Simulation and Modelling to Understand Change

Manuele Leonelli

2021-01-18

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. The module is given in the 2nd semester of the 1st year of the bachelor in Data & Business Analytics. 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.

```r
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.

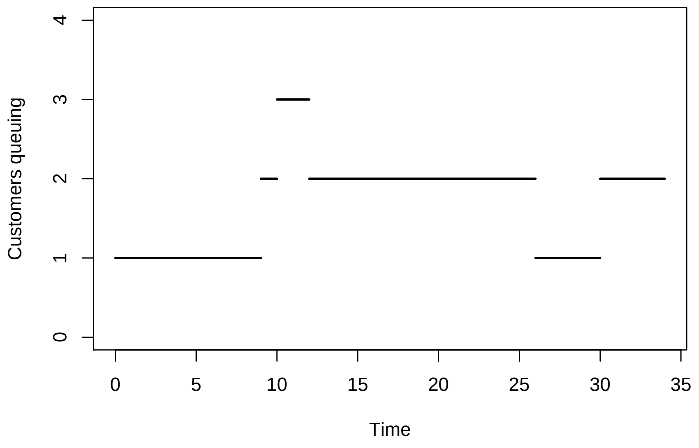

Figure 1.1: Example of a discrete dynamic simulation

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.

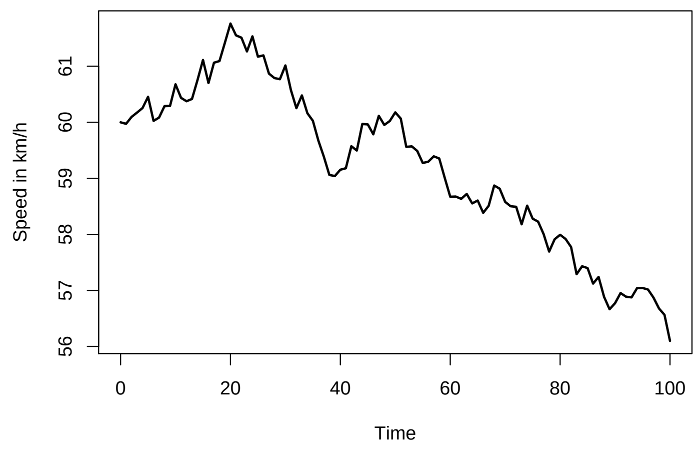

Figure 1.2: Example of a discrete dynamic simulation

In later chapters we will focus on discrete simulations, which are usually called *discrete-event simulation*. Continuous simulations will not be discussed in these notes.

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.

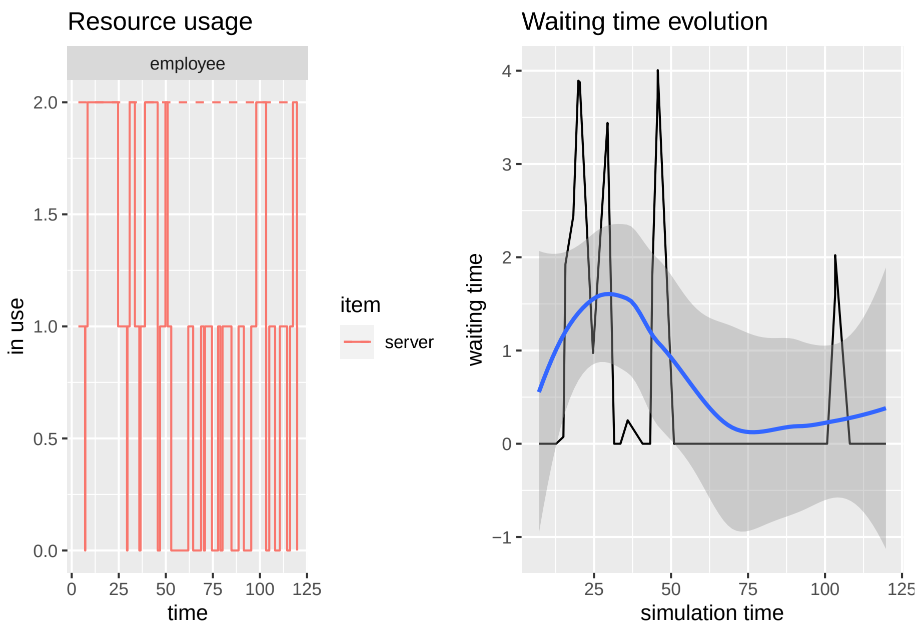

Figure 1.3: Graphical summaries from the simulation of the donut shop

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!

