Simulation and Modelling to Understand Change

Manuele Leonelli

2021-04-19

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.

```
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.

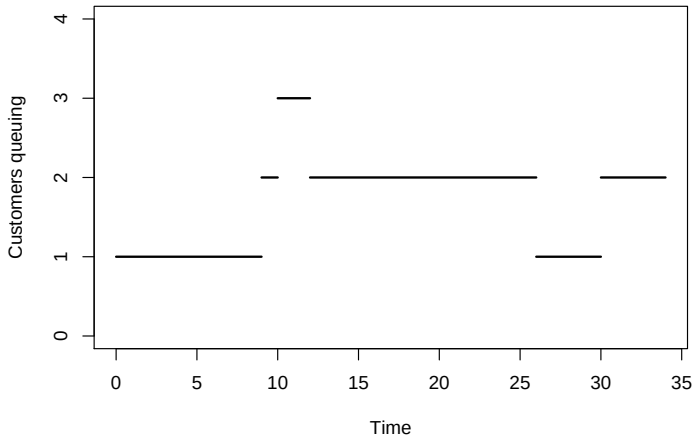


Figure 1.1: Example of a discrete dynamic simulation

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.

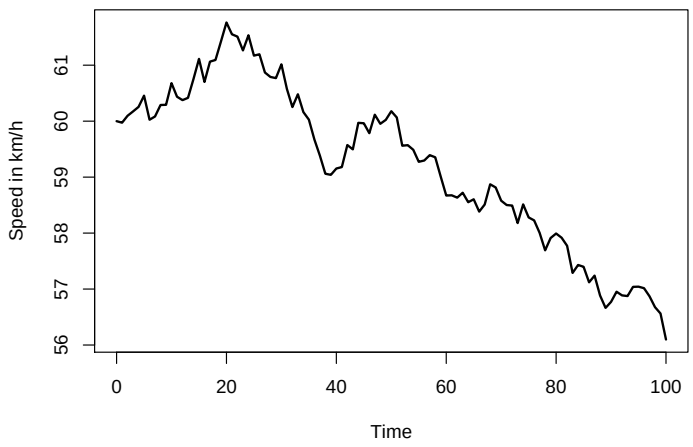


Figure 1.2: Example of a discrete dynamic simulation

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.

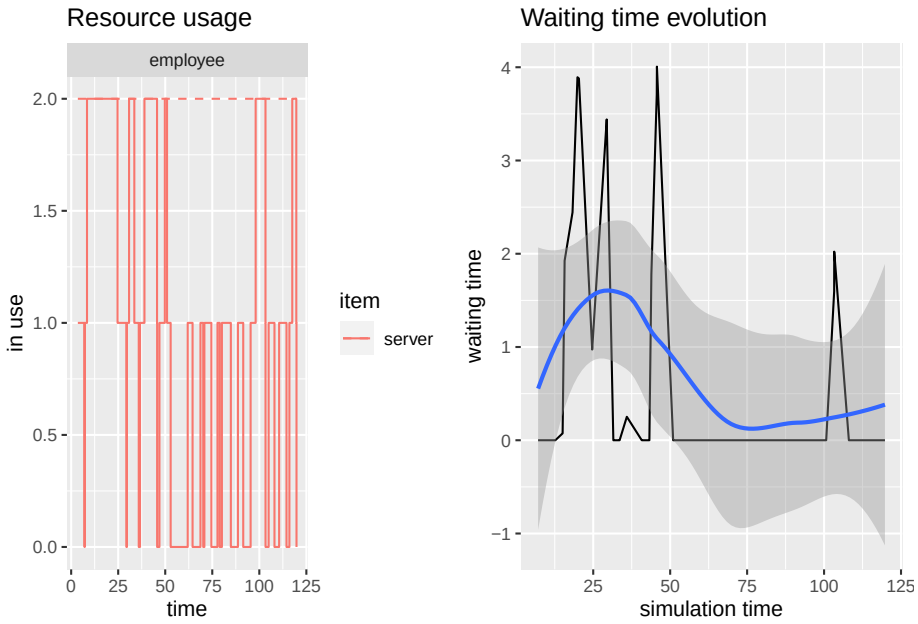


Figure 1.3: Graphical summaries from the simulation of the donut shop

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!

```
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")
```

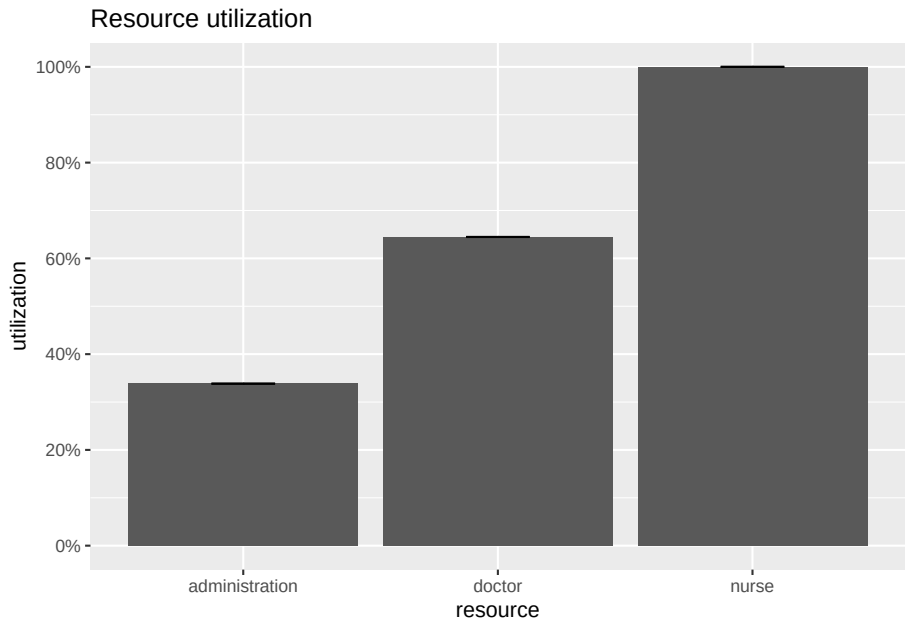


Figure 1.4: Utilization of the resources in the health center

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

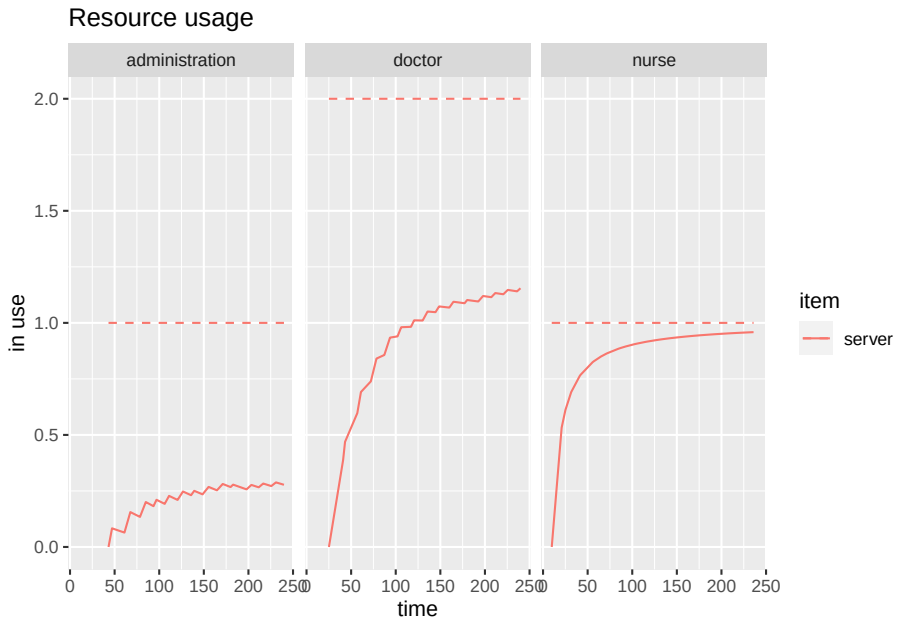


Figure 1.5: Usage of the resources in the health center

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;

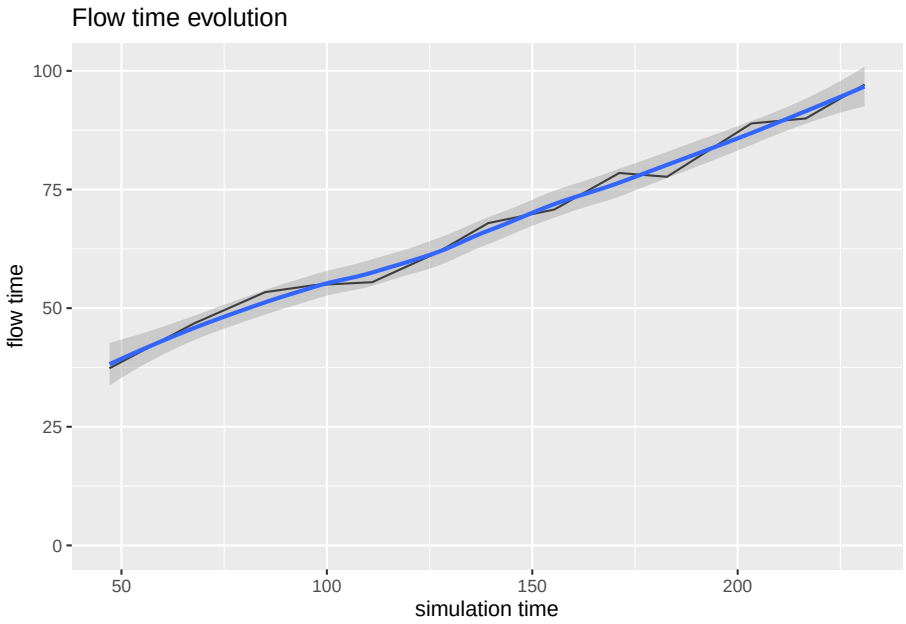


Figure 1.6: Time spent in the health center

- 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 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:

```
4 + 2
```

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

```
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:

```
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.

```
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 see 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.

```
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.

```
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.

```
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.

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

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

```
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 explicit 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.

```
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.

```
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`.

```
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`.

```
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

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.

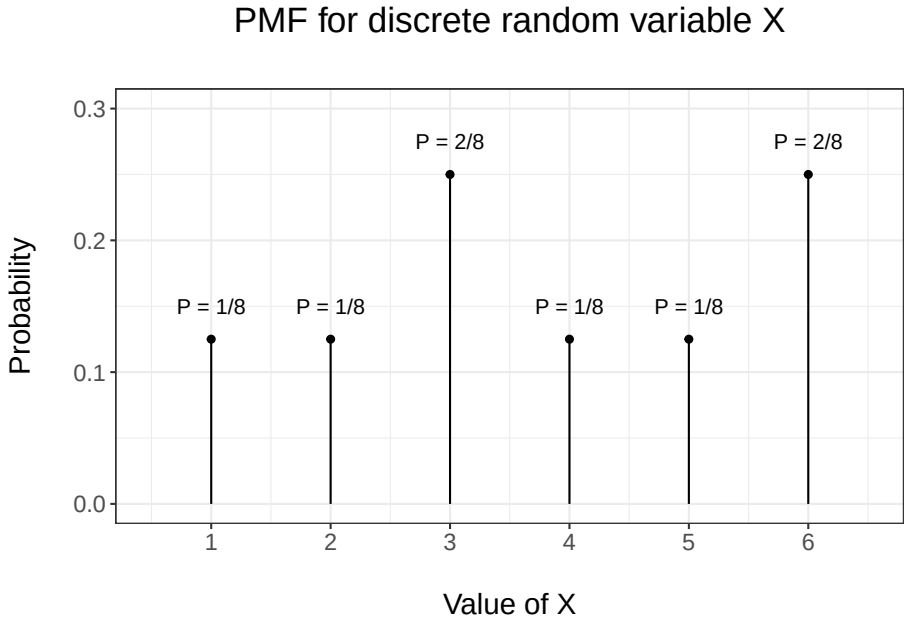


Figure 3.1: PMF for the biased dice example

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.

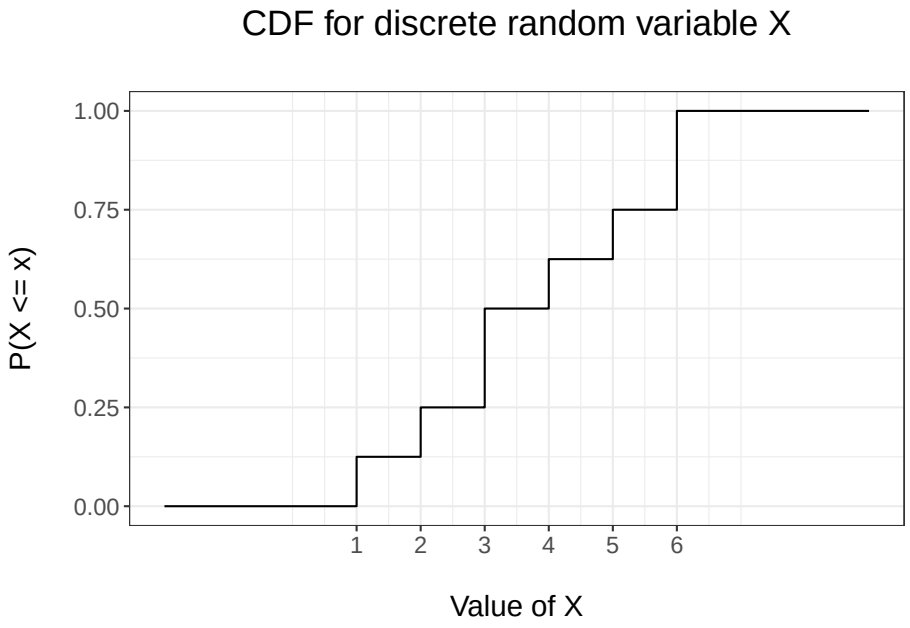


Figure 3.2: CDF for the biased dice example

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.

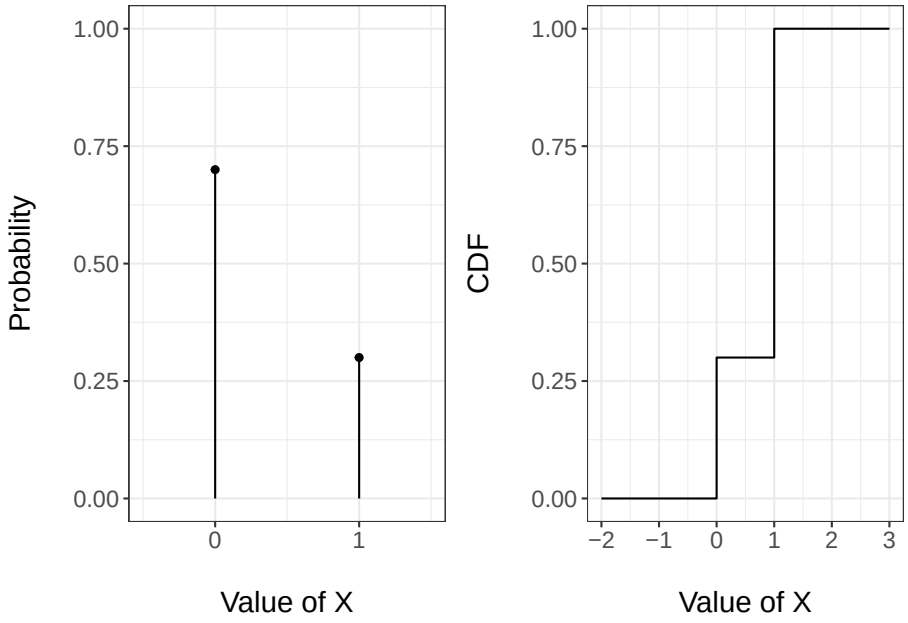


Figure 3.3: PMF (left) and CDF (right) of a Bernoulli random variable with parameter 0.3

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:

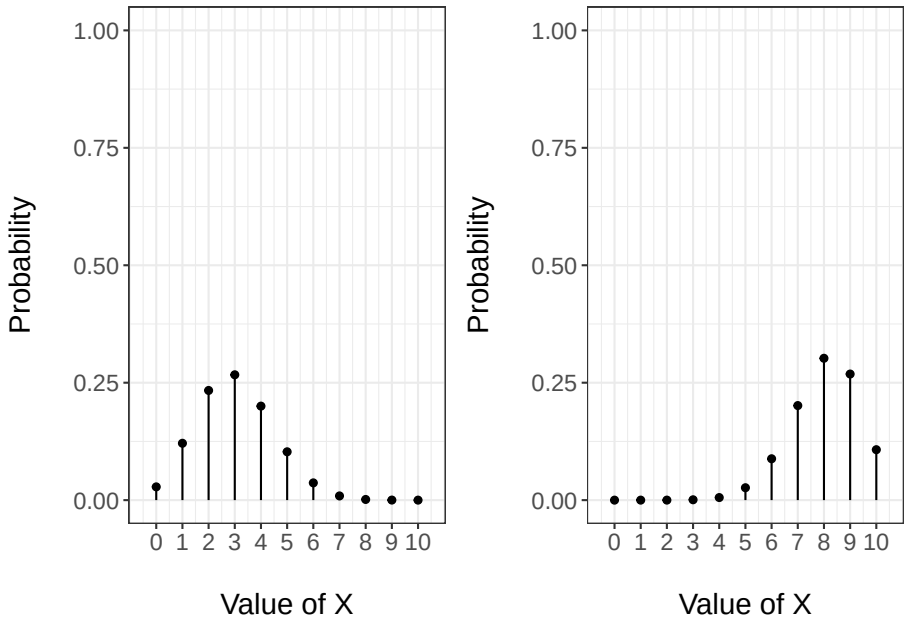


Figure 3.4: PMF of a Binomial random variable with parameters n = 10 and theta = 0.3 (left) and theta = 0.8 (right)

- 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.

