to the so-called *standard Normal* random variable Z, and viceversa, using the relationship:

$$Z = \frac{X - \mu}{\sigma}, \qquad X = \mu + \sigma Z. \tag{3.1}$$

It can be shown that Z is a Normal random variable with parameter $\mu = 0$ and $\sigma^2 = 1$.

The values of the cdf of the standard Normal random variable then need to be computed only once since μ and σ^2 are fixed. You have seen these numbers many many times in what are usually called the tables of the Normal distribution.

As a matter of fact you have also computed many times the cdf of a generic Normal random variable. First you computed Z using equation (3.1) and then looked at the Normal tables to derive that number.

Let's give some details about the standard Normal. Its pdf is

$$\phi(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-z^2/2\right).$$

It can be seen that it is the same as the one of the Normal by setting $\mu = 0$ and $\sigma^2 = 1$. Such a function is so important that it is given its own symbol ϕ.

The cdf is

$$\Phi(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} \exp\left(-x^2/2\right) dx$$

Again this cannot be computed exactly, there is no closed-form expression. This is why you had to look at the tables instead of using a simple formula. The cdf of the standard Normal is also so important that it is given its own symbol ϕ.

Instead of using the tables, we can use R to tell us the values of Normal probabilities. R provides an implementation of the Normal random variable with the functions `dnorm` and `pnorm` whose details are as follows:

- the first argument is the value at which to compute the function;
- the second argument, `mean`, is the parameter μ, by default equal to zero;
- the third argument, `sd`, is the standard deviation, that is $\sqrt{\sigma^2}$, by default equal to one.

So for instance

```
dnorm(3)
```

```
## [1] 0.004431848
```

computes the value of the standard Normal pdf at the value three.

Similarly,

```
pnorm(0.4,1,0.5)
```

```
## [1] 0.1150697
```

compute the value of the Normal cdf with parameters $\mu = 1$ and $\sqrt{\sigma^2} = 0.5$ at the value 0.4.

3.5 The Central Limit Theorem

As a final topic in probability we will briefly discuss why the Normal distribution is so important and widely known. The reason behind this is the existence of a theorem, called the *Central Limit Theorem* which is perhaps the most important theorem in probability which has far-reaching consequences in the world of statistics.

Let's first state theorem. Suppose you have random variables $X_1, \dots, X_n$ which have the following properties:

- they are all independent of each other;
- they all have the same mean μ;
- the all have the same standard deviation σ^2.

Consider the random variable

$$\bar{X}_n = \frac{X_1 + \cdots X_n}{n}.$$

Then it holds that

$$\lim_{n \to +\infty} \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} = Z$$

where Z is the standard normal random variable.

We can also state the theorem as

$$\lim_{n \to +\infty} \bar{X}_n = Y$$

where Y is a Normal random variable with mean μ and variance σ^2/n.

The interpretation of the Central Limit Theorem is as follows. The sample mean $\bar{X}_n$ of independent random variables with the same mean and variance can be approximated by a Normal distribution, if the sample size n is large. Notice that we made no assumption whatsoever about the distribution of the X_i's and still we were able to deduce the distribution of the sample mean.

The existence of this theorem is the reason why you used so often Normal probabilities to construct confidence intervals or to carry out tests of hypothesis. As you will continue study statistics, you will see that the assumption of Normality of data is made most often and is justified by the central limit theorem.

Chapter 4

Random Number Generation

At the hearth of any simulation model there is the capability of creating numbers that mimic those we would expect in real life. In simulation modeling we will assume that specific processes will be distributed according to a specific random variable. For instance we will assume that an employee in a donut shop takes a random time to serve customers distributed according to a Normal random variable with mean μ and variance σ^2. In order to then carry out a simulation the computer will need to generate random serving times. This corresponds to simulating number that are distributed according to a specific distribution.

Let's consider an example. Suppose you managed to generate two sequences of numbers, say `x1` and `x2`. Your objective is to simulate numbers from a Normal distribution. The histograms of the two sequences are reported in Figure 4.1 together with the estimated shape of the density. Clearly the sequence `x1` could be following a Normal distribution, since it is bell-shaped and reasonably symmetric. On the other hand, the sequence `x2` is not symmetric at all and does not resembles the density of a Normal.

```
p <- ggplot(data.frame(x1=rnorm(700)), aes(x=x1))+
   geom_histogram(aes(y=..density..), colour="black", fill="white")+
 geom_density(alpha=.2, fill="#FF6666")  + theme_bw()
```

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

In this chapter we will learn how to characterize randomness in a computer and how to generate numbers that appear to be random realizations of a specific random variable. We will also learn how to check if a sequence of values can be a random realization from a specific random variable.

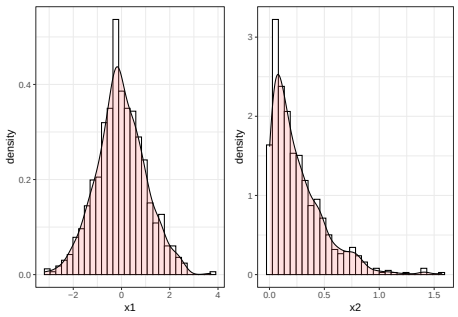


Figure 4.1: Histograms of two sequences of randomly generated numbers

4.1 Properties of Random Numbers

The first step to simulate numbers from a distribution is to be able to independently simulate random numbers $u_1, u_2, \ldots, u_N$ from a continuous uniform distribution between zero and one. From the previous chapter, you should remember that such a random variables has pdf

$$f(x) = \begin{cases} 1, & 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

and cdf

$$F(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x \leq 1 \\ 1, & \text{otherwise} \end{cases}$$

These two are plotted in Figure 4.2.

```
x <- c(seq(-1,0,0.01),seq(0,1,0.01),seq(1,2,0.01))
```

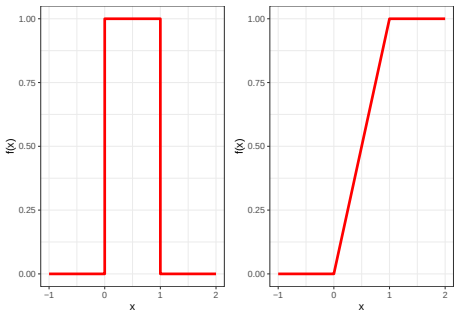


Figure 4.2: Pdf (left) and cdf (right) of the continuous uniform between zero and one.

Its expectation is $1/2$ and its variance is $1/12$.

This implies that if we were to divide the interval $[0, 1]$ into n sub-intervals of equal length, then we would expect in each interval to have N/n observations, where N is the total number of observations.

Figure 4.3 shows the histograms of two sequences of numbers between zero and one: whilst the one on the left resembles the pdf of a uniform distribution, the one on the right clearly does not (it is far from being flat) and therefore it is hard to believe that such numbers follow a uniform distribution.

```
p <- ggplot(data.frame(x1=runif(10000)), aes(x=x1))+
   geom_histogram(aes(y=..density..),binwidth=0.1,center = 0.05, colour="black", fill="white")  +
```

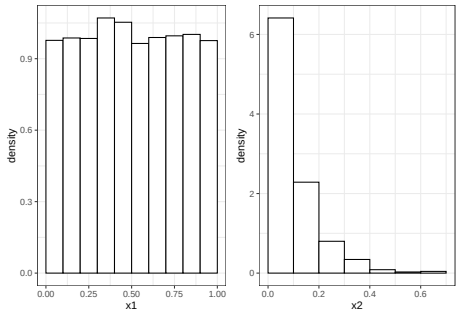


Figure 4.3: Histograms from two sequences of numbers between zero and one.

The second requirement the numbers $u_1, \ldots, u_N$ need to respect is independence. This means that the probability of observing a value in a particular sub-interval of $(0, 1)$ is independent of the previous values drawn.

Consider the following sequence of numbers:

$$0.25 \quad 0.72 \quad 0.18 \quad 0.63 \quad 0.49 \quad 0.88 \quad 0.23 \quad 0.78 \quad 0.02 \quad 0.52$$

We can notice that numbers below and above 0.5 are alternating in the sequence. We would therefore believe that after a number less than 0.5 it is much more likely to observe a number above it. This breaks the assumption of independence.

4.2 Pseudo Random Numbers

We will investigate ways to simulate numbers using algorithms in a computer. For this reason such numbers are usually called *pseudo-random* numbers. Pseudo means false, in the sense that the number are not really random! They

are generated according to a deterministic algorithm whose aim is to imitate as closely as possible what randomness would look like. In particular, for numbers $u_1, \ldots, u_N$ it means that they should look like independence instances of a Uniform distribution between zero and one.

Possible departures from ideal numbers are:

- the numbers are not uniformly distributed;
- the mean of the numbers might not be 1/2;
- the variance of the numbers might not be 1/12;
- the numbers might be discrete-valued instead of continuous;
- independence might not hold.

We already looked at examples of departures from the assumptions, but we will later study how to assess these departures more formally.

Before looking at how we can construct pseudo-random numbers, let's discuss some important properties/considerations that need to be taken into account when generating pseudo-random numbers:

- the random generation should be very fast. In practice, we want to use random numbers to do other computations (for example simulate a little donut shop) and such computations might be computationally intensive: if random generation were to be slow, we would not be able to perform them.
- the cycle of random generated numbers should be long. The cycle is the length of the sequence before numbers start to repeat themselves.
- the random numbers should be repeatable. Given a starting point of the algorithm, it should be possible to repeat the exact same sequence of numbers. This is fundamental for debugging and for reproducibility.
- the method should be applicable in any programming language/platform.
- and of course most importantly, the random numbers should be independent and uniformly distributed.

Repeatability of the pseudo-random numbers is worth further consideration. It is fundamental in science to be able to reproduce experiments so that the validity of results can be assessed. In R there is a specific function that allows us to do this, which is called `set.seed`. It is customary to choose as starting point of an algorithm the current year. So henceforth you will see the command:

```
set.seed(2021)
```

This ensures that every time the code following `set.seed` is run, the same results will be observed. We will give below examples of this.

4.3 Generating Pseudo-Random Numbers

The literature on generating pseudo-random numbers is now extremely vast and it is not our purpose to review it, neither for you to learn how such algorithms work.

4.3.1 Generating Pseudo-Random Numbers in R

R has all the capabilities to generate such numbers. This can be done with the function `runif`, which takes one input: the number of observations to generate. So for instance:

```
set.seed(2021)
runif(10)
```

```
##  [1] 0.4512674 0.7837798 0.7096822 0.3817443 0.6363238 0.7013460 0.6404389
##  [8] 0.2666797 0.8154215 0.9829869
```

generates ten random numbers between zero and one. Notice that if we repeat the same code we get the same result since we fixed the so-called *seed* of the simulation.

```
set.seed(2021)
runif(10)
```

```
##  [1] 0.4512674 0.7837798 0.7096822 0.3817443 0.6363238 0.7013460 0.6404389
##  [8] 0.2666797 0.8154215 0.9829869
```

Conversely, if we were to simply run the code `runif(10)` we would get a different result.

```
runif(10)
```

```
##  [1] 0.02726706 0.83749040 0.60324073 0.56745337 0.82005281 0.25157128
##  [7] 0.50549403 0.86753810 0.95818157 0.54569770
```

4.3.2 Linear Congruential Method

Although we will use the functions already implemented in R, it is useful to at least introduce one of the most classical algorithms to simulate random numbers, called the *linear congruential method*. This produces a sequence of integers x_1, x_2, x_3 between 0 and $m-1$ using the recursion:

$$x_i = (ax_{i-1} + c) \mod m, \qquad \text{for } i = 1, 2, ...$$

Some comments:

- $\mod m$ is the remainder of the integer division by m. For instance 5 mod 2 is one and 4 mod 2 is zero.
- therefore, the algorithm generates integers between 0 and $m-1$.
- there are three parameters that need to be chosen a, c and m.
- the value x_0 is the *seed* of the algorithm.

Random numbers between zero and one can be derived by setting

$$u_i = x_i/m.$$

It can be shown that the method works well for specific choices of a, c and m, which we will not discuss here.

Let's look at an implementation.

```
lcm <- function(N, x0, a, c, m){
   x <- rep(0,N)
   x[1] <- x0
   for (i in 2:N) x[i] <- (a*x[i-1]+c)%%m
   u <- x/m
   return(u)
}
```

```
lcm(N = 8, x0 = 4, a = 13, c = 0, m = 64)
```

```
## [1] 0.0625 0.8125 0.5625 0.3125 0.0625 0.8125 0.5625 0.3125
```

We can see that this specific choice of parameters is quite bad: it has cycle 4! After 4 numbers the sequence repeats itself and we surely would not like to use this in practice.

In general you should not worry of these issues, R does things properly for you!

4.4 Testing Randomness

Now we turn to the following question: given a sequence of numbers $u_1, \ldots, u_N$ how can we test if they are independent realizations of a Uniform random variable between zero and one?

We therefore need to check if the distribution of the numbers is uniform and if they are actually independent.

4.4.1 Testing Uniformity

A simple first method to check if the numbers are uniform is to create an histogram of the data and to see if the histogram is reasonably flat. We already saw how to assess this, but let's check if `runif` works well. Simple histograms can be created in `R` using `hist` (or if you want you can use `ggplot`).

```
set.seed(2021)
u <- runif(5000)
hist(u)
```

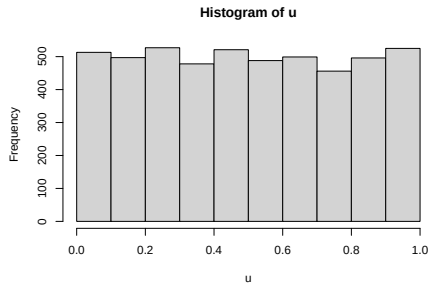


Figure 4.4: Histogram of a sequence of uniform numbers

We can see that the histogram is reasonably flat and therefore the assumption of uniformity seems to hold.