```r
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")
```

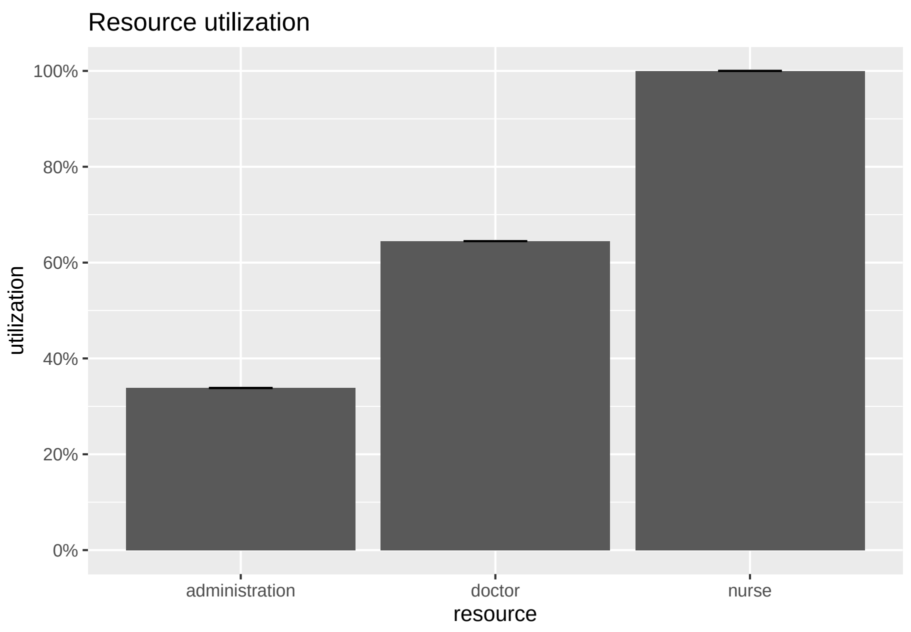

Figure 1.4: Utilization of the resources in the health center

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

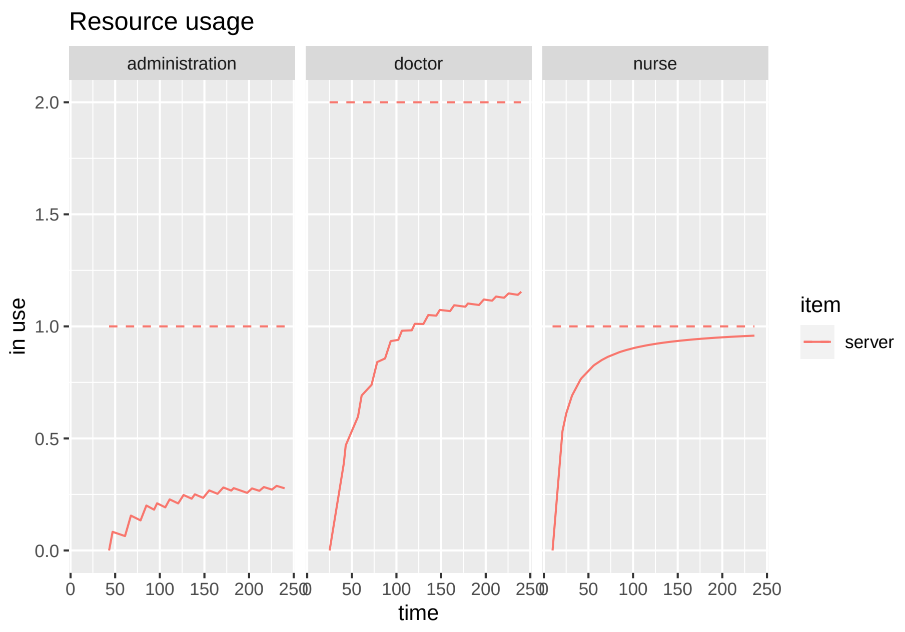

Figure 1.5: Usage of the resources in the health center

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;

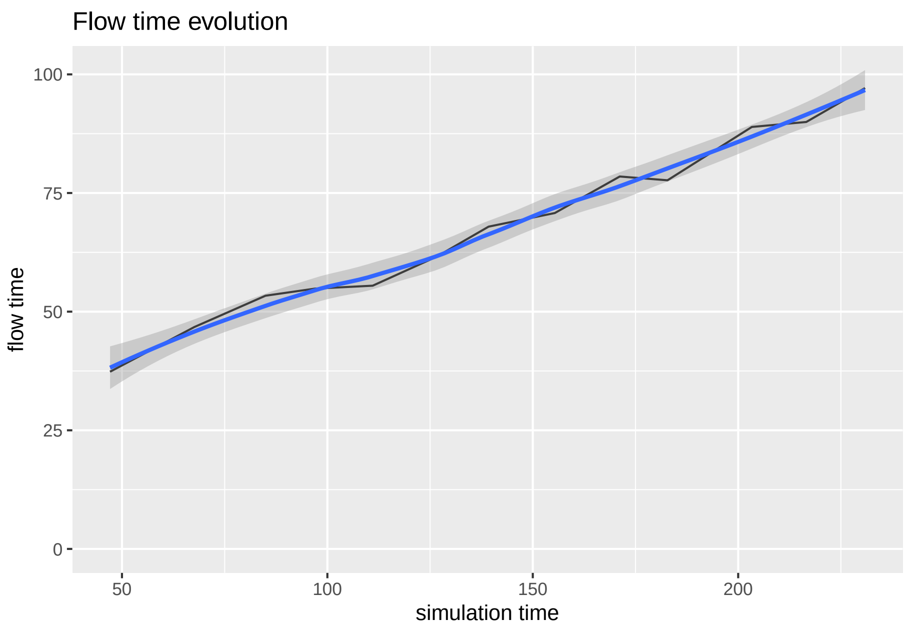

Figure 1.6: Time spent in the health center

- 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 capabalities 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 R basics

So let's get started with R programming!

2.2.1 R as a calculator

In its most basic usage, we can use R as a calculator. Basic algebraic operations can be carried out as you would expect. The symbol `+` is for sum, `-` for subtraction, `*` for multiplication and `/` for division. Here are some examples:

```r
4 + 2
```

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

```r
4 - 2
```

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

```
4 * 2
```

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

```
5 / 2
```

```
## [1] 2.5
```

2.2.2 Variable assignment

In R the symbol `<-` is used to assign a quantity to a variable. For instance, `a <- 4` assigns the number `4` to the variable `a` and `b <- 3` assigns the number `3` to `b`. It is much more common to work with variables in programming. Basic operations can then be performed over variables.

```
a <- 4
b <- 3
a + b
```

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

```
a - b
```

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

Notice for example that the code `a <- 4` does not show us the value of the variable `a`. It only creates this assignment. If we want to print the value of a variable, we have to explictly type the name of the variable.

```
a
```

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

2.2.3 Data types

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

- *Logicals* or *Booleans*: corresponding to `TRUE` and `FALSE`, also abbreviated as `T` and `F` respectively;

- *Doubles*: real numbers;
- *Characters*: strings of text surrounded by `"` (for example `"hi"`) or by `'` (for example 'by');
- *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.

Examples:

```
a <- TRUE
a
```

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

```
b <- "hello"
b
```

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

2.2.4 Vectors

In all previous examples the variables included one element only. More generally we can define sequences of elements or so-called *vectors*. They can be defined with the command `c`, which stands for combine.

```
vec <- c(1,3,5,7)
vec
```

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

So `vec` includes the sequence of numbers 1, 3, 5, 7. Notice that a vector can only include one data type. Consider the following:

```
vec <- c(1, "hello", TRUE)
vec
```

```
## [1] "1"     "hello" "TRUE"
```

We created a variable `vec` where the first entry is a number, then a character string, then a Boolean. When we print `vec`, we get that its elements are `"1"`, `"hello"` and `"TRUE"`: it has transformed the number `1` into the string `"1"` and the Boolean `TRUE` into `"TRUE"`.

2.2.5 Matrices

Matrices are tables of elements that are organized in rows and columns. You can think of them as an arrangement of vectors into a table. Matrices must have the same data type in all its entries, as for vectors. Matrices can be constructed in multiple ways. One way is by stacking vectors into a matrix row-by-row with the command `rbind`. Consider the following example.

