Financial and Actuarial Modelling in R (MATH377)

Lecture Notes

Jorge Yslas

Contents

Preface

The objective of this module is to give a set of skills used in practice in financial and insurance institutions. We aim at introducing students to the basic principles and practices of the R programming language.

Chapter 1

Introduction to R

1.1 Installation

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and macOS. To help you write code in R, RStudio is a free application that makes the task significantly easier. To get started, you need to acquire your own copies of these two programs. If you already have R and Rstudio installed on your computer, it is a good idea to update to the latest versions by following the same step since some of the packages employed later work only with recent versions of this software.

1.1.1 Installation of R

We start with the installation of R. The essential files for installation and packages can be found on *The Comprehensive R Archive Network* (CRAN) website, http://cran.r-project.org/, as shown in Figure 1.1 below.

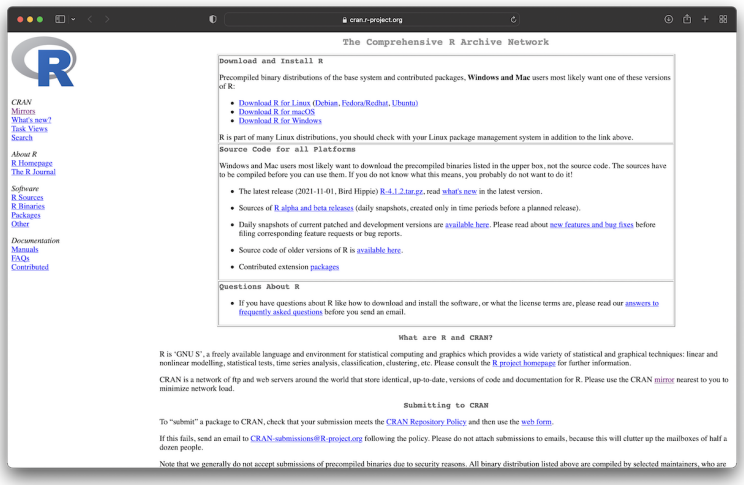


Figure 1.1: CRAN webpage.

For Windows users, click "Download R for Windows." Then, you will obtain a screen similar to Figure 1.2 (left) below. Next, click the "base" link. After that, click on the link at the top of the page that should say something like "Download R 4.2.2 for Windows" (see Figure 1.2, right). This will download an executable file for installation. Finally, open the executable and follow the installation wizard.

For macOS users, click "Download R for macOS." This will take you to a screen similar to Figure 1.3. Then, depending on your computer, click "R-4.2.2.pkg" for Mac computers with Intel processors or

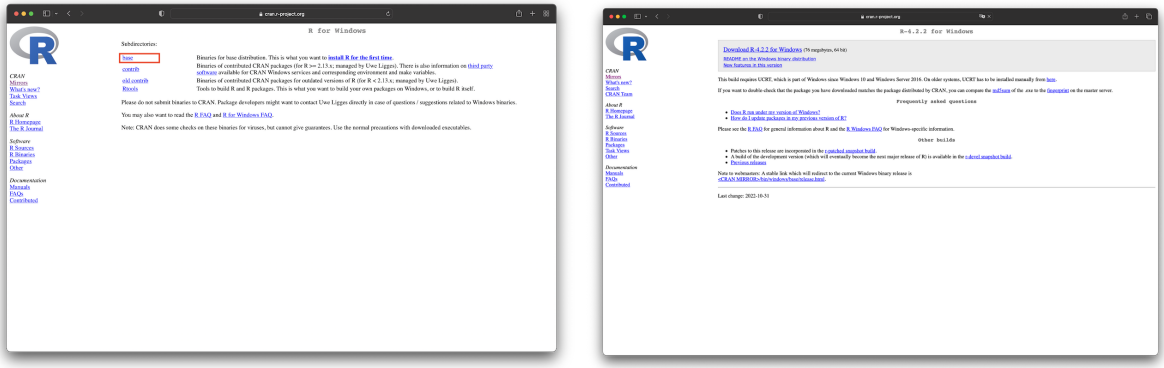


Figure 1.2: Download R for Windows.

"R-4.2.2-arm64.pkg" for Mac computers with Apple silicon. This will download an installation file. Finally, open the file and follow the installation steps.

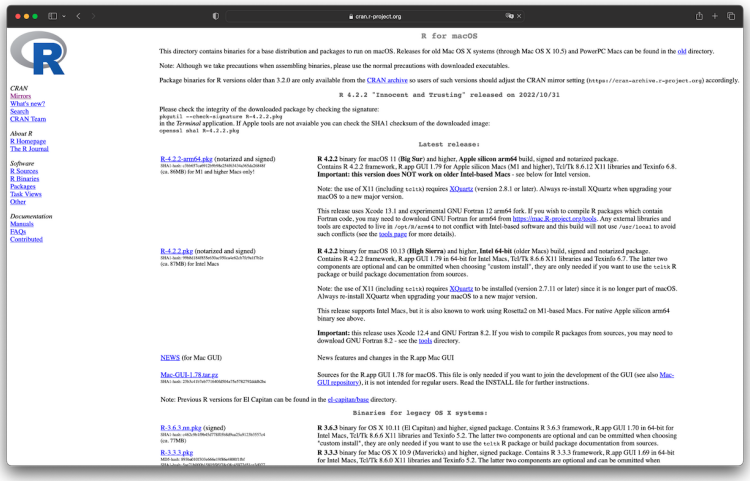


Figure 1.3: Download R for macOS.

If you followed the steps above, you should be able to open R. When starting R, a screen similar to Figure 1.4 should appear.

Remark. The steps above refer to version 4.2.2 of R, which corresponds to the most recent version at the time of writing. However, you should download the most current release of R.

1.1.2 Installation of Rstudio

Although the editor of R works fairly well, you should consider running R with RStudio. We will use RStudio here because it makes using R much easier and gives you access to many useful functionalities. For instance, these lecture notes were written using Rstudio. To download your copy, go to the Rstudio download page https://posit.co/download/rstudio-desktop/. Then, scroll down and click "DOWNLOAD RSTUDIO DESKTOP FOR MAC (WINDOWS)" under the heading "Step 2: Install RStudio Desktop" (see Figure 1.5), which will download the installer recommended according to your system. If the system suggested is incorrect, you can select the appropriate one from the installers list. Then, download the installation file, open it, and follow the installation steps.

Test the installation by opening Rstudio. You should see a screen similar to Figure 1.6.

Remark. Note that even if you use RStudio, you still need to download and install R. RStudio runs the version of R installed on your computer, and it does not come with a version of R on its own.

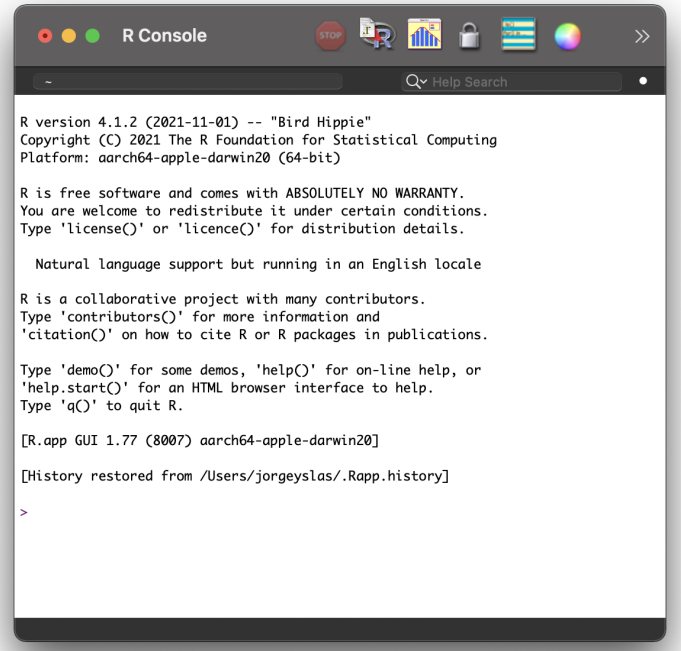


Figure 1.4: R console.

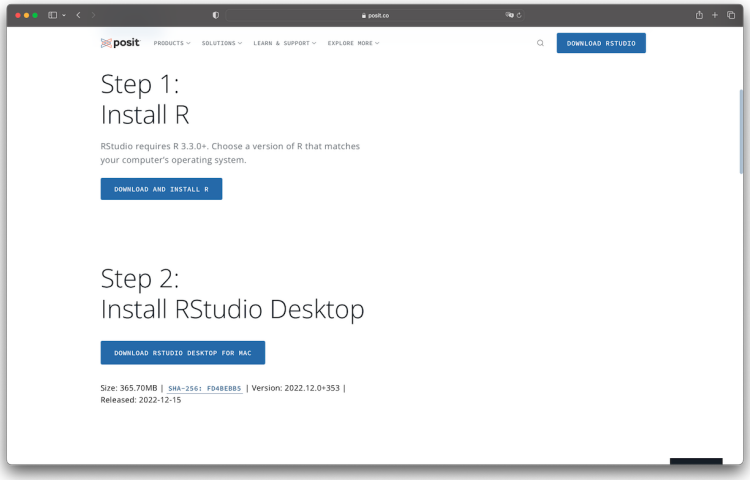


Figure 1.5: Download Rstudio.

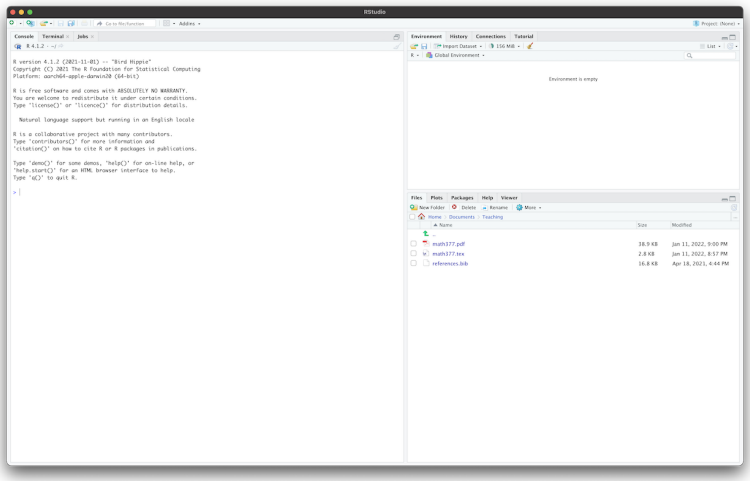


Figure 1.6: Rstudio.

1.2 R as a simple calculator

We will start by using R as a calculator. To run R code, go to the **console** tab (bottom left in the default Rstudio arrangement). You will see a `>` symbol called the **prompt**. Now, type `1 + 2` after the prompt and press Enter. RStudio will display the following:

```
> 1 + 2
[1] 3
```

The `[1]` that appears next to our result indicates that the line starts with the first value in our result. Some commands return more than one value, and their results may show up in multiple lines. For instance, type the command `10:50`, which produces 51 integer values from 10 to 50. As a result, you will see something like the following:

```
> 10:50
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
[24] 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Here, `[24]` indicates that the second line starts with the 24th value. We will come back to the `:` operator later.

If we type an incomplete command and hit Enter, R will display a `+` prompt. This means that R is waiting for the rest of our command. At this stage, we can finish the command and press Enter or hit the Escape key to start over.

```
> 2 -
+ 3 +
+ 5
[1] 4
```

In what follows, the code will be shown in this way:

```
1 + 2
```

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

Note that we no longer display the prompt and that the results are presented after two hashtags (`##`). The reason is that this makes it easier for you to copy and paste the code into your R console.

Returning to using R as a calculator, Table 1.1 shows the arithmetic operators available in R.

Lets now try some simple operations:

```
3^2 + 4 * 10
```

```
## [1] 49
```

Table 1.1: Arithmetic operators.

Addition	+
Subtraction	-
Multiplication	*
Division	/
Power	^
Integer division	%/%
Modulus (remainder of integer division)	%%

```
(3^2 + 4) * 10
```

```
## [1] 130
```

One need to be aware that the precedence for arithmetic operators in R follows the BODMAS order: Brackets ( ), Orders ^, Division / and Multiplication *, Addition + and Subtraction -.

Remark. If you want to learn more about how R gives precedence to all operators, type `?Syntax` in the console.

If we make a mistake in your code, R will let us know and indicate what the error is.

```
3 + *
```

```
## Error: <text>:1:5: unexpected '*'
## 1: 3 + *
##         ^
```

Example 1.1. Find the present value of an investment that gives 1 in 5 years, assuming that the interest rate is $r = 3\%$ compounded annually.

Solution.

```
(1 + 0.03)^(-5)
```

```
## [1] 0.8626088
```

R also has several common constants and mathematical functions. These include the exponential and log functions `exp()` and `log()`, and trigonometry functions such as `sin()` and `cos()`, among others.

```
pi
```

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

```
cos(2 * pi)
```

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

```
log(10) # The default option is base e
```

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

We can look at the full description of a function in R by using the "help." Let us try this with the `log()` function. For this, we need type `help(log)` or `?log`. This will display the documentation of the `log()` function in the tab **Help** (bottom right). Here, we will find out that `log()` computes by default the natural logarithm. Moreover, looking at the rest of the documentation, we will see that `log()` has, in fact, two *arguments*: `x` and `base`, the latter with the default value of the mathematical constant e. When calling this function, we can specify the argument names. In such a case, then the order does not matter:

```
log(x = 4, base = 10)
```

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

Table 1.2: Logical operators.

Less Than	<
Less Than or Equal To	<=
Greater Than	>
Greater Than or Equal To	>=
Equal To	==
Not Equal To	!=
Not	!
Or	|
And	&

```
log(base = 10, x = 4)
```

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

However, if we do not specify the argument names, R will decide that `x` is the first argument and `base` is the second one.

```
log(4, 10)
```

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

```
log(10, 4)
```

```
## [1] 1.660964
```

If we try to do an operation that is not mathematically defined, R will return `NaN` (Not a Number).

```
log(-2)
```

```
## Warning in log(-2): NaNs produced
```

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

```
0 / 0
```

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

1.2.1 Logical operators

Logical operators help evaluate certain conditions. For instance, we may be interested in checking if an insurance claim is greater than (`>`) a given deductible. If this is "true," then the insurance company may pay the amount above the deductible. Table 1.2 displays a list of logical operators in R. These operators return a *Boolean* value of either `TRUE` or `FALSE`.

Following, we have some simple examples:

```
4 <= 3
```

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

```
1 == 1
```

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

```
(3 != 2) & !(1 > 1)
```

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

1.3 R objects

An essential part of any programming language is the ability to store information in variables. R lets us save data by storing it inside an R object. An object is just a name that you can use to call up stored data. First, we need to see how to create objects.

1.3.1 Assignment

Objects in R are created using the assignment operator `<-`, that is, the less than symbol, `<`, followed by a minus sign, `-`. Simply choose a name followed by `<-` and the information you want to store. For example:

```
x <- 1
```

Our variable will now be part of what is called the "working space." We can print the value stored in this variable by typing its name

```
x
```

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

or, alternatively, we can use the `print()` function

```
print(x)
```

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

We need to make sure that the assignment operator is well written. Otherwise, we may obtain an error:

```
x < -1
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

There are a few rules regarding the name of an object in R. First, a name cannot start with a number. Second, a name cannot use some special symbols, like `^`, `!`, `$`, `@`, `+`, `-`, `/`, or `*`. Other than that, we can name our variables in any way we want. However, it is convenient to choose a name that will help us (and others) easily read our code. Note also that R is also case-sensitive. This means that in the following code, the variables `my_variable` and `My_variable` are different objects:

```
my_variable <- 1
My_variable <- 2
my_variable
```

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

If we use a name that is already taken, R will overwrite any previous information stored in that object:

```
x <- 3
x
```

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

```
x <- 10
x
```

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

Hence, it is good to check if the names we will use are already taken. We can see which object names we have already used with the function `ls()` or in the **Environment** tab of Rstudio:

```
ls()
```

```
## [1] "my_variable" "My_variable" "x"
```

We also need to be careful not to use the name of a function or constant that is already in R. This can lead to hard-to-identify errors or incorrect outputs:

```
pi <- 10

sin(2 * pi)
```

```
## [1] 0.9129453
```

To remove specific objects from the working space we can use the `rm()` function:

```
rm(my_variable)
ls()
```

```
## [1] "My_variable" "pi"          "x"
```

This can also be done using RStudio interface. We need to ensure the Environment tab is in Grid (not List) mode, tick the object(s) we want to remove from the environment and click the broom icon.

To clean the whole working space, we type

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

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

or click the broom icon in the Environment tab (List mode) in Rstudio.

Remark. Objects can also be created using `=` instead of `<-`. Although `=` is simpler to type, we do not recommend using it since it will make your code more difficult to read. Alternatively, one can use the shortcut `alt/opt` + `-` to generate `<-`.

Just as we performed calculations with numbers in the previous section, we can do the same with objects containing numbers references. Let us come back to Example 1.1. An alternative solution using objects would be as follow:

```
maturity <- 5
r <- 0.03
(1 + r)^(-maturity)
```

```
## [1] 0.8626088
```

As our analysis gets more complicated, we may want to save the results to access them later:

```
pv <- (1 + r)^(-maturity)
pv
```

```
## [1] 0.8626088
```

For instance, if we now want to compute the present value of an investment that gives 75 at the end of 5 years (assuming the same 3% interest rate), we can simply type:

```
75 * pv
```

```
## [1] 64.69566
```

Let us imagine that we need to repeat the analysis above, but this time using different inputs (for instance, a different interest rate). If we used the console solely, we would need to go back in the console history, change where needed, and rerun the rest of the code. This is particularly complicated when we have several lines of code. Hopefully, we can use **scripts** to avoid this situation. To open a new script, go to `File -> New File -> R script`. This will open a new script. Here we can write our code and run it afterward using shortcuts such as `Cmd+Enter` or using the **Run** button in Rstudio.

1.3.2 Data types

R has different types of variables. These are some of the most commonly used:

1. Numeric - Any number in $\mathbb{R}$.
2. Complex - Complex numbers ($\mathbb{C}$)
3. Character - Collection of characters. For instance, the name of a student.
4. Logical - TRUE and FALSE

A difference between R and other programming languages, such as C++, is that we can declare a variable without specifying its type. Instead, R will automatically determine the type.

Numeric

R is equipped with useful functions that can be used to find the type of a variable. For instance, the `class()` function:

```
x <- log(2)
x
```

```
## [1] 0.6931472
```

```
class(x)
```

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

We can also check if a variable is of a certain type. For instance, the function `is.numeric()` checks whether a variable is of the numeric type and returns a boolean value accordingly.

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

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

R can store data only up to a certain point. This means that, for instance, π, which has an infinite decimal representation, is computed up to a finite number of decimal points. This is the reason why expressions such as the next one do not give exactly 0 as a result:

```
exp(log(10)) - 10
```

```
## [1] 1.776357e-15
```

R has a special command to represent infinity: `Inf`. We can check which is the largest number below `Inf` using:

```
.Machine$double.xmax
```

```
## [1] 1.797693e+308
```

Any number above this will be converted to `Inf`.

```
2e+308
```

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

We can use logical operators to check if a number is below `Inf`.

```
2e+308 < Inf
```

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

```
2e+306 < Inf
```

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

Remark. The above notation using `e` is to specify scientific notation in R. For instance,

```
2e+2 # This corresponds to 2 * 10^2
```

```
## [1] 200
```

```
3.5e-1 # This corresponds to 3.5 * 10^(-1)
```

```
## [1] 0.35
```

`Inf` can also be used in calculations and to represent certain quantities:

```
1 / 0
```

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

```
1 / Inf
```

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

Remark. It is possible to work with **integers** in R. To specify that a number is an integer, we need to type `L` after the number:

```
x <- 2L
class(x)
```

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

However, integers can easily be converted to numeric objects when performing computations. Hence, it is very likely that one will never need to work with integers.

```
y <- 1 * x
class(y)
```

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

Complex

A complex number is specified by adding the suffix i.

```
x <- -3i
x
```

```
## [1] 0-3i
```

```
class(x)
```

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

We need to make sure to include a number before i. Otherwise, R will try to look for an object.

```
x <- i
```

```
## Error in eval(expr, envir, enclos): object 'i' not found
```

The order of real and complex parts does not matter.

```
x <- 2 - 1i
x
```

```
## [1] 2-1i
```

```
y <- -1i + 2
y
```

```
## [1] 2-1i
```

We can check if a variable is complex using the `is.complex()` function:

```
is.complex(x)
```

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

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

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

We can also perform calculations with this type of objects:

```
x + y
```

```
## [1] 4-2i
```

R has some built-in functions to transform the type of data. For instance, we could use `as.numeric()` to try to transform a complex into a numeric

```
as.numeric(x)
```

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

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

Note that a warning message is displayed, indicating that R only took the real part.

Character

A character string is specified by using quotation marks (").

```
x <- "Hello world"
x
```

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

```
class(x)
```

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

We can use single (') or double (") quotes, but not a mix.

```
x <- "Actuarial"
x
```

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

```
y <- 'Mathematics'
y
```

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

```
z <- 'Mix"
```

```
## Error: <text>:1:6: unexpected INCOMPLETE_STRING
## 1: z <- 'Mix"
##          ^
```

To check whether an object is a character string, we can use the `as.character()` function.

```
as.character(x)
```

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

We can concatenate two character strings by using the `paste()` function:

```
paste(x, y)
```

```
## [1] "Actuarial Mathematics"
```

Note that by default, `paste()` puts a space between the two character strings. We can change this by using the `sep` argument:

```
paste(x, y, sep = ".")
```

```
## [1] "Actuarial.Mathematics"
```

We can convert numeric variables into character variables by using the `as.character()` function.

```
x <- 2
as.character(x)
```

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

It is also possible to pass from character to numeric.

```
x <- "2"
as.numeric(x)
```

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

Finally, try the following code:

```
x <- "Hello"
as.numeric(x)
```

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

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

This will produce a `NA` (short for Not Applicable). `NA` is used to represent missing values, or unknown values, in R.

Logical

We have already mentioned the logical constants `TRUE` and `FALSE`, when describing logical operators. These can also be stored in an object as follows:

```
x <- TRUE
x
```

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

```
class(x)
```

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

Alternatively, we can use `T` and `F`:

```
y <- F
y
```

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

Again, we can check if a variable is of the logical type by using `is.logical()`:

```
is.logical(y)
```

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

When converting a logical variable into a numeric variable, we will obtain 1 (`TRUE`) or 0 (`FALSE`):

```
as.numeric(TRUE)
```

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

```
as.numeric(FALSE)
```

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

On the other hand, we can convert a numeric variable into a logical one using the `as.logical()` function. In such a case, any numeric value different to 0 will be converted to `TRUE` and only 0 to `FALSE`:

```
as.logical(2.3)
```

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

```
as.logical(0)
```

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

When performing computation, R automatically transforms `TRUE` to 1 and `FALSE` to 0, which can be very useful.