Although the histogram is quite informative, it is not a fairly formal method. We could on the other hand look at tests of hypotheses of this form:

$$H_0 : \ u_i \text{ is uniform between zero and one, } i = 1, 2, \ldots$$
$$H_a : \ u_i \text{ is not uniform between zero and one, } i = 1, 2, \ldots$$

The null hypothesis is thus that the numbers are indeed uniform, whilst the alternative states that the numbers are not. If we reject the null hypothesis,

which happens if the p-value of the test is very small (or smaller than a critical value α of our choice), then we would believe that the sequence of numbers is not uniform.

There are various ways to carry out such a test, but we will consider here only one: the so-called *Kolmogorov-Smirnov Test*. We will not give all details of this test, but only its interpretation and implementation.

In order to understand how the test works we need to briefly introduce the concept of *empirical cumulative distribution function* or *ecdf*. The ecdf $\hat{F}$ is the cumulative distribution function computed from a sequence of N numbers as

$$\hat{F}(t) = \frac{\text{numbers in the sequence } \leq t}{N}$$

Let's consider the following example.

```
data <- data.frame(u= c(0.1,0.2,0.4,0.8,0.9))
ggplot(data,aes(u)) + stat_ecdf(geom = "step") + theme_bw() + ylab("ECDF")
```

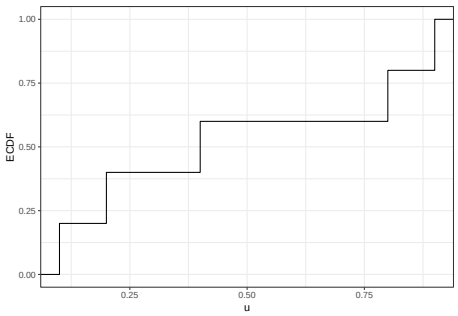


Figure 4.5: Ecdf of a simple sequence of numbers

For instance, since there are 3 numbers out of 5 in the vector `u` that are less than 0.7, then $\hat{F}(0.7) = 3/5$.

The idea behind the Kolmogorov-Smirnov test is to quantify how similar the ecdf computed from a sequence of data is to the one of the uniform distribution which is represented by a straight line (see Figure 4.2).

As an example consider Figure 4.6. The step functions are computed from two different sequences of numbers between one and zero, whilst the straight line is the cdf of the uniform distribution. By looking at the plots, we would more strongly believe that the sequence in the left plot is uniformly distributed, since the step function is much more closer to the theoretical straight line.

```
set.seed(2021)
```

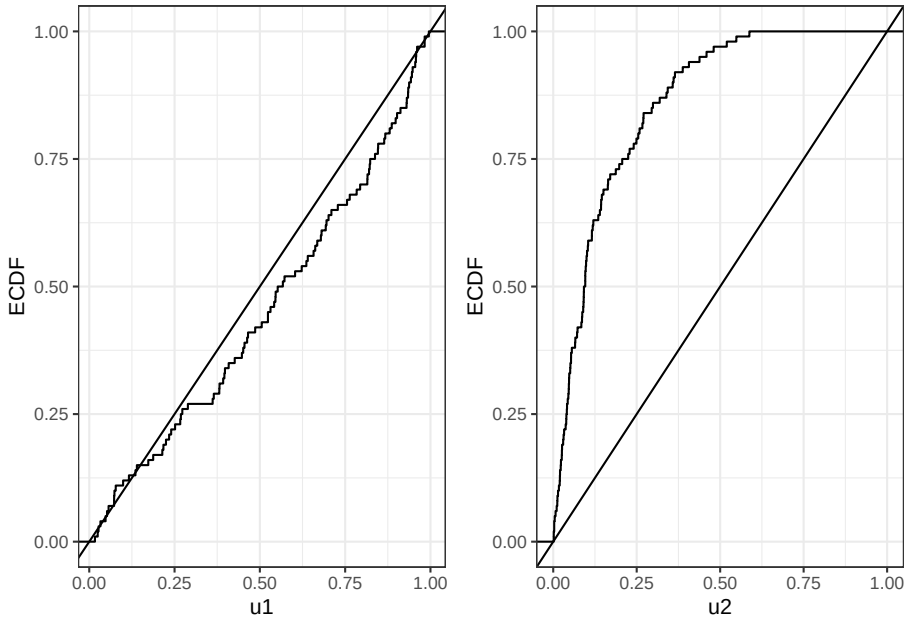


Figure 4.6: Comparison between ecdf and cdf of the uniform for two sequences of numbers

The Kolmogorov-Smirnov test formally embeds this idea of similarity between the ecdf and the cdf of the uniform in a test of hypothesis. The function `ks.test` implements this test in R. For the two sequences in Figure @ref{fig:Kol} `u1` (left plot) and `u2` (right plot), the test can be implemented as following:

```
ks.test(u1,"punif")
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  u1
## D = 0.11499, p-value = 0.142
## alternative hypothesis: two-sided
```

```
ks.test(u2,"punif")
```

```
##
```

```
##  One-sample Kolmogorov-Smirnov test
##
## data:  u2
## D = 0.56939, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

From the results that the p-value of the test for the sequence `u1` is 0.142 and so we would not reject the null hypothesis that the sequence is uniformly distributed. On the other hand the p-value for the test over the sequence `u2` has an extremely small p-value therefore suggesting that we reject the null hypothesis and conclude that the sequence is not uniformly distributed. This confirms our intuition by looking at the plots in Figure 4.6.

4.4.2 Testing Independence

The second requirement that a sequence of pseudo-random numbers must have is independence. We already saw an example of when this might happen: a high number was followed by a low number and vice versa.

We will consider tests of the form:

$$H_0 : u_1, \ldots, u_N \text{ are independent}$$
$$H_a : u_1, \ldots, u_N \text{ are not independent}$$

So the null hypothesis is that the sequence is of independent numbers against the alternative that they are not. If the p-value of such a test is small we would then reject the null-hypothesis of independence.

In order to devise such a test, we need to come up with a way to quantify how dependent numbers in a sequence are with each other. Again, there are many ways one could do this, but we consider here only one.

You should already be familiar with the idea of *correlation*: this tells you how to variables are linearly dependent of each other. There is a similar idea which extends in a way correlation to the case when it is computed between a sequence of numbers and itself, which is called *autocorrelation*. We will not give the details about this, but just the interpretation (you will learn a lot more about this in Time Series).

Let's briefly recall the idea behind correlation. Suppose you have two sequences of numbers $u_1, \ldots, u_N$ and $w_1, \ldots, w_N$. To compute the correlation you would look at the pairs $(u_1, w_1), (u_2, w_2), \ldots, (u_N, w_N)$ and assess how related the numbers within a pair (u_i, w_i) are. Correlation, which is a number between -1 and 1, assesses how related the sequences are: the closer the number is to one in absolute value, the stronger the relationship.

Now however we have just one sequence of numbers $u_1, \dots, u_N$. So for instance we could look at pairs $(u_1, u_2), (u_2, u_3), \dots, (u_{N-1}, u_N)$ which consider two consecutive numbers and compute their correlation. Similarly we could compute the correlation between $(u_1, u_{1+k}), (u_2, u_{2+k}), \dots, (u_{N-k}, u_N)$ between each number in the sequence and the one k-positions ahead. This is what we call *autocorrelation of lag k*.

If autocorrelations of various lags are close to zero, this gives an indication that the data is independent. If on the other hand, the autocorrelation at some lags is large, then there is an indication of dependence in the sequence of random numbers. Autocorrelations are computed and plotted in `R` using `acf` and reported in Figure 4.7.

```
set.seed(2021)
u1 <- runif(200)
acf(u1)
```

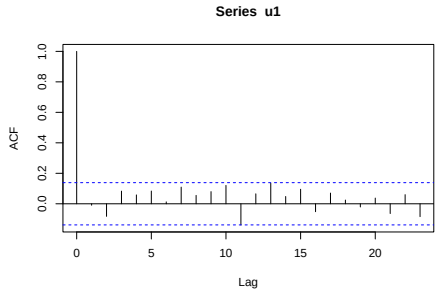


Figure 4.7: Autocorrelations for a sequence of random uniform numbers

The bars in Figure 4.7 are the autocorrelations at various lags, whilst the dashed blue lines are confidence bands: if a bar is within the bands it means that we cannot reject the hypothesis that the autocorrelation of the associated lag is equal to zero. Notice that the first bar is lag 0: it computes the correlation for the sample $(u_1, u_1), (u_2, u_2), \dots, (u_N, u_N)$ and therefore it is always equal to one. You should never worry about this bar. Since all the bars are within the confidence bands, we believe that all autocorrelations are not different from zero and consequently that the data is independent (it was indeed generated using 'runif").

Figure 4.8 reports the autocorrelations of a sequence of numbers which is not independent. Although the histogram shows that the data is uniformly distributed, we would not believe that the sequence is of independent numbers since autocorrelations are very large and outside the bands.

```
u2 <- rep(1:10,each = 4,times = 10)/10 + rnorm(400,0,0.02)
u2 <- (u2 - min(u2))/(max(u2)-min(u2))
par(mfrow=c(1,2))
hist(u2)
acf(u2)
```

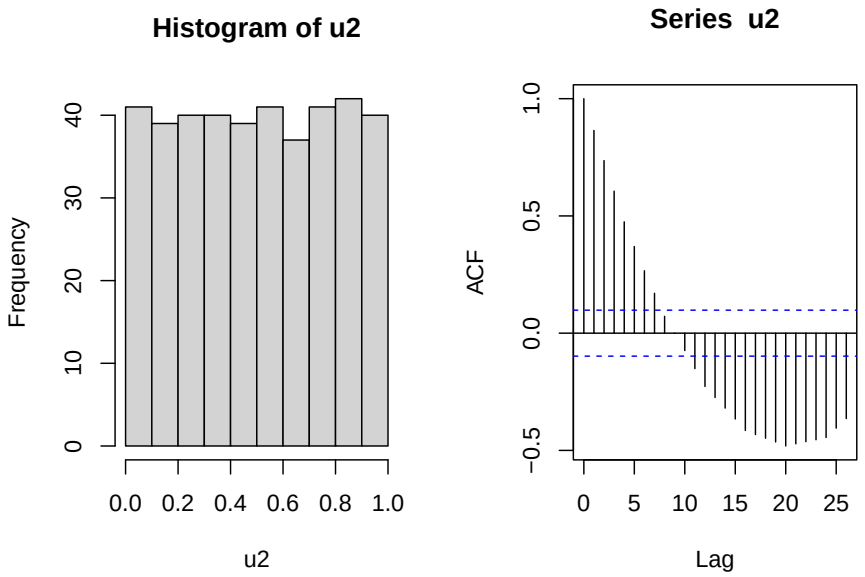


Figure 4.8: Histogram and autocorrelations of a sequence of uniform numbers which are not independent

A test of hypothesis for independence can be created by checking if any of the autocorrelations up to a specific lag are different from zero. This is implemented in the function `Box.test` in R. The first input is the sequence of numbers to consider, the second is the largest lag we want to consider. Let's compute it for the two sequences `u1` and `u2` above.

```
Box.test(u1, lag = 5)
```

```
##
##  Box-Pierce test
##
## data:  u1
## X-squared = 4.8518, df = 5, p-value = 0.4342
```

```
Box.test(u2, lag = 5)
```

```
##
##  Box-Pierce test
##
## data:  u2
## X-squared = 807.1, df = 5, p-value < 2.2e-16
```

Here we chose a lag up to 5 (it is usually not useful to consider larger lags). The test confirms our observations of the autocorrelations. For the sequence `u1` generated with `runif` the test has a high p-value and therefore we cannot reject the hypothesis of independence. For the second sequence `u2` which had very large autocorrelations the p-value is very small and therefore we reject the hypothesis of independence.

4.5 Random Variate Generation

Up to this point we have investigated how to generate numbers between 0 and 1 and how to assess the quality of those randomly generated numbers.

For simulation models we want to be more generally able to simulate observations that appear to be realizations of random variables with known distributions. We now study how this can be done. But before this, let's see how R implements random variate generation.

4.5.1 Random Generation in R

In the next few sections we will learn results that allow for the simulation of random observations from generic distributions. No matter how the methods work, they have a very simple and straightforward implementation in R.

We have already learned that we can simulate observations from the uniform between zero and one using the code `runif(N)` where `N` is the number of observations to simulate. We can notice that it is similar to the commands `dunif` and `punif` we have already seen for the pdf and cdf of the uniform.

Not surprisingly we can generate observations from any random variable using the syntax `r` followed by the naming of the variable chosen. So for instance:

- `runif` generates random observations from the Uniform;
- `rnorm` generates random observations from the Normal;
- `rexp` generates random observations from the Exponential;

- `rbinom` generates random observations from the Binomial;
- `rpois` generates random observations from the Poisson;

Each of these functions takes as first input the number of observations that we want to simulate. They then have additional inputs that can be given, which depend on the random variable chosen and are the same that we saw in the past.

So for instance

```
rnorm(10, mean = 1, sd = 2)
```

generates ten observations from a Normal distribution with mean 1 and standard deviation 2.

4.5.2 The Inverse Transform Method

The simplest method to simulate observations from generic random variables is the so-called *inverse transform method*.

Suppose we are interested in a random variable X whose cdf is F. We must assume that F is:

- known and written in closed-form;
- continuous;
- strictly increasing.

Then one can prove that

$$X = F^{-1}(U),$$

where F^{-1} is the inverse of F and U is a continuous uniform distribution between zero and one.

The above results gives us the following algorithm to simulate observations from a random variable X with distribution F:

1. compute the inverse F^{-1} of F;
2. generate independent random observations $u_1, u_2, \dots, u_N$ from a Uniform between zero and one;
3. compute $x_1 = F^{-1}(u_1), x_2 = F^{-1}(u_2), \dots, x_N = F^{-1}(u_n)$.

Then $x_1, x_2, \ldots, x_N$ are independent random observations of the random variable X.

So the above algorithm can be applied to generate observations from continuous random variables with a cdf in closed-form. Therefore, for instance the above algorithm cannot be straightforwardly used for Normals. However it can be used to simulate Exponential and Uniform distributions. Furthermore, it cannot be directly used to simulate discrete random variables. A simple adaptation of this method, which we will not see here, can however be used for discrete random variables.

4.5.2.1 Simulating Exponentials

Recall that if X is Exponential with parameter λ then its cdf is

$$F(x) = 1 - e^{-\lambda x}, \qquad \text{for } x \geq 0$$

Suppose we want to simulate observations $x_1, \ldots, x_N$ from such a distribution.

1. First we need to compute the inverse of F. This means solving the equation:
$$1 - e^{-\lambda x} = u$$
for x. This can be done following the steps:
$$\begin{aligned} 1 - e^{-\lambda x} &= u \\ e^{-\lambda x} &= 1 - u \\ -\lambda x &= \log(1-u) \\ x &= -\frac{1}{\lambda}\log(1-u) \end{aligned}$$
So $F^{-1}(u) = -\log(1-u)/\lambda$.
2. Second we need to simulate random uniform observations using `runif`.
3. Last, we apply the inverse function to the randomly simulated observations.

Let's give the R code.