```
row1 <- c(1,2,3)
row2 <- c(4,5,6)
row3 <- c(7,8,9)
mat <- rbind(row1,row2,row3)
mat
```

```
##      [,1] [,2] [,3]
## row1    1    2    3
## row2    4    5    6
## row3    7    8    9
```

So first we created vectors `row1 = (1,2,3)`, `row2 = (4,5,6)` and `row3 = (7,8,9)` and then organizing them together into the matrix `mat`.

The following code follows the same procedure but now organizes vectors by columns instead using the command `cbind`.

```
col1 <- c(1,2,3)
col2 <- c(4,5,6)
col3 <- c(7,8,9)
mat <- cbind(col1,col2,col3)
mat
```

```
##      col1 col2 col3
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Last, there is also a command called `matrix` to create a matrix. It takes a vector, defined using the command `c` and stores its entries into a matrix of `nrow` rows and `ncol` columns. Consider the following example.

```
vec <- c(1,2,3,4,5,6,7,8,9)
mat <- matrix(vec, nrow = 3, ncol = 3)
mat
```

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

So first we created a vector `vec` with numbers from 1 to 9 and then stored them in a matrix with 3 rows and 3 columns. Number are stored by column: the first element of `vec` is in entry (1,1), the second element of `vec` is in entry (2,1), and so on.

2.2.6 Dataframes

Dataframes are very similar as matrices, they are tables organized in rows and columns. However, different to matrices they can have columns with different data types. They can be created with the command `data.frame`.

```
data <- data.frame(X1 = c(1,2,3), X2 = c(TRUE,FALSE,FALSE),
                   X3 = c("male","male","female"))
data
```

```
##   X1    X2     X3
## 1  1  TRUE   male
## 2  2 FALSE   male
## 3  3 FALSE female
```

The dataframe `data` includes three columns: the first column `X1` of numbers, the second column `X2` of Boolean and the third column `X3` of characters. Dataframes are the objects that are most commonly used in real world data analysis.

2.2.7 `NULL` and `NA`

The expression `NA` is used in R to denote a missing value. Consider the following example.

```
vec <- c(3, NA, 5)
vec
```

```
## [1]  3 NA  5
```

Although the second element of `vec` is the expression `NA`, R recognizes that it is used for missing value and therefore the elements 3 and 5 are still considered numbers: indeed they are not printed as `"3"` and `"5"`.

`NULL` is an additional datatype. This can have various uses. For instance, it is associated to a vector with no entries.

```
c()
```

```
## NULL
```

2.3 Accessing and manipulating variables

Now that we have described the main objects we will work with in R, we can discuss how to access specific information.

2.3.1 Accessing a single element

Given a vector `vec` we can access its i-th entry with `vec[i]`.

```
vec <- c(1,3,5)
vec[2]
```

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

For a matrix or a dataframe we need to specify the associated row and column. If we have a matrix `mat` we can access the element in entry (i,j) with `mat[i,j]`.

```
mat <- matrix(c(1,2,3,4,5,6,7,8,9), ncol=3, nrow =3)
mat[1,3]
```

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

2.3.2 Acessing multiple entries

To access multiple entries we can on the other hand define a vector of indexes of the elements we want to access. Consider the following examples:

```
vec <- c(1,3,5)
vec[c(1,2)]
```

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

The above code accesses the first two entries of the vector `vec`. To do this we had to define a vector using `c(1,2)` stating the entries we wanted to look at. For matrices consider:

```
mat <- matrix(c(1,2,3,4,5,6,7,8,9), ncol=3, nrow =3)
mat[c(1,2),c(2,3)]
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
```

The syntax is very similar as before. We defined to index vectors, one for the rows and one for columns. The two statements `c(1,2)` and `c(2,3)` are separated by a comma to denote that the first selects the first and second row, whilst the second selects the second and third column.

If one wants to access full rows or full columns, the argument associated to rows or columns is left blank. Consider the following examples.

```
mat <- matrix(c(1,2,3,4,5,6,7,8,9), ncol=3, nrow =3)
mat[1,]
```

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

```
mat[,c(1,2)]
```

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

The code `mat[1,]` selects the first full row of `mat`. The code `mat[,c(1,2)]` selects the first and second column of `mat`. Notice that the comma has always to be included!