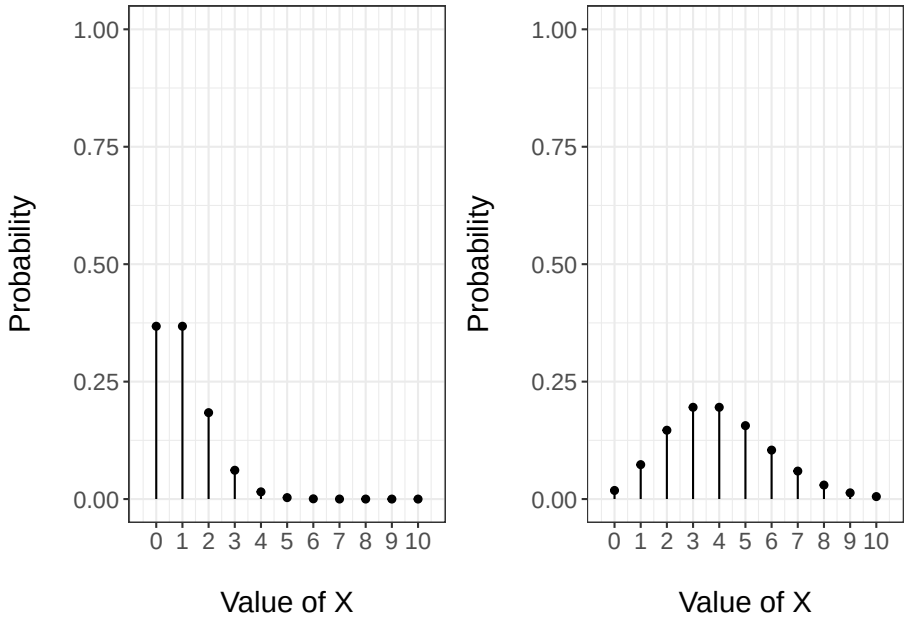


Figure 3.5: PMF of a Poisson random variable with parameter 1 (left) and 4 (right)

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 $\int_{-1}^{1} x^2 dx$.

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 $p(x) = P(X = x)$, that is the value of that function in the point x is exactly the probability that $X = x$.

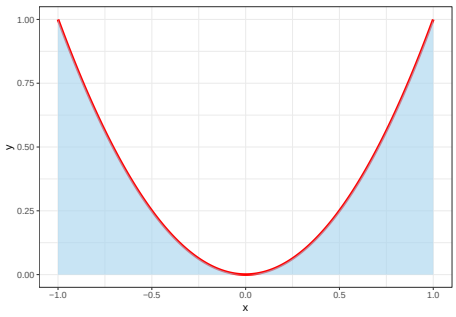


Figure 3.6: Plot of the squared function and the area under its curve

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

$$P(2 < X < 5) = \int_2^5 f(x)dx = \int_2^5 \frac{1}{4} e^{-x/4} dx = 0.32$$

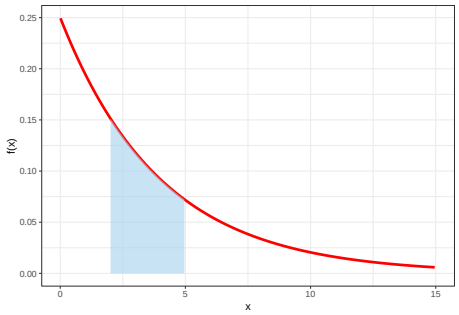


Figure 3.7: Probability density function for the waiting time in the donut shop example

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.

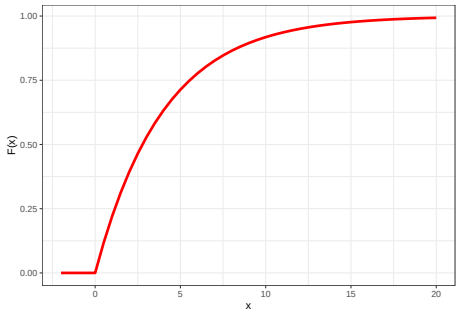


Figure 3.8: Cumulative distribution function for the waiting time at the donut shop

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 $b = 6$

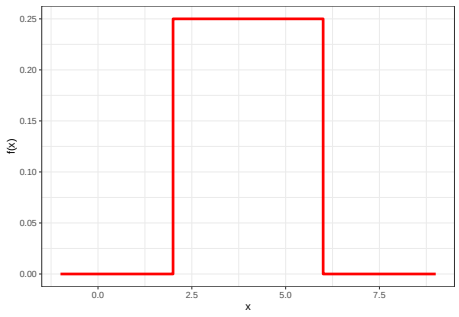


Figure 3.9: Probability density function for a uniform random variable with parameters a = 2 and b = 6

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.

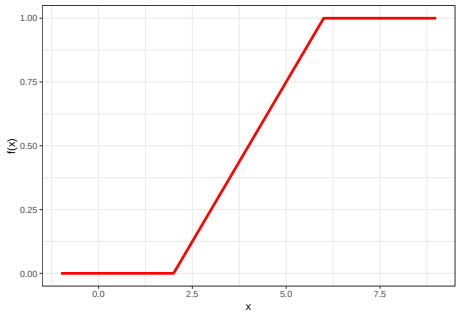


Figure 3.10: Cumulative distribution function for a uniform random variable with parameters a = 2 and b = 6

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.

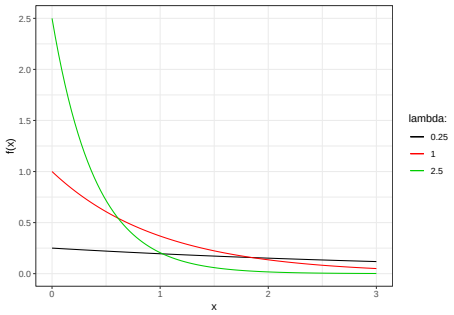


Figure 3.11: Probability density function for exponential random variables

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

$$F(x) = \begin{cases} 0, & x < 0 \\ 1 - e^{-\lambda x}, & x \geq 0 \end{cases}$$

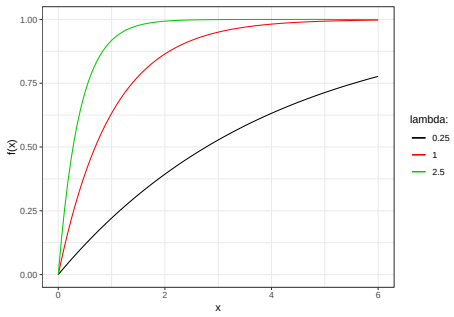


Figure 3.12: Cumulative distribution function for exponential random variables

and is reported in Figure 3.12 for the same choices of parameters.

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.

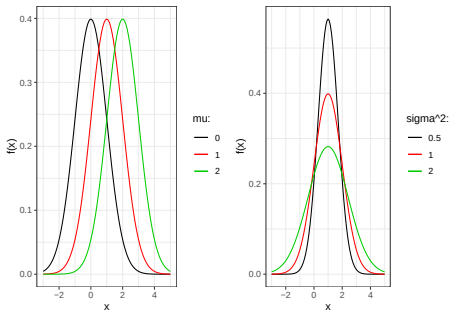


Figure 3.13: Probability density function for normal random variables

The form of the Normal pdf is the well-known so-called bell-shaped function. We can notice some properties:

- 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.

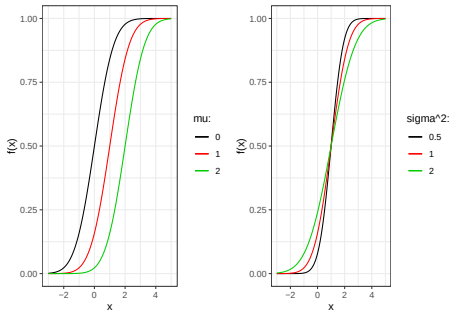


Figure 3.14: Cumulative distribution function for normal random variables

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.

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.

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

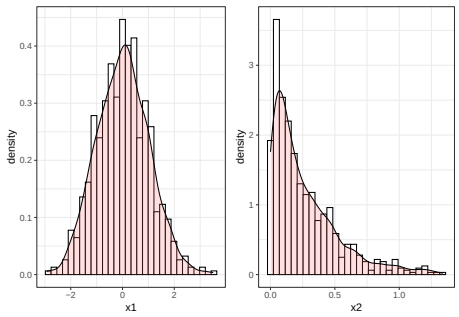


Figure 4.1: Histograms of two sequences of randomly generated numbers

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.

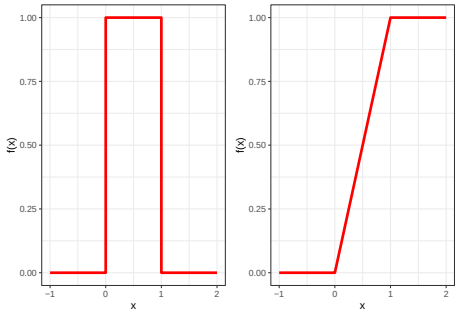


Figure 4.2: Pdf (left) and cdf (right) of the continuous uniform between zero and one.

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.

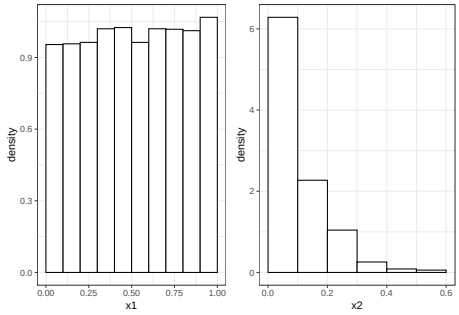


Figure 4.3: Histograms from two sequences of numbers between zero and one.

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 $u_1, \dots, u_N$ 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)
```

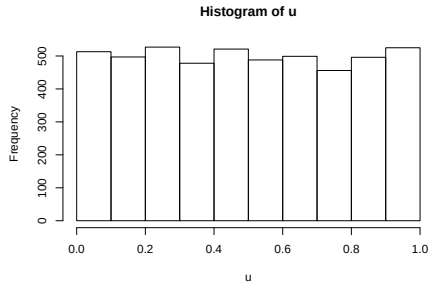


Figure 4.4: Histogram of a sequence of uniform numbers

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:

$$H_0 : \ u_i \text{ is uniform between zero and one, } i = 1, 2, ...$$
$$H_a : \ u_i \text{ is not uniform between zero and one, } i = 1, 2, ...$$

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")
```

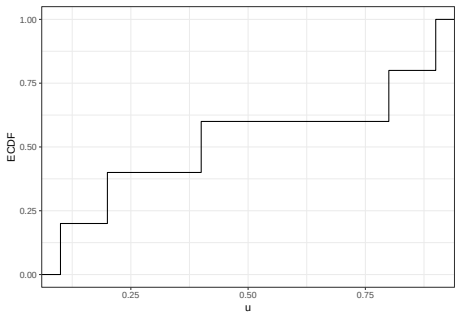


Figure 4.5: Ecdf of a simple sequence of numbers

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.

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
##
```

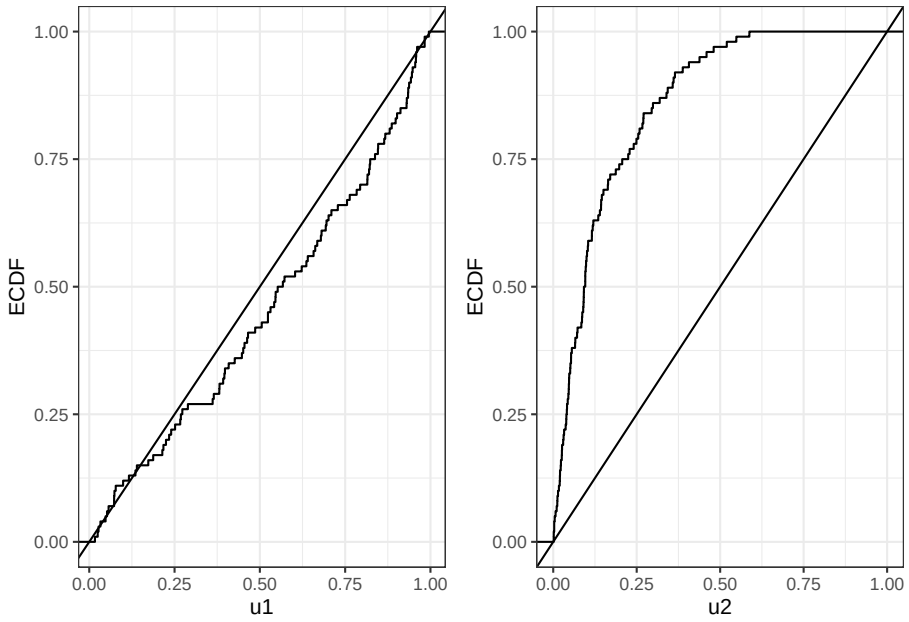


Figure 4.6: Comparison between ecdf and cdf of the uniform for two sequences of numbers

```
## 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 $u_1, \ldots, u_N$. So for instance we could look at pairs $(u_1, u_2), (u_2, u_3), \ldots, (u_{N-1}, u_N)$ which consider two consecutive numbers and compute their correlation. Similarly we could compute the correlation between $(u_1, u_{1+k}), (u_2, u_{2+k}), \ldots, (u_{N-k}, u_N)$ 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)
```

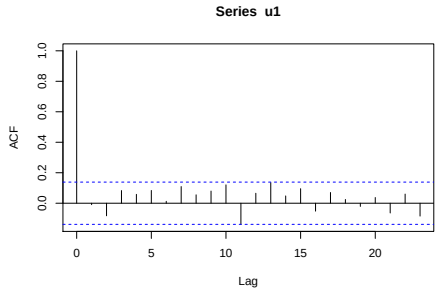


Figure 4.7: Autocorrelations for a sequence of random uniform numbers

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)
```

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.

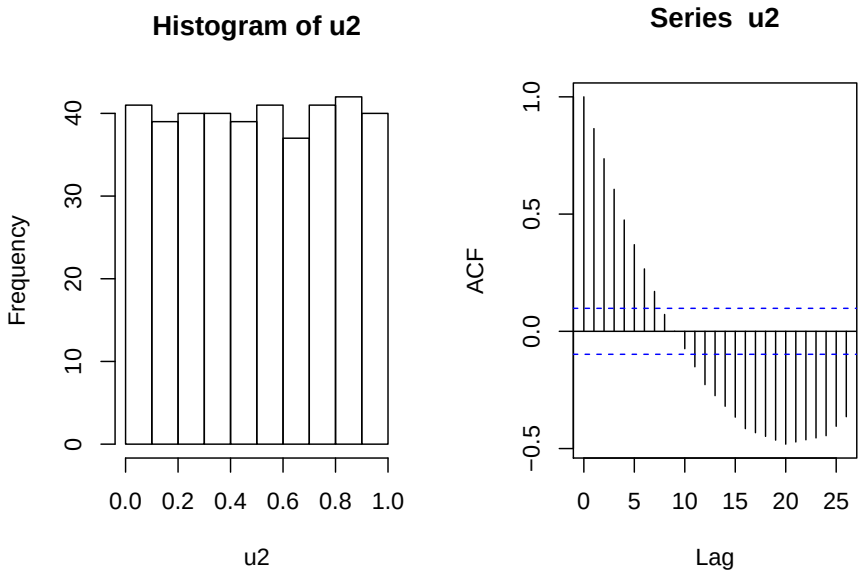


Figure 4.8: Histogram and autocorrelations of a sequence of uniform numbers which are not independent

```
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 $x_1, x_2, \dots, x_N$ 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.