```
set.seed(2021)
# Define inverse function
invF <- function(u,lambda) -log(1-u)/lambda
# Simulate 5 uniform observations
u <- runif(5)
# Compute the inverse
invF(u, lambda = 2)
```

```
## [1] 0.3000720 0.7657289 0.6183896 0.2404265 0.5057456
```

First we defined the inverse function of an Exponential with parameter `lambda` in `invF`. Then we simulated five observations from a uniform. Last we applied the function `invF` for a parameter `lambda` equal to two. The output are therefore five observations from an Exponential random variable with parameter 2.

4.5.2.2 Simulating Generic Uniforms

We know how to simulate uniformly between 0 and 1, but we do not know how to simulate uniformly between two generic values a and b.

Recall that the cdf of the uniform distribution between a and b is

$$F(x) = \frac{x-a}{b-a}, \qquad \text{for } a \leq x \leq b$$

The inverse transform method requires the inverse of F, which using simple algebra can be computed as

$$F^{-1}(u) = a + (b-a)u$$

So given a sequence $u_1, \dots, u_N$ of random observations from a Uniform between 0 and 1, we can simulate numbers at uniform between a and b by computing

$$x_1 = a + (b-a)u_1, \dots, x_N = a + (b-a)u_N$$

In R:

```
set.seed(2021)
a <- 2
b <- 6
a + (b-a)*runif(5)
```

```
## [1] 3.805069 5.135119 4.838729 3.526977 4.545295
```

The code simulates five observations from a Uniform between two and six. This can be equally achieved by simply using:

```
set.seed(2021)
runif(5, min = 2, max = 6)
```

```
## [1] 3.805069 5.135119 4.838729 3.526977 4.545295
```

Notice that since we fixed the seed, the two methods return exactly the same sequence of numbers.

4.5.3 Simulating Bernoulli and Binomial

Bernoulli random variables represent binary experiments with a probability of success equal to θ. A simple simulation algorithm to simulate one Bernoulli observation is:

1. generate u uniformly between zero and one;
2. if $u < \theta$ set $x = 0$, otherwise $x = 1$.

We will not prove that this actually works, but it intuitively does. Let's code it in R.

```
set.seed(2021)
theta <- 0.5
u <- runif(5)
x <- ifelse(u < theta, 0, 1)
x
```

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

So here we simulated five observations from a Bernoulli with parameter 0.5: the toss of a fair coin. Three times the coin showed head, and twice tails.

From this comment, it is easy to see how to simulate one observation from a Binomial: by simply summing the randomly generated observations from Bernoullis. So if we were to sum the five numbers above, we would get one random observations from a Binomial with parameter $n = 5$ and $\theta = 0.5$.

4.5.4 Simulating Other Distributions

There are many other algorithms that allow to simulate specific as well as generic random variables. Since these are a bit more technical we will not consider them here, but it is important for you to know that we now can simulate basically any random variable you are interested in!

4.6 Testing Generic Simulation Sequences

In previous sections we spent a lot of effort in assesing if a sequence of numbers could have been a random sequence of independent numbers from a Uniform distribution between zero and one.

Now we will look at the same question, but considering generic distributions we might be interested in. Recall that we had to check two aspects:

1. if the random sequence had the same distribution as the theoretical one (in previous sections Uniform between zero and one);
2. if the sequence was of independent numbers

We will see that the tools to perform these steps are basically the same.

4.6.1 Testing Distribution Fit

There are various ways to check if the random sequence of observations has the same distribution as the theoretical one.

4.6.1.1 Histogram

First, we could construct an histogram of the data sequence and compare it to the theoretical distribution. Suppose we have a sequence of numbers `x1` that we want to assess if it simulated from a Standard Normal distribution.

```
x1 <- data.frame(x1 =rnorm(500))
```

```
ggplot(x1, aes(x1)) +
   geom_line(aes(y = ..density.., colour = 'Empirical'), stat = 'density') +
   stat_function(fun = dnorm, aes(colour = 'Normal')) +
   geom_histogram(aes(y = ..density..), alpha = 0.4) +
   scale_colour_manual(name = 'Density', values = c('red', 'blue')) +
   theme_bw()
```

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

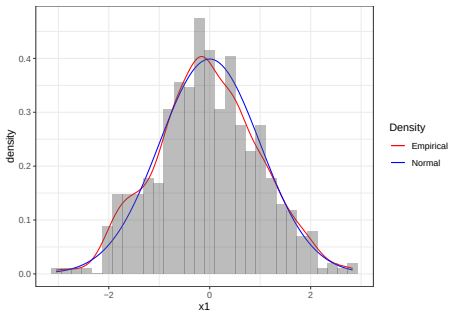


Figure 4.9: Histogram of the sequence x1 together with theoretical pdf of the standard Normal

Figure 4.9 reports the histogram of the sequence `x1` together with a smooth estimate of the histogram, often called density plot, in the red line. The blue line denotes the theoretical pdf of the standard Normal distribution. We can see that the sequence seems to follow quite closely a Normal distribution and therefore we could be convinced that the numbers are indeed Normal.

Let's consider a different sequence `x2`. Figure 4.10 clearly shows that there is a poor fit between the sequence and the standard Normal distribution. So we would in general not believe that these observations came from a Standard Normal.

```
x2 <- data.frame(x2 =rnorm(500,0.5,0.6))
```

```
ggplot(x2, aes(x2)) +
   geom_line(aes(y = ..density.., colour = 'Empirical'), stat = 'density') +
   stat_function(fun = dnorm, aes(colour = 'Normal')) +
   geom_histogram(aes(y = ..density..), alpha = 0.4) +
   scale_colour_manual(name = 'Density', values = c('red', 'blue')) +
   theme_bw()
```

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

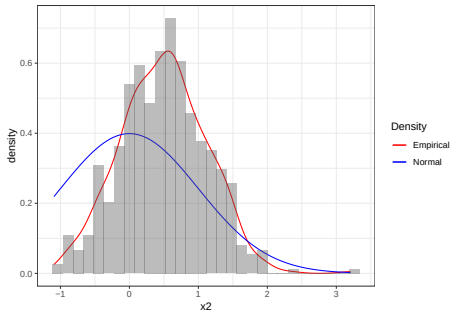


Figure 4.10: Histogram of the sequence x2 together with theoretical pdf of the standard Normal

4.6.1.2 Empirical Cumulative Distribution Function

We have already seen for uniform numbers that we can use the empirical cdf to assess if a sequence of numbers is uniformly distributed. We can use the exact same method for any other distribution.

Figure 4.11 reports the ecdf of the sequence of numbers `x1` (in red) together with the theoretical cdf of the standard Normal (in blue). We can see that the

two functions match closely and therefore we could assume that the sequence is distributed as a standard Normal.

```
ggplot(x1, aes(x1)) +
   stat_ecdf(geom = "step",aes(colour = 'Empirical')) +
   stat_function(fun = pnorm,aes(colour = 'Theoretical')) +
   theme_bw() +
   scale_colour_manual(name = 'Density', values = c('red', 'blue'))
```

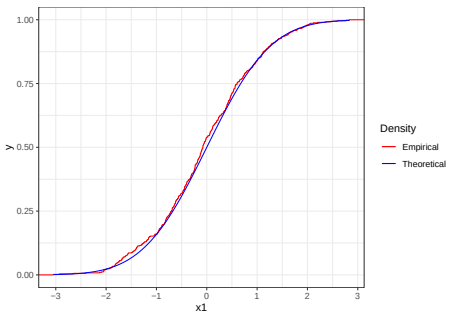


Figure 4.11: Empirical cdf the sequence x1 together with theoretical cdf of the standard Normal

Figure 4.12 reports the same plot but for the sequence `x2`. The two lines strongly differ and therefore it cannot be assume that the sequence is distributed as a standard Normal.

```
ggplot(x2, aes(x2)) +
   stat_ecdf(geom = "step",aes(colour = 'Empirical')) +
   stat_function(fun = pnorm,aes(colour = 'Theoretical')) +
   theme_bw() +
   scale_colour_manual(name = 'Density', values = c('red', 'blue'))
```

4.6.1.3 QQ-Plot

A third visualization of the distribution of a sequence of numbers is the so called *QQ-plot*. You may have already seen this when checking if the residuals of a linear regression follow a Normal distribution. But more generally, qq-plots can be used to check if a sequence of numbers is distributed according to any distribution.

We will not the details about how these are constructed but just their interpretation and implementation. Let's consider Figure 4.13. The plot is composed of

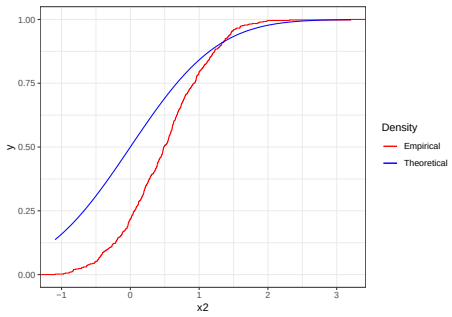


Figure 4.12: Empirical cdf the sequence x2 together with theoretical cdf of the standard Normal

a series of points, where each point is associated to a number in our random sequence, and a line, which describes the theoretical distribution we are targeting. The closest the points and the line are, the better the fit to that distribution.

In particular, in Figure 4.13 we are checking if the sequence `x1` is distributed according to a standard Normal (represented by the straight line). Since the points are placed almost in a straight line over the theoretical line of the standard Normal, we can assume the sequence to be Normal.

```
ggplot(x1, aes(sample = x1)) +
   stat_qq(distribution = qnorm) +
   stat_qq_line(distribution = qnorm) +
   theme_bw()
```

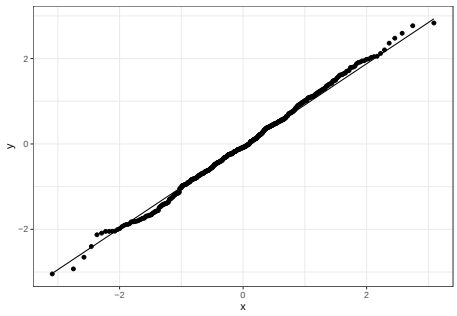


Figure 4.13: QQ-plot for the sequence x1 checking against the standard Normal

Figure 4.14 reports the qq-plot for the sequence `x2` to check if the data can be following a standard Normal. We can see that the points do not differ too much from the straight line and in this case we could assume the data to be Normal (notice that the histograms and the cdf strongly suggested that this sequence was not Normal).

```
ggplot(x2, aes(sample = x2)) +
   stat_qq(distribution = qnorm) +
   stat_qq_line(distribution = qnorm) +
   theme_bw()
```

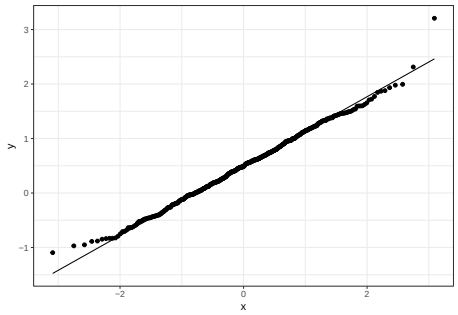


Figure 4.14: QQ-plot for the sequence x2 checking against the standard Normal

Notice that the form of the qq-plot does not only depend on the sequence of numbers we are considering, but also on the distribution we are testing it against. Figure 4.13 reports the qq-plot for the sequence `x1` when checked against an Exponential random variable with parameter $\lambda = 3$. Given that the sequence also includes negative numbers, it does not make sense to check if it is distributed as an Exponential (since it can only model non-negative data), but this is just an illustration.