Example 1.2. A car insurance policy covers any losses above a deductible of 80. Compute the amount the company pays if a claim for 100 is registered.

Solution.

```
deductible <- 80
claim <- 100
payment <- (claim > deductible) * (claim - deductible)
payment
```

```
## [1] 20
```

What if the claim was for 70?

```
deductible <- 80
claim <- 70
payment <- (claim >= deductible) * (claim - deductible)
payment
```

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

1.4 Vectors

Vectors are made of a set of objects. The `c()` function can be used to create vectors, and it is the easiest way to define and store more than one value in R.

```
x <- c(0, 2, 6, -1, -5, 1)
x
```

```
## [1]  0  2  6 -1 -5  1
```

We can check if an object is a vector by using the function `is.vector()`:

```
is.vector(x)
```

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

In fact, we have been working with vectors (of length one) for quite some time:

```
y <- 2
is.vector(y)
```

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

The `c()` function also allows you to give other vectors as inputs; in that case, it will concatenate the vectors.

```
x <- c(c(0, 2), c(6, -1), -5, 1)
x
```

```
## [1]  0  2  6 -1 -5  1
```

There are other ways to generate (numeric) vectors. Next, we describe how to use the `rep()` and `seq()` functions to do so. Let us start with the function `rep()`. The first argument in `rep()` is `x`, which is the vector we want to repeat. The next argument is `times`, corresponding to a vector with the number of times we desire to repeat the elements of `x`. The simplest example is using `x` and `times` as vectors of length one:

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

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

Note that in the above code, we omitted the name of the argument `times`, given that it is the first optional argument. Next, we can give a vector of any length as input in `x` and `times` as vectors of length one:

```
rep(c(1, 2), 4) # repeats c(1, 2) four times
```

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

If in `times` we give a vector of length larger than one, this should coincide with the length of the vector passed on `x`, and it will repeat each entry of `x` according to the entries of `times`.

```
rep(c(1, 2), c(4, 2)) # repeats 1 four times and 2 two times
```

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

The next optional argument is `each`, which should be a nonnegative integer. It indicates to `rep()` that each entry of `x` must be repeated the number of times given in `each`.

```
rep(c(1, 2), each = 3) # repeats 1 three times and 2 three times
```

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

Finally, we also have the optional argument `length.out`, which specifies the length of the output.

```
rep(c(1, 2), each = 3, length.out = 5)
```

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

Next, we will use the function `seq(from, to, by, length.out)` to generate sequences of numbers. Note that this function has four arguments, and the output will depend on the values provided for these arguments. For example, if we want to generate a sequence of integers from 10 to 20, we can type:

```
seq(from = 10, to = 20, by = 1)
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Recall that if we omit the names of the arguments, R will take them in the order provided. So, the following code will give the same output as before:

```
seq(10, 20, 1)
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

As a matter of fact, if we look at the documentation of `seq()`, we can see that the default value of `by` is 1, so we can simply type:

```
seq(10, 20)
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Sequences of integers are extensively used in R programming. That is why we have the shortcut operator `:` (you may remember this operator) to produce the same result:

```
10:20
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

Let us try now with a different increment:

```
seq(1, 5, by = 0.1)
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8
## [20] 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7
## [39] 4.8 4.9 5.0
```

The fourth argument, `length.out`, can be used to specify the number of values to generate. The increment will be adjusted accordingly to obtain precisely the given number.

```
seq(1, 5, length.out = 10)
```

```
##  [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667 4.111111
##  [9] 4.555556 5.000000
```

So far, we have only created numeric vectors. However, creating vectors of the other data types is also possible.

```
x <- c("string1", "string2")
x
```

```
## [1] "string1" "string2"
```

```
x <- c(FALSE, TRUE)
x
```

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

```
x <- c(1 + 2i, 3 - 2i)
x
```

```
## [1] 1+2i 3-2i
```

We can also check the data type of a vector:

```
x <- c("string1", "string2")
class(x)
```

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

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

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

At this point, you may be wondering what happens when you mix objects. Try it for yourself:

```
class(c(2.5, "string"))
class(c(TRUE, 10))
class(c("string", FALSE))
class(c(2.5, 1i))
```

You will see that the assignment hierarchy is given as follows: character > complex > numeric > logical.

1.4.1 Accessing vector elements

To access and modify elements of a vector, we can employ `[]`. It is important to point out that, in stark contrast to other programming languages (e.g., C++), where the vector index starts in 0, in R, the index begins in 1. Hence, if we want to access the first element of a vector, we need to write the following:

```
x <- c(0, 2, 6, -1, -5, 1)
x[1]
```

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

This operator also allows you to change the value of that element:

```
x[1] <- 10
x
```

```
## [1] 10  2  6 -1 -5  1
```

If we try to access an element beyond the last element, we will get `NA`:

```
x[10]
```

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

To obtain the size (or length) of a vector, we can use the function `length()`:

```
length(x)
```

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

Thus, one way to access the last element of a vector is the following:

```
x[length(x)]
```

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

We may be interested in selecting multiple elements of a vector. There are different ways to do it, depending on what is required. Consider the vector

```
x <- c(0, 2, 6, -1, -5, 1)
```

To access the second and third elements, we can type:

```
x[c(2, 3)]
```

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

To obtain the elements from positions 2 to 5, we can run the following:

```
x[2:5]
```

```
## [1]  2  6 -1 -5
```

We can use negative numbers if we require all the vector entries except for certain elements. For instance, the following code returns the whole vector without the first element:

```
x[-1]
```

```
## [1]  2  6 -1 -5  1
```

For multiple elements:

```
x[c(-1, -3)]
```

```
## [1]  2 -1 -5  1
```

```
x[-c(1, 3)] # equivalently
```

```
## [1]  2 -1 -5  1
```

It turns out that we can use logical indexing to subtract data, which is a very powerful tool. Let us start with a very simple example. Consider the vector

```
x <- c(0, 2, 5)
```

If we type

```
x[c(TRUE, FALSE, TRUE)]
```

```
## [1] 0 5
```

we can see that R selects the entries with `TRUE` values. Now, take again the vector

```
x <- c(0, 2, 6, -1, -5, 1)
```

We can use the logical operators to see, for example, which entries are nonnegative:

```
x >= 0
```

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

Thus, if we want to subtract the nonnegative entries of that vector, we can type:

```
x[x >= 0]
```

```
## [1] 0 2 6 1
```

Alternatively, we can make use of the function `which()`. This function returns the indices that give `TRUE` to a logical object. For example,

```
which(x >= 0) # These are indices, not the values contained in the vector
```

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

Hence, we can subtract the nonnegative entries of that vector as follows:

```
x[which(x >= 0)]
```

```
## [1] 0 2 6 1
```

If the length of the logical vector is different. R will "recycle" information:

```
x[c(TRUE, FALSE)]
```

```
## [1]  0  6 -5
```

```
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
```

```
## [1]  0  6 -5
```

There are two other special cases that are worth mentioning. If we put *nothing* inside `[]`, we obtain the original vector

```
x[]
```

```
## [1]  0  2  6 -1 -5  1
```

If we put 0, then we obtain a vector of size 0, which is represented by `numeric(0)`

```
x[0]
```

```
## numeric(0)
```

1.4.2 Operations with vectors

Now, we will review some operations involving vectors.

Scalars and vectors

Let us start with operations between scalars (vectors of length one) and vectors.

```
x <- c(0, 2, 6, -1, -5, 1)
x * 2
```

```
## [1]   0   4  12  -2 -10   2
```

As we can see, R will return a new vector with entries, the entries of the original vector multiplied by the given scalar. In the same way, we can make other operations:

```
x / 4 + 2
```

```
## [1] 2.00 2.50 3.50 1.75 0.75 2.25
```

```
x^2
```

```
## [1]  0  4 36  1 25  1
```

```
2^x
```

```
## [1]  1.00000  4.00000 64.00000  0.50000  0.03125  2.00000
```

Scalar functions applied to vectors

If we apply a scalar function, such as `exp()`, `abs()` (absolute value), `round()`, etc., to a vector, R will compute such a function element-wise:

```
exp(x)
```

```
## [1] 1.000000e+00 7.389056e+00 4.034288e+02 3.678794e-01 6.737947e-03
## [6] 2.718282e+00
```

```
abs(x)
```

```
## [1] 0 2 6 1 5 1
```

Operations among vectors

We can also perform operations among vectors. The operators typically work element-wise. Let us try first with a sum:

```
x <- c(0, 2, 6, -1, -5, 1)
y <- c(1, -1, 2, 7, -2, 8)
x + y
```

Table 1.3: Some vector functions

"min()"	Minimum value in a vector
"max()"	Maximum value in a vector
"length()"	Number of elements in a vector
"sum()"	Sum of the elements in a vector
"mean()"	Mean of the elements in a vector
"median()"	Median of the elements in a vector
"var()"	Variance of the elements in vector
"sd()"	Standard deviation of the elements in

```
## [1]  1  1  8  6 -7  9
```

We can try other operators:

```
x * y
```

```
## [1]  0 -2 12 -7 10  8
```

```
x^y
```

```
## [1]  0.00  0.50 36.00 -1.00  0.04  1.00
```

If the vectors are of different sizes, R will *recycle* information.

```
x <- c(0, 2, 6, -1, -5, 1)
y <- c(1, -1)
x + y
```

```
## [1]  1  1  7 -2 -4  0
```

```
# Equivalent to
x <- c(0, 2, 6, -1, -5, 1)
y <- c(1, -1, 1, -1, 1, -1)
x + y
```

```
## [1]  1  1  7 -2 -4  0
```

Finally, note that the `*` operator performs element-wise multiplication. However, when working with mathematical vectors may want to compute the inner product. This can be done with the `%*%` operator:

```
x <- c(0, 2, 6, -1, -5, 1)
y <- c(1, -1, 2, 7, -2, 8)
x %*% y
```

```
##      [,1]
## [1,]   21
```

Remark. Recall that for two vectors $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$, the inner product $< \mathbf{x}, \mathbf{y} >$ is defined as

$$< \mathbf{x}, \mathbf{y} >= \sum_{i=1}^{n} x_i y_i \, .$$

Some vector functions

Data will be typically stored inside vectors. In order to analyze such data, we can make use of different R functions. For instance, Table 1.3 shows some useful functions.

Remark. For given data $x_1, \ldots, x_n$, the `mean()` and `var()` functions compute the sample mean and variance given by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \, , \quad s^2_{n-1} = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \, ,$$

respectively.

We can now compute several quantities of interest with all these functionalities at hand.

Example 1.3. Let us assume that the following insurance claims have been reported to an insurance company:

```
claims <- c(80, 20, 60, 55, 20.5, 100, 70, 89.9, 120)
```

How many claims were reported?

```
length(claims)
```

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

How much is the total amount of claims?

```
sum(claims)
```

```
## [1] 615.4
```

For which amount are the minimum and maximum claim amounts?

```
min(claims)
```

```
## [1] 20
```

```
max(claims)
```

```
## [1] 120
```

What is the average claim amount?

```
mean(claims)
```

```
## [1] 68.37778
```

This can also be computer using the `sum()` and `length()` functions:

```
sum(claims) / length(claims)
```

```
## [1] 68.37778
```

What is the standard deviation of the claims?

```
sd(claims)
```

```
## [1] 33.81489
```

Again, this can be computed using other functions:

```
sqrt(sum((claims - mean(claims))^2) / (length(claims) - 1))
```

```
## [1] 33.81489
```

Now assume that all these claims have a deductible of 50 and the insurance company pays the difference between a claim and the deductible if the former is larger. On average, how much does the company have to pay for the claims above the deductible?

```
deductible <- 50
claims_to_pay <- claims[claims > deductible]
payment <- claims_to_pay - deductible
payment
```

```
## [1] 30.0 10.0  5.0 50.0 20.0 39.9 70.0
```

```
mean(payment)
```

```
## [1] 32.12857
```

1.5 Matrices, data frames, and lists

1.5.1 Matrices

Sometimes we are interested in storing information in a matrix form. The simplest way to generate a matrix is to use the `matrix()` function. This function has different arguments (see the help - `?matrix`), so let us see how to use some of these options. The main argument of `matrix()` is `data`, which should be a data vector.

```
x <- matrix(c(1, 2, 3, 4, 5, 6))
x
```

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

Recall that R will execute the default option if no further arguments are passed. In the above case, it creates a one-column matrix. The next two arguments are `nrow` and `ncol`. These are used to define the dimension of a matrix:

```
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
x
```

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

```
y <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
y
```

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

We can omit the argument names, and R will assign the input to the arguments in the order they are defined in the function. In this case, `nrow` first and `ncol` second (again, see help - `?matrix`).

```
x <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
x
```

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

```
y <- matrix(c(1, 2, 3, 4, 5, 6), 3, 2)
y
```

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

Note that the above matrices are filled by columns (the default). If we want to change this, we can use the next argument `byrow`.

```
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2, byrow = TRUE)
x
```

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

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

```
y <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)
y
```

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

In these notes, we will mainly use the default option of filling by columns.

If we only specify one of the arguments, `ncol` or `nrow`, R will attempt to find the other parameter from the length of the data.

```
x <- matrix(c(1, 2, 3, 4, 5, 6), ncol = 2)
x
```

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

```
y <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)
y
```

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

Finally, if we specified the dimensions of the matrix and provided a vector of length larger than `ncol` x `nrow` (i.e., the number of entries of the matrix), R will cut out the last elements.

```
x <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8), 3, 2)
```

```
## Warning in matrix(c(1, 2, 3, 4, 5, 6, 7, 8), 3, 2): data length [8] is not a
## sub-multiple or multiple of the number of rows [3]
```

```
x
```

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

The last argument, `dimnames`, helps us give names to a matrix's rows and columns.

```
x <- matrix(c(1, 2, 3, 4, 5, 6), 3, 2,
            dimnames = list(c("Row1", "Row2", "Row3"), c("Column1", "Column2")))
x
```

```
##      Column1 Column2
## Row1       1       4
## Row2       2       5
## Row3       3       6
```

Alternatively, names can be added (and modified) by using the `colnames()` and `rownames()` functions. Additionally, these functions can help to keep the code clearer.

```
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
colnames(x) <- c("Column1", "Column2")
rownames(x) <- c("Row1", "Row2", "Row3")
x
```

```
##      Column1 Column2
## Row1       1       4
## Row2       2       5
```

```
## Row3       3       6
```

Remark. Sometimes we will encounter matrices that are defined using data vectors created with the `c()` function applied multiple times. For instance,

```
x <- matrix(c(c(1, 2, 3), c(4, 5, 6)), 3, 2)
x
```

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

The reason is that this "notation" can help to distinguish the different columns of your input, so it is easier to read and identify mistakes. Remember that `c(c(1, 2, 3), c(4, 5, 6))` produces the same vector as `c(1, 2, 3, 4, 5, 6)`.

We can verify if an object is a matrix using the `is.matrix()` function:

```
x <- matrix(c(1, 2, 3, 4, 5, 6), 3, 2)
is.matrix(x)
```

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

```
y <- c(1, 2, 3)
is.matrix(y)
```

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

Another important function for matrices is `dim()`. It has two functionalities. The first one is to tell the dimension of a matrix object.

```
x <- matrix(c(1, 2, 3, 4, 5, 6), 3, 2)
dim(x)
```

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

If we only require the number of columns or rows, we can use the `ncol()` and `nrow()` functions:

```
nrow(x)
```

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

```
ncol(x)
```

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

The second functionality of `dim()` is transforming vectors into matrices.

```
x <- c(1, 2, 3, 4, 5, 6)
dim(x) <- c(3, 2)
x
```

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

A matrix can also be obtained by first generating two or more vectors and then combining them using the `cbind()` or the `rbind()` functions. The `cbind()` function combine the vectors by making them columns of a new matrix:

```
x <- 2:5
y <- 9:6
cbind(x, y) # Note that the columns have been named
```

```
##      x y
## [1,] 2 9
```

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

On the other hand, the `rbind()` function makes those vectors rows of a new matrix:

```
rbind(x, y) # Note that the rows have been named
```

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

We can also create diagonal matrices by using the `diag()` function.

```
diag(c(1, 2, 3, 4))
```

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

To create an identity matrix, we simply type:

```
diag(4)
```

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

Accessing matrix elements

To access elements of a matrix, we can use the `[]` notation. This is done in a similar way as in vectors, but in this case, we have to specify the rows and columns (`[row, column]`).

```
x <- matrix(1:6, 3)
x
```

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

```
x[3, 2]
```

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

We can also modify these values:

```
x[3, 2] <- 8
x
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    8
```

If we want a whole row of the matrix, we leave the second argument blank:

```
x[2, ]
```

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

Similarly, for columns:

```
x[, 1]
```

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

We can select multiple rows (or columns) using the `c()` function:

```
x[c(1, 3), ]
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    3    8
```

```
x[c(1, 3), 2]
```

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

Negative values can be used to discard certain rows (or columns):

```
x[, -2]
```

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

```
x[-c(1, 3), -2]
```

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

Additionally, we can also use logical operators to subtract data. For instance, consider the following matrix:

```
x <- matrix(c(c(10, -2, 3), c(8, 0, -5)), 3, 2)
x
```

```
##      [,1] [,2]
## [1,]   10    8
## [2,]   -2    0
## [3,]    3   -5
```

If, for example, we were required to subset the matrix in such a way that we only keep the rows with values in the first column that are nonnegative, this can be done as follows:

```
x[x[, 1] >= 0, ]
```

```
##      [,1] [,2]
## [1,]   10    8
## [2,]    3   -5
```

What about subsetting the matrix in such a way that we only keep rows that has only nonnegative in both columns:

```
x[x[, 1] >= 0 & x[, 2] >= 0, ]
```

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

Finally, if our matrix has column or row names, we can use those names to subtract information:

```
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
colnames(x) <- c("Column1", "Column2")
rownames(x) <- c("Row1", "Row2", "Row3")
x
```

```
##      Column1 Column2
## Row1       1       4
## Row2       2       5
## Row3       3       6
```

```
x["Row2", "Column1"]
```

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

```
x[c("Row2", "Row1"), "Column1"]
```

```
## Row2 Row1
##    2    1
```

Remark. In the last example, note that the output of the last command is a vector with names for its elements. We can also give names to the entries of a vector by using the `names()` function. For instance,

```
x <- c(1,2)
names(x) <- c("name1", "name2")
x
```

```
## name1 name2
##     1     2
```

However, this functionality is not used very often.

Operations with matrices

Matrices behave pretty much in the same way as vectors when dealing with scalars and functions of scalars:

```
x <- matrix(1:6, 3)
x
```

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

```
x^2 # Each element to the power 2
```

```
##      [,1] [,2]
## [1,]    1   16
## [2,]    4   25
## [3,]    9   36
```

```
exp(x) # Exp applied element-wise
```

```
##           [,1]      [,2]
## [1,]  2.718282  54.59815
## [2,]  7.389056 148.41316
## [3,] 20.085537 403.42879
```

When working with matrices of the same dimension, the computations will be performed element-wise:

```
x <- matrix(1:6, 3)
x
```

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

```
y <- matrix(6:1, 3)
y
```

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

```
x + y
```

```
##      [,1] [,2]
## [1,]    7    7
```

```
## [2,]    7    7
## [3,]    7    7
```

```
x^y
```

```
##      [,1] [,2]
## [1,]    1   64
## [2,]   32   25
## [3,]   81    6
```

Remark. We can also perform calculations between vectors and matrices. However, we need to be aware that R will recycle information to do so. For instance,

```
x <- matrix(1:6, 3)
y <- c(0.5, 2)
x * y
```

```
##      [,1] [,2]
## [1,]  0.5  8.0
## [2,]  4.0  2.5
## [3,]  1.5 12.0
```

```
# Equivalently
y <- matrix(rep(c(0.5, 2), 3), 3)
y
```

```
##      [,1] [,2]
## [1,]  0.5  2.0
## [2,]  2.0  0.5
## [3,]  0.5  2.0
```

```
x * y
```

```
##      [,1] [,2]
## [1,]  0.5  8.0
## [2,]  4.0  2.5
## [3,]  1.5 12.0
```

There are other important operations among matrices. Let us review some of them. To perform matrix multiplication, we need to use the `%*%` operator (remember `*` is an element-wise product):

```
x <- matrix(1:4, 2)
x
```

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

```
y <- matrix(4:1, 2)
y
```

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

```
x %*% y
```

```
##      [,1] [,2]
## [1,]   13    5
## [2,]   20    8
```

The transpose of a matrix can be computed with the `t()` function:

```
x <- matrix(1:6, 3)
x
```

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

```
t(x)
```

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

The inverse of a matrix can be obtained with the `solve()` function (we recommend checking the help for more details `?solve()`):

```
x <- matrix(1:4, 2)
solve(x)
```

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

```
x %*% solve(x) # To check the result
```

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

We may also be interested in performing operations column- or row-wise. For instance, compute the mean by row (or column). To do so, we can use vector functions in conjunction with the `apply(X, MARGIN, FUN, ...)` function. The argument of `apply()` are: `X` a matrix, `MARGIN`, which can be 1 for rows, 2 for columns, or `c(1, 2)` for rows and columns, and `FUN` the function to be applied. For instance, let us compute the mean by rows and columns:

```
x <- matrix(1:6, 3)
x
```

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

```
apply(x, 1, mean)
```

```
## [1] 2.5 3.5 4.5
```

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

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

Remark. Matrices can be seen as 2-dimensional arrays. We can create n-dimensional arrays by using the `array()` function.

```
x <- array(c(1:4, 11:14, 21:24), dim = c(2, 2, 3)) # We used dim for the dimensions
x
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]   11   13
```

```
## [2,]   12   14
##
## , , 3
##
##      [,1] [,2]
## [1,]   21   23
## [2,]   22   24
```

Access to an array's elements is done the same way as for matrices, using `[]`.

```
x[1, , ]
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    3   13   23
```

```
x[, 1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
```

```
x[, , 1]
```

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

However, array objects do not have the same versatility as matrices.