```
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()
```

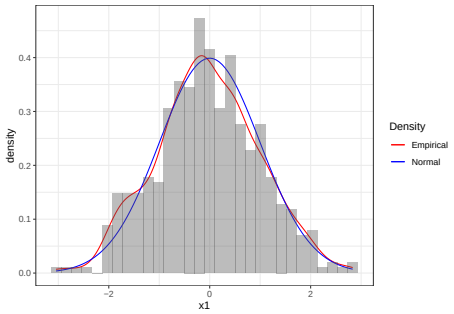


Figure 4.9: Histogram of the sequence x1 together with theoretical pdf of the standard Normal

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.

```
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()
```

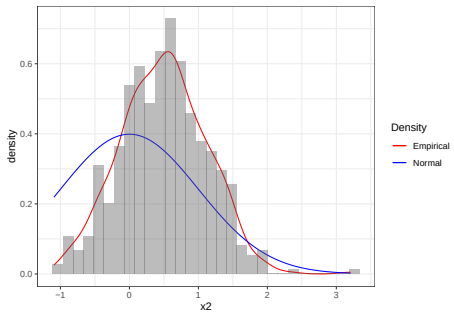


Figure 4.10: Histogram of the sequence x2 together with theoretical pdf of the standard Normal

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 4.12 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.

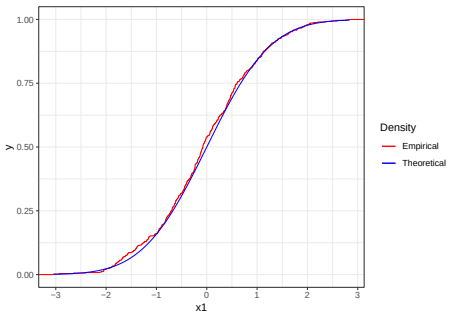


Figure 4.11: Empirical cdf the sequence x1 together with theoretical cdf of the standard Normal

```
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'))
```

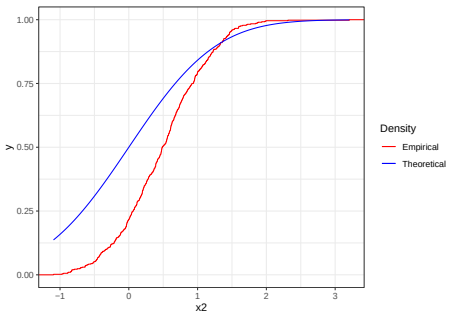


Figure 4.12: Empirical cdf the sequence x2 together with theoretical cdf of the standard Normal

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

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()
```

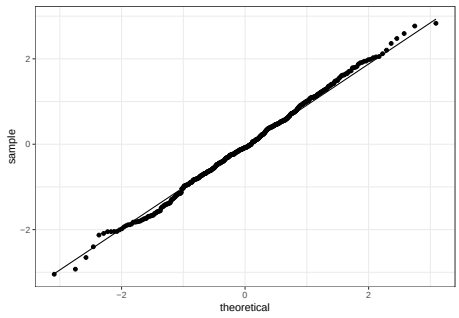


Figure 4.13: QQ-plot for the sequence x1 checking against the standard Normal

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()
```

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.

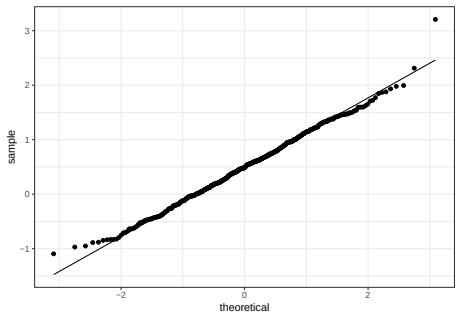


Figure 4.14: QQ-plot for the sequence x2 checking against the standard Normal

```
ggplot(x1, aes(sample = x1)) +
   stat_qq(distribution = qexp, dparams = (rate = 3)) +
   stat_qq_line(distribution = qexp, dparams = (rate = 3)) +
   theme_bw()
```

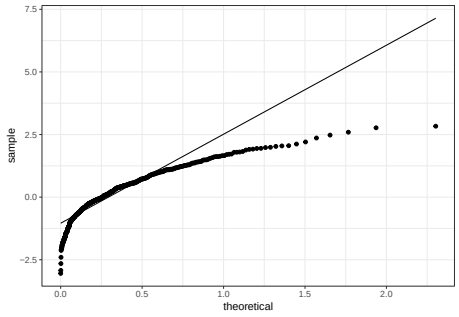


Figure 4.15: QQ-plot for the sequence x1 checking against an Exponential

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)
```

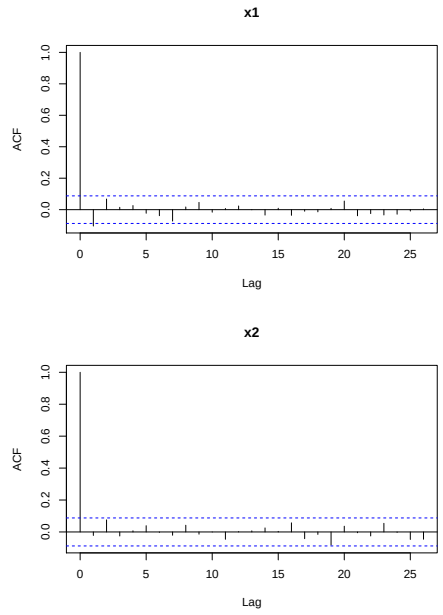


Figure 4.16: Autocorrelations for the sequences x1 and x2.

Let's run the Box test to assess if the assumption of independence is tenable for both sequences.

```
Box.test(x1, lag = 5)
```

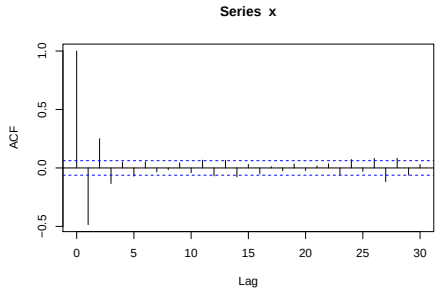
```
##
##  Box-Pierce test
##
## data:  x1
## X-squared = 8.4212, df = 5, p-value = 0.1345
```

```
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

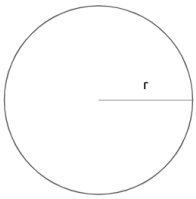
Monte Carlo methods vary, but tend to follow a particular pattern:

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

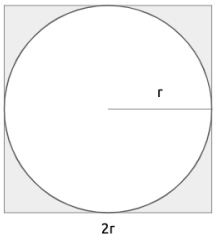
To better understand how Monte Carlo simulation works we will develop a classic experiment: The π number estimation.

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.

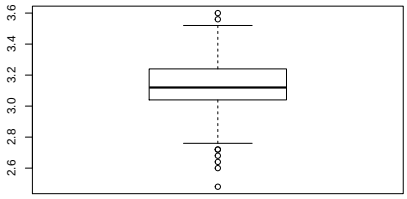
```
mean(pis)
```

```
## [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 = T)
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.

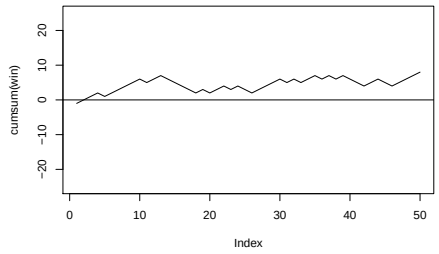


Figure 5.1: Peter's fortune throughout one simulated game.

```
set.seed(2021)
par(mfrow=c(2, 2))
for(j in 1:4){
  plot(cumsum(sample(c(-1, 1),size=50,replace=T)),type="l" ,ylim=c(-25, 25), ylab="Outcome")
  abline(h=0)}
```

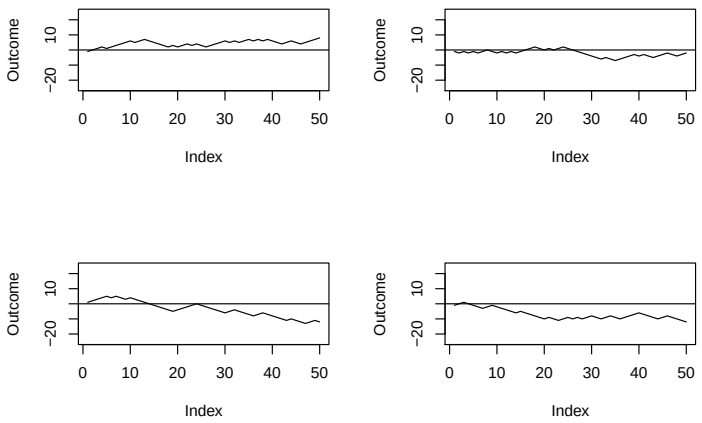


Figure 5.2: Outcome of four simulated game of chances.

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
```

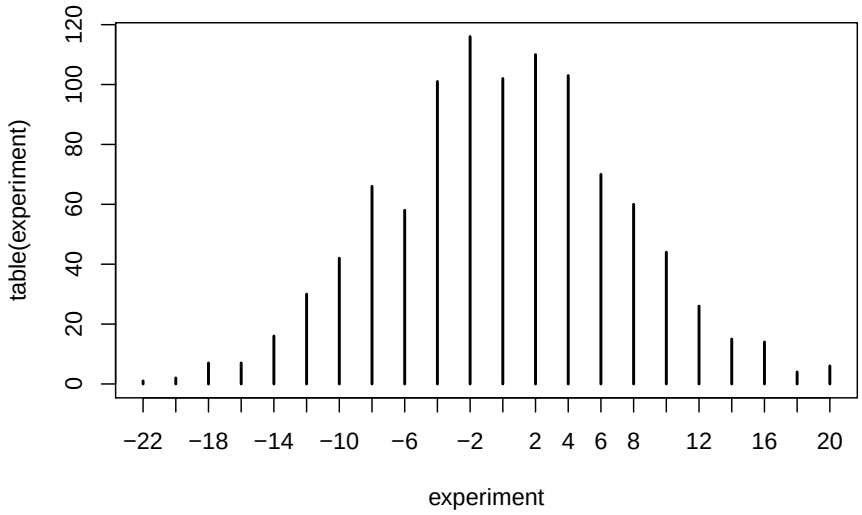
So the vector `experiment` stores Peter's final fortune in 1000 games. Since Peter's fortune is an integer-value variable, it is convenient to summarize it using the `table` function.

```
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 Playing the roulette

Consider a European-style roulette which includes numbers between 0 and 36. In roulette there are many different betting alternatives but we consider here the simplest case where we bet on a single number. If we pick the right number then we win 35 times what we bet.

Consider the following scenario. We start playing roulette with a fixed `budget`. Every sping of the roulette costs one unit of `budget`. We play until we run out of budget and we always bet on the same number. The following function `roulette` implements this game.

```
roulette <- function(budget, number){
  current <- c()
  while(budget > 0){
    outcome <- sample(0:36,1)
    if(outcome == number) {
      budget <- budget + 35
    }
    else {budget <- budget -1}
    current <- c(current,budget)
  }
  current
}
```

It takes two inputs: the `budget` and the number we decided to play all the time. It outputs our budget throughout the whole game until it ends.

Let's play one game with a budget of 15 and betting on the number 8.

```
set.seed(2021)
roulette(15,8)
```

```
##  [1] 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

In 15 spins the number 8 never comes up and therefore our game ends quite quickly.

We can ask ourselves many questions about such a game. For instance:

1. What is the probability that the game last exactly 15 spins if I start with a budget of 15 and bet on the number 8?

2. What is the average length of the game starting with a budget of 15 and betting on the number 8?

3. What is the average maximum wealth I have during a game started with a budget of 15 and betting on the number 8?

We will develop various Monte Carlo experiments to answer all the above questions. In each case, we simply need to modify the function `roulette` to output some summary about the game.

5.6.1 Question 1

1. What is the probability that the game last exactly 15 spins if I start with a budget of 15 and bet on the number 8?

If a game where I started with a budget of 15 ends after 15 spins, it means that the number 8 never showed up. We can adapt the function roulette to output `TRUE` if the length of the vector `wealth` is exactly equal to `budget`.

```
roulette1 <- function(budget, number){
  wealth <- c()
  current <- budget
  while(current > 0){
    outcome <- sample(0:36,1)
    if(outcome == number) {
      current <- current + 35
    }
    else {current <- current -1}
    wealth <- c(wealth,current)
  }
  if(length(wealth) == budget){return(TRUE)} else{return(FALSE)}
}
```

Therefore for the previous example the function should output `TRUE`.

```
set.seed(2021)
roulette1(15,8)
```

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

Let's replicate the experiment 1000 times. The proportion of `TRUE` we observe is our estimate of this probability.

```
set.seed(2021)
experiment <- replicate(1000,roulette1(15,8))
sum(experiment)/1000
```

```
## [1] 0.657
```

Notice that actually we could have also computed this probability exactly. This is the probability that a Binomial random variable with parameter $n = 15$ (15 spins of the roulette) and $\theta = 1/37$ (the number eight has probability $1/37$ of appearing in a single spin) is equal to zero (no 8 can happen). This is equal to:

```
dbinom(0,15,1/37)
```

```
## [1] 0.6629971
```

Therefore the Monte Carlo experiment approximates well the probability.

5.6.2 Question 2

2. What is the average length of the game starting with a budget of 15 and betting on the number 8?

We can answer this question by adapting the `roulette` function to output the length of the vector `wealth`.

```
roulette2 <- function(budget, number){
  wealth <- c()
  current <- budget
  while(current > 0){
    outcome <- sample(0:36,1)
    if(outcome == number) {
      current <- current + 35
    }
    else {current <- current -1}
    wealth <- c(wealth,current)
  }
  length(wealth)
}
```

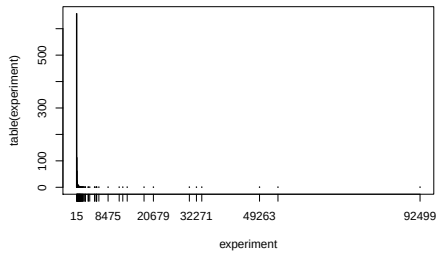
In our example, the output should be 15.

```
set.seed(2021)
roulette2(15,8)
```

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

Let's replicate the experiment 1000 times and summarize the results with a plot (it may take some time to run the code!).

```
set.seed(2021)
experiment <- replicate(1000,roulette2(15,8))
plot(table(experiment))
```



We can see that the distribution is very skewed, most often the length is 15 spins, but then sometimes the game has a length which is much longer. Therefore, the median of the data is a much better option to summarize the average length of the game. We can compute it, together with a range of plausible values, as

```
median(experiment)
```

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

```
c(sort(experiment)[25],sort(experiment)[975])
```

```
## [1]   15 1923
```

So we can see that the median is indeed 15: most of the times the number 8 does not appear and therefore the game ends in 15 spins.

5.6.3 Question 3

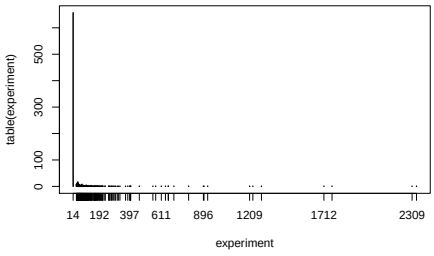
3. What is the average maximum wealth I have during a game started with a budget of 15 and betting on the number 8?

We can answer this question by adapting the `roulette` function to output the maximum of the vector `wealth`.

```
roulette3 <- function(budget, number){
  wealth <- c()
  current <- budget
  while(current > 0){
    outcome <- sample(0:36,1)
    if(outcome == number) {
      current <- current + 35
    }
    else {current <- current -1}
    wealth <- c(wealth,current)
  }
  max(wealth)
}
```

Let's again replicate the experiment and plot the result.