To access multiple entries it is often useful to define sequences of number quickly. The following command defines the sequence of integer numbers from 1 to 9.

```
1:9
```

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

More generally, one can define sequences of numbers using `seq` (see `?seq`).

2.3.3 Accessing entries with logical operators

If we want to access elements of an object based on a condition it is often easier to use logical operators. This means comparing entries using the comparisons you would usually use in mathematical reasoning, for instance being equal to, or being larger to. The syntax is as follows:

- `==` to check equality (notice the two equal signs)
- `!=` to check non-equality
- `>` bigger to
- `>=` bigger or equal to
- `<` less to
- `<=` less or equal to

Let's see some examples.

```
vec <- c(2,3,4,5,6)
vec > 4
```

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

We constructed a vector `vec` and check which entries were larger than 4. The output is a Boolean vector with the same number of entries as `vec` where only the last two entries are `TRUE`. Similarly,

```
vec <- c(2,3,4,5,6)
vec == 4
```

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

has a `TRUE` in the third entry only.

So if we were to be interested in returning the elements of `vec` that are larger than 4 we could use the code

```
vec <- c(2,3,4,5,6)
vec[vec > 4]
```

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

So we have a vector with only elements 5 and 6.

2.3.4 Manipulating dataframes

We have seen in the previous section that dataframes are special types of matrices where columns can include a different data type. For this reason they have special way to manipulate and access their entries.

First, specific columns of a dataframe can be accessed using its name and the `$` sign as follows.

```
data <- data.frame(X1 = c(1,2,3), X2 = c(TRUE,FALSE,FALSE),
                   X3 = c("male","male","female"))
data$X1
```

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

```
data$X3
```

```
## [1] male   male   female
## Levels: female male
```

So using the name of the dataframe `data` followed by `$` and then the name of the column, for instance `X1`, we access that specific column of the dataframe.

Second, we can use the `$` sign to add new columns to a dataframe. Consider the following code.

```
data <- data.frame(X1 = c(1,2,3), X2 = c(TRUE,FALSE,FALSE),
                   X3 = c("male","male","female"))
data$X4 <- c("yes","no","no")
data
```

```
##   X1    X2     X3  X4
## 1  1  TRUE   male yes
## 2  2 FALSE   male  no
## 3  3 FALSE female  no
```

`data` now includes a fourth column called `X4` coinciding to the vector `c("yes","no","no")`.

Third, we can select specific rows of a dataframe using the command `subset`. Consider the following example.

```
data <- data.frame(X1 = c(1,2,3), X2 = c(TRUE,FALSE,FALSE),
                   X3 = c("male","male","female"))
subset(data, X1 <= 2)
```

```
##   X1    X2   X3
## 1  1  TRUE male
## 2  2 FALSE male
```

The above code returns the rows of `data` such that `X1` is less or equal to 2. More complex rules to subset a dataframe can be combined using the and operator `&` and the or operator `|`. Let's see an example.

```
data <- data.frame(X1 = c(1,2,3), X2 = c(TRUE,FALSE,FALSE),
                   X3 = c("male","male","female"))
subset(data, X1 <= 2 & X2 == TRUE)
```

```
##   X1   X2   X3
## 1  1 TRUE male
```

So the above code selects the rows such that `X1` is less or equal to 2 and `X2` is `TRUE`. This is the case only for the first row of `data`.

2.3.5 Information about objects

Here is a list of functions which are often useful to get information about objects in R.

- `length` returns the number of entries in a vector.
- `dim` returns the number of rows and columns of a matrix or a dataframe
- `unique` returns the unique elements of a vector or the unique rows of a matrix or a dataframe.
- `head` returns the first entries of a vector or the first rows of a matrix or a dataframe
- `order` returns a re-ordering of a vector or a data.frame in ascending order.

Let's see some examples.

```
vec <- c(4,2,7,5,5)
length(vec)
```

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

```
unique(vec)
```

```
## [1] 4 2 7 5
```

```
order(vec)
```

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

`length` gives the number of elements of `vec`, `unique` returns the different values in `vec` (so 5 is not repeated), `order` returns in entry i the ordering of the i-th entry of `vec`. So the first entry of `order(vec)` is 2 since 4 is the second-smallest entry of `vec`.

```
data <- data.frame(X1 = c(1,2,3,4), X2 = c(TRUE,FALSE,FALSE,FALSE),
                   X3 = c("male","male","female","female"))
dim(data)
```

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

So `dim` tells us that `data` has four rows and three columns.