1.5.2 Data frames

Matrices can store only one data type. However, we may want to save more than one type of data in a matrix type of format — for instance, students' names and their grades. Data frames allow us to do precisely that. To create a data frame, we use the `data.frame()` function.

```
names <- c("student1", "student2", "student3", "student4")
grades <- c(90, 95, 85, 70)
students <- data.frame(names, grades)
students
```

```
##      names grades
## 1 student1     90
## 2 student2     95
## 3 student3     85
## 4 student4     70
```

Note that `data.frame()` has named the object's columns. We can check if an R object is a data frame with the `is.data.frame()` function:

```
is.data.frame(students)
```

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

Data frames allow access to their data via `[]` in the same way as matrices:

```
students[1, ] # Information of the first student
```

```
##      names grades
## 1 student1     90
```

```
students[, 1] # Grades
```

```
## [1] "student1" "student2" "student3" "student4"
```

```
students[2, 2] # Grades of second student
```

```
## [1] 95
```

```
students[students[, 2] > 80, ] # Students with grades above 80
```

```
##      names grades
## 1 student1     90
## 2 student2     95
## 3 student3     85
```

Data frames can also use the `$` operator to access data in a specific column. We simply type the name of the data frame object followed by the `$` operator and the column's name. For instance,

```
students$grades
```

```
## [1] 90 95 85 70
```

Moreover, to access a specific element of that column, we can type:

```
students$grades[1]
```

```
## [1] 90
```

Alternatively, we can use the name of the column inside `[]` to access its data:

```
students[, "grades"]
```

```
## [1] 90 95 85 70
```

The next thing we may want to do with a data frame is to add more information. This can be done with the `cbind()` function. For instance,

```
age <- c(20, 21, 21, 19)
students <- cbind(students, age)
students
```

```
##      names grades age
## 1 student1     90  20
## 2 student2     95  21
## 3 student3     85  21
## 4 student4     70  19
```

Alternatively, we can use the `$` operator:

```
students$email <- c("st1@liv.co", "st2@liv.co", "st3@liv.co", "st4@liv.co")
students
```

```
##      names grades age      email
## 1 student1     90  20 st1@liv.co
## 2 student2     95  21 st2@liv.co
## 3 student3     85  21 st3@liv.co
## 4 student4     70  19 st4@liv.co
```

Or even `[]`:

```
students[, 5] <- c(1, 2, 3, 4)
```

The names of the columns can be accessed (and changed) using the function `names()`:

```
names(students)
```

```
## [1] "names"  "grades" "age"    "email"  "V5"
```

```
names(students)[5] <- "id"
students
```

```
##      names grades age      email id
```

```
## 1 student1     90  20 st1@liv.co  1
## 2 student2     95  21 st2@liv.co  2
## 3 student3     85  21 st3@liv.co  3
## 4 student4     70  19 st4@liv.co  4
```

1.5.3 Lists

A list enables the storage of a variety of objects into a single one. In contrast to data frames, lists can contain different data types of different lengths. Lists can be generated in R using the function `list()`.

```
month <- c("Jan", "Feb", "Mar")
value <- c(1.21, 1.245, 1.402)
type <- "monthly -interest -rate"
rate <- list(Month = month, Value = value, Type = type)
rate
```

```
## $Month
## [1] "Jan" "Feb" "Mar"
##
## $Value
## [1] 1.210 1.245 1.402
##
## $Type
## [1] "monthly -interest -rate"
```

The structure of a list object can also be displayed (in a more compact format) using `str()`:

```
str(rate)
```

```
## List of 3
##  $ Month: chr [1:3] "Jan" "Feb" "Mar"
##  $ Value: num [1:3] 1.21 1.25 1.4
##  $ Type : chr "monthly -interest -rate"
```

As is the case with data frames, we can use `$` to access information:

```
rate$Value
```

```
## [1] 1.210 1.245 1.402
```

```
rate$Month[1]
```

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

We can also use `[]`. However, the notation, in this case, is slightly different:

```
rate[[1]] # First element of a list, in this case, the months
```

```
## [1] "Jan" "Feb" "Mar"
```

```
rate[[2]][2] # Second value
```

```
## [1] 1.245
```

To add more data to a list, we can do it in different ways. First, using the `$` operator:

```
rate$Year <- 2021
str(rate)
```

```
## List of 4
##  $ Month: chr [1:3] "Jan" "Feb" "Mar"
##  $ Value: num [1:3] 1.21 1.25 1.4
##  $ Type : chr "monthly -interest -rate"
##  $ Year : num 2021
```

Second, using `[]`:

```
rate[[5]] <- c("UK")
str(rate)
```

```
## List of 5
##  $ Month: chr [1:3] "Jan" "Feb" "Mar"
##  $ Value: num [1:3] 1.21 1.25 1.4
##  $ Type : chr "monthly -interest -rate"
##  $ Year : num 2021
##  $      : chr "UK"
```

Names can be handled in the same way as with data frames.

```
names(rate)[5] <- "Country"
str(rate)
```

```
## List of 5
##  $ Month  : chr [1:3] "Jan" "Feb" "Mar"
##  $ Value  : num [1:3] 1.21 1.25 1.4
##  $ Type   : chr "monthly -interest -rate"
##  $ Year   : num 2021
##  $ Country: chr "UK"
```

1.6 Functions

So far, we have been only using built-in R functions. However, R also allows us to create our own functions. The basic syntax to define a new R function is shown below:

```
function_name <- function(arg1, arg2, ...) {
  statement(s)
}
```

Let us try a very simple example first: compute the area of a circle given its radius. An implementation would be the following:

```
area_circle<- function(radius) {
   pi * radius^2
}
```

We can now call our function:

```
area_circle(2)
```

```
## [1] 12.56637
```

We now consider a slightly more complicated example: Program the density function of an exponential distributed random variable X with mean λ^{-1}, $\lambda > 0$. Recall that this density function is given by

$$f(x) = \lambda \exp(-\lambda x)\,, \quad x > 0\,.$$

We write $X \sim Exp(\lambda)$.

We now require two inputs for our function: the parameter λ and the point x where we evaluate the density.

```
den_exponential<- function(x, lambda) {
   lambda * exp(-lambda * x)
}
```

```
den_exponential(1, 0.5)
```

```
## [1] 0.3032653
```

When defining a function, we can give default values to the arguments. For example, let us make the default option of our exponential density the standard exponential, i.e., $\lambda = 1$:

```
den_exponential <- function(x, lambda = 1) {
   lambda * exp(-lambda * x)
}
```

Thus, if we do not provide the function with the second parameter, R will use 1.

```
den_exponential(1)
```

```
## [1] 0.3678794
```

We now give some important considerations when working with functions:

- You can return the result of a function using the `return()` function. In our functions above, we did not make use of this function, and the reason is that, by default, R returns the result of the last evaluated expression. However, `return()` can be used for early returns. For instance, the function below terminates after the first line and returns 1, ignoring the second line.

```
dummy_function <- function(x) {
  return(1)
  x * 2
}
dummy_function(10)
```

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

- We can create objects inside a function, and these will be erased after executing the function.

```
dummy_function <- function(x) {
  z <- 10
  x * z
}
dummy_function(10)
```

```
## [1] 100
```

```
z
```

```
## Error in eval(expr, envir, enclos): object 'z' not found
```

- R will look for objects in the global environment if they are not defined inside a function.

```
z <- 2
dummy_function <- function(x) {
  x * z
}
dummy_function(10)
```

```
## [1] 20
```

```
z # Keeps the value 2
```

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

However, if we create an object inside a function with the same name as an object in the global environment, the function will use the object defined inside the function's body.

```
z <- 2
dummy_function <- function(x) {
  z <- 5
  x * z
}
dummy_function(10)
```

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

```
z # Keeps the value 2
```

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

- R does not alter objects given as input of a function. Instead, it creates copies.

```
x <- c(2, 6, 1, 9)
dummy_function <- function(x) {
  x <- sort(x)
  x
}
dummy_function(x)
```

```
## [1] 1 2 6 9
```

```
x
```

```
## [1] 2 6 1 9
```

1.7 Packages

R comes with a preloaded selection of functions. However, contributors can make their functions available via an R package. In fact, you can make your function available in the form of an R package. The general way to install a package in CRAN is as follows:

```
install.packages("package_name")
```

Let us try this by installing the `derivmkts` package:

```
install.packages("derivmkts")
```

We can install multiple packages using `c()`.

```
install.packages(c("ggplot2", "devtools"))
```

Once a package is installed, we need to call it using the `library()` command in order to access all its functionalities.

```
library(package_name)
```

For instance,

```
library(derivmkts)
```

R packages in CRAN are updated regularly with new functions, fix bugs, performance improvements, etc. To update an R package, we can use the `update.packages()` function. The syntax is similar to `install.packages()`, for instance,

```
update.packages(c("ggplot2", "devtools"))
```

If a package in CRAN is not updated regularly, it can potentially be removed from CRAN. This means that sometimes when a package has not been updated for a while, we will not be able to install it using `install.packages()`. Let us try, for instance, with the `CASdatasets` package:

```
install.packages("CASdatasets")
```

However, the source code of an older version may still be available, and we can use that to install the package using the command

```
install.packages(path_to_file, repos = NULL, type = "source")
```

where `path_to_file` is the location and name of the file. Following the example of `CASdatasets` package, it is available to download at http://cas.uqam.ca. Try to install this package from source (you may need to install first the packages `xts` and `sp`). As previously mentioned, packages in CRAN are updated regularly. This means that their functionality may change over time by adding, modifying, or even erasing functions. Fortunately, CRAN keeps an archive of previous versions of each package. Hence, the previous method can also be used to install an older version of a package.

Remark. Submitting a package to CRAN requires passing certain tests, documenting properly, and constantly updating. That is why some developers make their code available in other places, for instance, GitHub. We can install a package from almost anywhere using the `devtools` package. For example, to install a package in GitHub, we can use the function `install_github()`.

A powerful characteristic of R is that it allows writing code in other computer languages, such as C++. However, this also means that some packages will require compilation for their installation. Thus, we need to set up our computer to do so. For Windows computers, we need to download and install Rtools, which can be found at the following link: https://cran.r-project.org/bin/windows/Rtools/. If you have installed version 4.2.2 of R (or any 4.2.x version), you will need RTools 4.2. Otherwise, you will need to select the version compatible with your R version. For Mac computers, we have two options: Download and install Xcode from the Apple store or install Xcode command line by running the following in the terminal

```
sudo xcode-select --install
```

In addition, some packages are written using Fortran, and this will require installing a Fortran compiler for macOS. The instruction can be found here: https://mac.r-project.org/tools/. However, installing Xcode or Xcode command line will work fine for the packages used in the present notes.

Test that your setup is working by installing the `actuar` package:

```
install.packages("actuar")
```

Finally, if after loading a package, we require to detach it, we can use the `detach()` function:

```
detach("package:ggplot2", unload = TRUE)
```

We can also remove packages from your library with the `remove.packages()` function:

```
remove.packages("ggplot2")
```

Remark. Installing, loading, updating, detaching, and erasing packages can also be done in the **Packages** tab of Rstudio.

1.8 Control Statements

1.8.1 Conditional statements

Conditional statements evaluate if a condition is `TRUE` and perform an action depending on the result. We start with the `if` statement. Its structure is as follows:

```
if (condition) {
  action(s)
}
```

Below, there is a very simple example:

```
claim <- 50
deductible <- 40
if (claim > deductible) {
  print("Pay claim")
}
```

```
## [1] "Pay claim"
```

In this case, the `claim` is larger (`>`) than our `deductible`, so our condition is `TRUE`. Thus, R will execute the code contained inside `{}`. In this case, print "Pay claim."

Remark. If our code after the `if` statement is only one line, `{}` can be committed. For instance, the above code could have been written as follow:

```
if (claim > deductible)
  print("Pay claim")
```

```
## [1] "Pay claim"
```

Sometimes we want to perform an action in case of condition is `FALSE`. In such a case, we can use the `else` statement.

```
if (condition) {
  action(s) in case of TRUE
} else {
  action(s) in case of FALSE
}
```

Here there is a simple example:

```
claim <- 30
deductible <- 40
if (claim > deductible) {
  print("Pay claim")
} else {
  print("Do not pay claim")
}
```

```
## [1] "Do not pay claim"
```

Remark. When if/else statements are simple, one can also make use of the `ifelse()` function:

```
ifelse(claim > deductible, "Pay claim", "Do not pay claim")
```

```
## [1] "Do not pay claim"
```

An advantage of the `ifelse()` function is that it can evaluate vectors.

```
x <- c(1, 2, 3, 4)
ifelse(x %% 2 == 0, "even", "odd")
```

```
## [1] "odd"  "even" "odd"  "even"
```

What if we want to evaluate several if/else conditions? We can use `if else` statements to do so.

```
if (condition1) {
  action(s) in case condition1 TRUE
} else if (condition2) {
  action(s) in case condition2 TRUE
} else {
  action(s) in case condition1 and condition2 FALSE
}
```

```
x <- 70
if (x < 50) {
  print("low number")
} else if (50 <= x & x <= 80) {
  print("medium number")
} else {
  print("high number")
}
```

```
## [1] "medium number"
```

1.8.2 Loop statements

Loops statements are used to repeat code. R has two loop statements: `for` and `while`.

while statements

A `while` statement repeats lines of code while a certain condition is `TRUE`, and it stops when it is `FALSE`. The general structure of a `while` loop is the following:

```
while (condition) {
```

```
  action(s)
}
```

We now give a simple example.

Example 1.4. Compute the cumulative sum of integers from 1 to 20 using a `while` loop.

Solution.

```
sum_int <- 0
i <- 1
while (i <= 20) {
  sum_int <- sum_int + i
  i <- i + 1
}
sum_int
```

```
## [1] 210
```

for statements

When we know exactly how many times we need to repeat a loop, we can use a `for` loop. The format of this statement is given below:

```
for (index in vector) {
  action(s)
}
```

The most common way to use a `for` loop is in conjunction with the `:` operator. For instance, if we want to solve Example 1.4 with a `for` loop, this can be done as follows:

```
sum_int <- 0
for (i in 1:20) {
  sum_int <- sum_int + i # value of i changes automatically
}
sum_int
```

```
## [1] 210
```

However, we can use a `for` loop with any other vector.

```
capital <- 1
yr_rates <- c(0.05, 0.03, 0.02)
for (r in yr_rates) {
  capital <- capital * (1 + r)
}
capital
```

```
## [1] 1.10313
```

1.9 Vectorized operations

R has been thoughtfully optimized for several operations with vector (and matrix) objects. This is commonly called "vectorization" and allows you to write concise, efficient, and easy-to-read code. For instance, let us consider the element-wise sum of two vectors. This can be done as follows:

```
x <- 1:10
y <- 11:20
x + y
```

```
##  [1] 12 14 16 18 20 22 24 26 28 30
```

However, the same calculation can be done with a `for` loop (without vectorization):

```
z <- numeric(length(x))
for (i in 1:length(x)) {
  z[i] <- x[i] + y[i]
}
z
```

```
##  [1] 12 14 16 18 20 22 24 26 28 30
```

We can see that the first code is simpler to read and easier to implement (less typing). Well, it turns out that it is also more efficient. To see this, we need to measure the time each code takes to perform the same calculation, and we can use the `microbenchmark` package to do it. First, we need to install the package.

```
install.packages("microbenchmark")
```

To use the functionalities of `microbenchmark`, we need to convert our previous code into functions:

```
sum_vect <- function(x, y) {
  x + y
}

sum_no_vect <- function(x, y) {
  z <- numeric(length(x))
  for (i in 1:length(x)) {
    z[i] <- x[i] + y[i]
  }
  z
}
```

We now use `microbenchmark()` to measure the running times of the above functions:

```
library(microbenchmark)
x <- 1:10
y <- 11:20
microbenchmark(sum_vect(x, y), sum_no_vect(x, y), times = 10)
```

```
## Warning in microbenchmark(sum_vect(x, y), sum_no_vect(x, y), times = 10): less
## accurate nanosecond times to avoid potential integer overflows
```

```
## Unit: nanoseconds
##               expr  min   lq     mean median   uq     max neval
##     sum_vect(x, y)  205  246  39987.3  246.0  328  394871    10
##  sum_no_vect(x, y) 1066 1189 216336.5 1250.5 1599 2151352    10
```

We can see that the performance of the vectorized version is, on average, around 5 times (this number may vary from computer to computer) faster than the implementation using a `for` loop. In conclusion, we should aim to do as many vectorized operations as possible in our code.

1.10 Reading and writing data

In practice, data is commonly provided in the form of an external file. This can be a plain text file (.txt), a comma-separated values file (.csv), an MS Excel file, etc. Hence, it is necessary that we learn how to read and write data in different file formats.

1.10.1 Working directory

One concept that is fundamental for reading and writing data is the "working directory." This is the directory where R will first look for files. To see our current working directory, we can use the `getwd()` function.

```
getwd()
```

If we want to change it to a different directory, this can be done with the `setwd()` function.

```
setwd("directory")
```

However, this can also be done using the Rstudio interface. For instance, to see our current working directory, we can go to the **Files** tab, click `More`, and then select `Go to Working Directory`. If we want to set a new working directory, this can be done by going to the **Files** tab, look for the directory that we want as the working directory, click `More`, and then select `Set as Working Directory`. Alternatively, we can go to the `Session` menu, `Set Working Directory`, and then select one of the options available.

1.10.2 Writing data

To store data in a file, we can use the `write.table()` and `write.csv()` functions. Typically, the object to be written would be a data frame or a matrix. Hence, let us consider the following data frame:

```
names <- c("student1", "student2", "student3", "student4")
grades <- c(90, 95, 85, 70)
students <- data.frame(names, grades)
students
```

```
##      names grades
## 1 student1     90
## 2 student2     95
## 3 student3     85
## 4 student4     70
```

Now we can use `write.table()` to write the data in a file. For instance, to generate a .txt file, we can use the following:

```
write.table(students, file = "students.txt")
```

This would produce a file that looks like this :

```
"names" "grades"
"1" "student1" 90
"2" "student2" 95
"3" "student3" 85
"4" "student4" 70
```

By default, `write.table()` separates the data with a space and includes the column and row names. We can adjust the above code to make a comma-separated values file (.csv) that does not include the row names.

```
write.table(students, file = "students.csv", sep = ",", row.names = FALSE)
```

Now the file looks like this:

```
"names","grades"
"student1",90
"student2",95
"student3",85
"student4",70
```

Since csv files are easier to read for most programs (including MS Excel), we have a specific function to write this type of files: `write.csv()`.

```
write.csv(students, file = "students.csv", row.names = FALSE)
```

The above code generates exactly the same csv file as before.

1.10.3 Reading data

Reading data in the form of text can be done with the `read.table()` and `read.csv()` functions.

For example, if we now want to read the csv file produced before, we can do it as follows:

```
the_students <- read.table("students.csv", header = TRUE, sep = ",")
# Our file is comma delimited and with headers
the_students
```

```
##      names grades
## 1 student1     90
## 2 student2     95
## 3 student3     85
## 4 student4     70
```

With the above, the data is now inside a data frame. Alternatively, we could have used the `read.csv()` function:

```
the_students <- read.csv("students.csv", header = TRUE)
```

Remark.

- Importing data can also be done with Rstudio. Simply click `Import Dataset` in the tab **Environment**.
- To import data from Excel, perhaps the easier method is to export the worksheet that contains your data to a csv file and then read it in R as described above. However, you can also use an R package such as `readxl`, which provides the function `read_excel()` to do so.

Chapter 2

R for Statistical Inference

2.1 Descriptive statistics

When working with real-life data, one of the first things we may want to do is get a general understanding of the data. For this purpose, we can use some descriptive statics and plots available in R.

R comes with some databases that can be accessed using the `data()` function. Moreover, we can see the list of data sets available by typing `data()`. For example, we can find the `iris` data set (use `?iris` for a complete description of the data). Although not an insurance or financial data set, we will use `iris` in this section since it is readily available, and the tools presented here do not depend on the source of the data.

First, we need to load the data into our working space using `data()`:

```
data(iris)
```

Now, the `iris` should appear in the **Environment** tab. Note that `iris` is a data frame, which is one of the most common ways of presenting data in R.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Remark. Some R packages include data sets. For instance, install and load the package `insuranceData`. Here you will find the `dataCar` data set.

The first thing that we may want to do is to compute some measures of central tendency and variability. Let us focus for now on the column `Sepal.Lenght`. Regarding measures of central tendency, these refer to measures such as the mean and the median. We already know that the function `mean()` can be used to compute the sample mean.

```
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

We can also compute the sample median using the `median()` function. Recall that the sample median refers to the value in the middle of the observations.

```
median(iris$Sepal.Length)
```

```
## [1] 5.8
```

Let us now move to some measures of variability, which help us understand how spread the data is. First, we can look at the minimum and maximum values, which can be obtained using the functions `min()` and

`max()` receptively.

```
min(iris$Sepal.Length)
```

```
## [1] 4.3
```

```
max(iris$Sepal.Length)
```

```
## [1] 7.9
```

Alternatively, we can use the `range()` function to compute both minimum and maximum simultaneously.