```
set.seed(2021)
experiment <- replicate(1000,roulette3(15,8))
plot(table(experiment))
```



We can then get an estimate of the maximum wealth as well as a range of plausible values as

```
median(experiment)
```

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

```
c(sort(experiment)[25],sort(experiment)[975])
```

```
## [1]  14 331
```

Since most often the game ends with no 8 appearing, the maximum wealth is most often 14.

5.7 Is Annie meeting Sam?

Sleepless in Seattle is a 1993 American Romantic Comedy. Annie and Sam are supposed to meet on top of the Empire State Building where they would at last meet. However, their arrival time depends on a variety of factors and may actually not coincide and therefore never fall in love.

We make the following assumptions:

- Sam and Annie arrive independently;
- Sam arrives between 10 pm and 11:30 pm uniformly;
- Annie arrives between 10:30 pm and midnight uniformly.

We want to answer the following questions:

- What is the probability that Annie arrives before Sam?
- What is the expected difference in arrival time?
- If they each wait only twenty minutes after their arrival, what is the probability that they meet?
- How much should they wait (assuming they wait the same amount of time), so that the probability they meet is at least 50%?

5.7.1 Question 1

- What is the probability that Annie arrives before Sam?

The following function outputs `TRUE` if Annie arrives before Sam and `FALSE` otherwise.

```
sam_annie1 <- function(){
  sam <- runif(1,0,90)
  annie <- runif(1,30,120)
  annie < sam
}
```

If we run a single simulation for instance

```
set.seed(2021)
sam_annie1()
```

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

We see that Annie does not arrive before Sam.

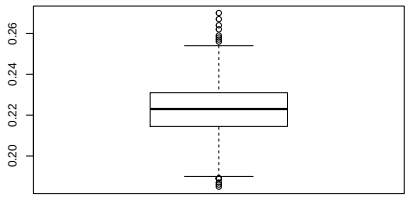
We can simply adapt `sam_annie1` to use 1000 random observations and give an estimate of this probability.

```
sam_annie1 <- function(){
  sam <- runif(1000,0,90)
  annie <- runif(1000,30,120)
  sum(annie < sam)/1000
}

set.seed(2021)
sam_annie1()
```

```
## [1] 0.223
```

If we want to have an estimate of the variability around this probability we can then replicate the experiment.

```
set.seed(2021)
experiment <- replicate(1000, sam_annie1())
boxplot(experiment)
```



The probability that Annie arrives before Sam is around 0.22.

5.7.2 Question 2

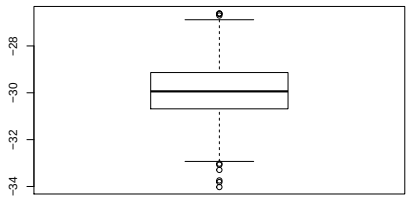
- What is the expected difference in arrival time?

The function `sam_annie2` computes this expectation using 1000 random observations.

```
sam_annie2 <- function(){
  sam <- runif(1000,0,90)
  annie <- runif(1000,30,120)
  mean(sam-annie)
}
```

We can replicate and create a boxplot.

```
set.seed(2021)
experiment <- replicate(1000,sam_annie2())
boxplot(experiment)
```



So on average Sam arrives 30 minutes before Annie.

5.7.3 Question 3

- If they each wait only twenty minutes after their arrival, what is the probability that they meet?

The function `sleepless` below returns `TRUE` if Annie and Sam meets,`FALSE` otherwise.

```
sleepless <- function(waiting = 20){
  u1 <- runif(1,0,150)
  u2 <- runif(1,50,200)
  if (u1 < u2){if(u1 + waiting > u2) return(TRUE)}
  else{if (u2 + waiting > u1) return(TRUE)}
  return(FALSE)
}
```

Let's run it 10.000 times to get an estimate of the probability.

```
set.seed(2021)
mean(replicate(10000,sleepless()))
```

```
## [1] 0.178
```

So we see that if they each wait 20 minutes, they have a probability of meeting of 0.178.

5.7.4 Question 4

- How much should they wait (assuming they wait the same amount of time), so that the probability they meet is at least 50%?

In order to answer this question we can use the `sapply` function. We already know that if they wait 20 minutes, the probability of meeting is 0.19. So we consider longer waiting times.

```
sapply(30:60,function(x) mean(replicate(10000,sleepless(x))))
```

```
##  [1] 0.2754 0.2769 0.2811 0.2925 0.3055 0.3104 0.3147 0.3248 0.3400 0.3482
## [11] 0.3571 0.3605 0.3746 0.3869 0.3958 0.4034 0.4104 0.4184 0.4237 0.4283
## [21] 0.4444 0.4653 0.4664 0.4763 0.4831 0.4846 0.4929 0.5067 0.5091 0.5222
## [31] 0.5312
```

It's the 28th entry of the vector `30:60`, that is 57, the first one for which the probability is at least 0.50. So they should wait 57 minutes if they want to have a probability of at least 50% to actually meet.

Chapter 6

Discrete Event Simulation

The complexity of many real-world systems involves unaffordable analytical models, and consequently, such systems are commonly studied by means of simulation. Different types of simulation apply depending on the nature of the system under consideration. In the last chapter we focused on static simulation, also called Monte Carlo simulation. In this chapter we discuss discrete-event simulation (DES) which 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.

To implement DES in R we will take advantage of the `simmer` package which provides capabilities to easily model a wide range of applications.

6.1 The donut shop

You may recall that in Chapter 1 we discussed the example of a simple donut shop where we were interested in the waiting time of costumers depending on the number of employees in the shop. We will slowly build a more and more realistic implementation of the shop using `simmer`. First, if you have never done this, you need to install the package using the code

```
install.packages("simmer")
```

Once it is installed (that you only need to do once), you then need to load it at the beginning of every R session using the code

```
library(simmer)
```

As we will see `simmer` takes advantage of the `magrittr` pipe operator.

6.1.1 A single customer at a fixed time

We first model a single customer who arrives at the shop for a visit, looks around at the decor for a time and then leaves. There is no queueing. First we will assume his arrival time and the time he spends in the shop are fixed.

The arrival time is fixed at 5, and the time spent in the shop is fixed at 10. We interpret '5' and '10' as '5 minutes' and '10 minutes'. The simulation runs for a maximum of 100 minutes, or until all the customers that are generated complete their visit to the shop.

Let's define step by step the code that implements this. First we define a variable called `customer` which describes the evolution of the customer in the shop. The evolution of the customer is called the `trajectory` and it is given some name. The function `log_` produces text which is shown as specific points during the simulation. Lastly, `timeout` specifies how long the customer will spend in the shop.

```
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  timeout(10) %>%
  log_("I must leave")
```

Now that we have described how the customer behaves in the shop, we must create a variable specifying how the shop itself works. The code below creates the variable `shop` where first we create a `simmer` object called `shop` and then we specify via the function `add_generator` that a customer arrives after 5 minutes.

```
shop <-
  simmer("shop") %>%
  add_generator("Customer", customer, at(5))
```

Now that the shop is created we have to run the simulation using the `run` command.

```
shop %>% run(until = 100)
```

```
## 5: Customer0: Here I am
## 15: Customer0: I must leave
```

```
## simmer environment: shop | now: 15 | next:
## { Monitor: in memory }
## { Source: Customer | monitored: 1 | n_generated: 1 }
```

The output summarizes what happened during the simulation. A customer arrives at minute 5 and leaves at minute 15. The simulation then stops since there are no other events that can happen in this simple case.

In order to have an overview of the events of the simulation we can also use the function `get_mon_arrivals`.

```
shop %>% get_mon_arrivals()
```

```
##        name start_time end_time activity_time finished replication
## 1 Customer0          5       15            10     TRUE           1
```

6.1.2 A customer arriving at random

Now we extend the model to allow our customer to arrive at a random simulated time though we will keep the time in the bank at 10, as before.

The change occurs in the arguments to the `add_generator` function. We will assume that the customer arrival time is generated from an Exponential distribution with parameter 1/5 (that is mean = 5).

```
set.seed(2021)

customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  timeout(10) %>%
  log_("I must leave")

shop <-
  simmer("shop") %>%
  add_generator("Customer", customer, at(rexp(1, 1/5)))

shop %>% run(until = 100)
```

```
## 5.92531: Customer0: Here I am
## 15.9253: Customer0: I must leave

## simmer environment: shop | now: 15.925307087372 | next:
## { Monitor: in memory }
## { Source: Customer | monitored: 1 | n_generated: 1 }
```

```
shop %>% get_mon_arrivals()
```

```
##        name start_time end_time activity_time finished replication
## 1 Customer0   5.925307 15.92531            10     TRUE           1
```

The trace shows that the customer now arrives at time 5.925307. Changing the seed value would change that time.

6.1.3 Many random customers

We now extend this model to allow multiple arrivals at random. In simulation this is usually interpreted as meaning that the times between customer arrivals are distributed as exponential random variables (we will see later on why this is so). There is little change in our program: we need to adapt the `add_generator` function. We will run the simulation a shorter time, since otherwise the output becomes massive.

```
set.seed(2021)

customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  timeout(10) %>%
  log_("I must leave")

shop <-
  simmer("shop") %>%
  add_generator("Customer", customer, function() rexp(1, 1/5))

shop %>% run(until = 30)
```

```
## 5.92531: Customer0: Here I am
## 12.0259: Customer1: Here I am
## 13.3892: Customer2: Here I am
## 15.4026: Customer3: Here I am
## 15.9253: Customer0: I must leave
```

```
## 16.807: Customer4: Here I am
## 20.6063: Customer5: Here I am
## 22.0259: Customer1: I must leave
## 23.3892: Customer2: I must leave
## 23.7606: Customer6: Here I am
## 23.8551: Customer7: Here I am
## 24.5296: Customer8: Here I am
## 25.4026: Customer3: I must leave
## 26.807: Customer4: I must leave
## 27.7301: Customer9: Here I am

## simmer environment: shop | now: 30 | next: 30.6063456850162
## { Monitor: in memory }
## { Source: Customer | monitored: 1 | n_generated: 11 }
```

```
shop %>% get_mon_arrivals()
```

```
##        name start_time end_time activity_time finished replication
## 1 Customer0   5.925307 15.92531            10     TRUE           1
## 2 Customer1  12.025928 22.02593            10     TRUE           1
## 3 Customer2  13.389166 23.38917            10     TRUE           1
## 4 Customer3  15.402626 25.40263            10     TRUE           1
## 5 Customer4  16.807015 26.80702            10     TRUE           1
```

So we see that in the 30 minutes we simulated, 10 customers arrived at the shop and 5 of them left after staying there for 10 minutes.

6.1.4 The donut shop with a service counter

So far, the model has been more like an art gallery, the customers entering, looking around, and leaving. Now they are going to require service from an employee. We extend the model to include a service counter that will be modelled as a 'resource'. The actions of a Resource are simple: a customer requests a unit of the resource (an employee). If one is free, then the customer gets service (and the unit is no longer available to other customers). If there is no free employee, then the customer joins the queue until it is the customer's turn to be served. As each customer completes service and releases the unit, the employee can start serving the next in line.

The service counter is created with the `add_resource` function. Default arguments specify that it can serve one customer at a time, and has infinite queuing capacity.

The `seize` function causes the customer to join the queue at the counter. If the queue is empty and the counter is available (not serving any customers),

then the customer claims the counter for itself and moves onto the `timeout` step. Otherwise the customer must wait until the counter becomes available. Behaviour of the customer while in the queue is controlled by the arguments of the `seize` function. Once the `timeout` step is complete, the `release` function causes the customer to make the counter available to other customers in the queue.

We will assume that serving time follows a Normal distribution with mean 10 and standard deviation 2.

```
set.seed(2021)

customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  seize("counter") %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter") %>%
  log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter") %>%
  add_generator("Customer", customer, function() rexp(1, 1/5))

shop %>% run(until = 30)
```

```
## 5.92531: Customer0: Here I am
## 12.0259: Customer1: Here I am
## 13.4303: Customer2: Here I am
## 16.6226: Customer0: Finished
## 17.2296: Customer3: Here I am
## 28.4187: Customer1: Finished
```

```
## simmer environment: shop | now: 30 | next: 36.6490003143151
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 1(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 5 }
```

So we see that 4 customers arrived in the shop and that 2 of them were served. Let's use the function `get_mon_arrivals` to have a summary of each customer. By default, the function does not tell us the waiting time for a customer, which we will need to compute.

```
shop %>%
  get_mon_arrivals() %>%
  transform(waiting_time = end_time - start_time - activity_time)
```

```
##        name start_time end_time activity_time finished replication
## 1 Customer0   5.925307 16.62261      10.69730     TRUE           1
## 2 Customer1  12.025928 28.41871      11.79611     TRUE           1
##    waiting_time
## 1 -1.776357e-15
## 2  4.596678e+00
```

6.1.5 Several service counters

Here we model a shop whose customers arrive randomly and are to be served at a group of counters, taking a random time for service, where we assume that waiting customers form a single first-in first-out queue.

The only difference between this model and the single-server model is in the `add_resource` function, where we have increased the capacity to two so that it can serve two customers at once.

```
set.seed(2021)

customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  seize("counter") %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter") %>%
  log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter",2) %>%
  add_generator("Customer", customer, function() rexp(1, 1/5))

shop %>% run(until = 30)
```

```
## 5.92531: Customer0: Here I am
## 12.0259: Customer1: Here I am
## 13.4303: Customer2: Here I am
## 13.5248: Customer3: Here I am
## 14.1994: Customer4: Here I am
## 16.6226: Customer0: Finished
```

```
## 17.3999: Customer5: Here I am
## 19.2911: Customer6: Here I am
## 20.7802: Customer1: Finished
## 25.2835: Customer2: Finished
## 26.8058: Customer7: Here I am
## 29.8574: Customer8: Here I am
## 29.971: Customer9: Here I am

## simmer environment: shop | now: 30 | next: 33.0973605697233
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 2(2) | queue status: 5(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 11 }
```

```
shop %>%
  get_mon_arrivals() %>%
  transform(waiting_time = end_time - start_time - activity_time)
```

```
##        name start_time end_time activity_time finished replication
## 1 Customer0   5.925307 16.62261     10.697299     TRUE           1
## 2 Customer1  12.025928 20.78016      8.754228     TRUE           1
## 3 Customer2  13.430317 25.28350      8.660893     TRUE           1
##    waiting_time
## 1 -1.776357e-15
## 2  0.000000e+00
## 3  3.192289e+00
```

Now that we also have a counter we can get some summary statistics from it using the function `get_mon_resources`.

```
shop %>% get_mon_resources()
```

```
##    resource      time server queue capacity queue_size system limit replication
## 1   counter  5.925307      1     0        2        Inf      1   Inf           1
## 2   counter 12.025928      2     0        2        Inf      2   Inf           1
## 3   counter 13.430317      2     1        2        Inf      3   Inf           1
## 4   counter 13.524818      2     2        2        Inf      4   Inf           1
## 5   counter 14.199351      2     3        2        Inf      5   Inf           1
## 6   counter 16.622606      2     2        2        Inf      4   Inf           1
## 7   counter 17.399880      2     3        2        Inf      5   Inf           1
## 8   counter 19.291124      2     4        2        Inf      6   Inf           1
## 9   counter 20.780156      2     3        2        Inf      5   Inf           1
## 10  counter 25.283499      2     2        2        Inf      4   Inf           1
## 11  counter 26.805825      2     3        2        Inf      5   Inf           1
## 12  counter 29.857358      2     4        2        Inf      6   Inf           1
## 13  counter 29.970950      2     5        2        Inf      7   Inf           1
```

6.1.6 Simple visualizations

We have learned to implement various simple simulations of our donut shop. The output we get is informative and comprehensive but nor particularly appealing to present, for instance in a report. The package `simmer.plot` provides plotting capabilities to summarize the results of a simulation. At this stage we will see two simple capabilities of the package. We will learn more about it in the following sections.

Before using `simmer.plot` you need to install it only once via

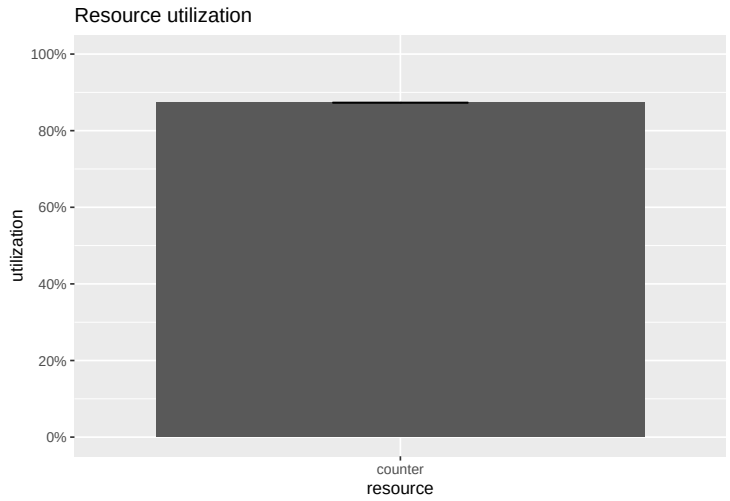
```
install.packages("simmer.plot")
```

and then load it at the beginning of every R session where you plan to use it.