2.4 Loops and conditions

This section reviews two of the most basic elements of any programming language: `if` statements and `cycles` or `loops`.

2.4.1 `if` statements

The basic form of an `if` statement in R is as follows:

```
if(condition){true_action}
```

Condition must return a Boolean, either `TRUE` or `FALSE`. If `TRUE` then the code follows the code within the curly brackets and performs the `true_action`. If `condition` is `FALSE` the code does nothing.

It is more customary to also give a chunk of code for the case `condition` is `FALSE`. This can be achieved with `else`.

```
if(condition){true_action} else {false_action}
```

Let's see an example.

```
a <- 5
if (a < 2){"hello"} else {"goodbye"}
```

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

The variable `a` is assigned the number 5. Then we impose a condition: if `a` is less than 2, we print the text `"hello"`, otherwise `"goodbye"` is printed. Since `a <- 5` the code prints correctly `"goodbye"`. On the other hand if `a` were assigned `1`.

```
a <- 1
if (a < 2){"hello"} else {"goodbye"}
```

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

2.4.2 `ifelse`

`if` works when checking a single element and the condition returns either `TRUE` or `FALSE`. The command `ifelse` can be used to quickly check a condition over all elements of a vector. Consider the following example.

```
vec <- c(1, 3, 5, 7, 9)
ifelse(vec > 5, "bigger", "smaller")
```

```
## [1] "smaller" "smaller" "smaller" "bigger"  "bigger"
```

`vec` contains the values 1, 3, 5, 7, 9 and the `condition` is if an elemenent of `vec` is larger than 5. If `TRUE` the code returns the string `bigger` and otherwise returns `smaller`. The code above returns therefore a vector of the same length of `vec` including either the string `bigger` or the string `smaller`.

2.4.3 Loops

`for` loops are used to iterate over items in a vector. They have the following skeleton:

```r
for(item in vector) {perform_action}
```

For each `item` in `vector`, `perform_action` is performed once and the value of `item` is updated each time.

Here is an example.

```r
for (i in c(1,2,3)){
  print(i)
}
```

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

Item is the variable `i` (it is costumary to use just a letter) and at each step `i` is set equal to a value in the vector `c(1,2,3)`. At each of these iterations, the command `print(i)`, which simply returns the value that `i` takes is called. Indeed we say that the output is the sequence of numbers 1, 2, 3.

2.5 Functions

Functions are chunks of code that are given a name so that they can be easily used multiple times. Perhaps without realising it, you have used functions already many times!

2.5.1 Defining your own function

A function is composed of the following elements:

- a name: in R functions are objects just like vectors or matrices and they are given a name.
- arguments: these are objects that will be used within the function.
- body: a chunk of code which is run within the function.
- output: an object that the function returns.

Let's consider an example.

```r
my.function <- function(x,y){
  z <- x + y
  return(z)
}
```

The above function computes the sum of two numbers `x` and `y`. Let's call it.

```r
my.function(2,3)
```

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

The sum between 2 and 3 is indeed 5.

Let's look at the code line by line. In the first line, we assigned a function using the command `function` to an object called `my.function`. `my.function` has two arguments called `x` and `y`. Then there is an opening curly bracket `{`. The last line of code has a closing curly bracket `}`: whatever is in between the two brackets is a chunk of code which is run when the function is run. The second line computes a new variable called `z` which stores the sum of `x` and `y`. The third line of code tells us that the function should return `z` as output.

Let's consider a slightly more complicated function.

```r
new.function <- function(x,y){
  z1 <- x^2
  z2 <- z1 + y
  return(z2)
}
```

The `new.function` returns the sum between the square of the first input `x` and the second input `y`. Let's call the function.

```r
new.function(2,3)
```

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

```r
new.function(3,2)
```

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

Notice that `new.function(2,3)` is different from `new.function(3,2)`: indeed in the fist case the sum between 2^2 and 3 is computed, whilst in the second the sum between 3^2 and 2 is computed. Furthermore, that the variable `z1` exists only within the function: when you call the function the output does not create a variable `z1`. The output does not create either a variable `z2` it simply returns the value that is stored in `z2`, which can the be assigned as in the following example.

```
value <- new.function(2,3)
value
```

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

We stored in `value` the output of `new.function(2,3)`.