```
range(iris$Sepal.Length)
```

```
## [1] 4.3 7.9
```

Next, we can look at the variance, which measures how the data values are dispersed around the mean. This can be computed with the `var()` function:

```
var(iris$Sepal.Length)
```

```
## [1] 0.6856935
```

A related measure is the standard deviation, which is simply the square root of the variance. To compute it, we can use `sd()`:

```
sd(iris$Sepal.Length)
```

```
## [1] 0.8280661
```

Finally, we may be interested in computing the sample quantiles of our data. The R command to do so is `quantile()`:

```
quantile(iris$Sepal.Length)
```

```
##   0%  25%  50%  75% 100%
##  4.3  5.1  5.8  6.4  7.9
```

Note that by default, R computes the quantiles of 0% (min), 25%, 50% (median), 75%, and 100% (max). However, we can change this with the argument `probs`:

```
quantile(iris$Sepal.Length, probs = seq(0.1, 0.9, by = 0.1))
```

```
##  10%  20%  30%  40%  50%  60%  70%  80%  90%
## 4.80 5.00 5.27 5.60 5.80 6.10 6.30 6.52 6.90
```

Although all the above measures can be computed separately, the `summary()` function provides an easier way to compute several of them at once:

```
summary(iris$Sepal.Length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   5.100   5.800   5.843   6.400   7.900
```

Moreover, if our data set has several columns, `summary()` can compute all these measures for all columns at the same time:

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
```

```
##  versicolor:50
##  virginica :50
##
##
##
```

Note that our data set has a column called `Species`, indicating the flower species. Although the summary function above already provides us with the list of species (three species), we can also use the `unique()` function to check how many species we have:

```
unique(iris$Species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

Now, we can use `Species` to give more insides into our data set. For example, we may be interested in computing the mean of `Sepal.Lenght` by species. For that, we can use the function `tapply()` as follows:

```
tapply(iris$Sepal.Length, iris$Species, mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Or we can use `tapply()` in conjunction with `summary()`.

```
tapply(iris$Sepal.Length, iris$Species, summary)
```

```
## $setosa
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   4.800   5.000   5.006   5.200   5.800
##
## $versicolor
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   5.600   5.900   5.936   6.300   7.000
##
## $virginica
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   6.225   6.500   6.588   6.900   7.900
```

We may also be interested in seeing if there is a relationship among the different columns of our data set. For that purpose, we can start, for instance, by computing their correlation, a measure of how a pair of variables are linearly related. Recall that for two vectors $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ the sample covariance is given by

$$\mathrm{Cov}(\mathbf{x}, \mathbf{y}) = \frac{1}{n-1} \sum_{i=1}^{n} \frac{(x_i - \bar{x})(y_i - \bar{y})}{s_x s_y},$$

where $s_{\mathbf{x}}$ and $s_{\mathbf{y}}$ are the sample standard deviations of $\mathbf{x}$ and $\mathbf{y}$, respectively. In R, the sample correlation is computed using the `cor()` function. For example,

```
cor(iris$Sepal.Length, iris$Sepal.Width)
```

```
## [1] -0.1175698
```

If we have several (numeric) columns, we can compute all correlations at once:

```
cor(iris[, -5])
```

```
##              Sepal.Length Sepal.Width Petal.Length Petal.Width
## Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
## Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
## Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
## Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

Remark. The `cor()` computes, by default, the Person's correlation coefficient described above. However, we can use the `method` argument to calculate Kendall's tau and Spearman's rho, two measures that assess how well the relationship between two variables can be described using a monotonic function. We will review these two other measures later on.

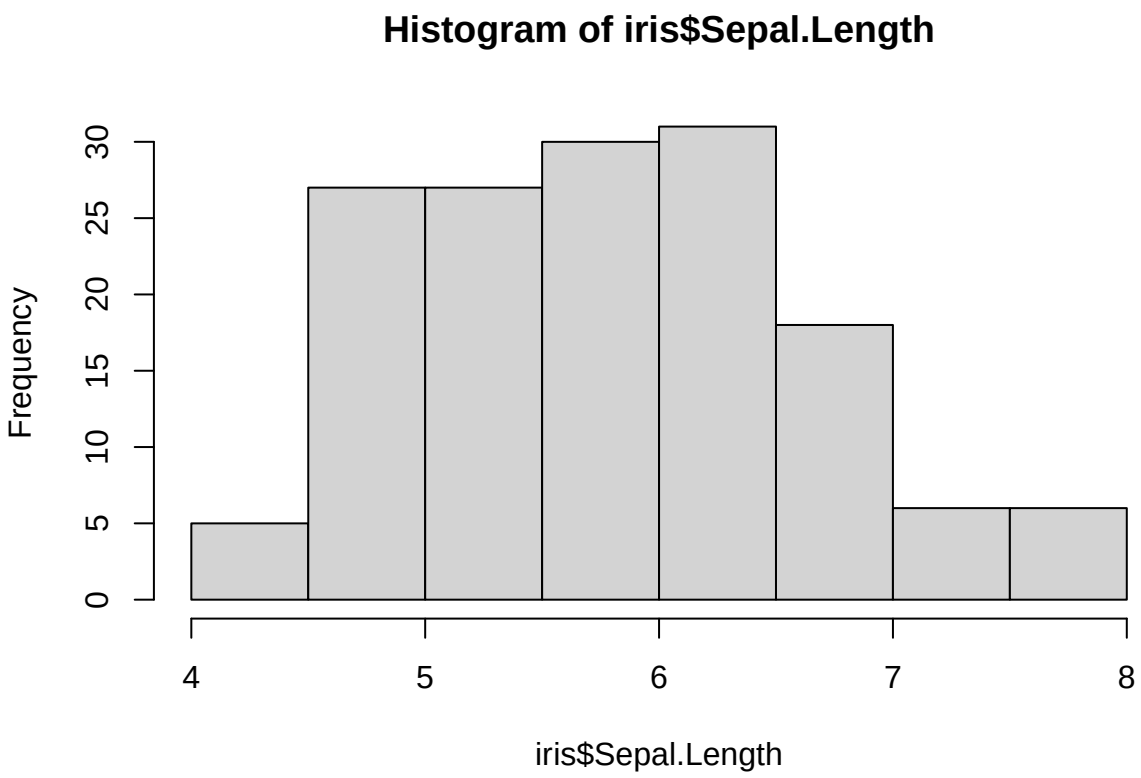
2.1.1 Visual tools

We now present some R tools that can be employed to visualize data. More specifically, we will see how to create histograms, box plots, and scatter plots.

Histograms

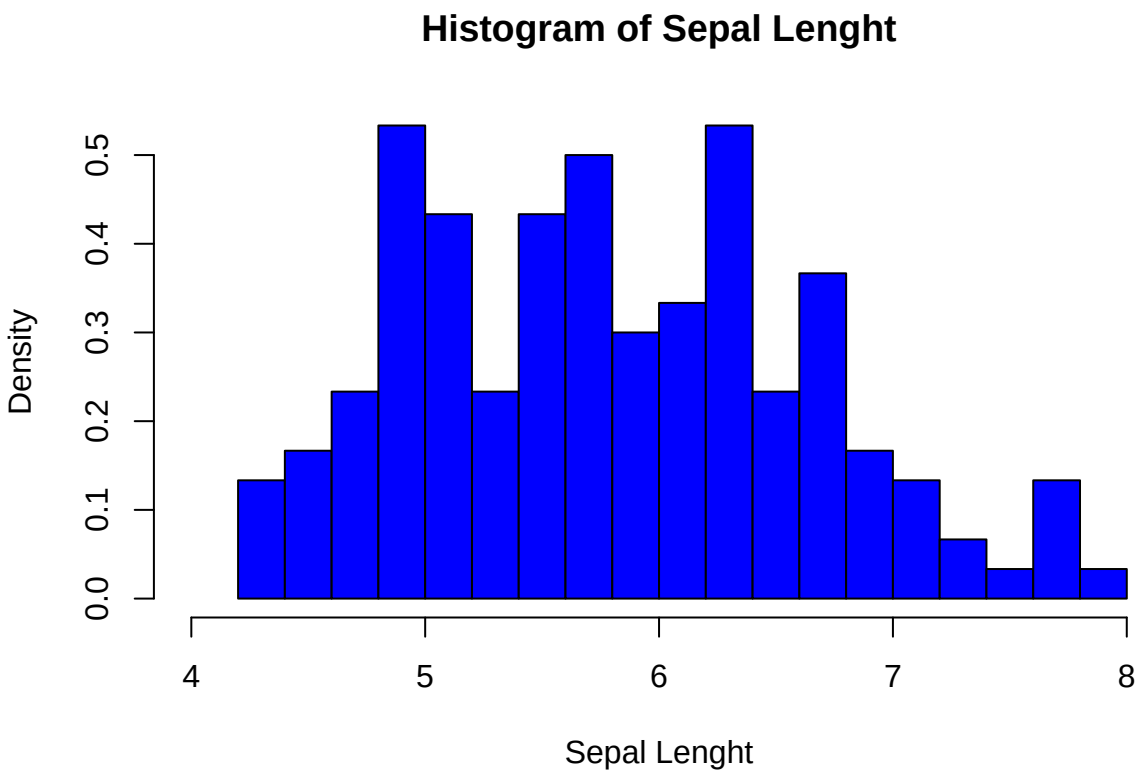
To create a histogram, we can use the `hist()` function. For instance, to draw a histogram for `Sepal.Length`, we type:

```
hist(iris$Sepal.Length)
```



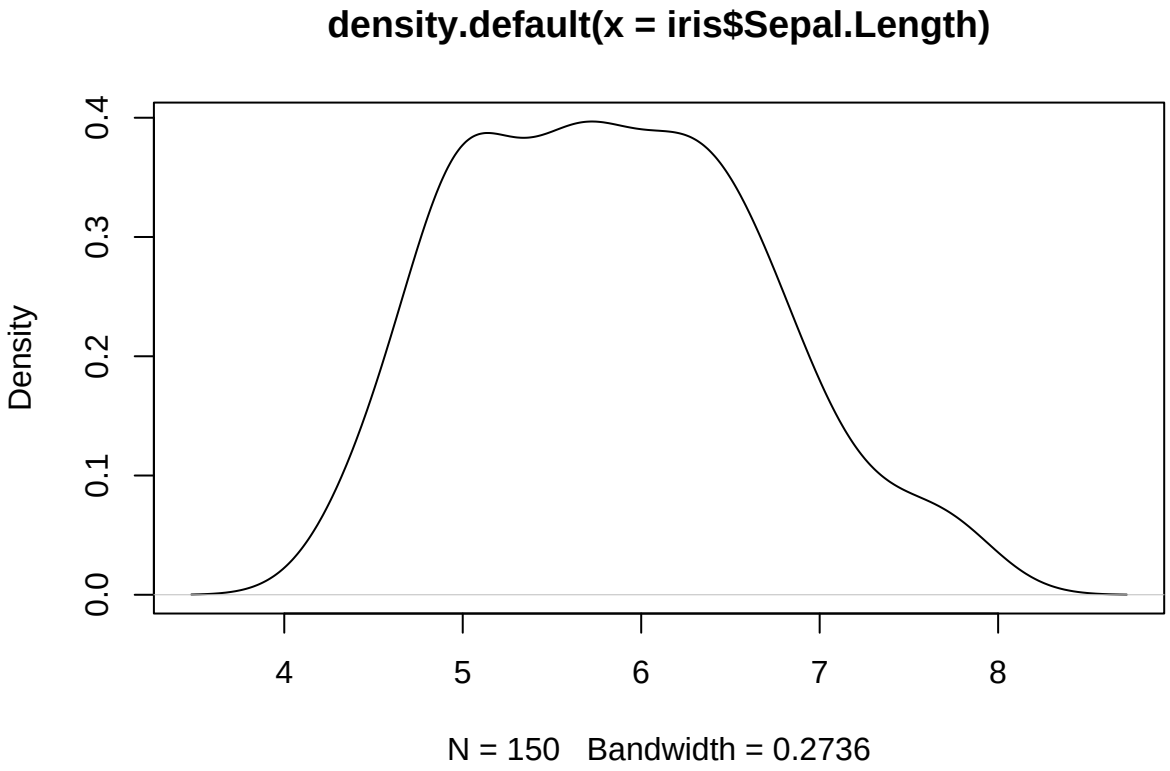
We can customize our plot further using the different arguments of `hist()` (see `?hist`):

```
hist(iris$Sepal.Length,
  breaks = 15, # Number of cells in the histogram
  main = "Histogram of Sepal Lenght", # Main tittle
  xlab = "Sepal Lenght", # Text on the x-axis
  freq = FALSE, # Probability density
  xlim = c(4, 8), # Range for the x-axis
  col = "blue" # Color to fill the bars
)
```



R also allows to compute kernel density estimates via the `density()` function.

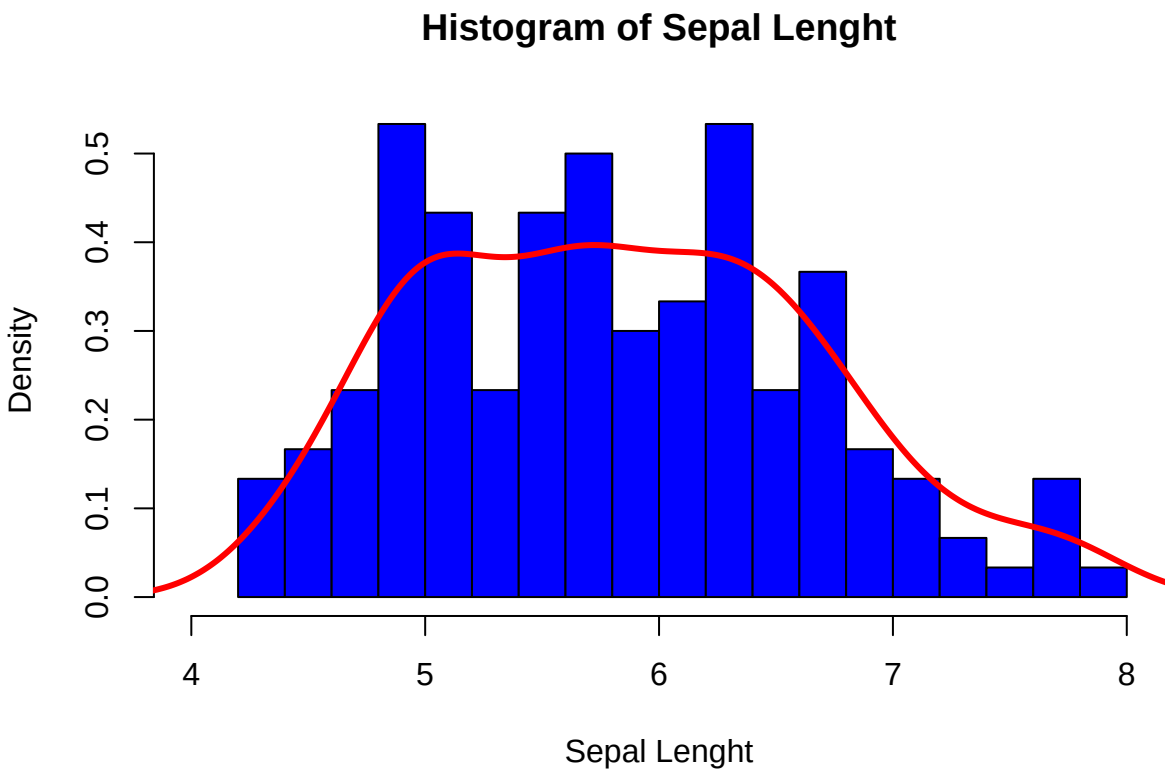
```
plot(density(iris$Sepal.Length))
```



We can combine our two previous plots using `lines()` as follows:

```
hist(iris$Sepal.Length,
  breaks = 15, # Number of cells in the histogram
  main = "Histogram of Sepal Lenght", # Main tittle
  xlab = "Sepal Lenght", # Text on the x-axis
  freq = FALSE, # Probability density
```

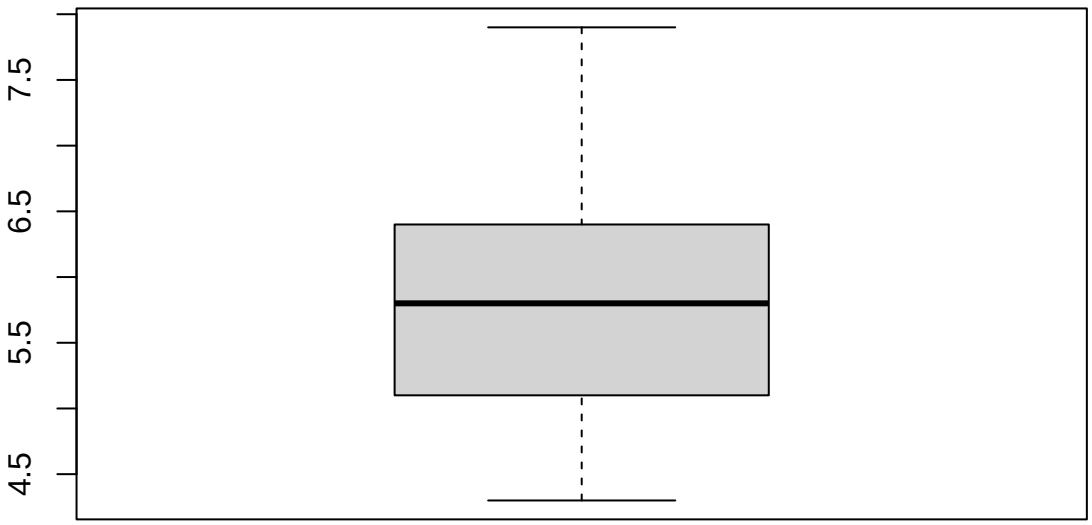
```
  xlim = c(4, 8), # Range for the x-axis
  col = "blue" # Color to fill the bars
)
lines(density(iris$Sepal.Length),
  lwd = 3, # Line width
  col = "red" # Color of the line
)
```



Box plots

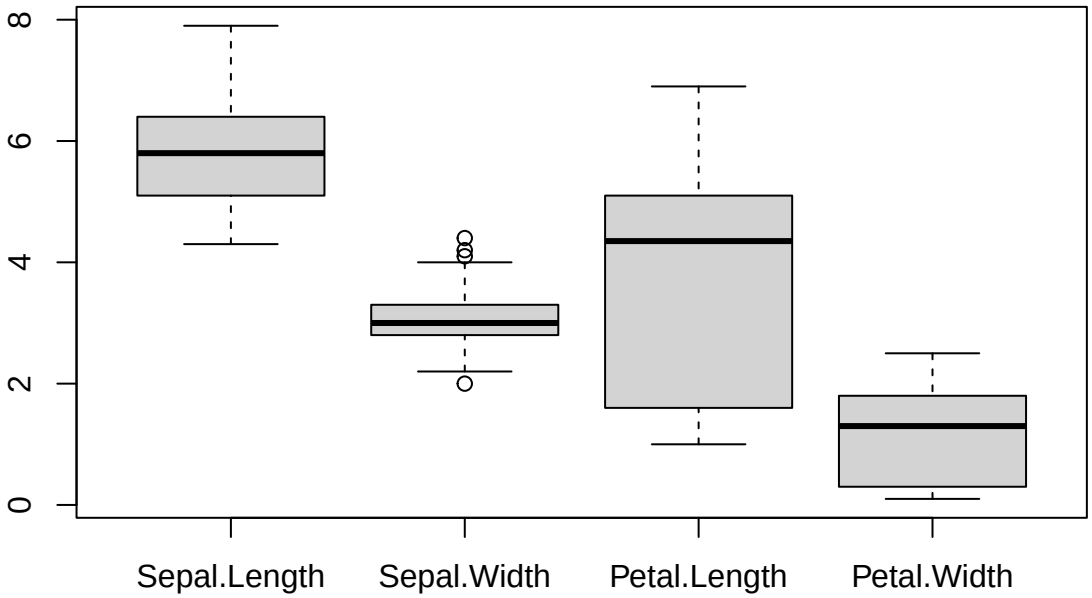
To produce box plots, R uses the `boxplot()` function. Let us try an example with the `iris` data set:

```
boxplot(iris$Sepal.Length)
```



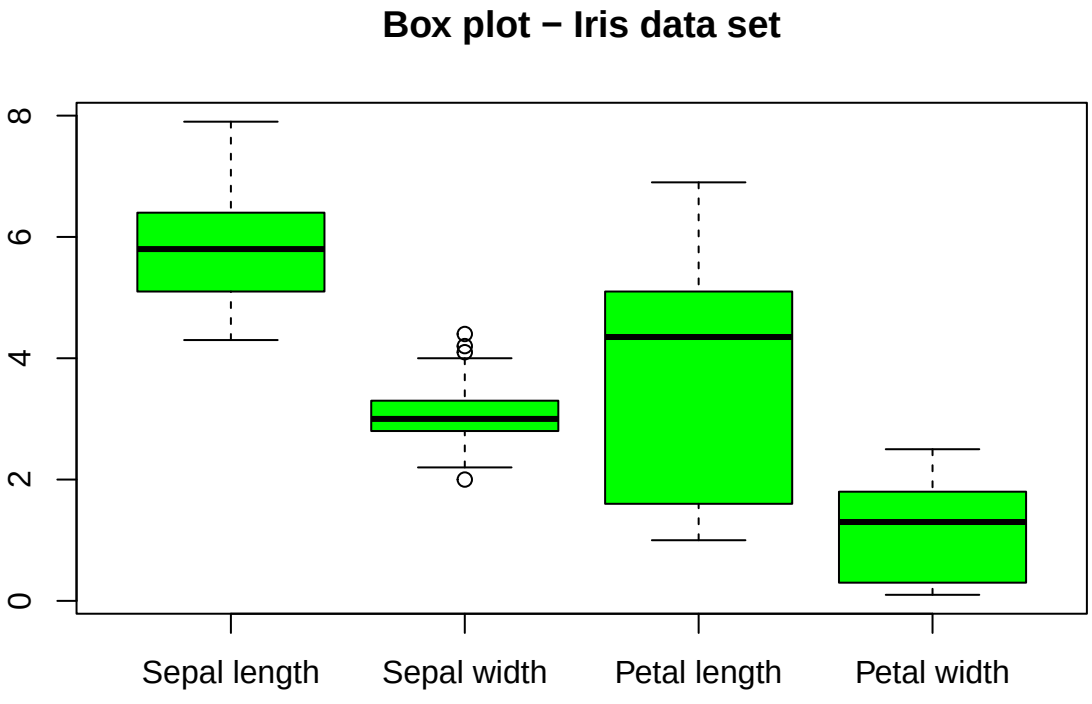
We can create box plots for the different (numeric) columns of `iris` at the same time:

```
boxplot(iris[, -5])
```



Let us now customize the above plot:

```
boxplot(iris[, -5],
  main = "Box plot - Iris data set",
  names = c(
    "Sepal length", "Sepal width",
    "Petal length", "Petal width"
  ), # Change names of x axis
  col = "green"
)
```

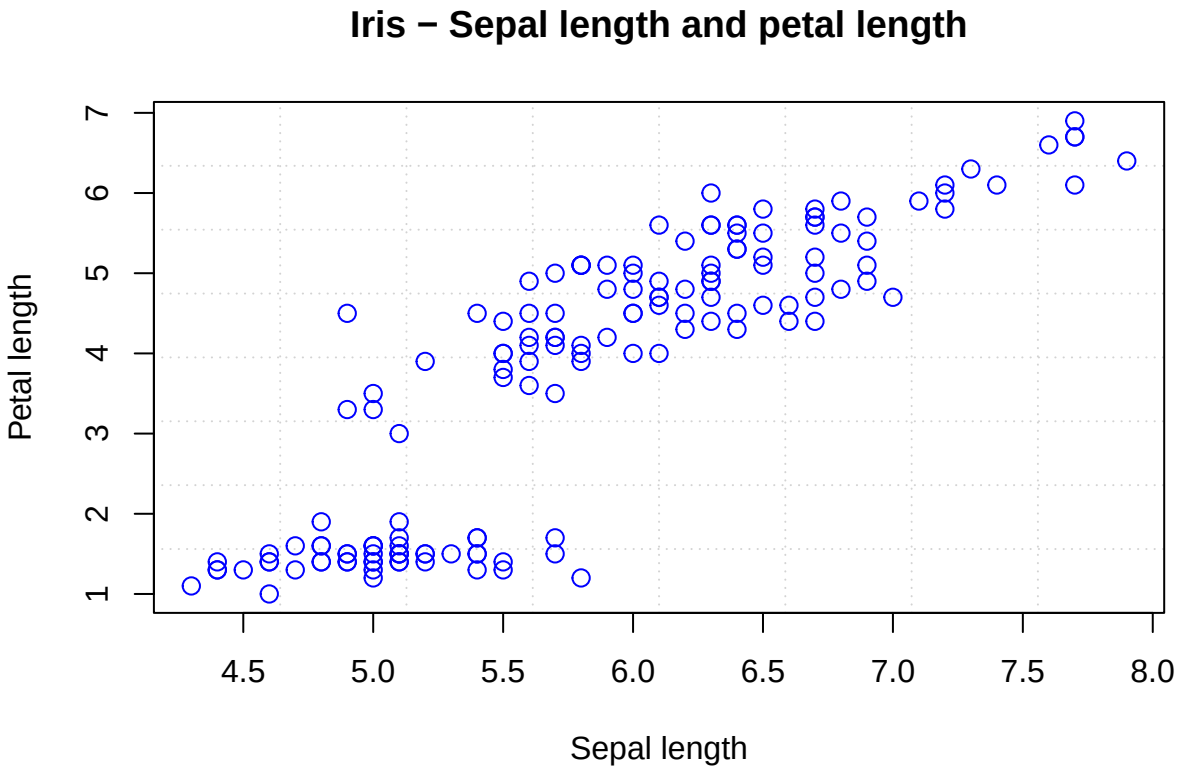


Scatter plots

Scatter plots can help us to visualize the relationship between the different data columns. We can use the `plot()` function to create a scatter plot. For instance, below, we plot Sepal Length against Petal Length:

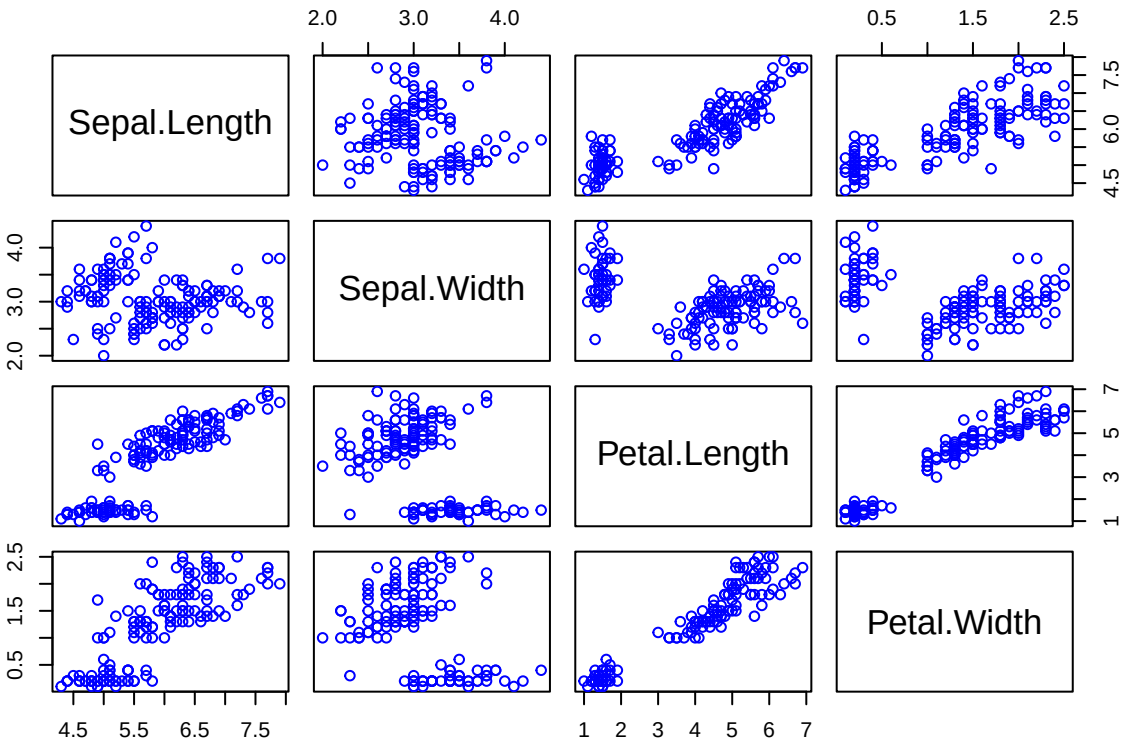
```
plot(iris$Sepal.Length, iris$Petal.Length,
  panel.first = grid(8, 8), # Adds a 8x8 grid
```

```
  cex = 1.2, # Size of the dots
  col = "blue",
  xlab = "Sepal length",
  ylab = "Petal length",
  main = "Iris - Sepal length and petal length"
)
```



Finally, for data with several columns, we can plot them all at once:

```
plot(iris[, -5],
  col = "blue",
)
```



2.2 Probability distributions

R comes with several parametric probability distributions that can help us to describe our data. Every distribution in R has four functions, which can be called by using the *root* name of the distribution (e.g., `exp`) preceded by one of the following letters:

- `d`: The probability density function (pdf).
- `p`: The cumulative distribution function (cdf).
- `q`: Quantile.
- `r`: Random generator for the specified distribution.

For instance, the exponential distribution has the four functions: `dexp()`, `pexp()`, `qexp()` and `rexp()`.

Table 2.1 shows the continuous distributions, and Table 2.2 the discrete distributions available by default in R. Note, however, that these two lists are far from comprehensive since there are several other distributions not contained there. Fortunately, other distributions may be available in different packages. For example, an implementation for the Pareto distribution can be found in the `actuar` R package.

Remark. When using these R built-in functions, we must be careful and check the parametric form implemented for these distributions since they may differ from the ones we are familiar with. For instance, in R, the pdf of the Gamma distribution has two parametrizations (see help - `?pgamma`). The first one is

$$f(x) = \frac{1}{\sigma^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\sigma}, \quad x \geq 0,$$

where $\alpha > 0$ and $\sigma > 0$ and is accessible by using the arguments `shape` (α) and `scale` (σ). The second parametrization is

$$f(x) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}, \quad x \geq 0,$$

where $\alpha > 0$ and $\lambda > 0$, which correspond to the use of the arguments `shape` (α) and `rate` (λ). However, note that both representations are equivalent. Indeed, take $s = 1/\lambda$ and $a = \alpha$.

Table 2.1: Continuous distributions.

Distribution	Root name
Beta	beta
Cauchy	cauchy
Chi-2	chisq
Exponential	exp
Fisher F	f
Gamma	gamma
Logistic	logis
Lognormal	lnorm
Normal	norm
Student t	t
Uniform	unif
Weibull	weibull

Table 2.2: Discrete distributions.

Distribution	Root name
Binomial	binom
Geometric	geom
Hypergeometric	hyper
Negative Binomial	nbinom
Poisson	pois

We now look at some examples of how to use these functions, and we will use the normal distribution to do so. Recall that a random variable X is said to be normal distributed with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma > 0$, if its density function is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \quad x \in \mathbb{R}.$$

We write $X \sim N(\mu, \sigma^2)$.

In R, this density can be evaluated using the `dnorm()` function. For instance,

```
dnorm(1) # mu = 0, sigma = 1
```

```
## [1] 0.2419707
```

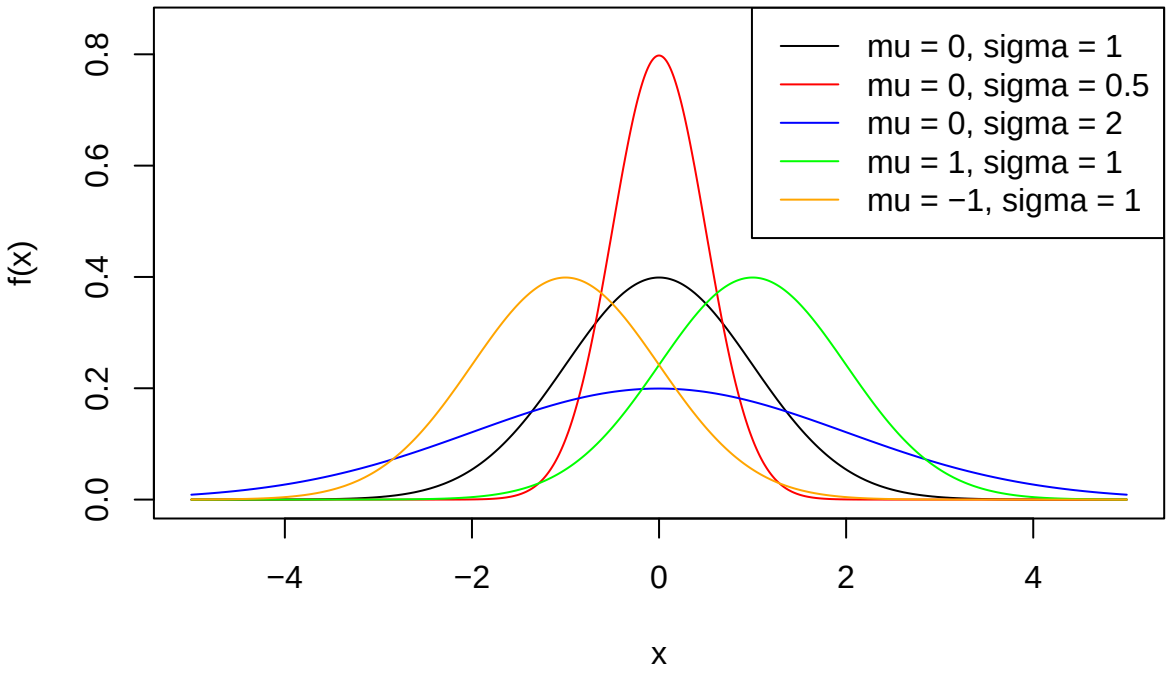
```
dnorm(1, 1, 2) # mu = 1, sigma = 2
```

```
## [1] 0.1994711
```

We will now plot this density function for different combinations of parameters. The usual way to plot a function in R is first generate a sequence of numbers, then evaluate the desired function at the generated sequence, and finally use `plot()`.

```
sq <- seq(-5, 5, by = 0.01)
plot(sq, dnorm(sq), type = "l", ylim = c(0, 0.85), ylab = "f(x)", xlab = "x")
lines(sq, dnorm(sq, 0, 0.5), col = "red")
lines(sq, dnorm(sq, 0, 2), col = "blue")
lines(sq, dnorm(sq, 1, 1), col = "green")
lines(sq, dnorm(sq, -1, 1), col = "orange")
legend("topright",
  leg = paste0("mu = ", c(0, 0, 0, 1, -1), ", sigma = ", c(1, 0.5, 2, 1, 1)),
  lty = 1,
```

```
  col = c("black", "red", "blue", "green", "orange")
)
```



Next, we consider the distribution function of $X \sim N(\mu, \sigma^2)$. Recall that this is given by

$$F(x) = \mathbb{P}(X \leq x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right) dy\,.$$

To evaluate this function in R, we can use `pnorm()`. For example, if we want to find $\mathbb{P}(X \leq 1.8)$ for $X \sim N(0, 1)$, we simply type:

```
pnorm(1.8)
```

```
## [1] 0.9640697
```

Note that `pnorm()` has an optional argument `lower.tail`. By default `lower.tail` takes the value `TRUE`, indicating that `pnorm()` will compute $\mathbb{P}(X \leq x)$. However, we can compute $\mathbb{P}(X > x) = 1 - \mathbb{P}(X \leq x)$ by using `lower.tail = FALSE`. For instance,

```
1 - pnorm(1.8)
```

```
## [1] 0.03593032
```

```
pnorm(1.8, lower.tail = FALSE)
```

```
## [1] 0.03593032
```

We can simulate random values following a $N(\mu, \sigma^2)$ distribution via the `rnorm()` function. For example,

```
rnorm(10)
```

```
##  [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430
##  [7] -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

Note that every time you run `rnorm()`, it will generate a different set of values:

```
rnorm(10)
```

```
##  [1]  1.10177950  0.75578151 -0.23823356  0.98744470  0.74139013  0.08934727
##  [7] -0.95494386 -0.19515038  0.92552126  0.48297852
```

However, sometimes it is convenient to generate the same random sequence at will. This can allow us, for example, to replicate a study. To do this, we need to give R an initial *seed* to generate the random numbers, which is done via the `set.seed()` function. For example, consider the following code

```
set.seed(1)
rnorm(10)
```

```
##  [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
##  [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

Then, if we run the code above again, we will get exactly the same sequence of values:

```
set.seed(1)
rnorm(10)
```

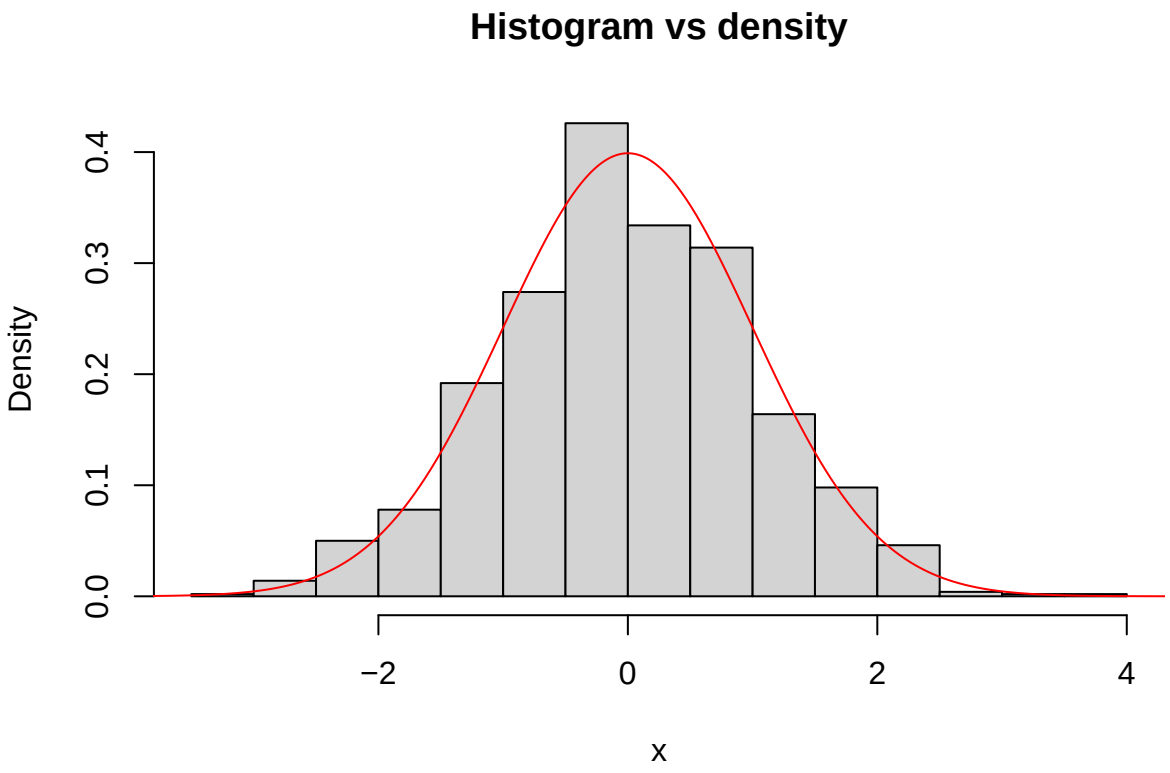
```
##  [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
##  [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

A visual way to evaluate if certain data follows a specific distribution is to compare the histogram generated by the data and the density function of the distribution. Let us look at a simple example: First, we generate a random sample following a $N(0,1)$ distribution.

```
set.seed(1)
x <- rnorm(1000)
```

Now, if the data would really follow a $N(0,1)$ distribution (which, in this case, it does by construction), the histogram of the data should be similar to the density of a $N(0,1)$ distribution. The following plot shows precisely this:

```
hist(x, freq = F, main = "Histogram vs density")
sq <- seq(-5, 5, by = 0.01)
lines(sq, dnorm(sq), type = "l", col = "red")
```



Finally, we will look at the function to compute the quantiles of a distribution. Recall that for a random variable X with distribution function F, its quantile function $F^{\leftarrow}$ is given by

$$F^{\leftarrow}(p) = \inf\{x : p \leq F(x)\}\,, \quad p \in [0,1]\,.$$

If the distribution function is continuous and strictly monotonic, then $F^{\leftarrow}(p)$ satisfies that

$$F(F^{\leftarrow}(p)) = p\,.$$

In other words, it corresponds to the inverse of F.

Now, consider $X \sim N(1, 2^2)$ and suppose that we want to find x such that $P(X \leq x) = 0.95$. This can be done with the `qnorm()` function:

```
x <- qnorm(0.95, 1, 2)
x
```

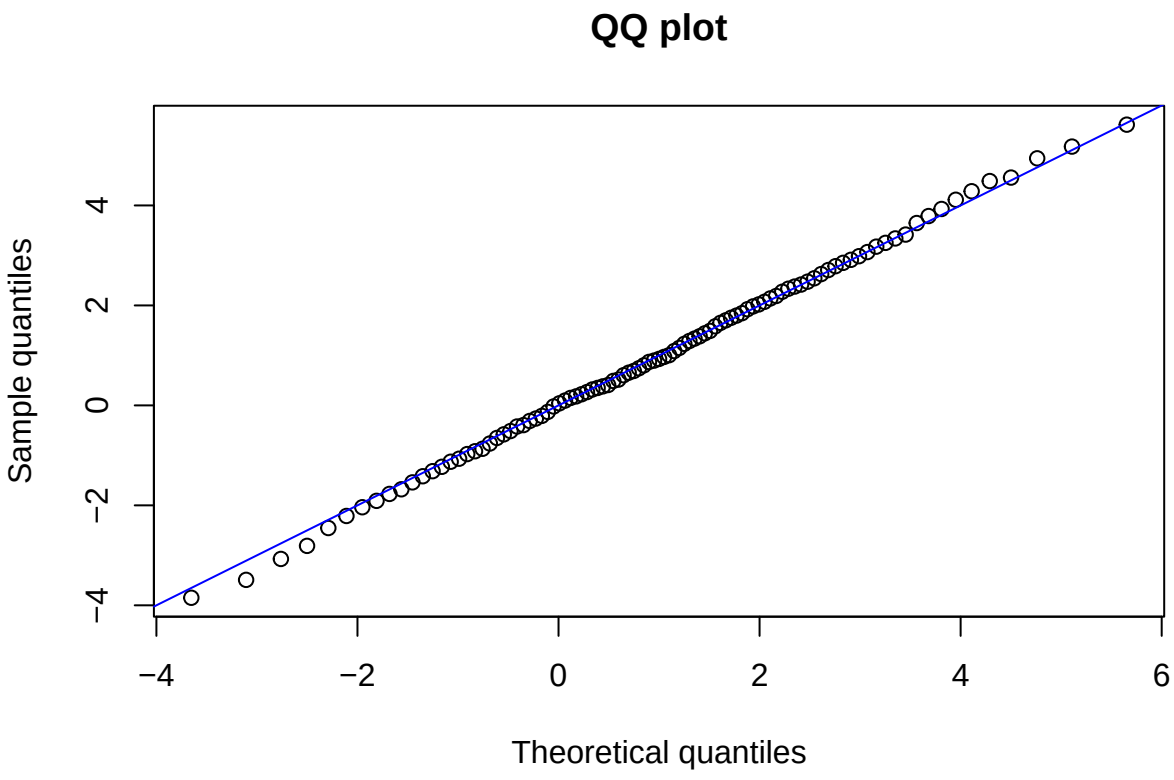
```
## [1] 4.289707
```

```
pnorm(x, 1, 2) # Check
```

```
## [1] 0.95
```

An alternative visual way of evaluating if our data comes from a certain distribution is to plot the sample quantiles against the theoretical quantiles of the distribution to check. This is called a QQ-plot. If the data comes from that specific distribution, then we should observe that the points form (approximately) an identity line. Let us look at an example:

```
set.seed(1)
x <- rnorm(1000, 1, 2)
p <- seq(0.01, 0.99, by = 0.01)
the_quantile <- qnorm(p, 1, 2)
sam_quantile <- quantile(x, p)
plot(the_quantile, sam_quantile,
  main = "QQ plot",
  xlab = "Theoretical quantiles",
  ylab = "Sample quantiles"
)
abline(0, 1, col = "blue") # Identity
```



2.2.1 Tranformations

Several distributions are obtained via transformations. For instance, a lognormal distributed random variable can be obtained by exponentiation of a normal distributed random variable. Here, we will illustrate how to use the previous implementations to work with transformations of random variables.

First, let us recall some results. Let Y be a continuous random variable with density and distribution functions f_Y and F_Y, respectively. Now, consider a strictly increasing transformation $g(\cdot)$, and define $X = g(Y)$. Then, the distribution function F_X of X is given by

$$F_X(x) = \mathbb{P}(X \leq x) = \mathbb{P}(g(Y) \leq x) = \mathbb{P}(Y \leq g^{-1}(x)) = F_Y(g^{-1}(x))\,.$$

From the expression above, it follows that the density function f_X of X is given by

$$f_X(x) = f_Y(g^{-1}(x))\frac{d}{dx}(g^{-1}(x))\,.$$

Now, let us consider an explicit example: the lognormal distribution. Recall that a lognormal distributed random variable with parameters μ and σ^2 is obtained via the transformation

$$X = \exp(Y)\,,$$

where $Y \sim N(\mu, \sigma^2)$. We write $X \sim LN(\mu, \sigma^2)$. Although already implemented in R under the root name `lnorm`, we can use this distribution to illustrate how to work with transformations and, at the same time, verify our computations. Note that in this case $g(y) = \exp(y)$, $y \in \mathbb{R}$, and $g^{-1}(x) = \log(x)$, $x > 0$.