```
library("simmer.plot")
```

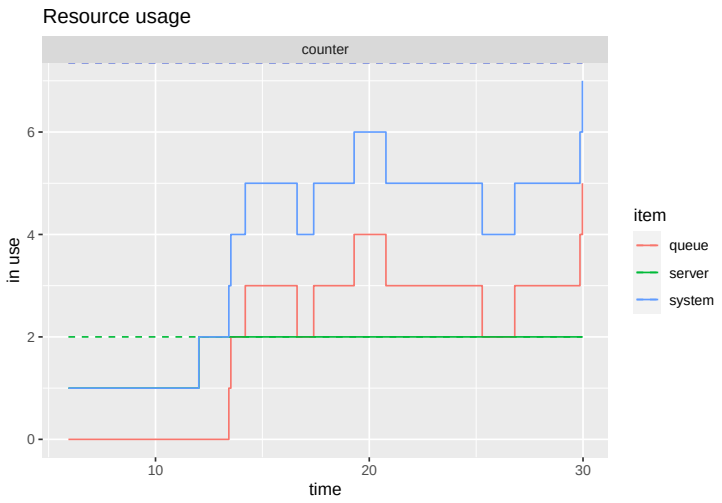
First, we can plot how much a resource, in this case our two employees, is utilized using the following code.

```
resources <- get_mon_resources(shop)
plot(resources, metric = "utilization")
```



So we see that our employees are busy around 90% of the time. We can also see when they are busy as well as how many people are queuing at each moment during the simulation using the code below.

```
plot(resources, metric = "usage",steps=T)
```



The green line reports the number of employees busy and we can see that most of the time they are both busy. The red line reports the number of people queuing and waiting to be served. The blue line is the total number of customers in the system: those queuing plus those being served.

6.2 Replication

For all previous examples, we ran a unique simulation and observed the results. As we have already learned, these results are affected by randomness and different runs will show different results.

Consider the last simulation we implemented where customers arrive at the shop where we have two counters. We can simulate the system 1000 times using the following code. We will not include `log` since otherwise the output will become cluttered.

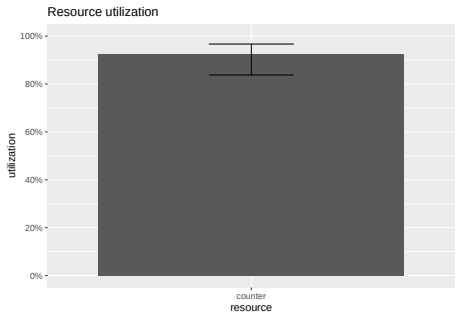
```
customer <-
  trajectory("Customer's path") %>%
  seize("counter") %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter")


envs <- lapply(1:100, function(i) {
```

```
    simmer("shop") %>%
    add_resource("counter",2) %>%
    add_generator("Customer", customer, function() rexp(1, 1/5)) %>%
    run(until=240)
}
)
```

Now `envs` stores the output of simulating the behavior of the shop for 4 hours 100 times.

We can summarize the results of these simulations using the `simmer.plot` package.

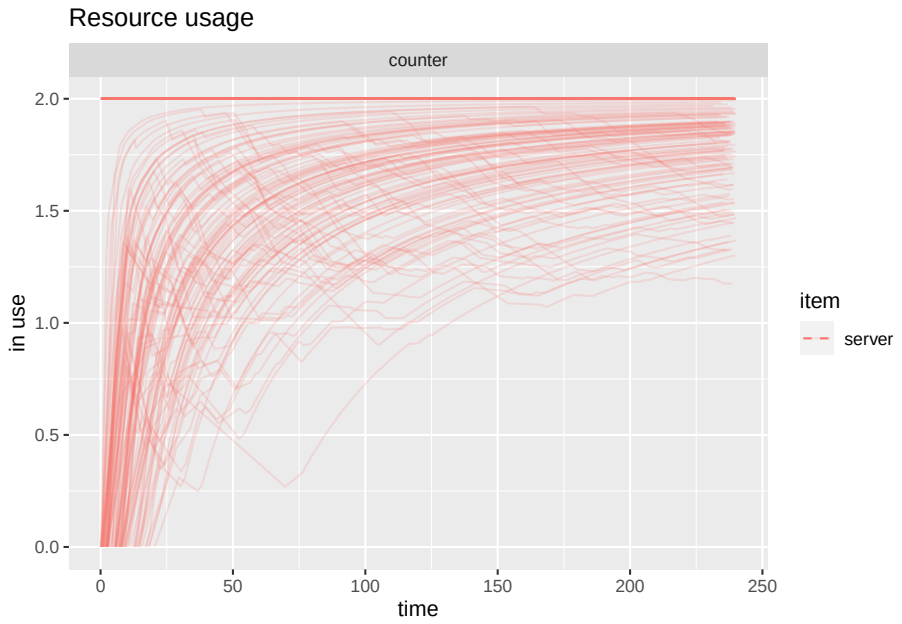
```
plot(get_mon_resources(envs), metric = "utilization")
```



Compared to previous plots we now notice that the output also has something that resembles a boxplot which tells us what was the utilization of the resource in different simulation runs.

We can also assess how busy were the employees in different simulations. As the shop opens the two employees become more and more busy and at the end of the four hours they are busy almost all the time.

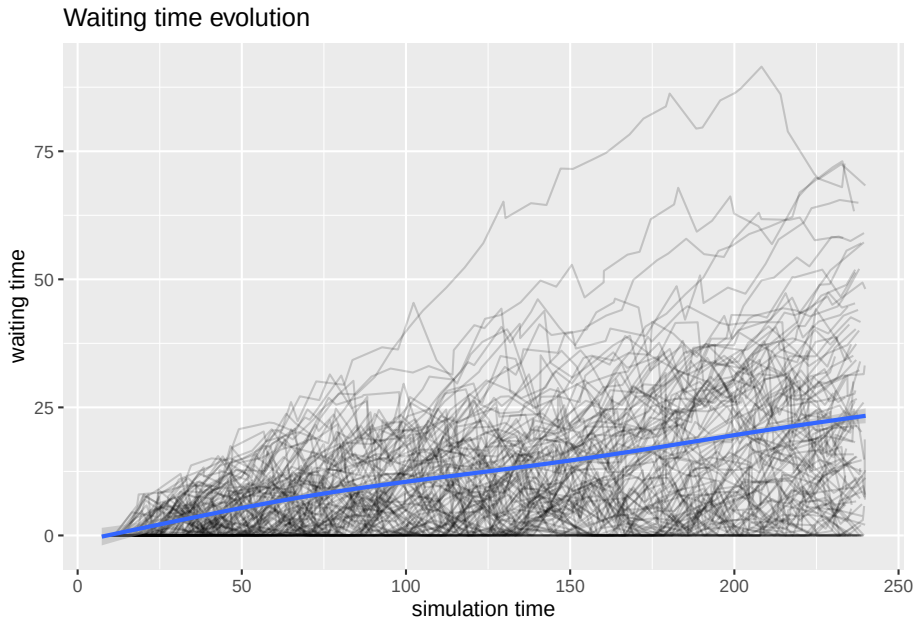
```
plot(get_mon_resources(envs), metric = "usage", items = "server")
```



Similarly, we can look at how long do customers queue in our shop.

```
plot(get_mon_arrivals(envs), metric = "waiting_time")
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Each black line represents a single simulation and the blue line gives an overall representation of the simulation. We can see that the waiting time seems to be linearly increasing with time.

6.3 The donut shop - advanced features

In many situations there is a system of priority service. Those customers with high priority are served first, those with low priority must wait. In some cases, preemptive priority will even allow a high-priority customer to interrupt the service of one with a lower priority.

`Simmer` implements priority requests with an extra integer priority argument to `add_generator()`. By default, priority is zero; higher integers have higher priority.

6.3.1 Priority customers without preemption

Suppose the donut shop have priority customers that when arrive at the shop, are served as soon as possible. We make the assumption that they arrive

```
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
```

```
  seize("counter") %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter") %>%
   log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter") %>%
  add_generator("Customer", customer, function() rexp(1, 1/5)) %>%
add_generator("Priority_Customer", customer, function() rexp(1, 1/15), priority = 1)

set.seed(2021)
shop %>% run(until = 45)
```

```
## 5.92531: Customer0: Here I am
## 7.28854: Customer1: Here I am
## 11.0879: Customer2: Here I am
## 14.2421: Customer3: Here I am
## 14.3366: Customer4: Here I am
## 15.0111: Customer5: Here I am
## 16.9819: Customer0: Finished
## 18.2117: Customer6: Here I am
## 18.3019: Priority_Customer0: Here I am
## 20.1029: Customer7: Here I am
## 21.0334: Customer8: Here I am
## 21.7555: Customer9: Here I am
## 24.807: Customer10: Here I am
## 24.9206: Customer11: Here I am
## 25.6428: Customer1: Finished
## 28.047: Customer12: Here I am
## 28.879: Customer13: Here I am
## 32.9443: Customer14: Here I am
## 35.9055: Priority_Customer0: Finished
## 36.8793: Customer15: Here I am
## 40.846: Priority_Customer1: Here I am
```

```
## simmer environment: shop | now: 45 | next: 47.5452877648588
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 14(Inf)
## { Source: Customer | monitored: 1 | n_generated: 17 }
## { Source: Priority_Customer | monitored: 1 | n_generated: 3 }
```

From the output we can see that whenever a priority customer joins the queue he is sold donuts as soon as the employee becomes available.

6.3.2 Priority customers with preemption

Now we allow priority customers to have preemptive priority. They will displace any customer in service when they arrive. That customer will resume when they finish (unless higher priority customers intervene). This requires only a change to one line of the program, adding the argument, `preemptive = TRUE` to the `add_resource` function call.

```
shop <-
  simmer("shop") %>%
  add_resource("counter", preemptive = TRUE) %>%
  add_generator("Customer", customer, function() rexp(1, 1/5)) %>%
add_generator("Priority_Customer", customer, function() rexp(1, 1/15), priority = 1)

set.seed(2021)
shop %>% run(until = 45)
```

```
## 5.92531: Customer0: Here I am
## 7.28854: Customer1: Here I am
## 11.0879: Customer2: Here I am
## 14.2421: Customer3: Here I am
## 14.3366: Customer4: Here I am
## 15.0111: Customer5: Here I am
## 16.9819: Customer0: Finished
## 18.2117: Customer6: Here I am
## 18.3019: Priority_Customer0: Here I am
## 20.1029: Customer7: Here I am
## 23.9383: Customer8: Here I am
## 24.6604: Customer9: Here I am
## 27.7119: Customer10: Here I am
## 27.8255: Customer11: Here I am
## 30.9519: Customer12: Here I am
## 31.4746: Customer13: Here I am
## 31.6998: Priority_Customer0: Finished
## 39.0407: Customer1: Finished
## 39.2381: Customer14: Here I am
## 40.846: Priority_Customer1: Here I am

## simmer environment: shop | now: 45 | next: 45.2787752879948
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 13(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 16 }
## { Source: Priority_Customer | monitored: 1 | n_generated: 3 }
```

In this other case, priority customers are served straight away. The customer

that was served when the priority customer arrived resumes is service as soon as the priority customer finishes.

6.3.3 Balking customers

Balking occurs when a customer refuses to join a queue if it is too long. Suppose that if there is one customer queuing in our shop then customers do not join the queue and leave. We can implement this by setting the `queue_size` option of `add_resource` and by adding some options of the `seize` function. Let's consider the following code.

```
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  seize("counter", continue = FALSE, reject =
          trajectory("Balked customer") %>% log_("Balking") ) %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter") %>%
   log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter", queue_size = 1) %>%
  add_generator("Customer", customer,
                function() rexp(1, 1/5))
```

The input `queue_size` is self-explanatory and simply sets how many people can queue for the counter. In the `seize` function we set the inputs `continue` and `reject`. With `continue = FALSE` we are saying that a rejected customer does not follow the rest of the trajectory. With `reject` we are specifying what trajectory the rejected customer will follow.

Let's run the simulation.

```
set.seed(2021)
shop %>% run(until = 45)
```

```
## 5.92531: Customer0: Here I am
## 12.0259: Customer1: Here I am
## 13.4303: Customer2: Here I am
## 13.4303: Customer2: Balking
## 16.6226: Customer0: Finished
## 17.2296: Customer3: Here I am
## 28.4187: Customer1: Finished
```

```
## 36.649: Customer4: Here I am
## 38.7585: Customer3: Finished
## 40.1462: Customer5: Here I am
## 40.6299: Customer6: Here I am
## 40.6299: Customer6: Balking
## 41.5604: Customer7: Here I am
## 41.5604: Customer7: Balking
## 42.2825: Customer8: Here I am
## 42.2825: Customer8: Balking
```

```
## simmer environment: shop | now: 45 | next: 45.3340111165536
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 1(1) }
## { Source: Customer | monitored: 1 | n_generated: 10 }
```

So now we see that often customers just leave the shop because they decide not to queue. We can count how many of them left for balking using:

```
sum(get_mon_arrivals(shop)$activity_time == 0)
```

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

and the hourly rate at which they leave

```
sum(get_mon_arrivals(shop)$activity_time == 0)/now(shop)*60
```

```
## [1] 5.333333
```

6.3.4 Reneging (or abandoning) customers

Often in practice an impatient customer will leave the queue before being served. Simmer can model this reneging behaviour using the `renege_in()` function in a trajectory. This defines the maximum time that a customer will wait before reneging, as well as an 'out' trajectory for them to follow when they renege.

If the customer reaches the server before reneging, then their impatience must be cancelled with the `renege_abort()` function.

```
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  renege_in(function() rnorm(1,5,1),
            out = trajectory("Reneging customer") %>%
```

```
              log_("I am off")) %>%
  seize("counter") %>%
  renege_abort() %>%
  timeout(function() rnorm(1,10,2)) %>%
  release("counter") %>%
   log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter") %>%
  add_generator("Customer", customer, function() rexp(1, 1/5))

run(shop, until = 45)
```

```
## 0.113593: Customer0: Here I am
## 0.892989: Customer1: Here I am
## 7.40631: Customer1: I am off
## 9.74378: Customer2: Here I am
## 10.2131: Customer3: Here I am
## 13.0758: Customer0: Finished
## 16.4212: Customer3: I am off
## 24.9321: Customer2: Finished
## 27.4101: Customer4: Here I am
## 27.7221: Customer5: Here I am
## 27.808: Customer6: Here I am
## 32.6284: Customer5: I am off
## 32.8674: Customer6: I am off
## 34.1951: Customer7: Here I am
## 36.7021: Customer4: Finished
```

```
## simmer environment: shop | now: 45 | next: 45.8000847848882
## { Monitor: in memory }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 0(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 9 }
```

6.3.5 Several counters with individual queues

Each counter is now assumed to have its own queue. The programming is more complicated because the customer has to decide which queue to join. The obvious technique is to make each counter a separate resource.

In practice, a customer might join the shortest queue. We implement this behaviour by first selecting the shortest queue, using the `select` function. Then we use `seize_selected` to enter the chosen queue, and later `release_selected`.