```
ggplot(x1, aes(sample = x1)) +
   stat_qq(distribution = qexp, dparams = (rate = 3)) +
   stat_qq_line(distribution = qexp, dparams = (rate = 3)) +
   theme_bw()
```

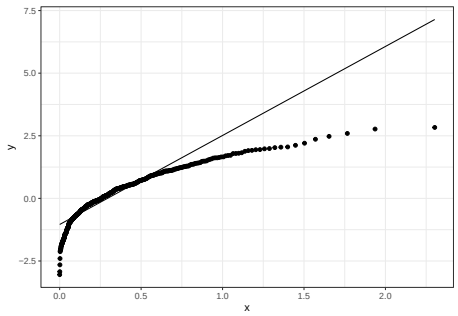


Figure 4.15: QQ-plot for the sequence x1 checking against an Exponential

4.6.1.4 Formal Testing

The above plots are highly informative since they provide insights into the shape of the data distribution, but these are not formal. Again, we can carry out tests of hypothesis to check if data is distributed as a specific random variable, just like we did for the Uniform.

Again, there are many tests one could use, but here we focus only on the Kolmogorov-Smirnov Test which checks how close the empirical and the theoretical cdfs are. It is implemented in the `ks.test` R function.

Let's check if the sequences `x1` and `x2` are distributed as a standard Normal.

```
ks.test(x1,pnorm)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  x1
## D = 0.041896, p-value = 0.3439
## alternative hypothesis: two-sided
```

```
ks.test(x2,pnorm)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  x2
## D = 0.2997, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

The conclusion is that `x1` is distributed as a standard Normal since the p-value of the test is large and for instance bigger than 0.10. On the other hand, the p-value for the Kolmogorov-Smirnov test over the sequence `x2` has a very small p-value thus leading us to reject the null hypothesis that the sequence is Normally distributed.

Notice that we can add extra inputs to the function. For instance we can check if `x1` is distributed as a Normal with `mean = 2` and `sd = 2` using:

```
ks.test(x1, pnorm, mean = 2, sd = 2)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
```

```
## data:  x1
## D = 0.54604, p-value < 2.2e-16
## alternative hypothesis: two-sided
```

The p-value is small and therefore we would reject the null hypothesis that the sequence is distributed as a Normal with mean two and standard deviation two.

4.6.2 Testing Independence

The other step in assessing if a sequence of numbers is pseudo-random is checking if independence holds. We have already learned that one possible way to do this is by computing the auto-correlation function with the R function `acf`. Let's compute the auto-correlations of various lags for the sequences `x1` and `x2`, reported in Figure 4.16. We can see that for `x2` all bars are within the confidence bands (recall that the first bar for lag $k = 0$ should not be considered). For `x1` the bar corresponding to lag 1 is slightly outside the confidence bands, indicating that there may be some dependence.

```
acf(x1)
acf(x2)
```

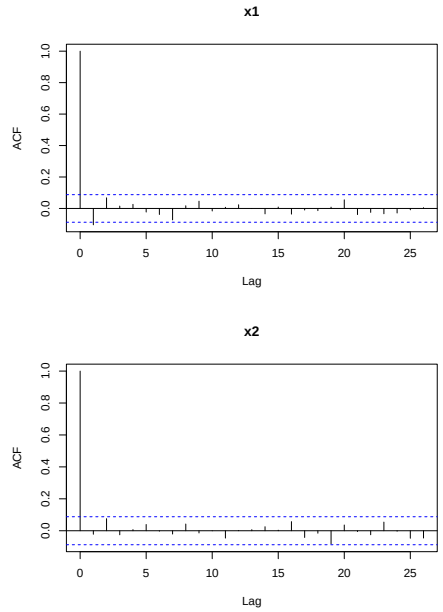


Figure 4.16: Autocorrelations for the sequences x1 and x2.

Let's run the Box test to assess if the assumption of independence is tenable for both sequences.

```
Box.test(x1, lag = 5)
```

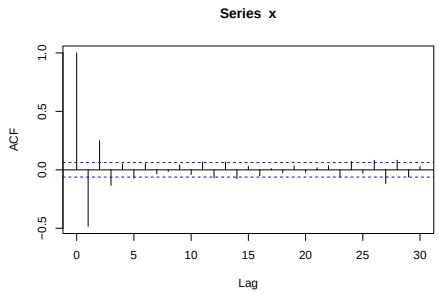
```
##
##  Box-Pierce test
##
## data:  x1
## X-squared = 8.4212, df = 5, p-value = 0.1345
```

```
Box.test(x2, lag = 5)
```

```
##
##  Box-Pierce test
##
## data:  x2
## X-squared = 4.2294, df = 5, p-value = 0.5169
```

In both cases the p-values are larger than 0.10, thus we would not reject the null hypothesis of independence for both sequences. Recall that `x1` is distributed as a standard Normal, whilst `x2` is not.

For the sequence `x1` we observed that one bar was slightly outside the confidence bands: this sometimes happens even when data is actually (pseudo-) random - I created `x1` using `rnorm`. The autocorrelations below are an instance of a case where independence is not tenable since we see that multiple bars are outside the confidence bands.



Chapter 5

Monte Carlo Simulation

The previous chapters laid the foundations of probability and statistics that now allow us to carry out meaningful simulation experiments. In this chapter we start looking at non-dynamic simulations which are often referred to as *Monte Carlo simulations*.

5.1 What does Monte Carlo simulation mean?

The definition of the Monte Carlo concept can be a bit confusing. For this reason, we will take *Sawilowsky's* example and distinguish between: Simulation, Monte Carlo method and Monte Carlo simulation.

- A **Simulation** is a fictitious representation of reality. For example: Drawing one pseudo-random uniform variable from the interval [0,1] can be used to simulate the tossing of a coin. If the value is less than or equal to 0.50 designate the outcome as heads, but if the value is greater than 0.50 designate the outcome as tails. This is a simulation, but not a Monte Carlo simulation.
- A **Monte Carlo method** is a technique that can be used to solve a mathematical or statistical problem. For example: Pouring out a box of coins on a table, and then computing the ratio of coins that land heads versus tails is a Monte Carlo method of determining the behavior of repeated coin tosses, but it is not a simulation.
- A **Monte Carlo simulation** uses repeated sampling to obtain the statistical properties of some phenomenon (or behavior). For example: drawing a large number of pseudo-random uniform variables from the interval [0,1] at one time, or once at many different times, and assigning values less than

> or equal to 0.50 as heads and greater than 0.50 as tails, is a Monte Carlo simulation of the behavior of repeatedly tossing a coin.

The main idea behind this method is that a phenomenon is simulated multiple times on a computer using random-number generation based and the results are aggregated to provide statistical summaries associated to the phenomenon.

Sawilowsky lists the characteristics of a high-quality Monte Carlo simulation:

- the (pseudo-random) number generator has certain characteristics (e.g. a long "period" before the sequence repeats)
- the (pseudo-random) number generator produces values that pass tests for randomness
- there are enough samples to ensure accurate results
- the algorithm used is valid for what is being modeled
- it simulates the phenomenon in question.

5.2 A bit of history

There were several approaches to the Monte Carlo method in the early 20th century, but it was in the mid-1940s during the Manhattan Project at "Los Alamos" that this method was first intentionally developed by Stanislaw Ulam and John von Neumann for early work relating to the development of nuclear weapons.

Below you can read a quote from Stanislaw Ulam in which he explains how he came up with this simple but powerful method:

> The first thoughts and attempts I made to practice [the Monte Carlo Method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations.

> Later [in 1946], I described the idea to John von Neumann, and we began to plan actual calculations.

Being secret, the work of von Neumann and Ulam required a code name. A colleague of von Neumann and Ulam, Nicholas Metropolis, suggested using the name Monte Carlo, which refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relatives to gamble.

5.3 Steps of Monte Carlo simulation

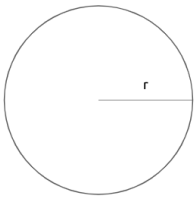
Monte Carlo methods vary, but tend to follow a particular pattern:

1. Define a domain of possible inputs
2. Generate inputs randomly from a probability distribution over the domain
3. Perform a deterministic computation on the inputs
4. Aggregate the results

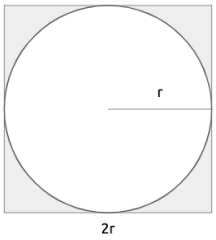
To better understand how Monte Carlo simulation works we will develop a classic experiment: The π number estimation.

π is the mathematical constant, which is equal to 3.14159265…, defined as the ratio of a circle's circumference to its diameter. It has been calculated in hundreds of different ways over the years. Today, with computational advances, a very useful way is through Monte Carlo Simulation.

Consider a circle with radius r, which is fixed and known.



Imagine that this circle is circumscribed within a square, which therefore has side $2r$ (also equal to the diameter).



What is the probability that if I choose a random point inside the square, it will also be inside the circle? If I choose any random point within the square, it can be inside the circle or just inside the square. A very simple way to compute this probability is the ratio between the area of the circle and the area of the square.

$$P(point\ inside\ the\ circle) = \frac{area\ of\ the\ circle}{area\ of\ the\ square} = \frac{\pi r^2}{2r \times 2r} = \frac{\pi}{4}$$

The probability that a random selected point in the square is in the circle is $\pi/4$. This means that if I were to replicate the selection of a random point in the square a large number of times, I could count the proportion of points inside the circle, multiply it by four and that would give me an approximation of π.

We will create a Monte Carlo experiment in R which implements the ideas above. We will carry out the experiment in 5 steps:

1. Generate 2 random numbers between -1 and 1 in total 100 times (x and y).

2. Calculate $x^2 + y^2$ (This is the circumference equation).

   - If the value is less than 1, the case will be inside the circle
   - If the value is greater than 1, the case will be outside the circle.

3. Calculate the proportion of points inside the circle and multiply it by four to approximate the π value.

4. Repeat the experiment a thousand times, to get different approximations to π.

5. Calculate the average of the previous 1000 experiments to give a final value estimate.

5.3.1 Estimating π: step 1

Generate two random numbers between -1 and 1, 100 times:

```
set.seed(2021)
nPoints <- 100
x <- runif(nPoints,-1,1)
y <- runif(nPoints,-1,1)
head(x)
```

```
## [1] -0.09746527  0.56755957  0.41936446 -0.23651148  0.27264754  0.40269205
```

```
head(y)
```

```
## [1] -0.38312966 -0.42837094 -0.97592119  0.76677917  0.03488941 -0.53043372
```

So both `x` and `y` are vectors of length 100 storing numbers between -1 and 1.

5.3.2 Estimating π: step 2

Calculate the circumference equation.

- If the value is less than 1, the case will be inside the circle
- If the value is greater than 1, the case will be outside the circle.

```
result <- ifelse(x^2 + y^2 <= 1, TRUE, FALSE)
head(result)
```

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

The vector `result` has in i-th position `TRUE` if `x[i]^2 + y[i]^2 <= 1`, that is if the associated point is within the circle. We can see that out of the first six simulated points, only one is outside the circle.

5.3.3 Estimating π: step 3

Calculate the proportion of points inside the circle and multiply it by four to approximate the π value.

```
4*sum(result)/nPoints
```

```
## [1] 2.92
```

So using our 100 simulated points, we came up with an approximation of 2.92 for the value of π. Of course this number depends on the random numbers that were generated. If we were to repeat it, we would get a different approximation.

```
set.seed(1988)
x <- runif(nPoints,-1,1)
y <- runif(nPoints,-1,1)
result <- ifelse(x^2 + y^2 <= 1, TRUE, FALSE)
4*sum(result)/nPoints
```

```
## [1] 3.08
```

5.3.4 Estimating π: step 4

Repeat the experiment a thousand times, to get different approximations to π.

We could do this by coding a `for` cycle, but we will take advantage of some features already implemented in R. In order to do this however, we first need to define a function which repeats our estimation of π given 100 random points.

```
piVal <- function(nPoints = 100){
  x <- runif(nPoints,-1,1)
  y <- runif(nPoints,-1,1)
  result <- ifelse(x^2+y^2 <= 1, TRUE, FALSE)
  4*sum(result)/nPoints
}

set.seed(2021)
piVal()
```

```
## [1] 2.92
```

```
set.seed(1988)
piVal()
```

```
## [1] 3.08
```

So we can see that the function works since it gives us the same output as the code above.

Now we can use the function `replicate` in R, to replicate the experiment 1000 times, or to put it differently, to compute the function `piVal` 1000 times. `replicate` takes two inputs:

- `n`: the number of times we want to replicate the experimet;
- `expr`: the function we want to be replicated

Therefore the following code replicates the experiment:

```
set.seed(2021)
N <- 1000
pis <- replicate(N, piVal())
head(pis)
```

```
## [1] 2.92 3.20 3.16 3.08 3.20 2.76
```

We can see that the first entry of the vector `pis` is indeed `pis[1]` which is the same value we obtained running the function ourselves (in both cases we fixed the same seed).

5.3.5 Estimating π: step 5

Calculate the average of the previous 1000 experiments to give a final value estimate.

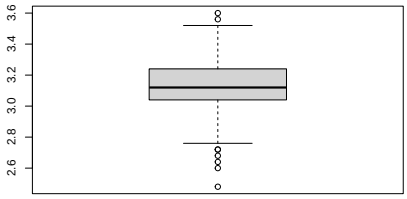
```
mean(pis)
```

```
## [1] 3.13828
```

The average gives us a good approximation of π.