We can now compute the distribution function $F_X(x)$ of $X \sim LN(\mu, \sigma^2)$ at x, by using `pnorm()` evaluated at $g^{-1}(x) = \log(x)$. For example, for $X \sim LN(1, 2^2)$, $\mathbb{P}(X \leq 3)$ can be computed as follows:

```
user_plnorm <- function(x, mu, sigma) {
  pnorm(log(x), mu, sigma)
}
user_plnorm(3, 1, 2)
```

```
## [1] 0.5196623
```

```
plnorm(3, 1, 2) # Check
```

```
## [1] 0.5196623
```

Now for the density evaluation, we need $\frac{d}{dx}(g^{-1}(x)) = 1/x$. Thus, the density evaluation at $x = 3$ for $X \sim LN(1, 2^2)$ can be computed as:

```
user_dlnorm <- function(x, mu, sigma) {
  dnorm(log(x), mu, sigma) / x
}
user_dlnorm(3, 1, 2)
```

```
## [1] 0.06640961
```

```
dlnorm(3, 1, 2) # Check
```

```
## [1] 0.06640961
```

In this case, since our transformation is strictly monotonic, we can compute the quantiles easily.

```
user_qlnorm <- function(p, mu, sigma) {
  exp(qnorm(p, mu, sigma))
}
user_qlnorm(0.95, 1, 2)
```

```
## [1] 72.94511
```

```
qlnorm(0.95, 1, 2) # Check
```

```
## [1] 72.94511
```

Simulation can also be performed using the relationship $X = \exp(Y)$:

```
user_rlnorm <- function(n, mu, sigma) {
  exp(rnorm(n, mu, sigma))
}
set.seed(1)
user_rlnorm(10, 1, 2)
```

```
##  [1]  0.7765396  3.9246872  0.5110656 66.0598801  5.2541358  0.5267987
##  [7]  7.2055971 11.9013211  8.5982845  1.4758340
```

In this very particular case, we can check that our simulation is correct by using `rlnorm()`:

```
set.seed(1)
rlnorm(10, 1, 2)
```

```
##  [1]  0.7765396  3.9246872  0.5110656 66.0598801  5.2541358  0.5267987
##  [7]  7.2055971 11.9013211  8.5982845  1.4758340
```

2.2.2 Law of large numbers and central limit theorem

We now recall two important results in probability theory and illustrate them using R, namely the *Law of Large Numbers* and the *Central Limit Theorem*. We start with the Law of Large Numbers:

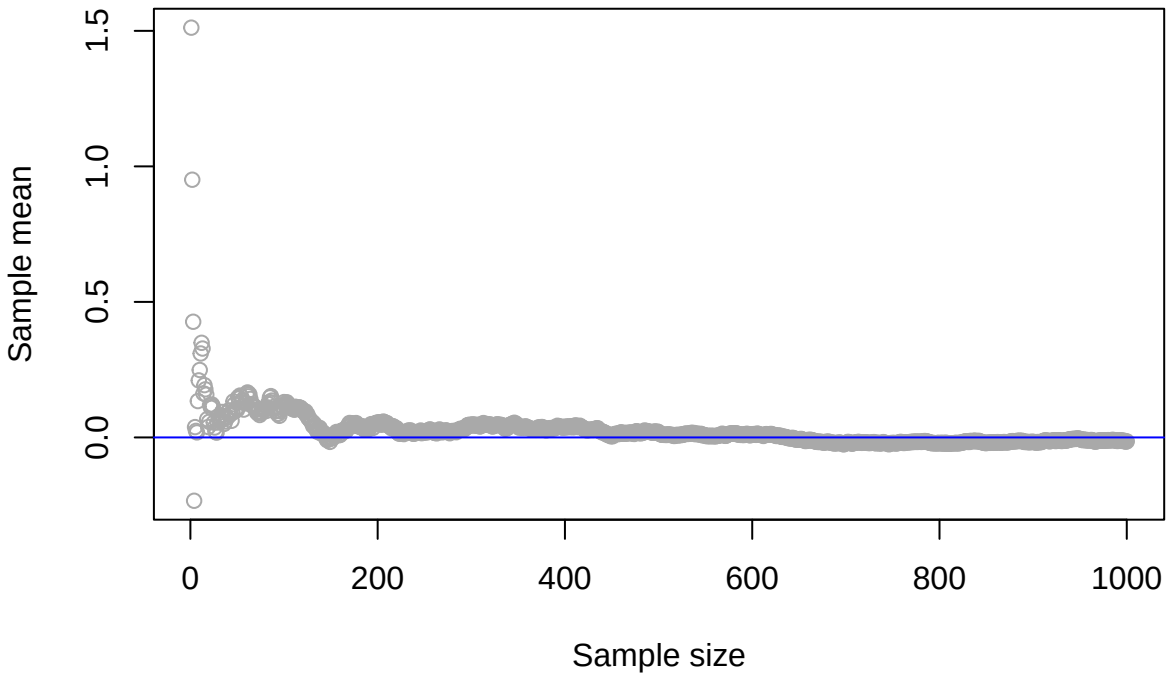
Theorem 2.1. *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables such that $\mathbb{E}[X_1] = \mu < \infty$. Then, for any $\epsilon > 0$*

$$\lim_{n\to\infty} \mathbb{P}(|\bar{X}_n - \mu| \geq \epsilon) = 0\,,$$

where $\bar{X}_n = n^{-1}\sum_{i=1}^{n} X_i$ denotes the sample mean. In other words, as the sample size n increases, the probability that the sample mean $\bar{X}_n$ is different from the population mean μ converges to zero.

We now illustrate this result via a simulation:

```
n <- 1000
plot(cumsum(rnorm(n)) / 1:n,
  xlab = "Sample size",
  ylab = "Sample mean",
  col = "darkgray"
)
abline(0, 0, col = "blue")
```



In the figure above, we can observe that as the sample size increases, the sample mean gets closer to the population mean (zero). Next, we recall the Central Limit Theorem:

Theorem 2.2. *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables with a finite expected value $\mathbb{E}[X_1] = \mu < \infty$ and variance $Var(X_1) = \sigma^2 < \infty$. Now, let Z_n be the standardized mean given by*

$$Z_n := \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}}.$$

Then, for n sufficiently large,

$$Z_n \simeq N(0,1).$$

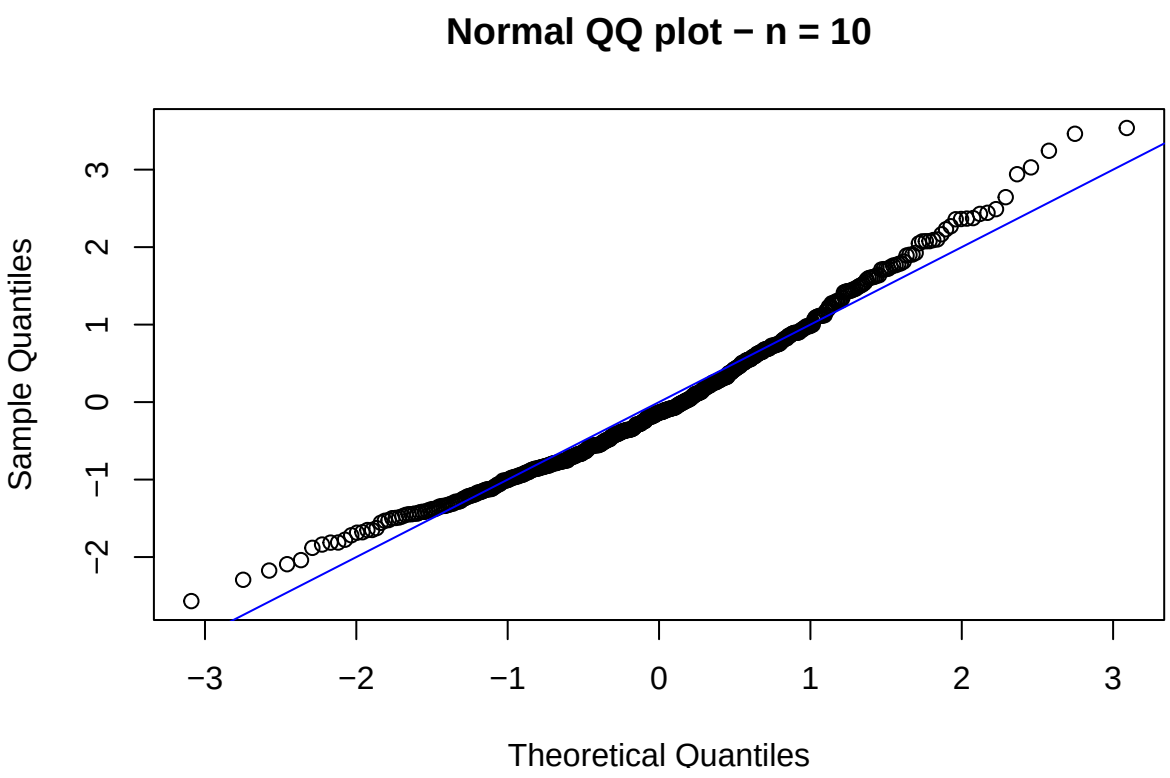
Or equivalently,

$$\bar{X}_n \simeq N(\mu, \sigma^2/n)\,,$$

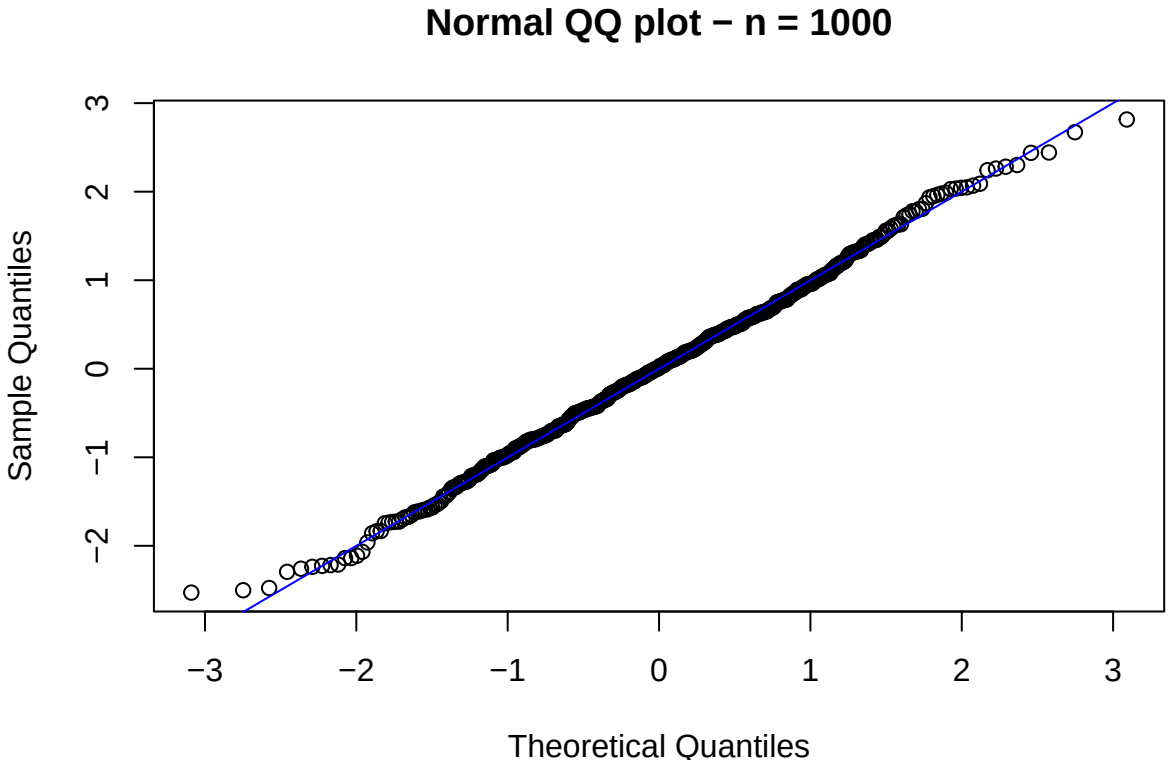
In other words, for sufficiently large n, the sample mean $\bar{X}_n$ is close to being normal distributed with mean μ and variance σ^2/n.

Let us exemplify this result via a simulation study. The idea is as follows: consider a sample of size n from an exponential distributed random variable with mean $1/\lambda$. If we were to observe several samples of size n from this distribution, let's say p samples, and compute $Z_n^{(j)}$ for each sample $j = 1, \ldots, p$, then for n large enough $Z_n^{(j)}$, $j = 1, \ldots, p$, should be approximately a sample from a standard normal distributed random variable. We could then check if $Z_n^{(j)}$, $j = 1, \ldots, p$ is truly normal distributed via, e.g., a QQ-plot. Let us try this, with $n = 10$, $p = 500$ and $\lambda = 0.5$:

```
n <- 10 # Sample size
p <- 500 # Replications of the experiment
lambda <- 0.5
sim_exp <- matrix(rexp(n * p, lambda), p, n) # Simulations in matrix form
samp_mean <- apply(sim_exp, 1, mean) # Sample mean for each replication
mu <- 1 / lambda # Population mean
sigma2 <- 1 / lambda^2 # Population variance
std_mean <- (samp_mean - mu) / (sqrt(sigma2 / n)) # Standardized mean (Z_n)
qqnorm(std_mean, main = "Normal QQ plot - n = 10") # QQ-plot with standard normal
abline(0, 1, col = "blue")
```



In the figure above, we observe that Z_n is not standard normal distributed (yet). However, if we now consider $n = 1000$, the conclusion changes (see figure below).



2.3 Parametric inference

In the last section, we presented some distributions in R that can be used to model our data. However, so far, we have not covered how to fit these models to given data, which is essential for their application in insurance and finance. Therefore, this section aims to present some estimation methods available.

2.3.1 Maximum likelihood estimation

Let $\boldsymbol{X} = (X_1, \ldots, X_n)$ be a random sample such that X_i are i.i.d. random variables with common distribution function $F(\cdot; \boldsymbol{\theta})$, where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_d) \in \boldsymbol{\Theta} \subset \mathbb{R}^d$. Here, $\boldsymbol{\Theta}$ is known as the parameter space. Given an observed data sample $\mathbf{x} = (x_1, \ldots, x_n)$ from $\boldsymbol{X}$, the *likelihood function* L is defined as the joint density (as a function of $\boldsymbol{\theta}$) evaluated at $\mathbf{x}$, that is,

$$L(\boldsymbol{\theta}; \mathbf{x}) = f_{\boldsymbol{X}}(\mathbf{x}; \boldsymbol{\theta}) = \prod_{i=1}^{n} f(x_i; \boldsymbol{\theta}) \, .$$

The *maximum likelihood estimator* (MLE) is defined as the value of $\boldsymbol{\theta}$ that maximizes the likelihood function, that is,

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} L(\boldsymbol{\theta}; \mathbf{x}) = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \prod_{i=1}^{n} f(x_i; \boldsymbol{\theta}) \, .$$

In practice, it is often more convenient to work with the *loglikelihood function* l, which is obtained by taking log of the likelihood function, i.e.,

$$l(\mathbf{x}, \boldsymbol{\theta}) = \log(L(\boldsymbol{\theta}; \mathbf{x})) = \sum_{i=1}^{n} \log\left(f(x_i; \boldsymbol{\theta})\right) \, .$$

Since log is an increasing continuous function, maximizing the loglikelihood is equivalent to maximizing the likelihood. In other words,

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} l(\boldsymbol{\theta}; \mathbf{x}) = \arg\max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \sum_{i=1}^{n} \log(f(x_i; \boldsymbol{\theta})) \,.$$

Lets us now give some examples of how to implement loglikelihood functions in R. We start by considering the exponential distribution. First, we simulate a sample following this distribution:

```
set.seed(1)
lambda <- 1.5
x_exp <- rexp(1000, lambda)
```

The following implementation computes the loglikelihood, assuming exponential distributed observations, for different parameters:

```
loglik_exp <- function(x, lambda) {
  sum(log(dexp(x, lambda)))
}
```

Given that our data originally comes from an exponential distribution with $\lambda = 1.5$, we expect that values close to 1.5 return larger values.

```
loglik_exp(x_exp, 0.5)
```

```
## [1] -1036.915
```

```
loglik_exp(x_exp, 1)
```

```
## [1] -687.5351
```

Moreover, we can plot the loglikelihood as a function of λ. To do so, we need to modify our loglikelihood implementation above to work with vector inputs for the parameter. Otherwise, we will obtain only one incorrect evaluation:

```
loglik_exp(x_exp, c(0.5, 1))
```

```
## [1] -865.5976
```

Note that the above is in fact is computing

```
loglik_exp(x_exp[2 * (1:500) - 1], 0.5) + loglik_exp(x_exp[2 * (1:500)], 1)
```

```
## [1] -865.5976
```

due to the recycling of information that R performs. One possible way to modify the function to work with vectors is the following:

```
loglik_exp <- function(x, lambda) {
  ll <- rep(0, length(lambda))
  for (i in 1:length(lambda)) {
    ll[i] <- sum(log(dexp(x, lambda[i])))
  }
  ll
}
```

With this new implementation, we obtain:

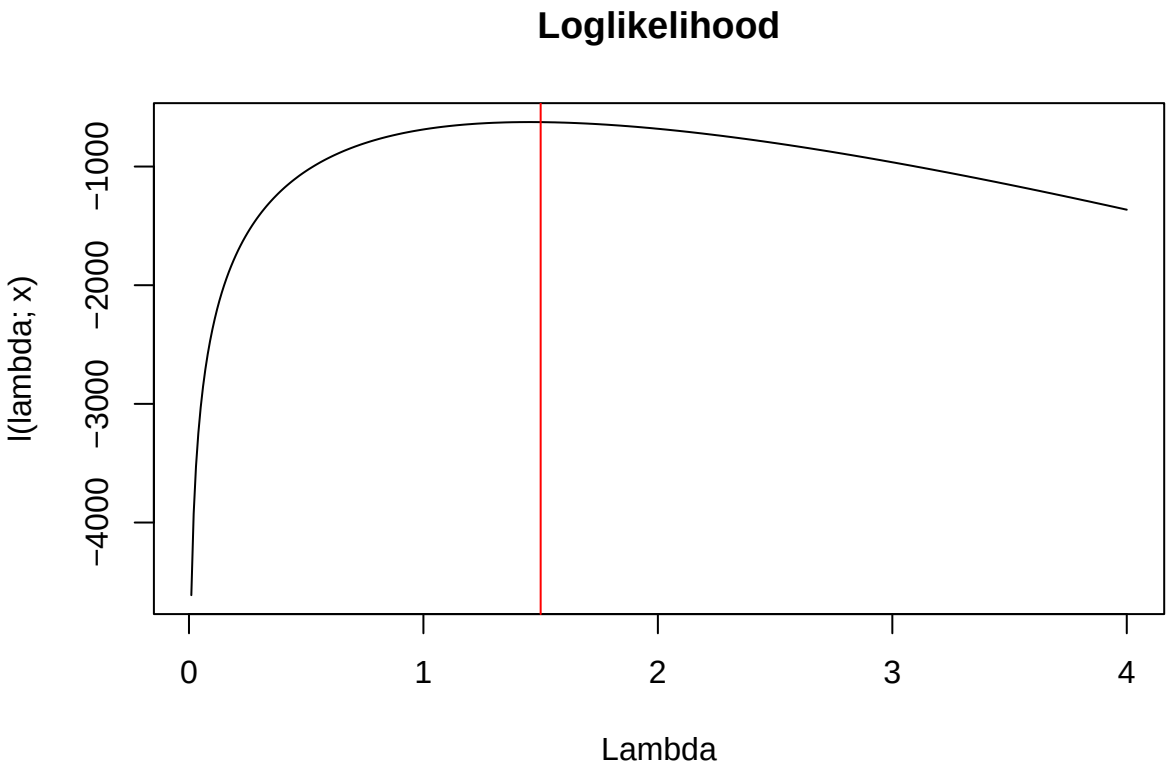
```
loglik_exp(x_exp, c(0.5, 1))
```

```
## [1] -1036.9147  -687.5351
```

Now, we can generate our plot.

```
lambda_sq <- seq(0.01, 4, by = 0.01)
plot(lambda_sq, loglik_exp(x_exp, lambda_sq),
  main = "Loglikelihood",
```

```
  ylab = "l(lambda; x)",
  xlab = "Lambda",
  type = "l"
)
abline(v = 1.5, col = "red")
```



The figure above shows that the loglikelihood is maximized around the original value $\lambda = 1.5$ (as expected).

Remark. The process of making a scalar function work with vectors is sometimes called: vectorization of a scalar function. There are other ways to solve the same problem, and here we present two more. The first one is to use the `sapply()` function. For instance, let us consider our initial implementation of the loglikelihood:

```
loglik_exp <- function(x, lambda) {
  sum(log(dexp(x, lambda)))
}
```

We can now evaluate this function in a vector using 'sapply() as follows:

```
sapply(c(0.5, 1), loglik_exp, x = x_exp)
```

```
## [1] -1036.9147  -687.5351
```

A second way to vectorize arguments of a function is to use the `Vectorize()` function, which creates a new function with vectorized arguments. The main arguments of this function are `FUN`, which is the function we need to vectorize, and `vectorize.args`, which is a vector with the arguments' names that we need to vectorize. Now, let us apply this function to our initial loglikelihood implementation:

```
loglik_exp_v <- Vectorize(loglik_exp, "lambda")
loglik_exp_v(x_exp, c(0.5, 1))
```

```
## [1] -1036.9147  -687.5351
```

Let us now consider a second example, where we have two parameters, namely a normal distribution. First, we simulate a sample:

```
set.seed(1)
mu <- 1
```

```
sigma <- 2
x_norm <- rnorm(1000, mu, sigma)
```

Next, we implement the loglikelihood:

```
loglik_norm <- function(x, mu, sigma) {
  sum(log(dnorm(x, mu, sigma)))
}
```

Let us test our function:

```
loglik_norm(x_norm, 0.5, 1)
```

```
## [1] -3172.521
```

```
loglik_norm(x_norm, 0.5, 1.5)
```

```
## [1] -2325.996
```

```
loglik_norm(x_norm, 0.8, 1)
```

```
## [1] -3074.51
```

```
loglik_norm(x_norm, 0.8, 1.5)
```

```
## [1] -2282.435
```

```
loglik_norm(x_norm, 2.5, 2.5)
```

```
## [1] -2363.257
```

Again, we can see that we obtain larger values when evaluating the function close to the real parameters $\mu = 1$ and $\sigma = 2$. As in the example of the exponential distribution, we need to modify the function above to work with vectors for plotting purposes.

```
loglik_norm <- function(x, mu, sigma) {
  ll <- rep(0, length(mu))
  for (i in 1:length(mu)) {
    ll[i] <- sum(log(dnorm(x, mu[i], sigma[i])))
  }
  ll
}
```

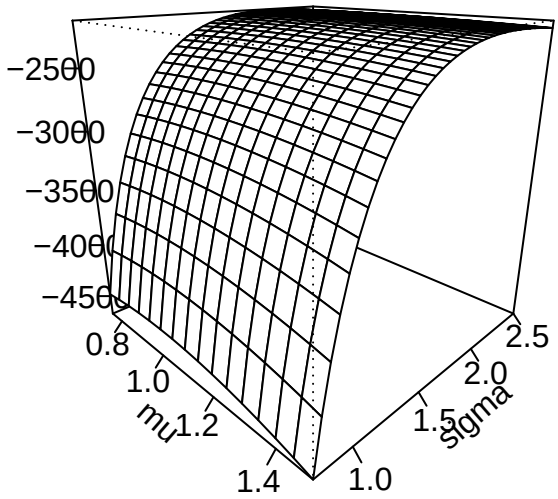
We can now perform computations for vector arguments:

```
loglik_norm(x_norm, c(1,2), c(2, 2))
```

```
## [1] -2147.143 -2277.967
```

With this implementation at hand, we can now create a surface plot for the loglikelihood. To that end, we need to create a grid of evaluation points and evaluate our loglikelihood at the generated points. This can be done as follows using the `outer()` function:

```
mu_sq <- seq(0.75, 1.5, by = 0.05)
sigma_sq <- seq(0.75, 2.5, by = 0.05)
ll_eval <- outer(mu_sq, sigma_sq, loglik_norm, x = x_norm) # Evaluates the LogLik at all points
persp(mu_sq, sigma_sq, ll_eval,
  phi = 20,
  theta = 45,
  ticktype = "detailed",
  xlab = "mu",
  ylab = "sigma",
  zlab = "",
  main = "Loglikelihood"
)
```



Remark. As in the exponential case, we have two alternative methods of evaluating the loglikelihood when passing vector inputs. The first one is to use the 'mapply() function, which is a multivariate version of sapply(). Let us consider our initial implementation of the loglikelihood:

```
loglik_norm <- function(x, mu, sigma) {
  sum(log(dnorm(x, mu, sigma)))
}
```

Then, we can evaluate this function in vector parameters using mapply() as follows:

```
mapply(loglik_norm, mu = c(1,2), sigma = c(2, 2), MoreArgs = list(x = x_norm))
```

```
## [1] -2147.143 -2277.967
```

The second way to vectorize the arguments of our loglikelihood function is to use Vectorize() as follows:

```
loglik_norm_v <- Vectorize(loglik_norm, c("mu", "sigma"))
loglik_norm_v(x_norm, c(1,2), c(2, 2))
```

```
## [1] -2147.143 -2277.967
```

Maximization

Our next step to finding the MLE is to maximize the loglikelihood with respect to the parameters. In R, this can be done by using optim(), which is a function that performs minimization. Nevertheless, remember that maximizing a function is equivalent to minimizing the negative of that function. Thus, we need to work with the negative loglikelihood in order to use optim(). Let us exemplify the above with the exponential distribution.