The rest of the program is the same as before.

```
set.seed(2021)
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am") %>%
  select(c("counter1", "counter2"), policy = "shortest-queue") %>%
  seize_selected() %>%
  timeout(function() rnorm(1,10,2)) %>%
  release_selected() %>%
   log_("Finished")

shop <-
  simmer("shop") %>%
  add_resource("counter1", 1) %>%
  add_resource("counter2", 1) %>%
  add_generator("Customer", customer, function() rexp(1, 1/5))

run(shop, until = 45)
```

```
## 5.92531: Customer0: Here I am
## 12.0259: Customer1: Here I am
## 13.4303: Customer2: Here I am
## 13.5248: Customer3: Here I am
## 14.1994: Customer4: Here I am
## 16.6226: Customer0: Finished
## 17.3999: Customer5: Here I am
## 19.2911: Customer6: Here I am
## 20.7802: Customer1: Finished
## 25.2835: Customer2: Finished
## 26.8058: Customer7: Here I am
## 29.8574: Customer8: Here I am
## 29.971: Customer9: Here I am
## 33.0974: Customer10: Here I am
## 33.62: Customer11: Here I am
## 34.0487: Customer4: Finished
## 34.1781: Customer3: Finished
## 41.3835: Customer12: Here I am
## 42.2932: Customer6: Finished

## simmer environment: shop | now: 45 | next: 45.3185680974989
## { Monitor: in memory }
## { Resource: counter1 | monitored: TRUE | server status: 1(1) | queue status: 3(Inf) }
## { Resource: counter2 | monitored: TRUE | server status: 1(1) | queue status: 2(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 14 }
```

There are several policies implemented internally that can be accessed by name:

- `shortest-queue`: The resource with the shortest queue is selected.
- `round-robin`: Resources will be selected in a cyclical nature.
- `first-available`: The first available resource is selected.
- `random` A resource is randomly selected.

6.3.6 Opening times

Customers arrive at random, some of them getting to the shop before the door is opened by a doorman. They wait for the door to be opened and then rush in and queue to be served.

This model defines the door as a resource, just like the counter. The capacity of the door is defined according to the `schedule` function, so that it has zero capacity when it is shut, and infinite capacity when it is open. Customers 'seize' the door and must then wait until it has capacity to 'serve' them. Once it is available, all waiting customers are 'served' immediately (i.e. they pass through the door). There is no timeout between 'seizing' and 'releasing' the door.

```
customer <-
  trajectory("Customer's path") %>%
  log_(function()
    if (get_capacity(shop, "door") == 0)
      "Here I am but the door is shut."
    else "Here I am and the door is open."
  ) %>%
  seize("door") %>%
  log_("I can go in!") %>%
  release("door") %>%
  seize("counter") %>%
  timeout(function() {rexp(1, 10)}) %>%
  release("counter")

door_schedule <- schedule(c(1,7,9,13), c(Inf,0,Inf,0), period = 13)

shop <-
  simmer("shop") %>%
  add_resource("door", capacity = door_schedule) %>%
  add_resource("counter") %>%
  add_generator("Customer", customer, function() rexp(1, 1))

shop %>% run(26)
```

```
## 3.53354: Customer0: Here I am and the door is open.
## 3.53354: Customer0: I can go in!
## 6.97294: Customer1: Here I am and the door is open.
## 6.97294: Customer1: I can go in!
## 7.05686: Customer2: Here I am but the door is shut.
## 9: Customer2: I can go in!
## 9.30068: Customer3: Here I am and the door is open.
## 9.30068: Customer3: I can go in!
## 10.4405: Customer4: Here I am and the door is open.
## 10.4405: Customer4: I can go in!
## 11.3978: Customer5: Here I am and the door is open.
## 11.3978: Customer5: I can go in!
## 15.4491: Customer6: Here I am and the door is open.
## 15.4491: Customer6: I can go in!
## 17.7096: Customer7: Here I am and the door is open.
## 17.7096: Customer7: I can go in!
## 18.4016: Customer8: Here I am and the door is open.
## 18.4016: Customer8: I can go in!
## 19.1858: Customer9: Here I am and the door is open.
## 19.1858: Customer9: I can go in!
## 22.0271: Customer10: Here I am and the door is open.
## 22.0271: Customer10: I can go in!
## 22.9206: Customer11: Here I am and the door is open.
## 22.9206: Customer11: I can go in!
## 24.0738: Customer12: Here I am and the door is open.
## 24.0738: Customer12: I can go in!
## 24.9154: Customer13: Here I am and the door is open.
## 24.9154: Customer13: I can go in!
## 25.3469: Customer14: Here I am and the door is open.
## 25.3469: Customer14: I can go in!
## 25.7375: Customer15: Here I am and the door is open.
## 25.7375: Customer15: I can go in!


## simmer environment: shop | now: 26 | next: 26
## { Monitor: in memory }
## { Resource: door | monitored: TRUE | server status: 0(Inf) | queue status: 0(Inf) }
## { Resource: counter | monitored: TRUE | server status: 1(1) | queue status: 0(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 17 }
```

6.3.7 Batching clients

Customers arrive at random, some of them getting to the shop before the door is open. This is controlled by an automatic machine called the doorman which opens the door only at intervals of 30 minutes (it is a very secure shop). The

customers wait for the door to be opened and all those waiting enter and proceed to the counter. The door is closed behind them.

One possible solution is using batching. Customers can be collected into batches of a given size, or for a given time, or whichever occurs first. Here, they are collected for periods of 30, and the number of customers in each batch is unrestricted.

After the batch is created with `batch` it is then separated with `separate`.

```
set.seed(2021)
customer <-
  trajectory("Customer's path") %>%
  log_("Here I am, but the door is shut.") %>%
  batch(n = Inf, timeout = 30) %>%
  separate() %>%
  log_("The door is open!") %>%
  seize("counter") %>%
  timeout(function() {rexp(1, 1/2)}) %>%
  release("counter") %>%
  log_("Finished.")

shop <- simmer("shop")
shop %>%
  add_resource("door") %>%
  add_resource("counter") %>%
  add_generator("Customer",
                customer, function() rexp(1, 1/20))
```

```
## simmer environment: shop | now: 0 | next: 0
## { Monitor: in memory }
## { Resource: door | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Resource: counter | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 0 }
```

```
shop %>% run(65)
```

```
## 23.7012: Customer0: Here I am, but the door is shut.
## 48.1037: Customer1: Here I am, but the door is shut.
## 53.5567: Customer2: Here I am, but the door is shut.
## 53.7012: Customer0: The door is open!
## 53.7012: Customer1: The door is open!
## 53.7012: Customer2: The door is open!
## 54.263: Customer0: Finished.
## 55.7827: Customer1: Finished.
```

```
## 57.0444: Customer2: Finished.
## 61.6105: Customer3: Here I am, but the door is shut.
## 61.9885: Customer4: Here I am, but the door is shut.
## 64.6866: Customer5: Here I am, but the door is shut.


## simmer environment: shop | now: 65 | next: 77.4887534806737
## { Monitor: in memory }
## { Resource: door | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Resource: counter | monitored: TRUE | server status: 0(1) | queue status: 0(Inf) }
## { Source: Customer | monitored: 1 | n_generated: 7 }
```

The function `balk` takes two inputs:

- `n`: the batch size;
- `timeout`: set an optional timer which triggers batches every `timeout` time units even if the batch size has not been fulfilled.

6.4 Simulating a simple health center

Let's say we want to simulate an ambulatory consultation where a patient is first seen by a nurse for an intake, next by two doctors for the consultation and finally by administrative staff to schedule a follow-up appointment.

We can construct a patient trajectory which defines all these steps.

```
patient <- trajectory("patients' path") %>%
  ## add an intake activity
  seize("nurse", 1) %>%
  timeout(function() rnorm(1, 15)) %>%
  release("nurse", 1) %>%
  ## add a consultation activity
  seize("doctor", 2) %>%
  timeout(function() rnorm(1, 20)) %>%
  release("doctor", 2) %>%
  ## add a planning activity
  seize("administration", 1) %>%
  timeout(function() rnorm(1, 5)) %>%
  release("administration", 1)
```

```
plot(patient)
```

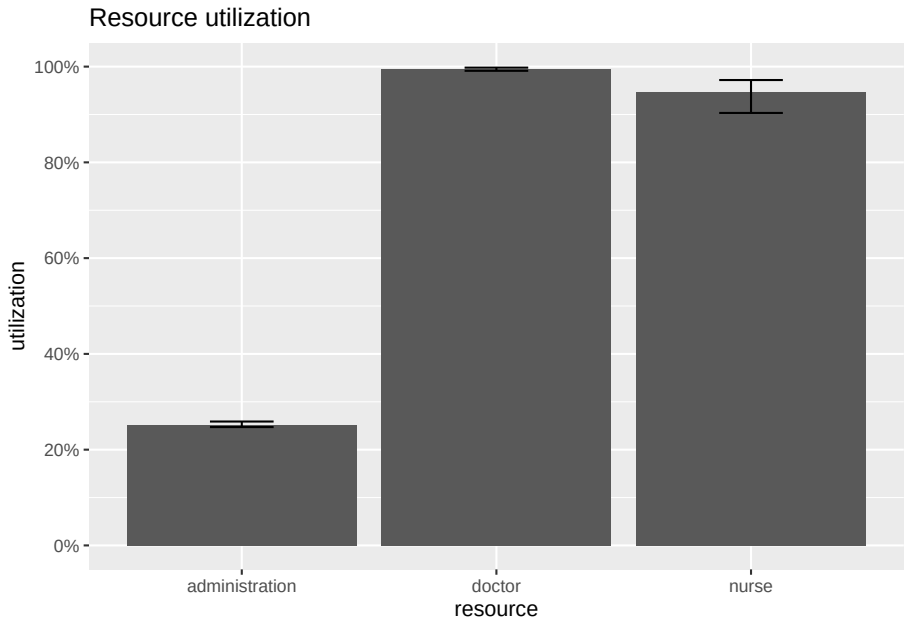
Once the trajectory is known, you may attach arrivals to it and define the resources needed. In the example below, three types of resources are added: the nurse and administration resources, each one with a capacity of 2, and the doctor resource, with a capacity of 4. The last method adds a generator of arrivals (patients) following the trajectory patient. The time between patients is about 5 minutes.

Let's run the simulation using replication.

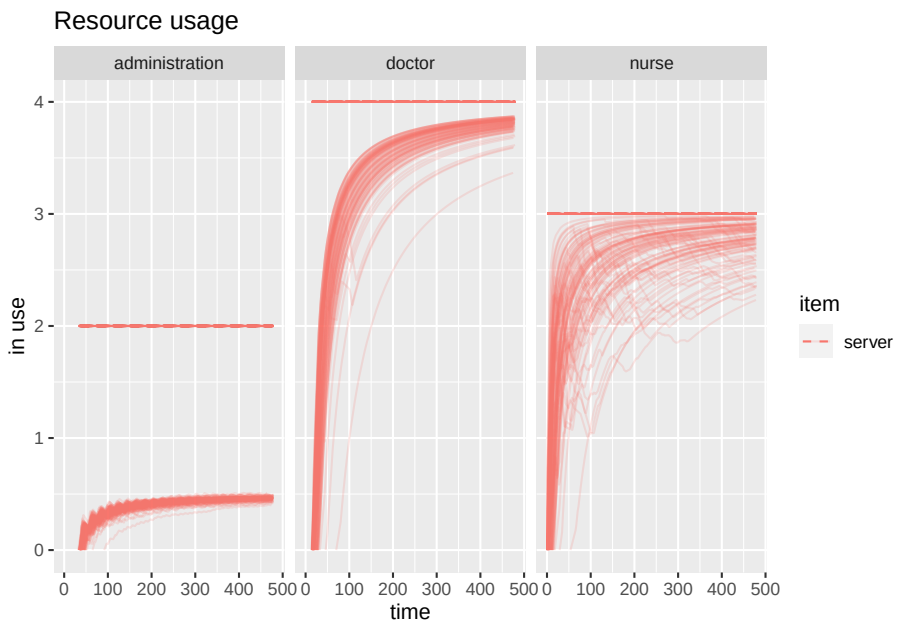
```
envs <- lapply(1:100, function(i) {
simmer("health center") %>%
  add_resource("nurse", 3) %>%
  add_resource("doctor", 4) %>%
  add_resource("administration", 2) %>%
  add_generator("patient", patient, function() rexp(1, 1/5)) %>%
    run(until=480)
}
)
```

Let's observe the results.

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

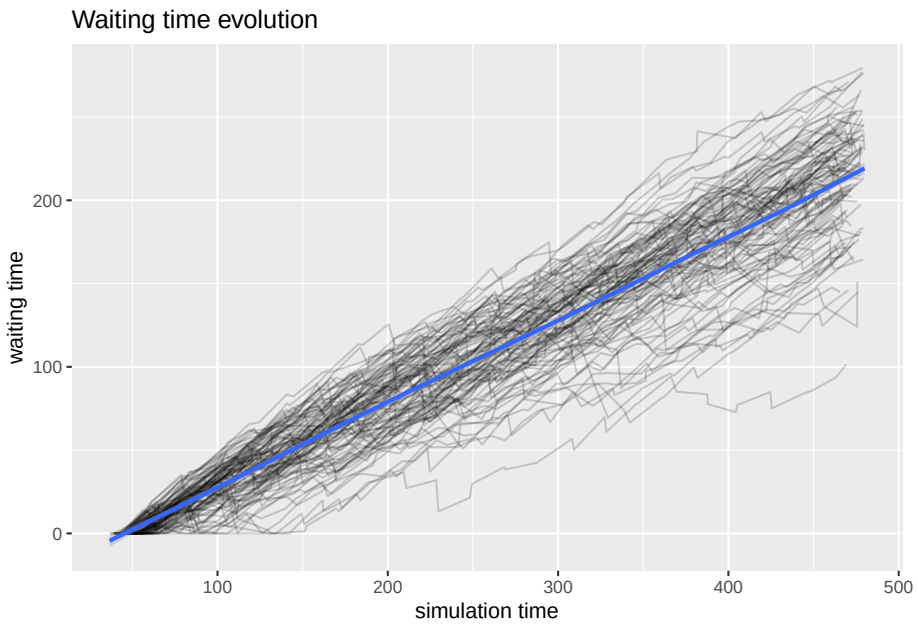



```
plot(get_mon_resources(envs), metric = "usage", items = "server")
```



```
plot(get_mon_arrivals(envs), metric = "waiting_time")
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



6.5 A production process simulation

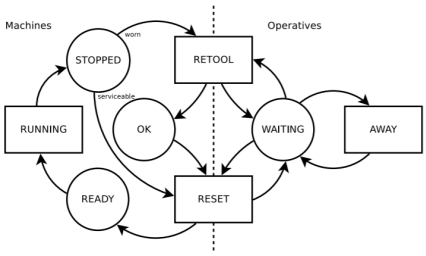
Consider a simple engineering job shop that consists of several identical machines. Each machine is able to process any job and there is a ready supply of jobs with no prospect of any shortages. Jobs are allocated to the first available machine. The time taken to complete a job is variable but is independent of the particular machine being used. The machine shop is staffed by operatives who have two tasks:

- RESET machines between jobs if the cutting edges are still OK
- RETOOL those machines with cutting edges that are too worn to be reset

In addition, an operator may be AWAY while attending to personal needs

The figure below shows the activity cycle diagram for the considered system. Circles (READY, STOPPED, OK, WAITING) represent states of the machines or the operatives respectively, while rectangles (RUNNING, RETOOL, RESET,

AWAY) represent activities that take some (random) time to complete. Two kind of processes can be identified: shop jobs, which use machines and degrade them, and personal tasks, which take operatives AWAY for some time.



Notice that after a job is completed by a machine there may be two possible trajectories to follow:

- either the machine needs only to be reset by an operator;
- or it first needs to be retool and then reset by the operator.

We can implement such a situation using `branch`. A branch is a point in a trajectory in which one or more sub-trajectories may be followed. The `branch()` activity places the arrival in one of the sub-trajectories depending on some condition evaluated in a dynamical parameter called `option`. It is the equivalent of an if/else in programming, i.e., if the value of `option` is `i`, the i-th sub-trajectory will be executed.

Let's implement the system. First of all, let us instantiate a new simulation environment and define the completion time for the different activities as random draws from exponential distributions. Likewise, the interarrival times for jobs and tasks are defined (`NEW_JOB`, `NEW_TASK`), and we consider a probability of 0.2 for a machine to be worn after running a job (`CHECK_JOB`).

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

RUNNING <- function() rexp(1, 1)
RETOOL <- function() rexp(1, 2)
RESET <- function() rexp(1, 3)
AWAY <- function() rexp(1, 1)
CHECK_WORN <- function() runif(1) < 0.2
NEW_JOB <- function() rexp(1, 5)
NEW_TASK <- function() rexp(1, 1)
```