A boxplot can give us a visualization of the results.

```
boxplot(pis)
```



The boxplot importantly tells us two things:

1. if we were to take the average of the 1000 approximatations of π we would get a value close to the true value (look at the horizontal line within the box).
2. if we were to choose a value for π based on a single simulation, then we could pick values between 2.48 and 3.6.

5.3.6 Estimating π: conclusions

One thing you might wonder now is the following. Why did we replicate the experiment 1000 times and each time took only 100 points. Could have we not taken a much larger number of points only once (for example 1000×100)?

On one hand that would have clearly given us a good approximation, using the same total number of simulated points. Indeed

```
set.seed(2021)
piVal(1000*100)
```

```
## [1] 3.1416
```

which is very close to the true value.

However this approach does not give us any information about uncertainty or about how good our approximation is. We have just one single value. On the other hand, using replication we have 1000 possible approximations of π and we can construct intervals of plausible values. For instance, we would believe that the true value π is with 95% probability in the interval

```
c(sort(pis)[25],sort(pis)[975])
```

```
## [1] 2.84 3.44
```

which includes 95% of the central approximations of π. Such intervals are in spirit similar to the confidence intervals you should be familiar with, but there are some technicalities that makes them different (which we will not discuss here).

5.4 The `sample` function

We have now carried out our first Monte Carlo experiment!! We will carry out others in the rest of this chapter and interpret their results. There is one function, the `sample` function which we will often use.

In the previous chapter we discussed how to simulate numbers distributed according to a specific random variable. One possible class of numbers we may want to simulate in some situations are integers. Suppose for example you want to simulate a game of dice: then we must be able to simulate in R one number from the set $\{1, 2, 3, 4, 5, 6\}$ where each has the same probability of appearing. We have not introduced yet a function that does this.

For this specific purpose there is the function `sample`. This takes four inputs:

1. `x`: a vector of values we want to sample from.
2. `size`: the size of the sample we want.
3. `replace`: if `TRUE` sampling is done with replacement. That is if a value has been selected, it can be selected again. By default equal to `FALSE`
4. `prob`: a vector of the same length of `x` giving the probabilities that the elements of `x` are selected. By defaul equal to a uniform probability.

So for instance if we wanted to simulate ten tosses of a fair dice we can write.

```
set.seed(2021)
sample(1:6, size = 10, replace = TRUE)
```

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

Notice that the vector `x` does not necessarily needs to be numeric. It could be a vector of characters. For instance, let's simulate the toss of 5 coins, where the probability of heads is 2/3 and the probability of tails is 1/3.

```
set.seed(2021)
sample(c("heads","tails"), size = 5, replace = TRUE, prob = c(2/3,1/3))
```

```
## [1] "heads" "tails" "tails" "heads" "heads"
```

5.5 A game of chance

For the rest of this chapter we will develop various Monte Carlo simulations. We start simulating a little game of chance.

Peter and Paul play a simple game involving repeated tosses of a fair coin. In a given toss, if heads is observed, Peter wins 1€ from Paul; otherwise if tails is tossed, Peter gives 1€ to Paul. If Peter starts with zero euros, we are interested in his fortune as the game is played for 50 tosses.

We can simulate this game using the R `sample()` function. Peter's winning on a particular toss will be 1€ or -1€ with equal probability. His winnings on 50 repeated tosses can be considered to be a sample of size 50 selected with replacement from the set {1€, -1€}.

```
set.seed(2021)
win <- sample(c(-1,1),size = 50, replace = TRUE)
head(win)
```

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

For this particular game Peter lost the first game, then won the second, the third and the fourth and so on.

Suppose Peter is interested in his cumulative winnings as he plays this game. The function `cumsum()` computes the cumulative winnings of the individual values and we store the cumulative values in a vector named `cumul.win`.

```
cumul.win <- cumsum(win)
cumul.win
```

```
##  [1] -1  0  1  2  1  2  3  4  5  6  5  6  7  6  5  4  3  2  3  2  3  4  3  4  3
## [26]  2  3  4  5  6  5  6  5  6  7  6  7  6  7  6  5  4  5  6  5  4  5  6  7  8
```

So at the end of this specific game Peter won 8€. Figure 5.1 reports Peter's fortune as the game evolved. We can notice that Peter was in the lead throughout almost the whole game.

```
plot(cumsum(win), type="l" ,ylim=c(-25, 25))
abline(h=0)
```

Of course this is the result of a single simulation and the outcome may be totally different than the one we saw. Figure 5.2 reports four simulated games: we can see that in the first one Peter wins, in the second he almost breaks even, whilst in the third and fourth he clearly loses.

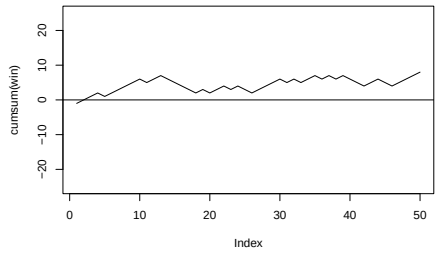


Figure 5.1: Peter's fortune throughout one simulated game.

```
set.seed(2021)
par(mfrow=c(2, 2))
for(j in 1:4){
  plot(cumsum(sample(c(-1, 1),size=50,replace=TRUE)),type="l" ,ylim=c(-25, 25), ylab="Outcome")
  abline(h=0)}
```

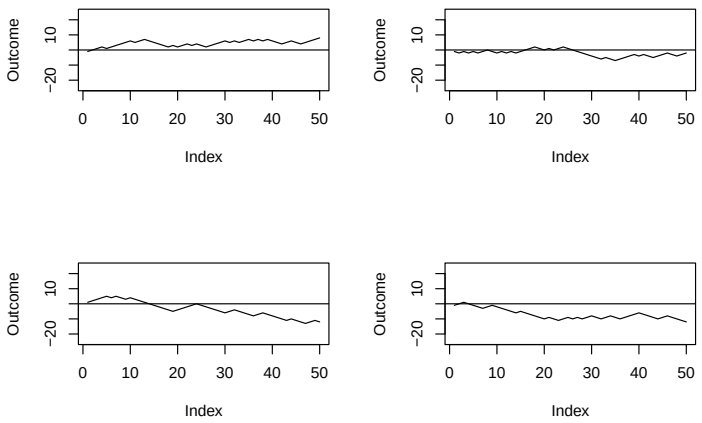


Figure 5.2: Outcome of four simulated game of chances.

Suppose we are interested in the following question.

- What is the probability that Peter breaks even at the end of the game?

Evidently we cannot cannot answer by simply looking at the outputs of the previous simulations. We need to do a formal Monte Carlo study. In this type

of experiment, we simulate the random process and compute the statistic of interest. By repeating the random process many times, we obtain a collection of values of the statistic, which can then be used to approximate probabilities or expectations that answer the questions.

As you may recall from the estimation of π experiment, we first need to write a function that simulates the experiment. In particular we need to write a function which outputs Peter's winning at the end of the game. To make this function more general, we define `n` to be the number of tosses and let the default value of `n` be 50.

```
peter.paul <- function(n=50){
  sum(sample(c(-1, 1), size=n, replace=TRUE))
}
set.seed(2021)
peter.paul()
```

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

The output is the same as the previous code, so it seems that our function works correctly.

Let's replicate the experiment many times.

```
set.seed(2021)
experiment <- replicate(1000,peter.paul())
head(experiment)
```

```
## [1]   8  -2 -12 -12 -14   8
```

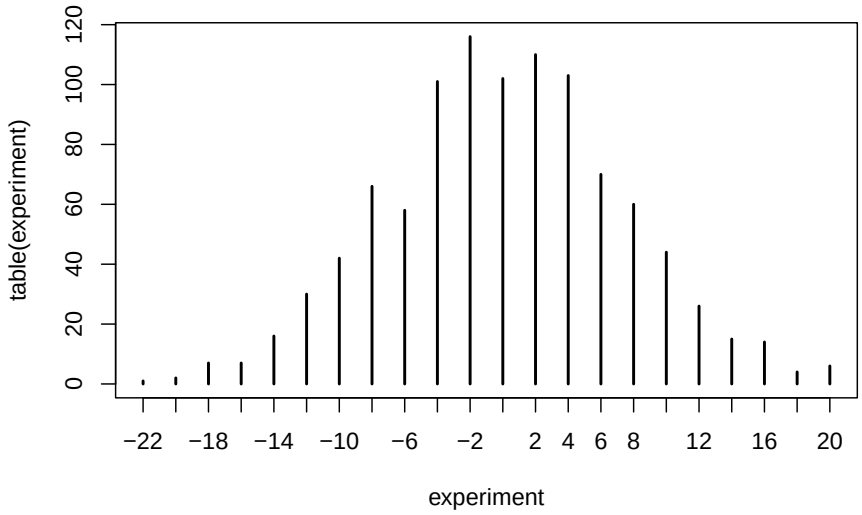
So the vector `experiment` stores Peter's final fortune in 1000 games. Since Peter's fortune is an integer-value variable, it is convenient to summarize it using the `table` function.

```
table(experiment)
```

```
## experiment
## -22 -20 -18 -16 -14 -12 -10  -8  -6  -4  -2   0   2   4   6   8  10  12  14  16
##   1   2   7   7  16  30  42  66  58 101 116 102 110 103  70  60  44  26  15  14
##  18  20
##   4   6
```

A graphical representation of the table is also useful.

```
plot(table(experiment))
```



So we can see that Peter breaks even 102 out of 1000 times. Furthermore the plot shows us that most commonly Peter will win/lose little money and that big wins/losses are unlikely.

To conclude our experiment we need to calculate our estimated probability of Peter breaking even. Clearly this is equal to 102/1000= 0.102. In R:

```
sum(experiment==0)/1000
```

```
## [1] 0.102
```

Notice that we could have also answered this question exactly. The event Peter breaking even coincides with a number of successes $n/2$ in a Binomial experiment with parameters $n = 50$ and $\theta = 0.5$. This can be computed in R as

```
dbinom(25, size = 50, prob = 0.5)
```

```
## [1] 0.1122752
```

So our approximation is already quite close to the true value. We would get even closer by replicating the experiment a larger number of times.

```
set.seed(2021)
experiment <- replicate(10000,peter.paul())
length(experiment[experiment==0])/1000
```

```
## [1] 1.096
```

5.6 Takeaways

Throughout this session we have understood:

- What the Monte Carlo method is, and how it differs from Monte Carlo simulation.
- The history of the creation of the Monte Carlo method
- To generate random numbers and random variables with R
- To establish a seed
- To conduct a simulation experiment using the Monte Carlo method.

Chapter 6

Monte Carlo methods for inference

Monte Carlo Methods may refer to any statistical or numerical method where simulation is used. When we talk about Monte Carlo methods for inference we just look at these inference processes. In this way, we can use Monte Carlo to estimate:

- Parameters of sampling
- Distribution of a statistic
- Mean squared error (MSE)
- Percentiles
- Other measures of interest.

In statistical inference there is uncertainty in any estimate. The methods we are going to see use repeated sampling from a given probability model, known as parametric bootstrap. We simulate the stochastic process that generated the data, repeatedly drawing samples under identical conditions. Other MC methods known as nonparametric use repeated sampling from an observed sample.

6.1 Monte Carlo for estimation

Let's begin with simply estimating a probability. Sometimes this is referred to as computing an expectation of a random variable. If you have a random variable X with a density function $f_X(x)$ and we want to compute the expectation of a function $g(x)$ which models the probability of $f_X(x)$, (or the area under the curve of $f_X(x)$), then $g(x)$ can be expressed as the integral of $f_X(x)$.

6.1.1 Sam and Annie from 'Sleepless in Seattle'

We can take a look to the Sleepless in Seattle video to understand the problem. It is a 1993 American romantic comedy.

Now, let A and S represent Sam's and Annie's arrival times at the Empire State Building, where we measure the arrival time as the number of hours after noon. We assume:

- A and S are independent and uniformly distributed
- Annie arrives somewhere between 10:30 and midnight
- Sam arrives somewhere between 10:00 and 11:30PM.

Our Questions are:

1) What is the probability that Annie arrives before Sam?
2) What is expected difference in arrival times?