We will consider the previous exponentially distributed sample (x_exp). We start by implementing the negative loglikelihood:

```
nloglik_exp <- function(x, lambda) {
  -sum(log(dexp(x, lambda)))
}
```

With this implementation at hand, we can find the MLE of λ using optim().

```
mle_exp <- optim(
  par = 1, # Initial value for the parameter to be optimized over
  fn = nloglik_exp, # Function to be minimized
  x = x_exp # Further parameters
)
```

```
## Warning in optim(par = 1, fn = nloglik_exp, x = x_exp): one-dimensional optimization by Nelder-Me
## use "Brent" or optimize() directly
```

```
mle_exp$par # MLE - note that it is close to the original parameter
```

```
## [1] 1.454297
```

```
mle_exp$value # Negative loglikelihood
```

```
## [1] 625.3576
```

In this particular case, we can verify our result using the fact that the MLE for the exponential distribution has an explicit solution given by

$$\hat{\lambda} = 1/\bar{x}_n \, .$$

With our simulated sample, we then obtain

```
lambda_hat <- 1 / mean(x_exp)
lambda_hat
```

```
## [1] 1.454471
```

which is very close to our solution with `optim()`. The difference is because the first solution is solved using numerical methods, while the second one is a closed-form expression.

Let us now try the same procedure with our normal distributed sample. In this case, we require to estimate two parameters. The first step is to implement the negative loglikelihood.

```
nloglik_norm <- function(x, par) {
  -sum(log(dnorm(x, par[1], par[2])))
}
```

Note that in the above function, we passed both parameters as a single argument. We need to do it this way to be able to use `optim()`.

```
mle_norm <- optim(
  par = c(0.5, 1.5), # Initial values for the parameters to be optimized over
  fn = nloglik_norm,
  x = x_norm
)
mle_norm$par # MLE - note that it is close to the original parameter
```

```
## [1] 0.9779522 2.0690633
```

```
mle_norm$value # Negative loglikelihood
```

```
## [1] 2145.906
```

Again, we can verify our results using that for the normal distribution, the MLEs of μ and σ are explicit and given by

$$\hat{\mu} = \bar{x}_n \, , \quad \hat{\sigma} = \left(n^{-1} \sum_{i=1}^{n} (x_i - \bar{x}_n)^2 \right)^{1/2} \, .$$

Thus, for our simulated sample, we obtain

```
mu_hat <- mean(x_norm)
sigma_hat <- sqrt(sum((x_norm - mean(x_norm))^2) / length(x_norm))
mu_hat
```

```
## [1] 0.9767037
```

```
sigma_hat
```

```
## [1] 2.068797
```

Note that our results using `optim()` are very close to the ones obtained via the closed-form formulas.

Properties of the MLE

We now review some properties of maximum likelihood estimators. For illustration purposes, we state the results when the parameter space is a subset of $\mathbb{R}$. However, the results can be extended to higher dimensions.

Consistency

Theorem 2.3. *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables with common density function $f(\cdot;\theta)$. Then, under mild conditions, for any $\epsilon > 0$*

$$\lim_{n\to\infty} \mathbb{P}(|\hat{\theta}_n - \theta| \geq \epsilon) = 0\,,$$

where $\hat{\theta}_n$ is the maximum likelihood estimator of θ based on a sample of size n. In other words, as the sample size n increases, the probability that the MLE $\hat{\theta}_n$ is different from the true parameter θ converges to zero.

Let us now illustrate this result with a simulation.

```
lambda <- 1.5
set.seed(1)
x_10 <- rexp(10, rate = lambda)
x_100 <- rexp(100, rate = lambda)
x_1000 <- rexp(1000, rate = lambda)
mle_exp_10 <- suppressWarnings(optim(par = 1, fn = nloglik_exp, x = x_10)$par)
mle_exp_100 <- suppressWarnings(optim(par = 1, fn = nloglik_exp, x = x_100)$par)
mle_exp_1000 <- suppressWarnings(optim(par = 1, fn = nloglik_exp, x = x_1000)$par)
mle_exp_10
```

```
## [1] 1.780078
```

```
mle_exp_100
```

```
## [1] 1.419922
```

```
mle_exp_1000
```

```
## [1] 1.480859
```

We can see that as the sample size increases, the MLE gets closer to the true parameter.

Efficiency

Theorem 2.4. *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables with common density function $f(\cdot;\theta)$. Now, let $\hat{\theta}_n$ be the maximum likelihood estimator of θ based on a sample of size n. Then, under mild conditions, for n sufficiently large,*

$$\sqrt{n}(\hat{\theta}_n - \theta) \simeq N(0, (\mathcal{I}(\theta))^{-1})\,, \tag{2.1}$$

where $\mathcal{I}(\theta)$ is the Fisher information

$$\mathcal{I}(\theta) = \mathbb{E}_\theta\left[-\frac{d^2}{d\theta^2}\log(f(X;\theta))\right].$$

We now give an equivalent representation for (2.1). First, we define the information of the sample $\mathcal{I}_n(\theta)$ as

$$\mathcal{I}_n(\theta) = n\mathcal{I}(\theta) = \mathbb{E}_\theta\left[-\frac{d^2}{d\theta^2}\log(L(\theta;\mathbf{X}))\right].$$

Next, we define the *standard error* se as

$$se = \sqrt{1/\mathcal{I}_n(\theta)}$$

Then, (2.1) can be rewritten as

$$\frac{(\hat{\theta}_n - \theta)}{se} \simeq N(0,1)\,. \tag{2.2}$$

In practice, $\mathcal{I}_n(\theta)$ can be approximated by the observed information $\hat{\mathcal{I}}_n(\hat{\theta}_n)$ given by

$$\hat{\mathcal{I}}_n(\hat{\theta}_n) = -\frac{d^2}{d\theta^2} \log(L(\theta; \mathbf{x})) \mid_{\theta=\hat{\theta}_n} .$$

Consequently, the standard error can be approximated as $\hat{se} = \sqrt{1/\mathcal{I}_n(\hat{\theta}_n)}$.

In R, we can compute $\hat{\mathcal{I}}_n(\hat{\theta}_n)$ by using the argument `hessian` in `optim()`. For example,

```
mle_exp <- optim(
  par = 1, # Initial value for the parameter to be optimized over
  fn = nloglik_exp, # Function to be minimized
  hessian = TRUE, # Computes the Hessian
  x = x_exp # Further parameters
)
```

```
## Warning in optim(par = 1, fn = nloglik_exp, hessian = TRUE, x = x_exp): one-dimensional optimizat
## use "Brent" or optimize() directly
```

```
obs_inf_exp <- mle_exp$hessian
obs_inf_exp
```

```
##          [,1]
## [1,] 472.8183
```

Then, we can approximate the standard error as well.

```
se_exp <- sqrt(1 / obs_inf_exp)
se_exp
```

```
##             [,1]
## [1,] 0.04598888
```

Now, we can use the asymptotic normality of the MLE to compute confidence intervals. More specifically, an $(1-\alpha)$ confidence interval for θ is given by

$$\hat{\theta}_n \pm q_{(1-\alpha/2)}\, \hat{se}\,,$$

where $q_{(1-\alpha/2)}$ is the $(1-\alpha/2)$-quantile of the standard normal distribution.

Further properties of the MLE The following theorem provides further properties of the MLE

Theorem 2.5.

a) Let $\hat{\boldsymbol{\theta}}$ be the MLE of $\boldsymbol{\theta}$. Then, given a function $g : \mathbb{R}^d \to \mathbb{R}$, the MLE of $g(\boldsymbol{\theta})$ is $g(\hat{\boldsymbol{\theta}})$. This is called the invariance property of the MLE.

b) If $\mathbf{Y} = \mathbf{h}(\mathbf{X})$, where $\mathbf{h}$ is invertible in the domain of $\mathbf{X}$, then the MLE based on $\mathbf{Y}$ is the same as the MLE based on $\mathbf{X}$.

R packages for MLE

There are several R packages that include functions to perform maximum likelihood estimation. For instance, `EstimationTools`, `fitdistrplus`, and `MASS`, among others. Here, we illustrate the use of the `fitdistrplus` package.

```
library(fitdistrplus)
```

```
## Loading required package: MASS
```

```
## Loading required package: survival
```

The `fitdistrplus` package comes with the `fitdist()` function to perform MLE (and other estimation methods). The distribution is specified with the argument `distr`, but the density (`d`) and distribution (`p`) functions must be available for the given distribution (see help for details `?fitdist`). Note that the argument `start` can be omitted for some distributions (the list can be found in the help); otherwise, it has to be specified. Let us give a couple of examples.

First, the MLE for our exponentially distributed sample:

```
fit_exp <- fitdist(x_exp, distr = "exp")
summary(fit_exp)
```

```
## Fitting of the distribution ' exp ' by maximum likelihood
## Parameters :
##      estimate Std. Error
## rate 1.454471  0.0459944
## Loglikelihood:  -625.3576   AIC:  1252.715   BIC:  1257.623
```

Secondly, the MLE for our normally distributed sample:

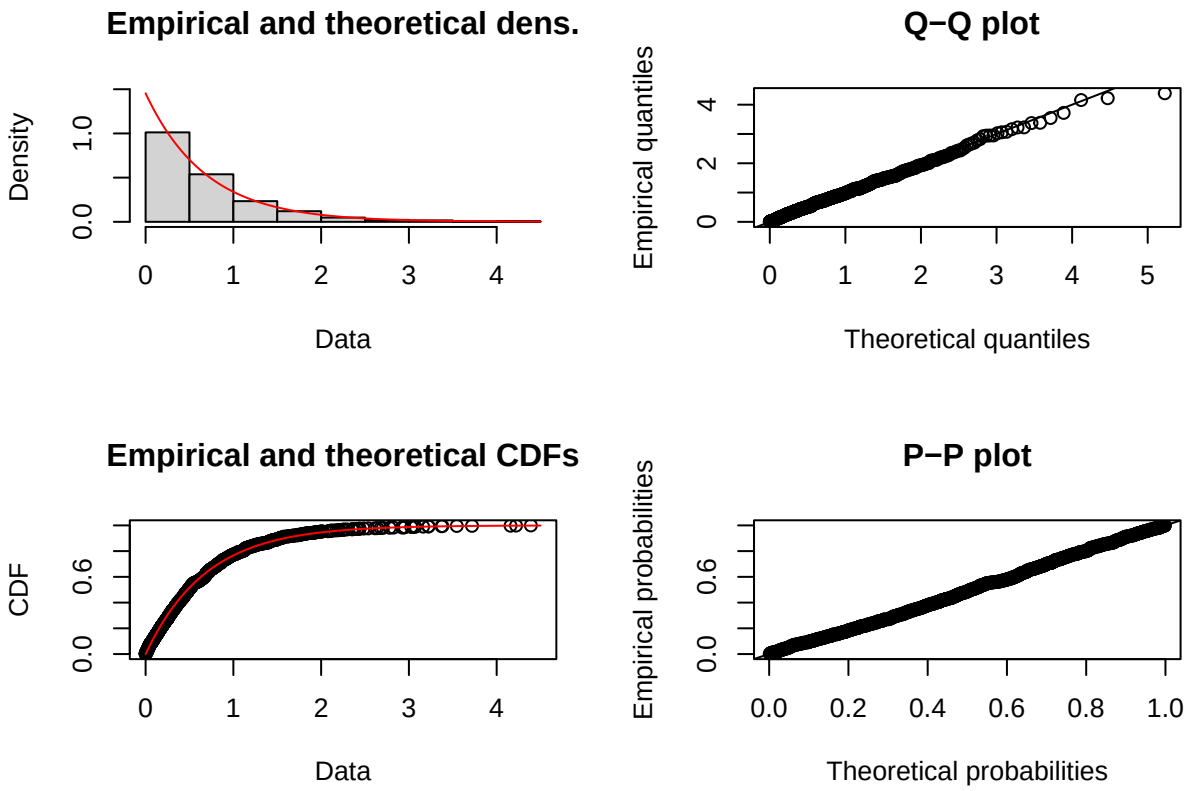
```
fit_norm <- fitdist(x_norm, distr = "norm")
summary(fit_norm)
```

```
## Fitting of the distribution ' norm ' by maximum likelihood
## Parameters :
##       estimate Std. Error
## mean 0.9767037 0.06542109
## sd   2.0687965 0.04625965
## Loglikelihood:  -2145.906   AIC:  4295.811   BIC:  4305.627
## Correlation matrix:
##      mean sd
## mean    1  0
## sd      0  1
```

Note that we obtain (approximately) the same values as our implementations above.

Remark. Another useful implementation in `fitdistrplus` is that it can generate plots to evaluate the quality of the fit by simply using the `plot()` function. For example,

```
plot(fit_exp)
```



Finally, we provide an example with a user-defined distribution, namely, the Gumbel distribution with location parameter $\mu \in \mathbb{R}$ and scale parameter $\beta > 0$. Recall that the density and distribution functions of this model are given by

$$f(x) = \exp\left(-\exp\left(-\frac{(x-\mu)}{\beta}\right)\right)\exp\left(-\frac{(x-\mu)}{\beta}\right)\frac{1}{\beta}, \quad x \in \mathbb{R},$$

$$F(x) = \exp\left(-\exp\left(-\frac{(x-\mu)}{\beta}\right)\right), \quad x \in \mathbb{R}.$$

An implementation of these functions is the following:

```
dgumbel <- function(x, mu, beta) {
  exp((mu - x) / beta) * exp(-exp((mu - x) / beta)) / beta
}
dgumbel(1, 1, 2)
```

```
## [1] 0.1839397
```

```
pgumbel <- function(q, mu, beta) {
  exp(-exp((mu - q) / beta))
}
pgumbel(1, 1, 2)
```

```
## [1] 0.3678794
```

With these implementations at hand, we can now call `fitdist()` to perform MLE. For example, we consider the `groundbeef` data set in the `fitdistrplus` package:

```
data(groundbeef)
```

Then, we fit a Gumbel distribution to the serving sizes:

```
fit_gum <- fitdist(groundbeef$serving,
  distr = "gumbel",
  start = list(mu = 1, beta = 2)
)
summary(fit_gum)
```

```
## Fitting of the distribution ' gumbel ' by maximum likelihood
## Parameters :
##      estimate Std. Error
## mu   56.97836   1.924291
## beta 29.08311   1.431894
## Loglikelihood:  -1255.717   AIC:  2515.435   BIC:  2522.509
## Correlation matrix:
##             mu      beta
## mu   1.0000000 0.3180636
## beta 0.3180636 1.0000000
```

2.3.2 Adequacy of the fit

We now focus on assessing the adequacy of a fit either via graphical methods or numerical methods. To illustrate the methods, we consider the Danish fire insurance data set (`danishuni`) available in `fitdistrplus`. More specifically, we will consider the losses above 1 million danish kroner and subtract 1 million to all data points to bring the data to the origin, that is,

```
data(danishuni)
danish_loss <- danishuni$Loss[danishuni$Loss > 1] - 1
summary(danish_loss)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##   0.00289   0.33092   0.78178   2.39726   1.97253 262.25037
```

We will consider the following potential models for the transformed data:

a) Gamma - `gamma`
b) Pareto - `pareto` (available in the `actuar` R package)
c) Burr - `burr` (available in the `actuar` R package)
d) Mixture of Gamma and Pareto - To be implemented

The idea is to select the model that "best" describes the data from the above list.

First, we need to load the `actuar` package to get access to the implementations of the Pareto and Burr distributions.

```
library(actuar)
```

```
##
## Attaching package: 'actuar'
```

```
## The following objects are masked from 'package:stats':
##
##     sd, var
```

```
## The following object is masked from 'package:grDevices':
##
##     cm
```

We can now perform MLE for the Gamma, Pareto and Burr distributions

```
fit_gamma <- fitdist(danish_loss,
  distr = "gamma"
)
fit_burr <- fitdist(danish_loss,
  distr = "burr",
  start = list(shape1 = 1, shape2 = 1, scale = 1)
)
fit_pareto <- fitdist(danish_loss,
  distr = "pareto",
  start = list(shape = 1, scale = 1)
)
```

To perform MLE of the mixture of Gamma and Pareto, we need to program its density and distribution functions. This is easily done using the corresponding functions for the `gamma` and `pareto` distributions:

```
dmgp <- function(x, shapeg, rateg, shapep, scalep, prob) {
  prob * dgamma(x, shapeg, rateg) + (1 - prob) * dpareto(x, shapep, scalep)
}
pmgp <- function(q, shapeg, rateg, shapep, scalep, prob) {
  prob * pgamma(q, shapeg, rateg) + (1 - prob) * ppareto(q, shapep, scalep)
}
dmgp(1, 1, 1, 1, 1, 0.5)
```

```
## [1] 0.3089397
```

```
pmgp(1, 1, 1, 1, 1, 0.5)
```

```
## [1] 0.5660603
```

We can now find the MLE for the above model:

```
fit_mgp <- fitdist(danish_loss,
  distr = "mgp",
  start = list(shapeg = 1, rateg = 1, shapep = 1, scalep = 1, prob = 0.5),
  lower = 0 # Avoid negative values
)
```

Remark. Although numerical maximization of mixture models can be done directly, aka by "brute force," there are more efficient ways to perform MLE of these models. For example, by using the expectation-maximization (EM) algorithm.

Now, let us look at the results of our fits:

```
summary(fit_gamma)
```

```
## Fitting of the distribution ' gamma ' by maximum likelihood
## Parameters :
##        estimate  Std. Error
## shape 0.5506529 0.013970187
## rate  0.2296712 0.008853186
## Loglikelihood:  -3712.443   AIC:  7428.887   BIC:  7440.239
## Correlation matrix:
##           shape      rate
## shape 1.0000000 0.6581388
## rate  0.6581388 1.0000000
```

```
summary(fit_pareto)
```

```
## Fitting of the distribution ' pareto ' by maximum likelihood
## Parameters :
##       estimate Std. Error
## shape 1.654915  0.0906339
## scale 1.566186  0.1265238
## Loglikelihood:  -3339.701   AIC:  6683.403   BIC:  6694.755
## Correlation matrix:
##           shape     scale
## shape 1.0000000 0.9194348
## scale 0.9194348 1.0000000
```

```
summary(fit_burr)
```

```
## Fitting of the distribution ' burr ' by maximum likelihood
## Parameters :
##        estimate Std. Error
## shape1 1.232000 0.10508632
## shape2 1.134183 0.03620174
```

```
## scale  1.029599 0.11953332
## Loglikelihood:  -3331.881   AIC:  6669.761   BIC:  6686.789
## Correlation matrix:
##            shape1     shape2      scale
## shape1  1.0000000 -0.8285421  0.9641895
## shape2 -0.8285421  1.0000000 -0.8099389
## scale   0.9641895 -0.8099389  1.0000000
```

```
summary(fit_mgp)
```

```
## Fitting of the distribution ' mgp ' by maximum likelihood
## Parameters :
##          estimate Std. Error
## shapeg 5.58594329         NA
## rateg  8.40560052         NA
## shapep 1.54732512         NA
## scalep 1.50836064         NA
## prob   0.09576093         NA
## Loglikelihood:  -3327.25   AIC:  6664.501   BIC:  6692.881
## Correlation matrix:
## [1] NA
```

The first number we can look at to select a model is the likelihood. Remember that we aim to maximize the likelihood; hence we would prefer a model with the highest likelihood possible. In our case, the mixture model has the highest likelihood. However, we also need to take into account the complexity of a model. Let us explain the last point in detail: In general, statistical models with more parameters allow for more flexibility, and thus, we can expect improvements in the fits as the number of parameters increases. However, having too many parameters can result in a fit that only describes the data at hand and fails to fit additional data or predict future observations. This is known as overfitting. On the other hand, underfitting occurs when a model cannot adequately capture the underlying structure of the data. Information criteria, such as the Akaike information criterion (AIC) and the Bayesian information criterion (BIC), deal with the problem of overfitting by introducing a penalty term for the number of parameters in the model. More specifically, if we let $\hat{L}$ be the maximum value of the likelihood function, d the number of parameters in our model, and n the sample size, the AIC and BIC are computed as

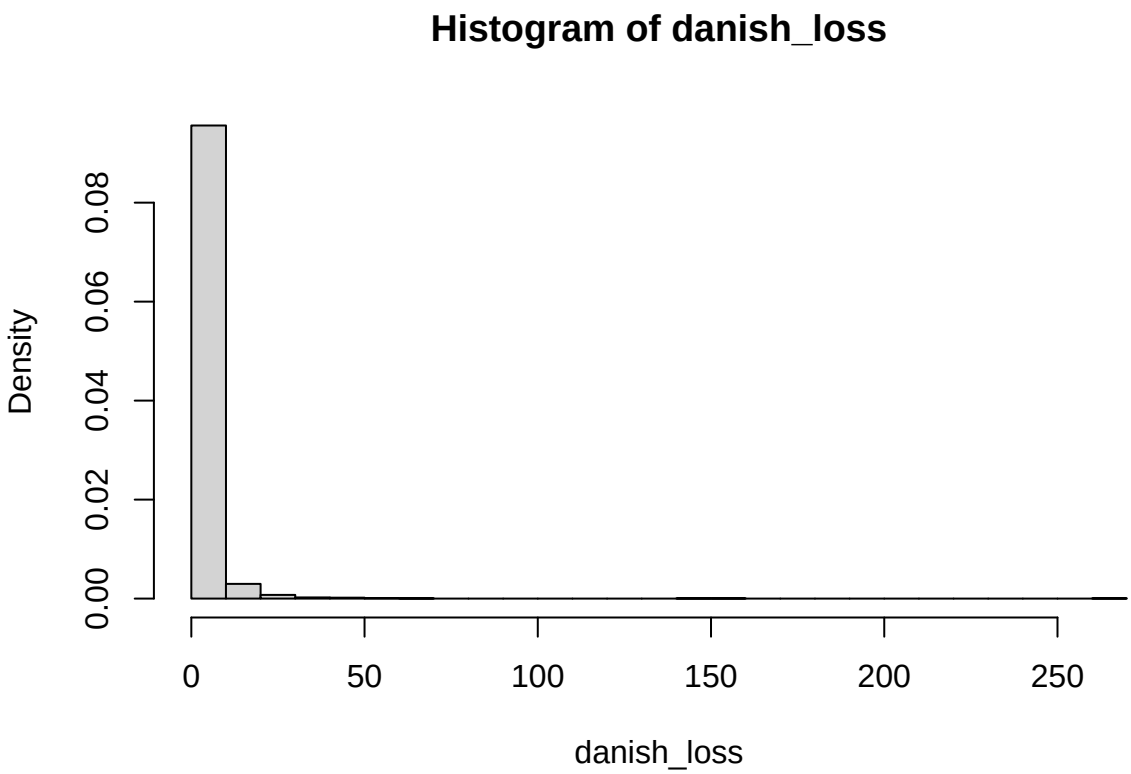
$$\text{AIC} = 2d - 2\ln(\hat{L})$$

$$\text{BIC} = d\log(n) - 2\ln(\hat{L})$$

Given a set of candidate models for our data, the preferred model would be the one with the minimum AIC (or BIC) value. In our current example, these numbers indicate that the mixture model is still preferred.