The trajectory of an incoming job starts by seizing a machine in `READY` state. It takes some random time for `RUNNING` it after which the machine's serviceability is checked. An operative and some random time to `RETOOL` the machine may be needed, and either way an operative must `RESET` it. Finally, the trajectory releases the machine, so that it is `READY` again. On the other hand, personal tasks just seize operatives for some time.

```
task <- trajectory() %>%
  seize("operative") %>%
  timeout(AWAY) %>%
  release("operative")

job <- trajectory() %>%
  seize("machine") %>%
  timeout(RUNNING) %>%
  branch(
   CHECK_WORN, continue = TRUE,
   trajectory() %>%
   seize("operative") %>%
   timeout(RETOOL) %>%
   release("operative")) %>%
  seize("operative") %>%
  timeout(RESET) %>%
  release("operative") %>%
  release("machine")
```

Once the processes' trajectories are defined, we append 10 identical machines and 5 operatives to the simulation environment, as well as two generators for jobs and tasks.

```
env %>%
  add_resource("machine", 10) %>%
  add_resource("operative", 5) %>%
  add_generator("job", job, NEW_JOB) %>%
  add_generator("task", task, NEW_TASK) %>%
  run(until=10)
```

```
## simmer environment: Job Shop | now: 10 | next: 10.2238404207112
## { Monitor: in memory }
## { Resource: machine | monitored: TRUE | server status: 9(10) | queue status: 0(Inf)
## { Resource: operative | monitored: TRUE | server status: 3(5) | queue status: 0(Inf
## { Source: job | monitored: 1 | n_generated: 49 }
## { Source: task | monitored: 1 | n_generated: 11 }
```

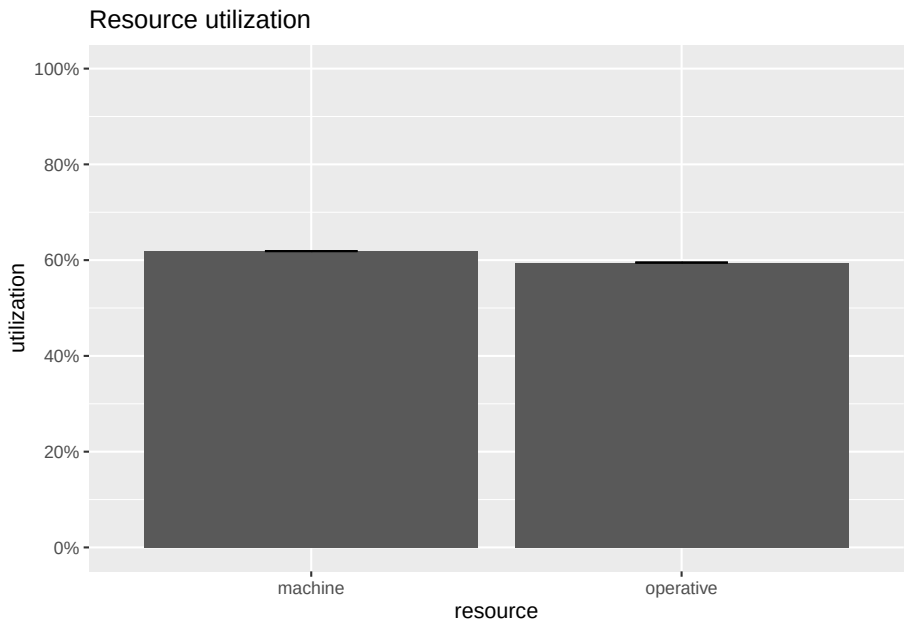
Let's extract a history of the resource's state to analyze the average number of

machines/operatives in use as well as the average number of jobs/tasks waiting for an assignment.

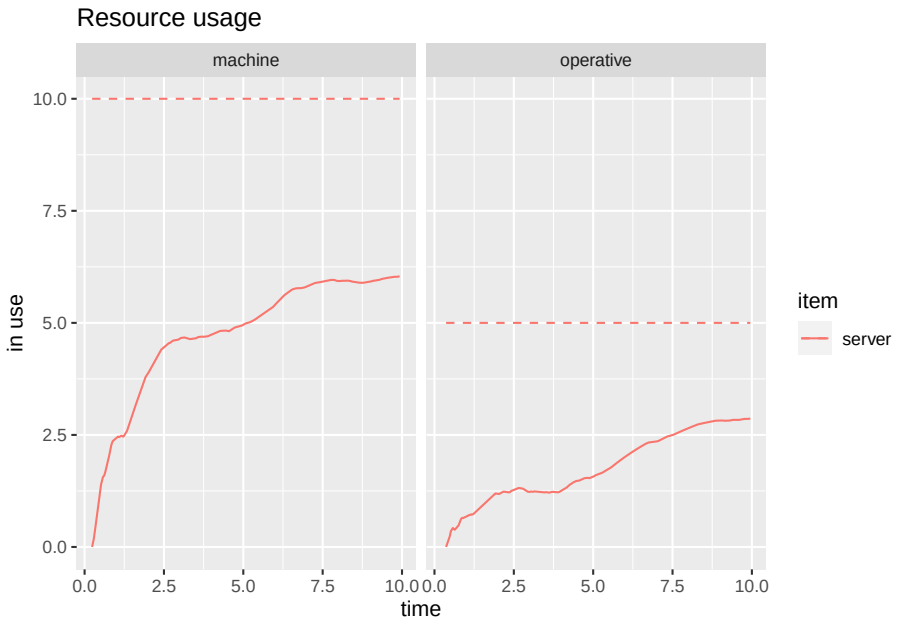
```
aggregate(cbind(server, queue) ~ resource, get_mon_resources(env), mean)
```

```
##    resource   server      queue
## 1   machine 5.839080 0.04597701
## 2 operative 3.153153 0.30630631
```

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




```
plot(get_mon_resources(env),"usage", items = "server")
```


Resource usage
machine
operative
10.0
7.5
5.0
2.5
0.0
in use
0.0
2.5
5.0
7.5
10.0
time
item
server

Chapter 7

Queuing Theory

In the previous chapter we have learned to implement complex discrete-event simulations using the `Simmer` package. Basically all simulation models we implemented involved some queue of customers requiring a service. There is a whole area of probability called, *queuing theory*, which studies the mathematical foundations and properties of such models. In this chapter we will define a bit more formally what queues are and how they work. We will also learn that we can actually already know a lot about the behavior of the queue by knowing a few key parameters.

7.1 Poisson Process

Consider random events such as the arrival of customer at a shop, the arrival of emails to a mail server or the arrival of calls to a call-center. These events can be described by a counting function $N(t)$ defined for all $t \geq 0$. This counting function represents the number of events that occurred in $[0, t]$. For each interval $[0, t]$ the value $N(t)$ is an observation of a random variable where the only possible values are the integers $0, 1, 2, ...$

The counting process $\{N(t) : t \geq 0\}$ is said to be a Poisson process with mean rate λ if the following assumptions are fulfilled:

- $N(0) = 0$;
- it has independent increments: that is the number of arrivals during non-overlapping time intervals are independent random variables.
- the number of events in any interval of length t is a Poisson random variable with parameter λt.

Therefore

$$P(N(t) = n) = \frac{e^{-\lambda t}(\lambda t)^n}{n!}$$

The last assumption implies that the distribution of the number of arrivals between, say, t and $t+s$ depends only on the length of the interval s and not on the starting point t. This property is usually called *stationarity*. Consequently:

$$P(N(t) - N(s) = n) = \frac{e^{-\lambda(t-s)}(\lambda(t-s))^n}{n!}$$

and because of the properties of the Poisson distribution:

$$E(N(t) - N(s)) = V(N(t) - N(s)) = \lambda(t-s).$$

Now consider the time at which arrivals occurs in a Poisson process. Let the first arrival occur at time A_1, the second occur at time $A_1 + A_2$ and so on. Thus $A_1, A_2, ...$ are successive inter-arrival times. The first arrival occurs after time t if and only if there are no arrivals in the interval $[0, t]$ so it is seen that

$$\{A_1 > t\} = \{N(t) = 0\}$$

and consequently

$$P(A_1 > t) = P(N(t) = 0) = e^{-\lambda t}$$

Thus the probability that the first arrival will occur in $[0, t]$ is given by

$$P(A_1 \leq t) = 1 - P(A_1 > t) = 1 - e^{-\lambda t}$$

which is the cumulative density function of an exponential distribution with parameter λ. Hence, A_1 is distributed exponentially with mean $E(A_1) = 1/\lambda$. It can also be shown that all inter-arrival times $A_1, A_2, ...$ are exponentially distributed and independent with mean $1/\lambda$.

An alternative definition of a Poisson process is of a counting process whose inter-arrival times are distributed exponentially and independently.

Exponential distributions have the property of being *memoryless*, which is deeply connected to Poisson processes. For an exponential distribution X it holds that

$$P(X > s + t | X > s) = P(X > t).$$

Suppose X represents the life of a light bulb. The above equation states that the probability that the light bulb lives for at least $s + t$ hours, given it has survived s hours, is that same as the initial probability that it lives for at least t hours. That is, the light bulb it does not remember that it has already been

in use for a time s. Let's show the equality is true:

$$\begin{aligned} P(X > s+t | X > s) &= \frac{P(X > s+t, X > s)}{P(X > s)} \\ &= \frac{P(X > s+t)}{P(X > s)} \\ &= \frac{1 - P(X \leq s+t)}{1 - P(X \leq s)} \\ &= \frac{e^{-\lambda(s+t)}}{e^{-\lambda s}} \\ &= e^{-\lambda t} \\ &= P(X > t) \end{aligned}$$

7.2 Characteristics of Queuing Systems

The key elements of queuing systems are customers and servers.

- The term *customer* can refer to people, machines, trucks, airplanes etc etc. Anything that arrive at a facility and requires service.
- The term *server* can refer to receptionist, repair personnel, runways in airport, washing machines etc etc. Any resource that provides the requested service.

Below we describe the elements of queuing systems in more details.

7.2.1 The Calling Population

The population of potential customers, referred to as the *calling population*, will be assumed to be infinite, even though the number of potential customers is actually finite. When the population of potential customers is large this assumption is innocuous and actually can simplify the model. This is especially true when we believe that at any given time the number of customers being served or waiting for service is a small proportion of the whole population.

The assumption of an infinite population is such that the rate of arrival of customers is not affected by the number of customers that have already joined the queuing system. In addition, this will usually entail that the rate of arrival is constant throughout time.

7.2.2 System Capacity

In many queuing systems there is a limit to the number of customers that may be in the waiting line or system. An arriving customer who finds the system full does not enter but returns immediately to the calling population. However, there are other systems that may simply have an infinite capacity. As we will see later, when a system has a limited capacity, a distinction is made between the arrival rate (i.e. the number of arrivals per time unit) and the effective arrival rate (the number who arrive and enter the system per time unit).

7.2.3 The Arrival Process

The arrival process for infinite population models is usually characterized in terms of inter-arrival times of successive customers. Arrivals may occur at scheduled times or random times. When at random times, the inter-arrival times are usually characterized by a probability distribution. In addition, customers may arrive one at a time or in batches. The batch may be of constant size or of random size.

The most important model, and the only one we will consider, for random arrivals is the Poisson arrival process. If A_n represents the inter-arrival time between customer $n-1$ and customer n, then for a Poisson arrival process A_n is exponentially distributed with mean $1/\lambda$ per time unite. The arrival rate is λ customers per time unit. The number of arrivals in a time interval of length t has the Poisson distribution with mean λt customers.

7.2.4 Queue Behavior and Queue Discipline

Queue behavior refers to the actions of customers while in a queue waiting for service to begin. In some situation, there is a possibility that incoming customers will balk, renege, or jockey (move from one line to another if they think they have chosen a slow line).

Queue discipline refers to the logical ordering of customers in a queue and determines which customer will be chosen for service when a server becomes free. Common queue disciplines include first-in-first-out (FIFO), last-in-first-out (LIFO), service in random order (SIRO) etc. Notice that a FIFO queue discipline implies that services begin in the same order as arrivals, but that customers could leave the system in a different order because of different length service times.

7.2.5 Service Times and Service Mechanism

The service times of successive arrivals are denoted by $S_1, S_2, ...$ They may be constant or of random duration. In the latter case $\{S_1, S_2, S_3, ...\}$ is usually

characterized as a sequence of independent and identically distributed random variables. The Exponential, Normal etc. are often used to model service times. Sometimes services are identically distributed for all customers of a given type or class or priority, whereas customers of different types might have completely different service-time distributions. In addition in some systems service times depend upon the time of the day or upon the length of the waiting line.

A queuing system consists of a number of service counters and interconnecting queues. Each service center consists of some number of server, c, working in parallel; that is, upon getting to the head of the line, a customer takes the first available server. Parallel service mechanisms are either single server ($c = 1$), multiple server ($1 < c < \infty$), or unlimited servers ($c = \infty$).

7.3 Queuing Notation

Recognizing the diversity of queuing systems, a notational system was introduced in the 50's which has been widely adopted. The convention is based on the format $A/B/c/N/K$, where the letters represent the following system characteristics:

- A represents the inter-arrival time distribution
- B represents the service-time distribution
- c represents the number of parallel servers
- N represents the system capacity
- K represents the size of the calling population

Common symbols for A and B are include M (exponential or Markov), D (constant or deterministic) and G (arbitrary or general).

For example, $M/M/1/\infty/\infty$ indicates a single-server system that has unlimited queue capacity and infinite population of potential arrivals. The inter-arrival times and service times are exponentially distributed. When N and K are infinity, they may be dropped from the notation. For example $M/M/1/\infty/\infty$ is often shorted to $M/M/1$.

All systems will be assumed to have a FIFO queue discipline.

7.4 Measures of Performance

There are various measures that one can use to assess the quality of a queuing system. These are:

- L: the average number of customers in the system;
- L_Q: the average number of customers in the system;
- w: the average time spent in the system;
- w_Q: the average time spent in the queue;
- ρ: the server utilization; the proportion of time that a server is busy.

The term system refers to the waiting line plus the service mechanism, whilst the term queue refers to the waiting line alone.

7.4.1 Average Number of Customers in the System L

Consider a queuing system over a period of time T and let $L(t)$ denote the number of customers in the system at time t. Let T_i be the total time in $[0, T]$ in which the system contained exactly i customers. In general $\sum_{i=0}^{\infty} T_i = T$. The *average number of customers in the system* is estimated by

$$\hat{L} = \frac{1}{T} \sum_{i=0}^{\infty} iT_i = \sum_{i=0}^{\infty} i \frac{T_i}{T}.$$

Notice that T_i/T is the proportion of time the system contains exactly i customers.

Let's consider an example. Figure 7.1 gives a simulation of a queue in an interval of 20 time units. It can be seen that $T_0 = 3$, $T_1 = 11$, $T_2 = 5$ and $T_3 = 1$, and therefore $\hat{L} = (0 \cdot 3 + 1 \cdot 11 + 2 \cdot 5 + 3 \cdot 1)/20 = 24/20 = 1.2$ customers.

By looking at Figure 7.1 it can be seen that the total area under the function $L(t)$ can be decomposed into rectangles of length T_i and height i, thus having area iT_i. It follows that the total area is given by

$$\sum_{i=0}^{\infty} iT_i = \int_0^T L(t)dt$$

and therefore

$$\hat{L} = \frac{1}{T} \sum_{i=0}^{\infty} iT_i = \frac{1}{T} \int_0^T L(t)dt$$

Many queuing systems exhibit some kind of long-run stability in terms of their average performance. For such systems, as time T gets large, the observed average number of customers in the system $\hat{L}$ approaches a limiting value, say L, which is called the *long-run average number in system.* With probability 1 we have that

$$\hat{L} = \frac{1}{T} \int_0^T L(t)dt \to L \text{ as } T \to \infty$$

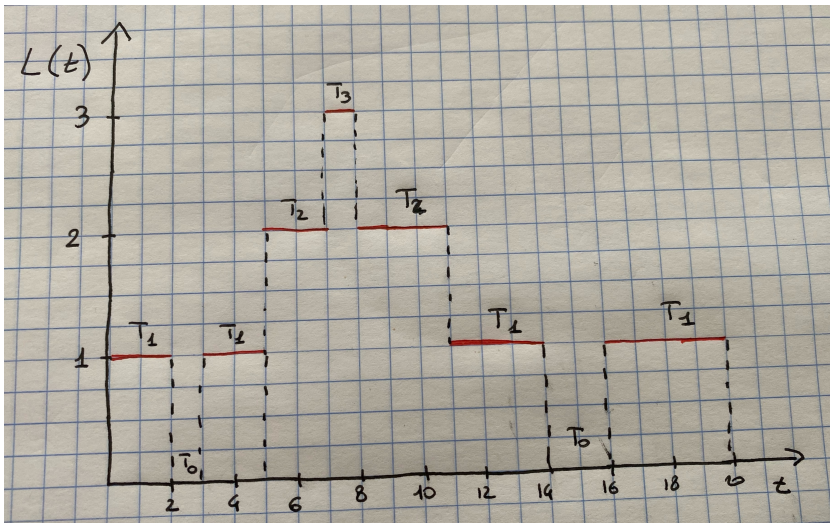


Figure 7.1: Number in system, L(t), at time t.