We start simulating a large number of values from distribution of (A, S) say, `1000`, where A and S are independent:

```
set.seed(2021)
sam = runif(1000, 10, 11.5)
annie = runif(1000, 10.5, 12)
```

We want the probability $P(A < S)$ which is estimated by the proportion of simulated pairs (a, s) where a is smaller than s

```
prob = sum(annie < sam) / 1000
prob
```

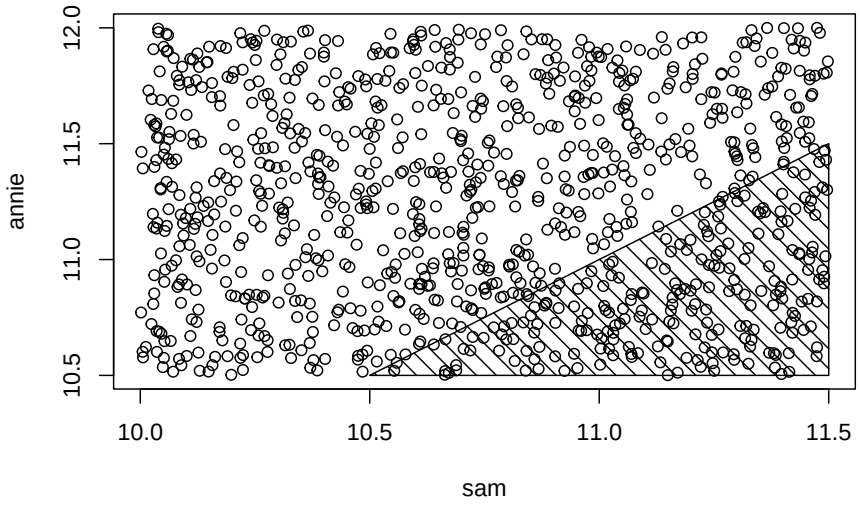
```
## [1] 0.223
```

The estimated probability that Annie arrives before Sam is 0.223, and the *standard error* of this estimation is:

```
sqrt(prob * (1 - prob) / 1000)
```

```
## [1] 0.01316324
```

In the next plot we can see that the shaded region shows the area in which $A < S$



Now, what is the expected difference in the arrival times? Annie is more likely to arrive later, so we model $E(A - S)$

```
difference = annie - sam
```

Now we can estimate the mean of the differences using Monte Carlo methods

```
estimatedMean = mean(difference)
estimatedMean
```

```
## [1] 0.5079882
```

The estimated standard error is the standard desviation of the difference divided by the square root of the simulation sample size:

```
SE = sd(difference) / sqrt(1000)
c("Mean" = estimatedMean, "Standard Error" = SE)
```

```
##           Mean Standard Error
##     0.50798819     0.01972754
```

So we estimate that Annie will arrive 0.508 hours after Sam arrives. Since standard error is only 0.02 hours, we can be 95% confident that the true difference is between 0.528 and 0.488 hours

6.2 General Case with Standard Normal Distributions

In probability theory and statistics, a collection of random variables is independent and identically distributed (iid) if each random variable has the same probability distribution as the others and all are mutually independent. Following, we are going to perform a Monte Carlo simulation to check two main iid properties:

- The expected value of absolute difference is: $E(|X-Y|) = \frac{2}{\sqrt{\pi}}$
- The variance of absolute difference is: $V(|X-Y|) = 2 - \frac{4}{\pi}$

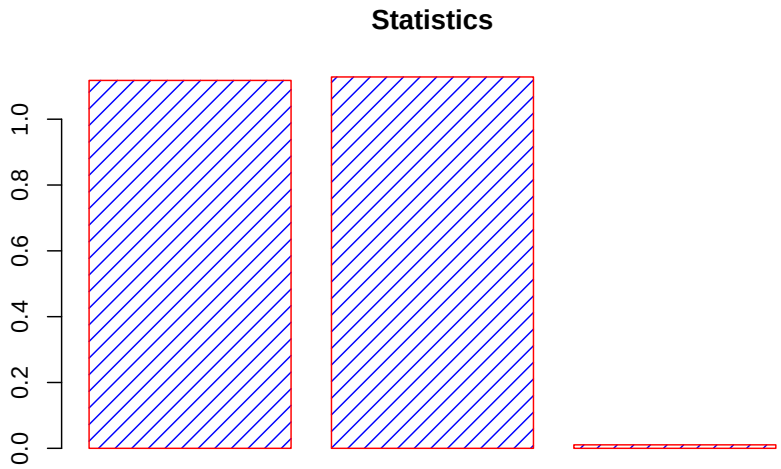
We generate a large number of random samples of size 2 from a standard normal distribution, then compute the replicate pairs' differences, and then the mean of those differences:

```
set.seed(2021)
n = 10000
g = numeric(n)

for (i in 1:n) {
  x = rnorm(2)
  g[i] = abs(x[1] - x[2])
}

estMean = mean(g)
expectedValue = 2/sqrt(pi)
diffMeanExpectation = abs(estMean-expectedValue)

c("Estimated Mean" = estMean, "Expected Value" = expectedValue, "Difference" = diffMean
```

```
## Estimated Mean Expected Value     Difference
##     1.11782094     1.12837917     0.01055823
```

Next, we can see a plot that represents the results.

```
library(ggplot2)

dataPlot = c(estMean, expectedValue, diffMeanExpectation)
barplot(dataPlot, main = "Statistics", border="red", col="blue", density=12)
```



6.3 The taxi problem (comparing estimators)

Finally, we will see a case in which we want to compare two different estimators. Imagine that a person is walking through the streets of a city and notices the following numbers of 5 taxis that pass by: 34,100,65,81,120. Can he/she make an intelligent guess at the number of taxis in the city?: Is a problem of statistical inference where population is collection of taxis driven in city and one wishes to know unknown number of taxis N.

Assume taxis are numbered from 1 to N, each equally likely to be observed and consider two possible estimates:

1. The largest taxi number observed
2. Twice the sample mean.

Which is a better estimator of the number of taxis N? We will compare these two estimators using a Monte Carlo Simulation:

1. Simulate taxi numbers from a uniform distribution with a known number of taxis N and compute the two estimates.
2. Repeat many times and obtain two empirical sampling distributions.
3. Then we can compare the two estimators by examining various properties of their respective sampling distributions

The `taxi()` function will implement a single simulation. We have two arguments:

1. The actual number of taxis `N`.
2. The sample size `n`.

```
taxi = function(N, n){
  y = sample(N, size=n, replace=TRUE)
  estimate1 = max(y)
  estimate2 = 2 * mean(y)
  c(estimate1=estimate1, estimate2=estimate2)
}
```

The `sample()` function simulates the observed taxi numbers and values of the two estimates are stored in variables `estimate1` and `estimate2`. Let's say actual number of taxis in city is `100` and we observe numbers of `n=5` taxis.

```
set.seed(2021)
taxi(100, 5)
```

```
## estimate1 estimate2
##      58.0      64.4
```

We get values `estimate1=58` and `estimate2=64.4`

Let's simulate sampling process 1000 times. We are going to create a matrix with two rows (`estimate1` and `estimate2`), and `1000` columns. This colums will hold the estimated values of `estimate1` and `estimate2` for `1000` simulated experiments.

```
set.seed(2021)
EST = replicate(1000, taxi(100, 5))
```

Here we are looking for "unbiasedness", which means that the average value of the estimator must be equal to the parameter. We know that the number of taxis is 100. So we can calculate the mean standard error for our estimators as follows:

```
c(mean(EST["estimate1", ]) -100, sd(EST["estimate1", ]) / sqrt(1000))
```

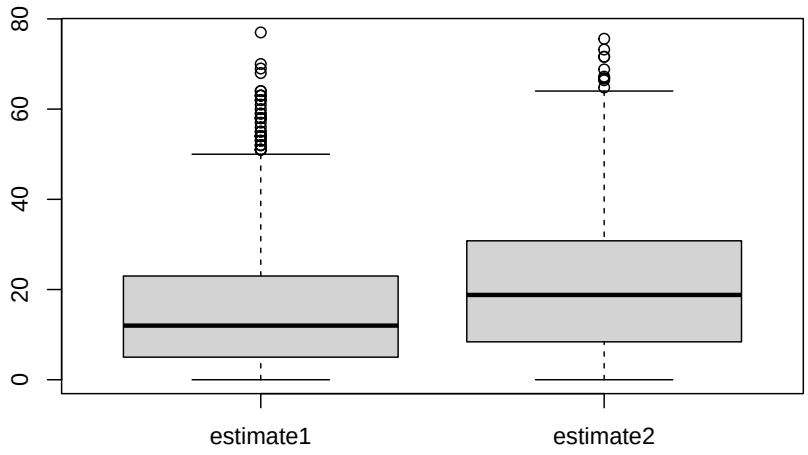
```
## [1] -15.7170000   0.4511901
```

```
c(mean(EST["estimate2", ]) -100, sd(EST["estimate2", ]) / sqrt(1000))
```

```
## [1] 1.4248000 0.8360629
```

Seems that `estimate2` is "less biased", but we can also compare them with respect to the mean distance from the parameter `N` (mean absolute error). Now, we are going to compute the mean absolute error and draw a boxplot:

```
absolute.error = abs(EST - 100)
boxplot(t(absolute.error))
```



In this case, seems that `estimate1` has smaller estimation errors. We can also find the sample mean of the absolute errors and its standard error:

```
c(mean(absolute.error["estimate1", ]), sd(absolute.error["estimate1", ]) / sqrt(1000))
```

```
## [1] 15.7170000  0.4511901
```

```
c(mean(absolute.error["estimate2", ]), sd(absolute.error["estimate2", ]) / sqrt(1000))
```

```
## [1] 21.463200  0.489799
```

Again, the `estimate1` looks better than the `estimate2`.

Chapter 7

Discrete Events Simulation with R

We are not going to explore this subject in depth, but at the end of this chapter we will understand what the discrete events simulation consists of, and we will carry out an example in R. Unlike Monte Carlo methods, which allow us to simulate events based on variables that change constantly over time, the discrete events simulation will help us to model the behavior of systems based on a sequence of discrete events over time. Remember that for both cases we use stochastic models.

Recall that different types of simulation apply depending on the nature of the system under consideration. A common model taxonomy classifies simulation problems along three main dimensions:

1. deterministic vs. stochastic
2. static vs. dynamic (depending on whether they require a time component)
3. continuous vs. discrete (depending on how the system changes).

For instance, Monte Carlo methods are well-known examples of static stochastic simulation techniques. On the other hand, discrete-event simulation (DES) is a specific technique for modelling stochastic, dynamic and discretely evolving systems. As opposed to continuous simulation, which typically uses smoothly-evolving equational models, DES is characterized by sudden state changes at precise points of (simulated) time.

Customers arriving at a bank, products being manipulated in a supply chain, or packets traversing a network are common examples of such systems. The discrete nature of a given system arises as soon as its behavior can be described in terms of events, which is the most fundamental concept in DES. An event

is an instantaneous occurrence that may change the state of the system, while, between events, all the state variables remain constant.

There are several world views, or programming styles, for DES (Banks 2005):

- In the **activity-oriented** approach, a model consists of sequences of activities, or operations, waiting to be executed depending on some conditions. The simulation clock advances in fixed time increments. At each step, the whole list of activities is scanned, and their conditions, verified. Despite its simplicity, the simulation performance is too sensitive to the election of such a time increment. -Instead, the **event-oriented** approach completely bypasses this issue by maintaining a list of scheduled events ordered by time of occurrence. Then, the simulation just consists in jumping from event to event, sequentially executing the associated routines.
- Finally, the **process-oriented** approach refines the latter with the addition of interacting processes, whose activation is triggered by events. In this case, the modeller defines a set of processes, which correspond to entities or objects of the real system, and their life cycle.

In this course we will use the `simmer` package to perform our DES. `simmer` (Ucar and Smeets 2019a) is a DES package for R which enables high-level **process-oriented** modelling, in line with other modern simulators. But in addition, it exploits the novel concept of **trajectory**: a common path in the simulation model for entities of the same type. In other words, a trajectory consist of a list of standardized actions which defines the life cycle of equivalent processes. This design pattern is flexible and simple to use. It is important to note that `simmer` package uses pipes in order to simplify the structure of the simulation.

7.1 `simmer` terminology

In this course we have already used some DES-specific terminology, e.g., event, state, entity, process or attribute. But there are, however, some simmer-specific terms, and some elements that require further explanation to understand the package architecture.

- **Resource**: A passive entity, as it is commonly understood in standard DES terminology. However, simmer resources are conceived with queuing systems in mind, and therefore they comprise two internal self-managed parts:
  - Server: which, conceptually, represents the resource itself. It has a specified capacity and can be seized and released.
  - Queue: A priority queue of a certain size.

- **Manager**: An active entity, i.e., a process, that has the ability to adjust properties of a resource (capacity and queue size) at run-time.
- **Source**: A process responsible for creating new arrivals with a given inter-arrival time pattern and inserting them into the simulation model.
- **Arrival**: A process capable of interacting with resources or other entities of the simulation model. It may have some attributes and prioritization values associated and, in general, a limited lifetime. Upon creation, every arrival is attached to a given trajectory.
- **Trajectory**: An inter-linkage of activities constituting a recipe for arrivals attached to it, i.e., an ordered set of actions that must be executed. The simulation model is ultimately represented by a set of trajectories.
- **Activity**: The individual unit of action that allows arrivals to interact with resources and other entities, perform custom routines while spending time in the system, move back and forth through the trajectory dynamically, and much more.

The `R` API (Application Programming Interface) exposed by `simmer` comprises two main elements: the **simmer environment** (or simulation environment) and the **trajectory object**.

7.2 The Trajectory Object

A trajectory can be defined as a recipe and consists of an ordered set of activities. The `trajectory()` method instantiates the object, and activities can be appended using the `%>%` operator:

```
library(simmer)
library(magrittr)

traj0 <- trajectory() %>%
  log_("Entering the trajectory") %>%
  timeout(10) %>%
  log_("Leaving the trajectory")
```

The trajectory above illustrates the two most basic activities available: displaying a message (`log_()`) and spending some time in the system (`timeout()`).

An arrival attached to this trajectory will execute the activities in the given order, i.e., it will display "Entering the trajectory", then it will spend 10 units of (simulated) time, and finally it will display "Leaving the trajectory".

The example uses fixed parameters: a string and a numeric value respectively. However, at least the main parameter for all activities can also be what we will call a dynamical parameter, i.e., a function.

```
traj1 <- trajectory() %>%
  log_(function() "Entering the trajectory") %>%
  timeout(10) %>%
  log_(function() "Leaving the trajectory")
```

Let's check what happens if we print the trajectory:

```
traj1
```

```
## trajectory: anonymous, 3 activities
## { Activity: Log          | message: function(), level: 0 }
## { Activity: Timeout      | delay: 10 }
## { Activity: Log          | message: function(), level: 0 }
```

We see the three activities we have created in our trajectory: message + delay + message. There are many activities available. We will briefly review them by categorizing them into different topics.