An equivalent way to write `new.function` is as follows:

```
new.function <- function(x,y){
  x^2 + y
}
new.function(2,3)
```

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

The output is the same. We did not create any variable within the function and we did not explicitly use the `return` command. R understands that the last line of code is what the function should return.

2.5.2 Calling functions

In R functions can be called in various ways. Before we have seen function calls as

```
new.function(2,3)
```

How did it work?

- The function `new.function` has a first argument `x` and a second argument `y`.
- R matched the first argument in `new.function(2,3)` to `x`, that is `x=2`, and the second argument to `y`, that is `y=3`.

We could have also been more explicity and state what `x` and `y` were.

```
new.function(x=2, y=3)
```

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

So now explicitly we state that the input `x` of `new.function` is 2 and that the input `y` is 3. Notice that the two ways of specifying inputs give the exact same results.

2.5.3 Mathematical and statistical functions

The number of functions available in R is massive and it would be impossible to mention them all. Here I just give you a list of mathematical and statistical functions that we may use in the following.

- `exp` computes the exponential of the entries of an object
- `log` computes the logarithm of the entries of an object
- `sqrt` computes the square root of the entries of an
- `sum` computes the sum of the entries of an object
- `abs` computes the absolute value of the entries of an object
- `mean` computes the mean of the entries of an object
- `sd` computes the standard deviation of the entries of an object
- `var` computes the variance of the entries of an object

2.6 The `apply` family of functions

One of the biggest limitation of R is that it is slow in performing cycles. For this reason, one should aim at avoiding as much as possible to use of loops.

There are various functions which are designed to help you in avoiding these loops and they are in the family of so called `apply` functions. There are many of these but we will only see two here.

2.6.1 The function `apply`

Consider the following code.

```r
x <- matrix(c(1:9), ncol=3 , nrow = 3)
y <- c()
for (i in 1:3){
  y[i] <- sum(x[i,])
}
y
```

```
## [1] 12 15 18
```

The code first defines a matrix `x` and an empty vector `y` (recall that this is bad practice, but for this example it does not matter). Then there is a `for` cycle which assigns to the i-th entry of `y` the sum of the entries of the i-th row of `x`. So the vector `y` includes the row-totals.

For this simple example the `for` cycle is extremely quick, but this is just to illustrate how we can replace it using the `apply` function.

```r
apply(x, 1, sum)
```

```
## [1] 12 15 18
```

Let's look at the above code. The first input of `apply` is the object we want to operate upon, in this case the matrix `x`. The second input specifies if the operation has to act over the rows of the matrix (input equal to 1) or over the columns (input equal to 2). The third input is the operation we want to use, in this case `sum`.

Beside being faster, the above code is also a lot more compact than using a for loop.

The following example computes the mean of each column of `x`.

```r
apply(x, 2, mean)
```

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

2.6.2 The function `sapply`

Consider again our function `new.function` which computes the sum of the squared of a number `x` with another number `y`.

```r
new.function <- function(x,y){ x^2 + y}
```

Suppose that we want to compute such a sum for all numbers `x` from 1 to 10. Suppose that `y` is chosen as 2. We can achieve this with a `for` cycle as follows.

```
x <- 1:10
z <- c()
for (i in 1:10){
  z[i] <- new.function(x[i],2)
}
z
```

```
##  [1]   3   6  11  18  27  38  51  66  83 102
```

The function `sapply` can be used for this specific purpose.

```
x <- 1:10
sapply(x,new.function, y=2)
```

```
##  [1]   3   6  11  18  27  38  51  66  83 102
```

The first argument of `sapply` is a vector of values we want to use as input of a function. The second argument is the function we want to apply multiple times. If the function has more than one input we can then specify what their value is, in this specific case `y=2`.

Notice that a function can also be defined within `sapply`.

```
x <- 1:10
sapply(x, function(i) i^2 + 2)
```

```
##  [1]   3   6  11  18  27  38  51  66  83 102
```

So we defined the vector `x` and we want to apply the function defined within `sapply` multiple times: once for each entry in the vector `x`.

2.7 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.

2.8 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

We describe our methods in this chapter.