If a simulation run length T is sufficiently long, the estimator $\hat{L}$ becomes arbitrarily close to L.

The above equations can be applied to any subsystem of a queuing system. If $L_Q(t)$ denotes the number of customers waiting in queue, and T_i^Q denotes the total time in $[0, T]$ in which exactly i customers are waiting in queue, then

$$\hat{L}_Q = \frac{1}{T}\sum_{i=0}^{\infty} iT_i^Q = \int_0^T L_Q(t)dt \to L_Q \text{ as } T \to \infty$$

7.4.2 Average Time Spent in System per Customer w

If we simulate a queuing system for some period of time, say T, then we can record the time that each customer spends in the system during $[0, T]$, say $W_1, W_2, \ldots, W_N$ where N is the number of arrivals in $[0, T]$. The average time spent in system per customer, called the *average system time*, is given by

$$\hat{w} = \frac{1}{N}\sum_{i=1}^{N} W_i$$

For stable systems, as $N \to \infty$

$$\hat{w} \to w$$

with probability 1, where w is called the *long-run average system time*.

If the system under consideration is the queue alone, we can re-write the above equations as

$$\widehat{w}_Q = \frac{1}{N}\sum_{i=1}^{N} W_i^Q \to w_Q \text{ as } N \to \infty$$

where W_i^Q is the total time customer i spends waiting in queue, $\widehat{w}_Q$ is the observed average time spend in queue (or delay) and w_Q is the long-run average delay per customer.

Consider the system in Figure 7.1. It can be seen that there are 5 customers who waited $W_1 = 2$, $W_2 = 5$, $W_3 = 6$, $W_4 = 7$ and $W_5 = 4$ and therefore

$$\widehat{w} = \frac{2+5+6+7+4}{5} = 4.8 \text{ time units}$$

Thus on average these customers spent 4.8 time units in the system. As for the time spent in the queue, it can be computed as $W_1^Q = 0$, $W_2^Q = 0$, $W_3^Q = 3$, $W_4^Q = 4$ and $W_5^Q = 0$, thus:

$$\widehat{w}_Q = \frac{0+0+3+4+0}{5} = 1.4 \text{ time units.}$$

7.4.3 Little's Law

For the example in Figure 7.1 there were $N = 5$ arrivals in $T = 20$ time units, and thus the observed arrival rate was $\hat{\lambda} = N/T = 1/4$ customer per time unit. Recall that $\hat{L} = 1.2$ and $\widehat{w} = 4.8$. Hence it follows that

$$\hat{L} = \hat{\lambda}\widehat{w},$$

since $1.2 = \frac{1}{4}4.8$.

This relationship is not coincidental: it holds for almost all queuing systems. Allowing $T \to \infty$ and $N \to \infty$, the above expression becomes

$$L = \lambda w,$$

where $\hat{\lambda} \to \lambda$ and λ is the long-run average arrival rate. This equality is usually called *Little's law*. It says that the average number of customers in the system is equal to the average number of arrivals per time unit times the average time spent in the system. For Figure 7.1 there is one arrival every 4 minutes (on average) and each arrival spends 4.6 minutes in the system (on average), so at an arbitrary point in time there will be $(1/4)(4.9) = 1.2$ customers present (on average).

Little's law can also be derived reconsidering Figure 7.1 as follows. Figure 7.2 shows the exact same system history, but reporting the waiting time of each customer in the system, W_i. Again we can see that the area under the function

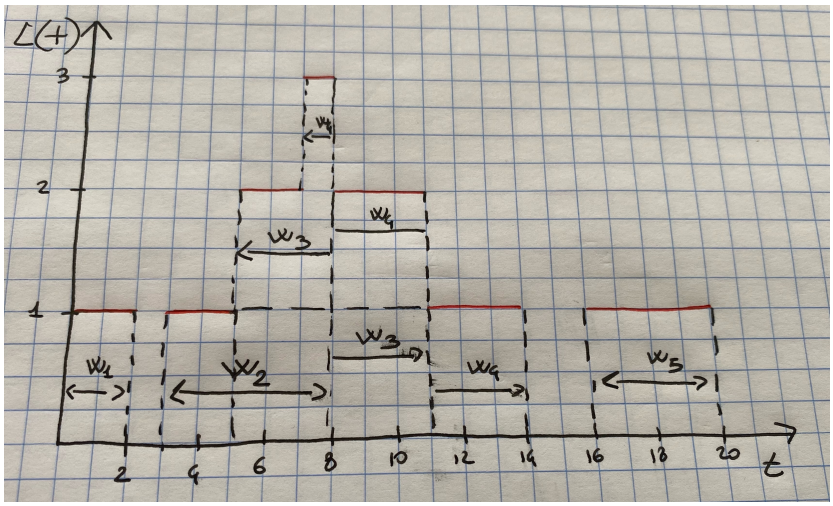


Figure 7.2: System times Wi for the example system.

$L(t)$ can be decomposed into rectangles of height 1 and length W_i, for each $i = 1, 2, \ldots, N$. Therefore

$$\sum_{i=1}^{N} W_i = \int_0^T L(t)dt$$

Therefore, by using that $\hat{\lambda} = N/T$ we have that

$$\hat{L} = \frac{1}{T}\int_0^T L(t)dt = \frac{1}{T}\sum_{i=1}^{N} W_i = \frac{N}{T}\frac{1}{N}\sum_{i=1}^{N} W_i = \hat{\lambda}\hat{w},$$

which is indeed Little's law!

7.4.4 Server Utilization

Server utilization is defined as the proportion of time that a server is busy. Observed served utilization, denoted by $\hat{\rho}$, is defined over a specified time interval $[0, T]$. Long-run server utilization is denoted by ρ. For systems that exhibit long-run stability,

$$\hat{\rho} \to \rho \text{ as } T \to \infty$$

7.4.4.1 Server Utilization in G/G/1 Queues

Consider any single-server queuing system with average arrival rate λ customers per time unit, average service time $E(S) = 1/\mu$ time units, and infinite queue capacity and calling population. Notice that $E(S) = 1/\mu$ implies that when

busy the server is working at the rate μ customers per time unit, on average: μ is called the *service rate*.

Notice that the server alone is a subsystem that can be considered as a queuing system itself. hence, Little's law $L = \lambda w$ can be applied to the server. For stable systems, the average arrival rate to the server, say λ_s must be identical to the average arrival rate to the system λ (certainly $\lambda_s \leq \lambda$ since customers cannot be served faster than they arrive, but if $\lambda_s < \lambda$ the the waiting line would tend to grow in length and so we would have an unstable system). For the server subsystem, the average system time is $w = E(S) = 1/\mu$. The actual number of customers in the server subsystem is either zero or one as shown in Figure 7.3 for the system in Figure 7.1. Hence the average number in the server subsystem $\hat{L}_s$ is given by

$$\hat{L}_s = \frac{T - T_0}{T}$$

In the example $\hat{L}_s = 17/20 = \hat{\rho}$.

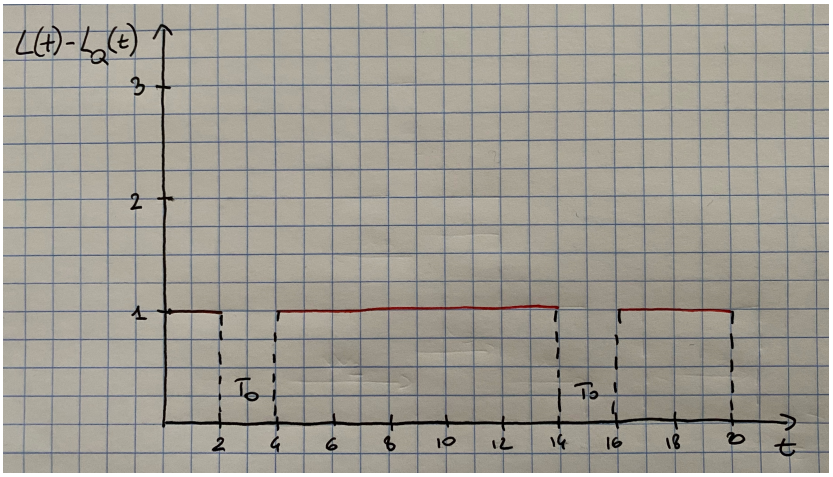


Figure 7.3: Number of customers being served at time t

In general, for a single-server queue, the average number of customers being served at an arbitrary point in time is equal to server utilization. As $T \to \infty$, $\hat{L}_s = \hat{\rho} \to L_s = \rho$. Combining these results into $L = \lambda w$ for the server subsystem yields

$$\rho = \lambda E(S) = \frac{\lambda}{\mu}$$

that is the long-run server utilization in a single-server queue is equal to the average arrival rate divided by the average service rate. For a single-server queue to be stable, the arrival-rate λ must be less then the service rate μ:

$$\lambda < \mu$$

or

$$\rho = \frac{\lambda}{\mu} < 1$$

If the arrival rate is greater than the service rate ($\lambda > \mu$) the server will eventually get further and further behind. After time, the server will always be busy, and the waiting line will tend to grow in length. For stable single-server systems long-run measures of performance such as average queue length L_Q are well defined and have a meaning. For unstable systems long-run server utilization is 1 and the long-run average queue length is infinite.

7.4.4.2 Server Utilization in G/G/c Queues

Consider now a queuing system with c identical servers in parallel. If an arriving customer finds more than one server idle, the customer chooses a server without favoring any particular server. Arrivals occur at rate λ from an infinite calling population and each server works at rate μ customers per time unit.

Using a similar argument as above, one can derive that long-run average server utilization is defined by

$$\rho = \frac{\lambda}{c\mu}$$

and the system is stable if $\rho < 1$ or equivalently if

$$\lambda < c\mu.$$

The average number of busy servers is

$$L_s = \frac{\lambda}{\mu}.$$

7.4.4.3 An Example

Consider a doctor who schedules patients every 10 minutes and who spends S_i minutes with the i-th patient, where

$$S_i = \begin{cases} 9 \text{ minutes with probability } 0.9 \\ 12 \text{ minutes with probability } 0.1 \end{cases}$$

Thus, arrivals are deterministic (every 10 minutes) but services are stochastic with mean and variance given by

$$E(S_i) = 9 \cdot 0.9 + 12 \cdot 0.1 = 9.3 \text{ minutes}$$

and

$$V(S_i) = 0.9 \cdot (9 - 9.3)^2 + 0.1 \cdot (12 - 9.3)^2 = 0.81 \text{ minutes}^2$$

Here $\rho = \lambda/\mu = 9.3/10 = 0.93 < 1$ and so the system is stable and the doctor will be busy 93% of the time in the long-run. In the short-run, sometimes queues will build up temporarily because of the variability of the service distribution, as shown in Figure 7.4.

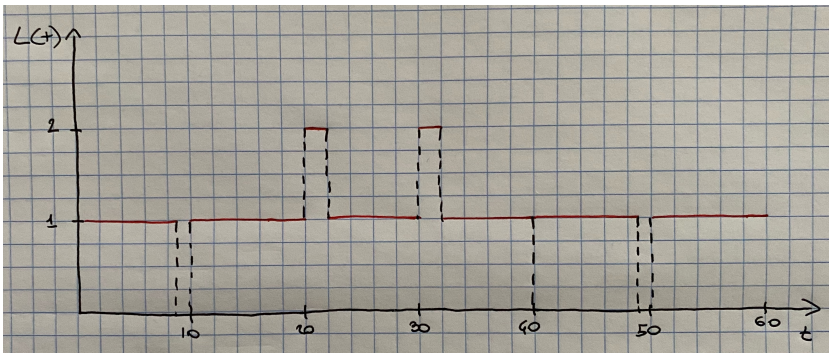


Figure 7.4: Number of patients in the doctor's office at time t

7.5 Steady-State Behavior of the M/M/1 Model

Henceforth, we assume that the calling population is infinite, the arrivals are assumed to follow a Poisson process with rate λ arrivals per time unit - that is, the inter-arrival times are assumed to be exponentially distributed with mean $1/\lambda$. We also assume that service times are exponentially distributed with mean $1/\mu$. The queue discipline will be FIFO. Because of the exponential distribution assumption on the arrival process, these models are called *Markovian*.

A queuing system is said to be in *statistical equilibrium* or in *steady state* if the probability that the system is in a given state is not time-dependent: that is

$$P(L(t) = n) = P_n(t) = P_n$$

is independent of time t. Two properties of such systems:

- they approach statistical equilibrium from any starting state,
- they remain in statistical equilibrium once it is reached.

For the models that we will consider next the steady-state parameter L, the average number of customers in the system, can be computed as

$$L = \sum_{n=0}^{\infty} nP_n$$

where $\{P_0, P_1, ...\}$ are the steady-state probabilities of finding n customers in the system. Once L is given, the other steady-state parameters can be computed

easily from Little's law:

$$
\begin{aligned}
w &= \frac{L}{\lambda} \\
w_Q &= w - \frac{1}{\mu} \\
L_Q &= \lambda w_Q,
\end{aligned}
$$

where λ is the arrival rate and μ is the service rate.

Recall that for queues to reach statistical equilibrium a necessary and sufficient condition is that $ /c <1 $.

For the M/M/1 model the steady-state parameters can be shown to be equal to:

$$
\begin{aligned}
\rho &= \frac{\lambda}{\mu} \\
L &= \frac{\lambda}{\mu - \lambda} \\
w &= \frac{1}{\mu - \lambda} \\
w_Q &= \frac{\lambda}{\mu(\mu - \lambda)} \\
L_Q &= \frac{\lambda^2}{\mu(\mu - \lambda)} \\
P_n &= \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^n
\end{aligned}
$$

Let's consider an example. Suppose that the interarrival times and service times of a single-chair unisex hair-styling shop have been shown to be exponentially distributed. The values of λ and μ are 2 per hour and 3 per hour, respectively: that is, the time between arrivals averages 1/2 hour and the service time averages 20 minutes. The server utilization and the probabilities for 0, 1, 2, 3, and 4 or

more customers in the shop are computed as follows:

$$\rho = \frac{\lambda}{\mu} = 2/3$$
$$P_0 = 1 - \frac{\lambda}{\mu} = 1/3$$
$$P_1 = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right) = \frac{2}{9}$$
$$P_2 = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^2 = \frac{4}{27}$$
$$P_3 = \left(\frac{1}{3}\right)\left(\frac{2}{3}\right)^3 = \frac{8}{81}$$
$$P_{\geq 4} = 1 - \sum_{n=0}^{3} = \frac{16}{81}$$

From the above calculations the probability that the hair stylist is busy is $1 - P_0 = 0.67$ and thus the probability that he is free is 0.33. The average number of customer in the system is

$$L = \frac{\lambda}{\mu - \lambda} = \frac{2}{3-2} = 2 \text{ customers}$$

The average time in the system is

$$w = \frac{L}{\lambda} = \frac{1}{\mu - \lambda} = \frac{2}{2} = 1 \text{ hour}$$

The average time the customer spends in the queue is

$$w_Q = w - \frac{1}{\mu} = 1 - \frac{1}{3} = \frac{2}{3} \text{ hour}$$

and the number of customer in the queue is

$$L_Q = \frac{\lambda^2}{\mu(\mu - \lambda)} = \frac{4}{3} \text{ customers}$$

Let's consider another example. Arrivals occur at rate $\lambda = 10$ per hour and management has a choice of two servers, one who works at rate $\mu_1 = 11$ customers per hour and the second at rate $\mu_2 = 12$ customers per hour. The utilizations are $\rho_1 = 10/11 = 0.909$ and $\rho_2 = 10/12 = 0.833$. Assume a M/M/1 model, then

$$L_1 = \frac{\lambda}{\mu_1 - \lambda} = \frac{10}{11 - 10} = 10$$

whilst

$$L_2 = \frac{\lambda}{\mu_2 - \lambda} = \frac{10}{12 - 10} = 5$$

Thus an increase in service rate from 11 to 12 customers per hour, a mere 9.1% increase, would result in a decrease in average number in system from 10 to 5, which is a 50% decrease!