- **Arrival properties**: Arrivals are able to store attributes and modify these using `set_attribute()`. Arrivals also hold a set of three prioritization values for accessing resources, `priority`, `preemptible` and `restart`, and we can use them inside the `set_priorization()` function.
- **Interaction with resources**: The two main activities for interacting with resources are `seize()` and `release()`. In their most basic usage, they seize/release a given amount of a resource specified by name.
- **Interaction with sources**: There are four activities specifically intended to modify arrival sources. An arrival may `activate()` or `deactivate()` a source, but also modify with `set_trajectory()` the trajectory to which it attaches the arrivals created, or set a new inter-arrival distribution with `set_source()`.
- **Branching**: A branch is a point in a trajectory in which one or more sub-trajectories may be followed. Two types of branching are supported in `simmer`. The `branch()` activity places the arrival in one of the sub-trajectories depending on some condition evaluated in a dynamical parameter called option. On the other hand, the `clone()` activity is a parallel branch. It does not take any option.
- **Loops**: There is a mechanism, `rollback()`, for going back in a trajectory and thus executing loops over a number of activities. This activity causes the arrival to step back a given amount of activities a number of times.

- **Batching**: Batching consists of collecting a number of arrivals before they can continue their path in the trajectory as a unit 2 . This means that if, for instance, 10 arrivals in a batch try to seize a unit of a certain resource, only one unit may be seized, not 10.
- **Asynchronous programming**: There are a number of methods enabling asynchronous events. The `send()` activity broadcasts one or more **signals** to all the arrivals subscribed to them. Signals can be triggered immediately or after some **delay**. Arrivals are able to block and `wait()` until a certain signal is received.
- **Reneging**: Besides being rejected while trying to seize a resource, arrivals are also able to leave the trajectory at any moment, synchronously or asynchronously. Namely, reneging means that an arrival abandons the trajectory at a given moment. The most simple activity enabling this is `leave`, which immediately triggers the action given some probability.

7.3 The Simulation Enviroment

The simulation environment manages resources and sources, and controls the simulation execution. The `simmer()` method instantiates the object, after which resources and sources can be appended using the `%>%` operator:

```
env <- simmer() %>%
  add_resource("res_name", 1) %>%
  add_generator("arrival", traj0, function() 25) %>%
  print()
```

```
## simmer environment: anonymous | now: 0 | next: 0
## { Monitor: in memory }
## { Resource: res_name | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Source: arrival | monitored: 1 | n_generated: 0 }
```

Then, the simulation can be executed, or `run()`, until a stop time:

```
env %>% run(until = 30)
```

```
## 25: arrival0: Entering the trajectory
```

```
## simmer environment: anonymous | now: 30 | next: 35
## { Monitor: in memory }
## { Resource: res_name | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Source: arrival | monitored: 1 | n_generated: 2 }
```

There are a number of methods for extracting information, such as the simulation time (`now()`), future scheduled events (`peek()`), and getters for obtaining resources' and sources' parameters (capacity, queue size, server count and queue count; number of arrivals generated so far). There are also several setters available for resources and sources (capacity, queue size; trajectory, distribution).

- **Resources**: A simmer resource comprises two internal self-managed parts: a **server** and a **priority queue**. Three main parameters define a resource: `name` of the resource, `capacity` of the server and `queue_size` (0 means no queue). Resources are monitored, non-preemptive and assumes a first-in-first-out (FIFO) policy by default.
- **Sources**: Three main parameters define a source: a `name_prefix` for each generated arrival, a `trajectory` to attach them to and a source of inter-arrival times. There are two kinds of source: **generators** and **data sources**. A generator (`add_generator` method) is a dynamic source that draws inter-arrival times from a user-provided function. The `add_dataframe` method allows the user to set up a data source which draws arrivals from a provided data frame.

7.4 Monitoring and data retrieval

There are three methods for obtaining monitored data about arrivals, resources and attributes. They can be applied to a single simulation environment or to a list of environments, and the returning object is always a data frame.

- `get_mon_arrivals()`: Returns timing information per arrival: `name` of the arrival, `start_time`, `end_time`, `activity_time` (time not spent in resource queues) and a flag, `finished`, that indicates whether the arrival exhausted its activities (or was rejected). By default, this information is referred to the arrivals' entire lifetime, but it may be obtained on a per-resource basis by specifying `per_resource = TRUE`.
- `get_mon_resources()`: Returns state changes in resources.
- `get_mon_attributes()`: Returns state changes in attributes.

7.5 `simmer` Example

Let's see an example in R of discrete event simulation in which we simulate the behavior of patients in a hospital.

First we have to load the `simmer` and `magrittr` libraries to do the simulation

```
library(simmer)
library(magrittr)
```

With the `simmer` library we create a simulation environment that we are going to call `outpatient clinic`

```
env = simmer("outpatient clinic")
env
```

```
## simmer environment: outpatient clinic | now: 0 | next:
## { Monitor: in memory }
```

Second, we create the patient's trajectory. We do this using the `trajectory` function of the `simmer` library.

Within this path we create the activities of the entities in the trajectory using the `seize` and `release` functions. Among these functions we introduce the time that each patient takes to carry out each event, that would correspond with the activity of the entities.

For example, in the case of the nurse, we seize the entity with the function `seize`, we give it a time limit of 15 minutes (we do this simulating a random number with a normal distribution of mean 15 and standard deviation 1), and we release the entity with the function `release`.

```
patient = trajectory("patients path") %>%
  seize("nurse", 1) %>%
  timeout(function() rnorm(1,15)) %>%
  release("nurse", 1) %>%

  seize("doctor", 1) %>%
  timeout(function() rnorm(1,20)) %>%
  release("doctor", 1) %>%

  seize("administration", 1) %>%
  timeout(function() rnorm(1,5)) %>%
  release("administration", 1)
```

Thirdly, we add the number of elements of the system using the `add_resource` function.

In this case we introduce in the system two elements of the nurse type, three of the doctor type and two of the administration type.

We also generate the patient entity that will interact with the whole system using the `add_generator` function. We will generate a patient every five minutes with a standard deviation of 0.5 minutes.

```
env %>%
  add_resource("nurse", 2) %>%
  add_resource("doctor", 3) %>%
  add_resource("administration", 2) %>%
  add_generator("patient", patient, function() rnorm(1,5,0.5))
```

```
## simmer environment: outpatient clinic | now: 0 | next: 0
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 0(2) | queue status: 0(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 0(3) | queue status: 0(Inf) }
## { Resource: administration | monitored: TRUE | server status: 0(2) | queue status: 0
## { Source: patient | monitored: 1 | n_generated: 0 }
```

We finally launch the simulation for 540 minutes. When the simulation is finished we can see how our resources are working in the simulated system.

```
env %>%
  run(until=540)
```

```
## simmer environment: outpatient clinic | now: 540 | next: 540.960538344821
## { Monitor: in memory }
## { Resource: nurse | monitored: TRUE | server status: 2(2) | queue status: 34(Inf) }
## { Resource: doctor | monitored: TRUE | server status: 3(3) | queue status: 0(Inf) }
## { Resource: administration | monitored: TRUE | server status: 1(2) | queue status: 0
## { Source: patient | monitored: 1 | n_generated: 108 }
```

It is very easy to replicate a simulation multiple times using standard R functions.

```
envs <- lapply(1:100, function(i) {
  simmer("outpatient clinic") %>%
    add_resource("nurse", 2) %>%
    add_resource("doctor", 3) %>%
    add_resource("administration", 2) %>%
    add_generator("patient", patient, function() rnorm(1,5,0.5)) %>%
    run(540)
})
```

This package provides some basic visualization tools to help you take a glance at your simulations quickly. There are three types of plot implemented with different metrics available:

- Plot of resources. Two metrics available:

  - the usage of a resource over the simulation time frame.
  - the utilization of specified resources in the simulation.

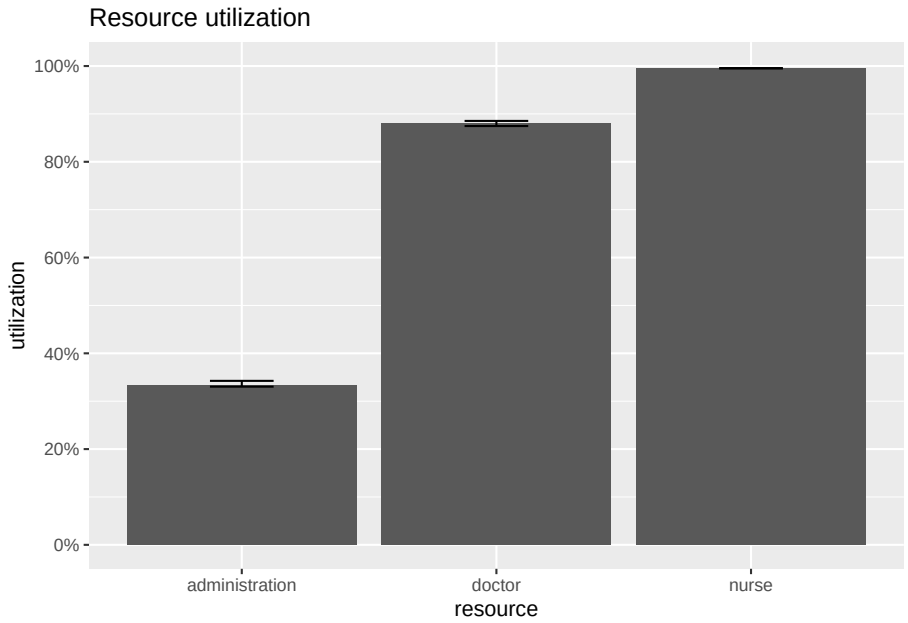
- Plot of arrivals. Three metrics available:

  - activity time.
  - waiting time.
  - flow time.

- Plot of trayectories

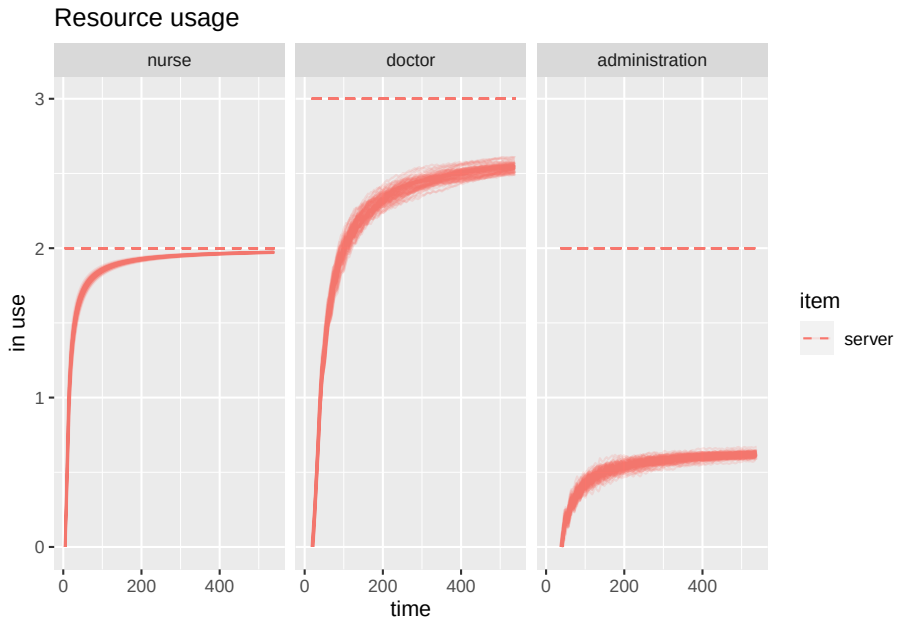
With this graphs we can have a look at the overall resource utilization. The top and bottom of the error bars show respectively the 25th and 75th percentile of the utilization across all the replications. The top of the bar shows the median utilization.

```
library(simmer.plot)

resources <- get_mon_resources(envs)
plot(resources, metric = "utilization")
```



It is also possible to have a look at resources' activity during the simulation.

```
plot(resources, metric = "usage", c("nurse", "doctor", "administration"), items = "server")
```

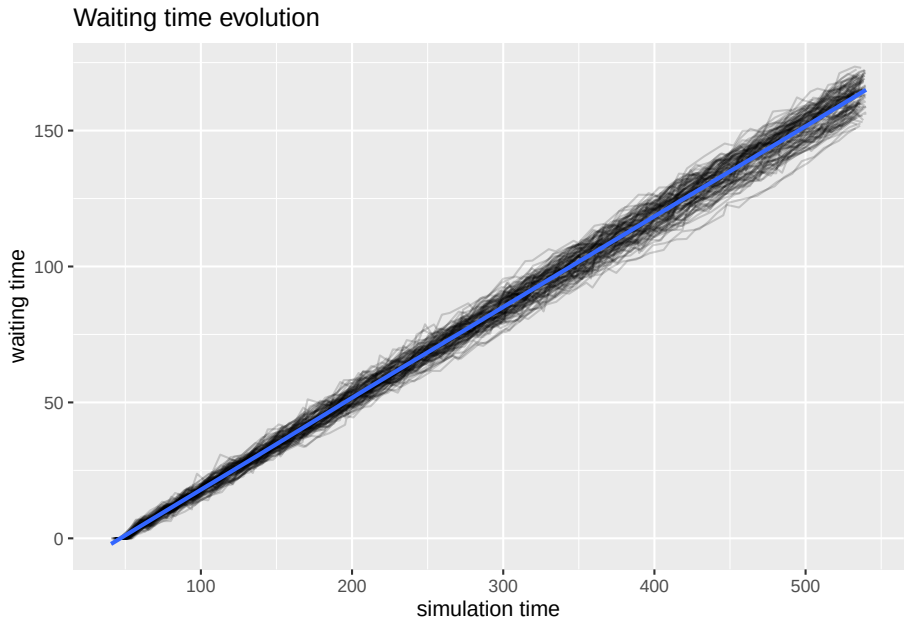


Next we can have a look at the evolution of the arrivals' flow time during the simulation. In the plot below, each individual line represents a replication. A smooth line is drawn over them. All arrivals that didn't finish their entire trajectory are excluded from the plot.

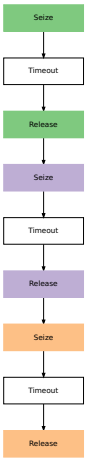
Similarly one can have a look at the evolution of the activity times with `metric = "activity_time"` and waiting times with `metric = "waiting_time"`.

```
arrivals <- get_mon_arrivals(envs)
plot(arrivals, metric = "waiting_time")
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



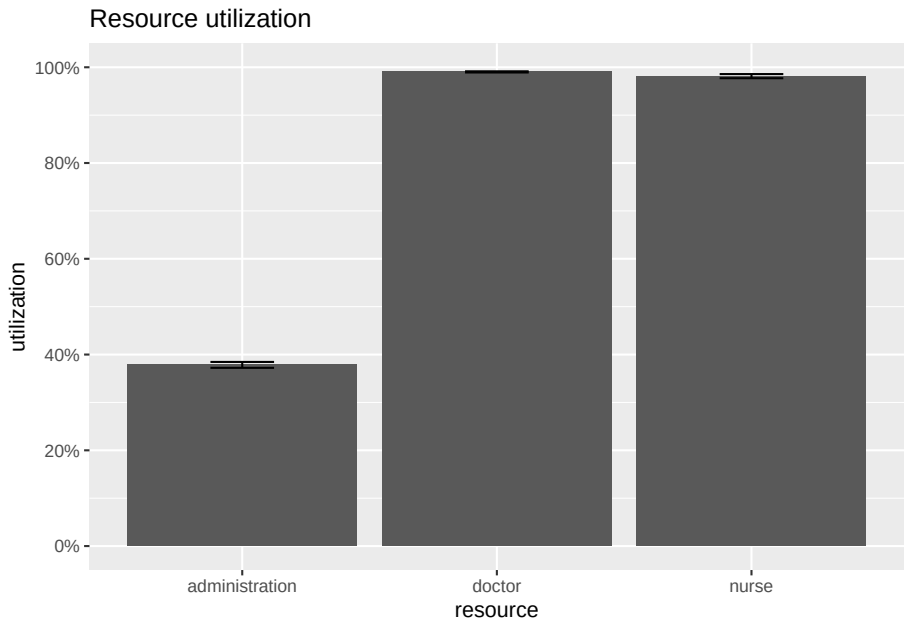
Simulation trajectories may grow considerably, and they are not always easy to inspect to ensure their compliance with the model that we are trying to build. In this example we are going to expose a simple one:

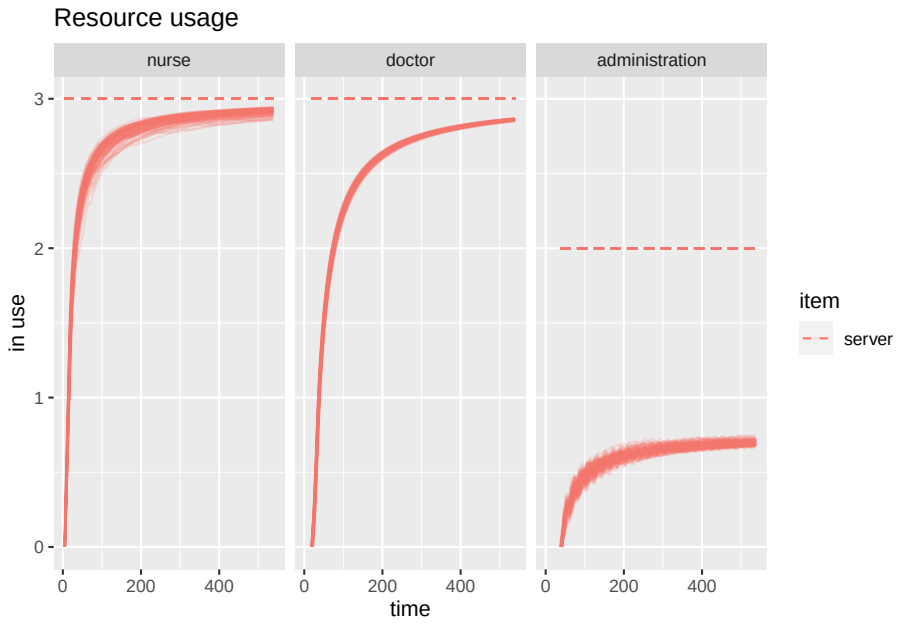


Now that we know a bit about the `simmer` tool we are going to make some changes in the system to try to improve its efficiency. From what we have seen in the examples, it seems that the nurses are overworked. We will first try to "hire" a new nurse to improve the system.

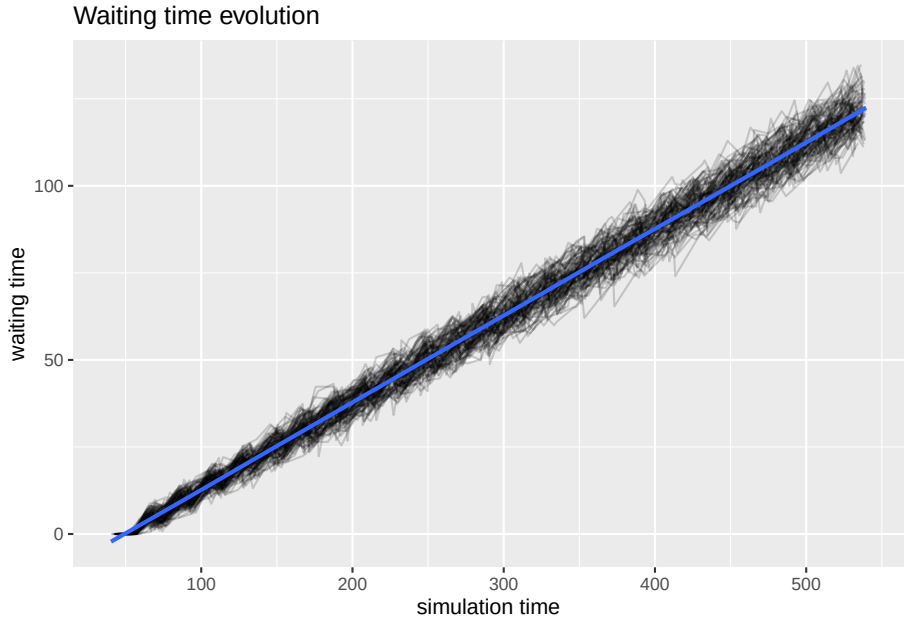
```
envs2 <- lapply(1:100, function(i) {
  simmer("outpatient clinic") %>%
    add_resource("nurse", 3) %>%
    add_resource("doctor", 3) %>%
    add_resource("administration", 2) %>%
    add_generator("patient", patient, function() rnorm(1,5,0.5)) %>%
    run(540)
})
```

Let's see how the system works with a new nurse through the plots:






```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

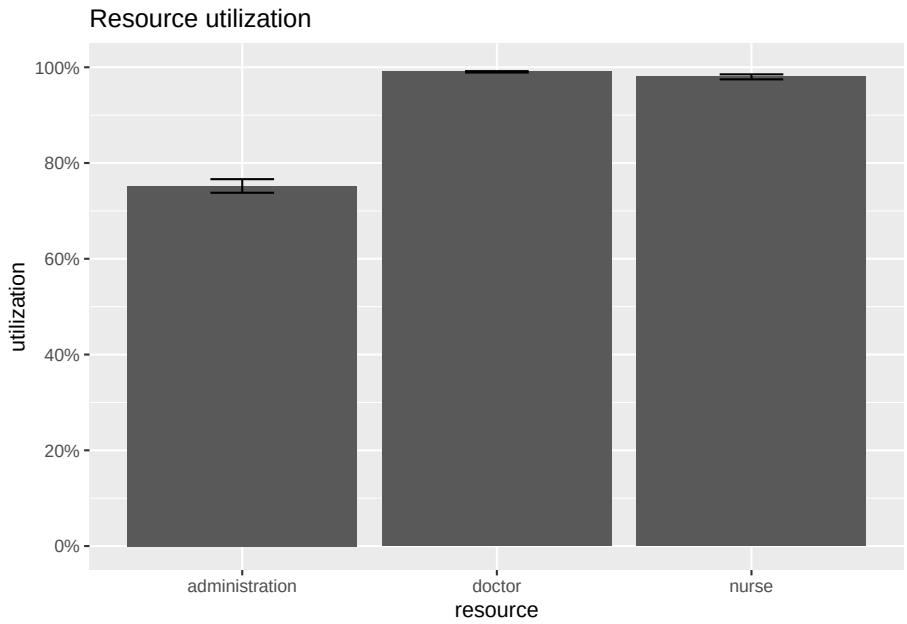


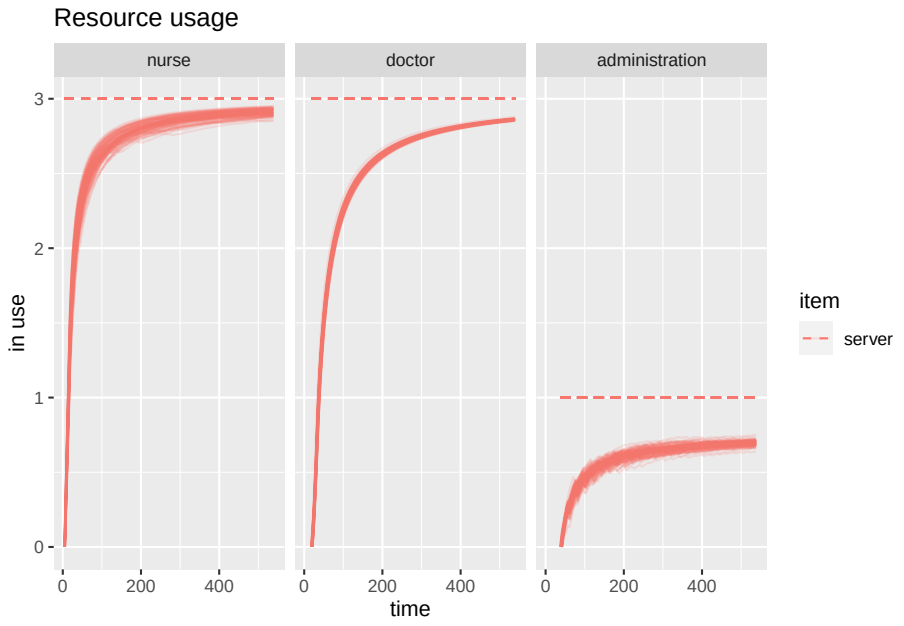
We can see that both doctors and nurses are now working at 100% capacity. Now the problem seems to be coming from the administration. We are going to

"fire" one of the workers in this area to try to improve the system.

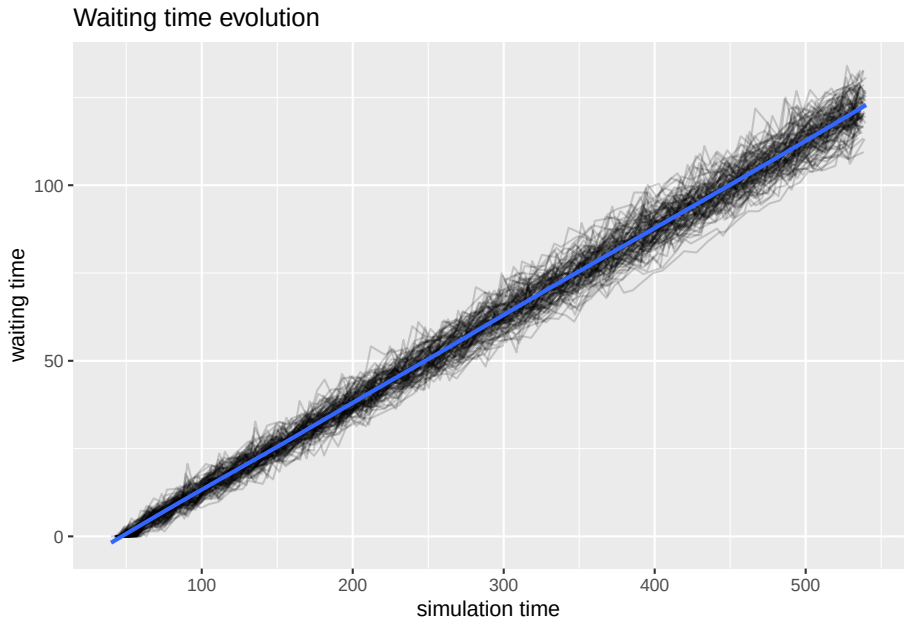
```
envs3 <- lapply(1:100, function(i) {
  simmer("outpatient clinic") %>%
    add_resource("nurse", 3) %>%
    add_resource("doctor", 3) %>%
    add_resource("administration", 1) %>%
    add_generator("patient", patient, function() rnorm(1,5,0.5)) %>%
    run(540)
})
```

Let's see how the system works without a administration worker through the plots:






```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Great! We've found the perfect balance for our little hospital.