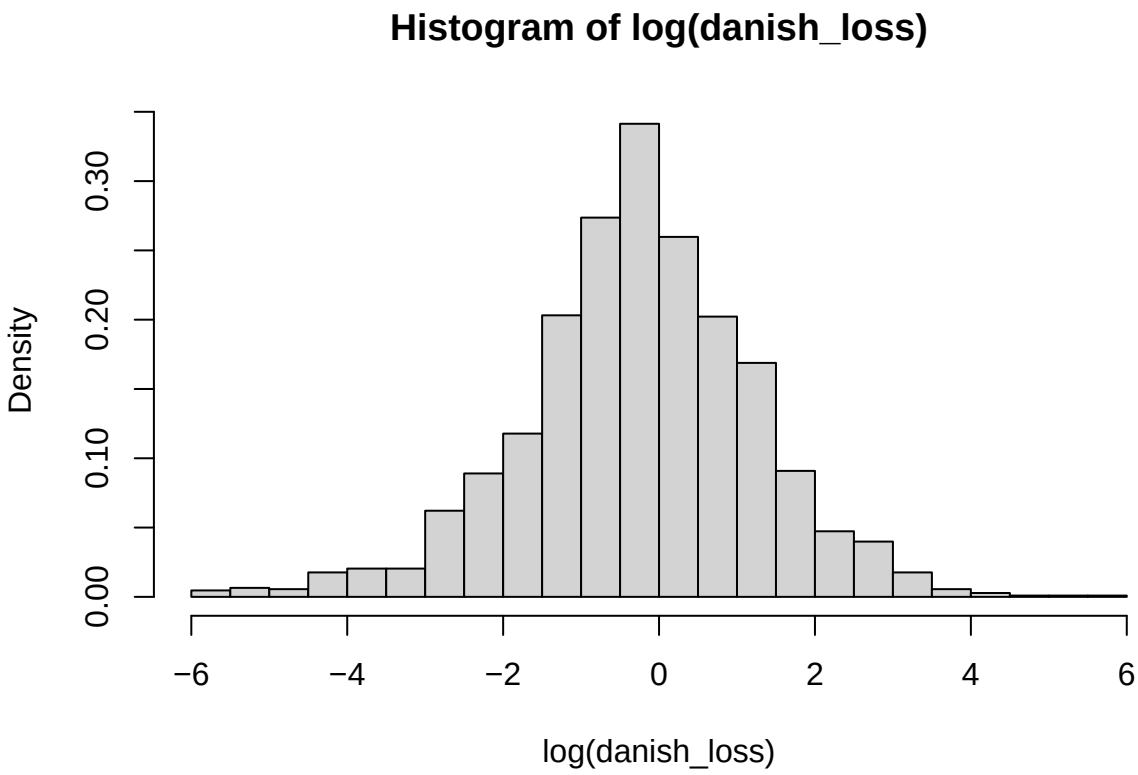
We can also use visual tools to assess the quality of the fit. Let us start by plotting the histogram of the data

```
hist(danish_loss, freq = F, breaks = 30)
```



Given that our data has losses that are quite large, in this case, it is more convenient to plot the logarithm of the data

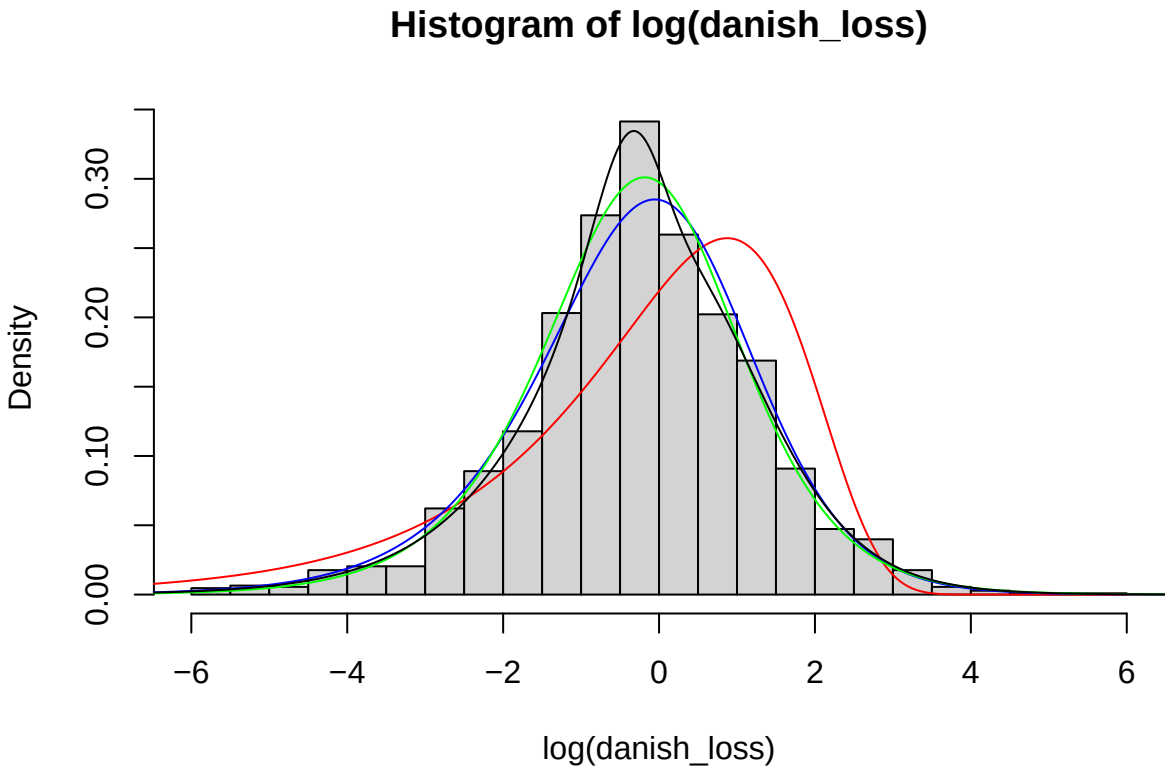
```
hist(log(danish_loss), freq = F, breaks = 30)
```



We now compare the histogram with the fitted distributions. Note, however, that we need to adapt our density functions accordingly to compare with the logarithm of the data by using the change of variable theorem.

```
sq <- seq(-7, 7, by = 0.01)
```

```
hist(log(danish_loss), freq = F, breaks = 30)
lines(sq,
  dgamma(exp(sq), fit_gamma$estimate[1], fit_gamma$estimate[2]) * exp(sq),
  col = "red"
)
lines(sq,
  dpareto(exp(sq), fit_pareto$estimate[1], fit_pareto$estimate[2]) * exp(sq),
  col = "blue"
)
lines(sq,
  dburr(exp(sq), fit_burr$estimate[1], fit_burr$estimate[2]) * exp(sq),
  col = "green"
)
lines(
  sq,
  dmgp(exp(sq), fit_mgp$estimate[1], fit_mgp$estimate[2], fit_mgp$estimate[3], fit_mgp$estimate[4],
)
```



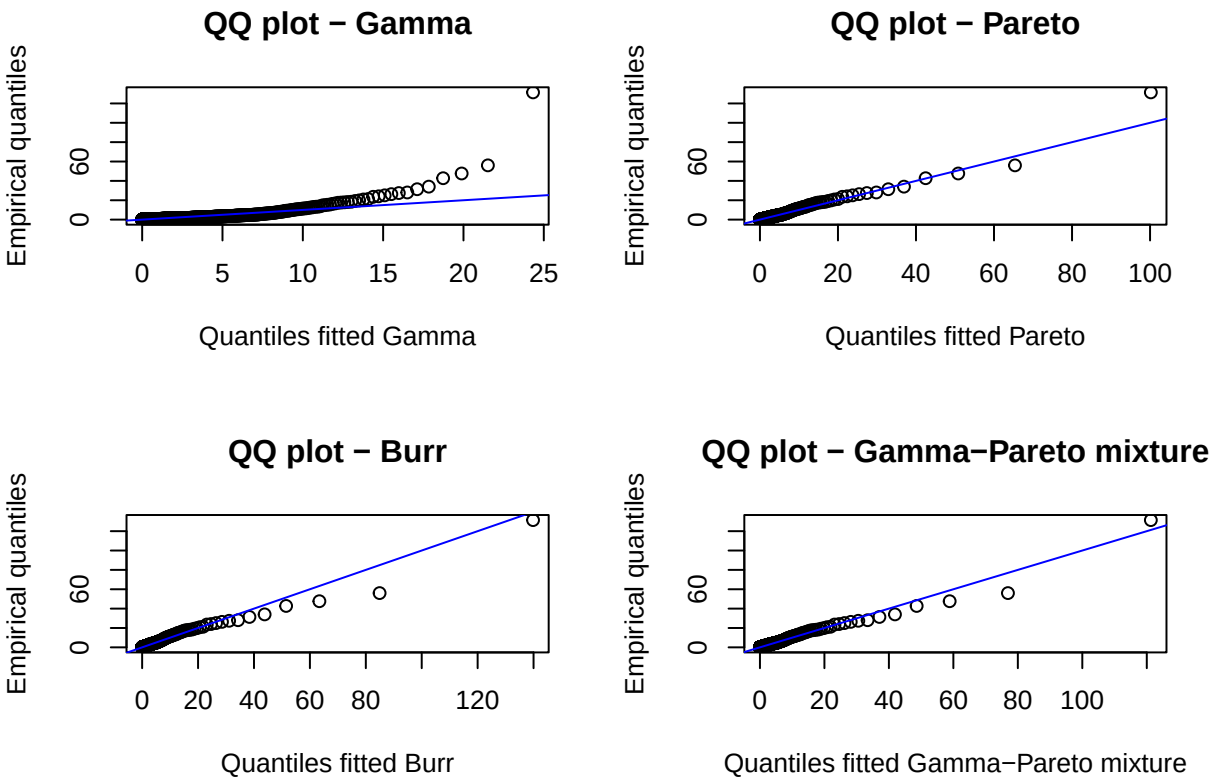
We observe that the density mixture distribution is closer to the histogram. Finally, we can also create QQ-plots to evaluate the fit. Notice that to create the QQ-plot of the mixture distribution, we need to implement a function to compute its quantiles. An implementation is the following:

```
qmgp <- function(p, shapeg, rateg, shapep, scalep, prob) {
  L2 <- function(q, p) {
    (p - pmgp(q, shapeg, rateg, shapep, scalep, prob))^2
  }
  sapply(p, function(p) optimize(L2, c(0, 10^3), p = p)$minimum)
}
qmgp(0.5, 1, 1, 1, 1, 0.5)
```

```
## [1] 0.8064592
```

We can now create the corresponding QQ-plots:

```r
p <- seq(0.001, 0.999, by = 0.001)
danish_quant <- quantile(danish_loss, p)

par(mfrow = c(2, 2))
plot(qgamma(p, fit_gamma$estimate[1], fit_gamma$estimate[2]),
  danish_quant,
  main = "QQ plot - Gamma",
  xlab = "Quantiles fitted Gamma",
  ylab = "Empirical quantiles"
)
abline(0, 1, col = "blue")

plot(qpareto(p, fit_pareto$estimate[1], fit_pareto$estimate[2]),
  danish_quant,
  main = "QQ plot - Pareto",
  xlab = "Quantiles fitted Pareto",
  ylab = "Empirical quantiles"
)
abline(0, 1, col = "blue")

plot(qburr(p, fit_burr$estimate[1], fit_burr$estimate[2]),
  danish_quant,
  main = "QQ plot - Burr",
  xlab = "Quantiles fitted Burr",
  ylab = "Empirical quantiles"
)
abline(0, 1, col = "blue")

plot(qmgp(p, fit_mgp$estimate[1], fit_mgp$estimate[2], fit_mgp$estimate[3], fit_mgp$estimate[4], fit
  danish_quant,
  main = "QQ plot - Gamma-Pareto mixture",
  xlab = "Quantiles fitted Gamma-Pareto mixture",
  ylab = "Empirical quantiles"
)
abline(0, 1, col = "blue")
```



One of the difficulties of this data set is the large values, which make it challenging to find a model that adequately models the body and tail simultaneously. However, the problem can be split. On the one hand, one can find a model for the body of the distribution and, on the other, a model for the tail of the distribution by, for example, using Extreme value theory statistics. In fact, this data set has been analyzed extensively using Extreme Value Theory techniques.

2.3.3 Other estimation methods

Although MLE is the most commonly used estimation method, we now review some alternatives, namely the methods of moments, quantile-matching, and maximum goodness-of-fit. Note that all of these methods are available in the `fitdistrplus` R package.

Method of moments Let $X_1, \dots, X_n$ be i.i.d. random variables with common distribution function $F(\cdot; \boldsymbol{\theta})$, where $\boldsymbol{\theta} = (\theta_1, \dots, \theta_d) \in \boldsymbol{\Theta} \subset \mathbb{R}^d$. Assuming that the moments up to order d of $X \sim F(\cdot; \boldsymbol{\theta})$ exist, we set

$$\mu_k(\boldsymbol{\theta}) = \mathbb{E}\left[X^k\right] = \mu_k\,, \quad k = 1, \dots, d\,.$$

In particular, this means that given $\mu_1, \dots, \mu_d$ we have d nonlinear equations with d unknown θ's. Therefore, we can find $\boldsymbol{\theta}$ from the raw moments of X, assuming that a unique solution exists, which is often the case. In practice, we can use the empirical raw moments $\hat{\mu}_k$ given by

$$\hat{\mu}_k = \frac{1}{n}\sum_{i=1}^{n} x_i^k\,, \quad k = 1, \dots, d\,.$$

Then, the moment matching estimator $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$ is the solution to

$$\mu_k(\boldsymbol{\theta}) = \hat{\mu_k}\,, \quad k = 1, \dots, d\,.$$

Sometimes it is more convenient to base the estimation on central moments. Let

$$m_1(\boldsymbol{\theta}) = \mathbb{E}\left[X\right] \quad \text{and} \quad m_k(\boldsymbol{\theta}) = \mathbb{E}\left[\left(X - \mathbb{E}\left[X\right]\right)^k\right]\,, k = 2, \dots, d\,,$$

and

$$\hat{m}_1 = \bar{x}_n \quad \text{and} \quad \hat{m}_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_n)^k\,, \quad k = 2, \dots, d\,.$$

Then, we can solve for $\boldsymbol{\theta}$ in

$$m_k(\boldsymbol{\theta}) = \hat{m}_k\,, \quad k = 1, \ldots, d\,,$$

to find the moment matching estimator $\hat{\boldsymbol{\theta}}$.

In general, there are no closed-form formulas for the moment-matching estimators, and numerical methods must be employed. In R, we can use, for example, the `fitdist()` function in the `fitdistrplus` R package to perform moment-matching estimation by simply adjusting the argument `method`. For example, for our exponential distributed sample

```
set.seed(1)
lambda <- 1.5
x_exp <- rexp(1000, lambda)
```

we compute the moment matching estimator as follows:

```
fit_exp_mme <- fitdist(x_exp, distr = "exp", method = "mme")
summary(fit_exp_mme)
```

```
## Fitting of the distribution ' exp ' by matching moments
## Parameters :
##      estimate
## rate 1.454471
## Loglikelihood:  -625.3576   AIC:  1252.715   BIC:  1257.623
```

Remark. You may have noticed that the maximum likelihood and the moment-matching estimators coincide for our exponentially distributed sample. This is a particular property of the exponential distribution. However, in general, this is not the case for other distributions.

Quantile-matching Quantile-matching estimation follows a similar idea to the moment-matching estimation. The main difference is that we now match the empirical quantiles of the given sample and the theoretical quantiles of the parametric distribution to be fitted. More specifically, we have the following equations

$$F^{\leftarrow}(p_k; \boldsymbol{\theta}) = Q_{n,p_k}\,, \quad k = 1, \ldots, d\,,$$

where Q_{n,p_k} are the empirical quantiles of the data for specified probabilities $p_k \in [0,1]$, $k = 1, \ldots, d$. The solution $\hat{\boldsymbol{\theta}}$ for the above equations is the quantile-matching estimator and is (typically) computed using numerical optimization. Note that the probabilities $p_k \in [0,1]$, $k = 1, \ldots, d$ must be pre-selected and, thus, the resulting estimator would depend on this selection. We can use the `fitdist()` function in R to perform this estimation method. We simply change the input of the argument `method` to "qme". Note that in this case, we need to specify the probabilities via the argument `probs`. Let us give a specific example

```
fit_exp_qme <- fitdist(x_exp, distr = "exp", method = "qme", probs = 0.5)
summary(fit_exp_qme)
```

```
## Fitting of the distribution ' exp ' by matching quantiles
## Parameters :
##      estimate
## rate 1.415335
## Loglikelihood:  -625.7263   AIC:  1253.453   BIC:  1258.36
```

Maximum Goodness-of-fit The last method that we present here is the Goodness-of-fit estimation method, also known as minimum distance estimation. The idea of this method is to find a parameter $\boldsymbol{\theta}$ that minimizes the "distance" between the empirical distribution function $F_n(x) = n^{-1} \sum_{i=1}^{n} \mathbf{1}(x_i \leq x)$ and the parametric distribution $F(x; \boldsymbol{\theta})$. The distance can be measure, for example, using the Cramer–von Mises distance defined as

$$D(\boldsymbol{\theta}) = \int_{-\infty}^{\infty} (F_n(x) - F(x; \boldsymbol{\theta}))^2 dx\,,$$

which in practice can be estimated as

$$\hat{D}(\boldsymbol{\theta}) = \frac{1}{12n} + \sum_{i=1}^{n} \left(F(x;\boldsymbol{\theta}) - \frac{2i-1}{2n} \right)^2 dx\,.$$

Thus, maximum goodness-of-fit estimation translates to finding $\boldsymbol{\theta}$ that minimizes $\hat{D}(\boldsymbol{\theta})$. In R, we can access this method by changing the argument `method` to "mge" in the `fitdist()` function. Let us finish this section by trying this method in our exponential sample

```
fit_exp_mge <- fitdist(x_exp, distr = "exp", method = "mge")
```

```
## Warning in fitdist(x_exp, distr = "exp", method = "mge"): maximum GOF
## estimation has a default 'gof' argument set to 'CvM'
```

```
summary(fit_exp_mge)
```

```
## Fitting of the distribution ' exp ' by maximum goodness-of-fit
## Parameters :
##      estimate
## rate  1.41384
## Loglikelihood:  -625.7552   AIC:  1253.51   BIC:  1258.418
```

Note that the Cramer-von Mises distance is the default option, but it can be changed to other distances (see help for details).

2.4 Multivariate distributions

In this section, we review some parametric models for random vectors. First, we review one of the most well know multivariate models, namely the multivariate normal distribution, and then we present a more general multivariate modeling approach using copulas.

2.4.1 Multivariate normal distribution

Recall that a p-dimensional random vector $\mathbf{X} = (X_1, \ldots, X_p)$ is said to be multivariate normal distributed with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ if its joint density function $f_{\mathbf{X}}$ is given by

$$f_{\mathbf{X}}(x_1, \ldots, x_p) = \frac{1}{\sqrt{(2\pi)^p |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}) \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})^\top \right),$$

where $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$. We write $\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

There are several implementations of the multivariate normal distribution in R. However, we will employ the one available in the `mvtnorm` package.

```
library(mvtnorm)
```

We start by exemplifying how to compute the joint density function via the `dmvnorm()` function:

```
dmvnorm(c(0, 0))
```

```
## [1] 0.1591549
```

```
dmvnorm(c(0, 0, 1))
```

```
## [1] 0.03851084
```

We can also compute density evaluations for several data points by giving them in a matrix form. For instance,

```
x <- matrix(c(c(0, 0), c(1, 1)), 2, 2, byrow = T)
x
```

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

```
dmvnorm(x)
```

```
## [1] 0.15915494 0.05854983
```

Note that the default value for the mean vector is the vector of 0's, and for the covariance matrix, the identity matrix. We can change these default values by providing the desired information in the respective arguments. For example,

```
mu <- c(1, 1)
sigma <- matrix(c(4, 2, 2, 3), ncol = 2)
sigma
```

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

```
dmvnorm(c(0, 0), mean = mu, sigma = sigma)
```

```
## [1] 0.04664928
```

Recall that for a random $\mathbf{X}$, its joint distribution function $F_{\mathbf{X}}$ is given by

$$F_{\mathbf{X}}(x_1, \ldots, x_p) = \mathbb{P}(X_1 \leq x_1, \ldots, X_p \leq x_p)\,.$$

The joint distribution function of $\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ can be evaluated in R using the `pmvnorm()` function. For instance, we can compute

$$\mathbb{P}(X_1 \leq 1, X_2 \leq 4)\,,$$

for $\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ given by

```
mu <- c(1, 1)
sigma <- matrix(c(4, 2, 2, 3), ncol = 2)
```

as follows

```
pmvnorm(mean = mu, sigma = sigma, upper = c(1, 4))
```

```
## [1] 0.4970487
## attr(,"error")
## [1] 1e-15
## attr(,"msg")
## [1] "Normal Completion"
```

Moreover, `pmvnorm()` allow us to compute more complicated probabilities. For example, to compute

$$\mathbb{P}(X_1 \leq 1, -1 < X_2 \leq 4)\,,$$

we simply specify values in the `lower` argument of `pmvnorm()` as follows

```
pmvnorm(mean = mu, sigma = sigma, lower = c(-Inf, -1), upper = c(1, 4))
```

```
## [1] 0.3892527
## attr(,"error")
## [1] 1e-15
## attr(,"msg")
## [1] "Normal Completion"
```

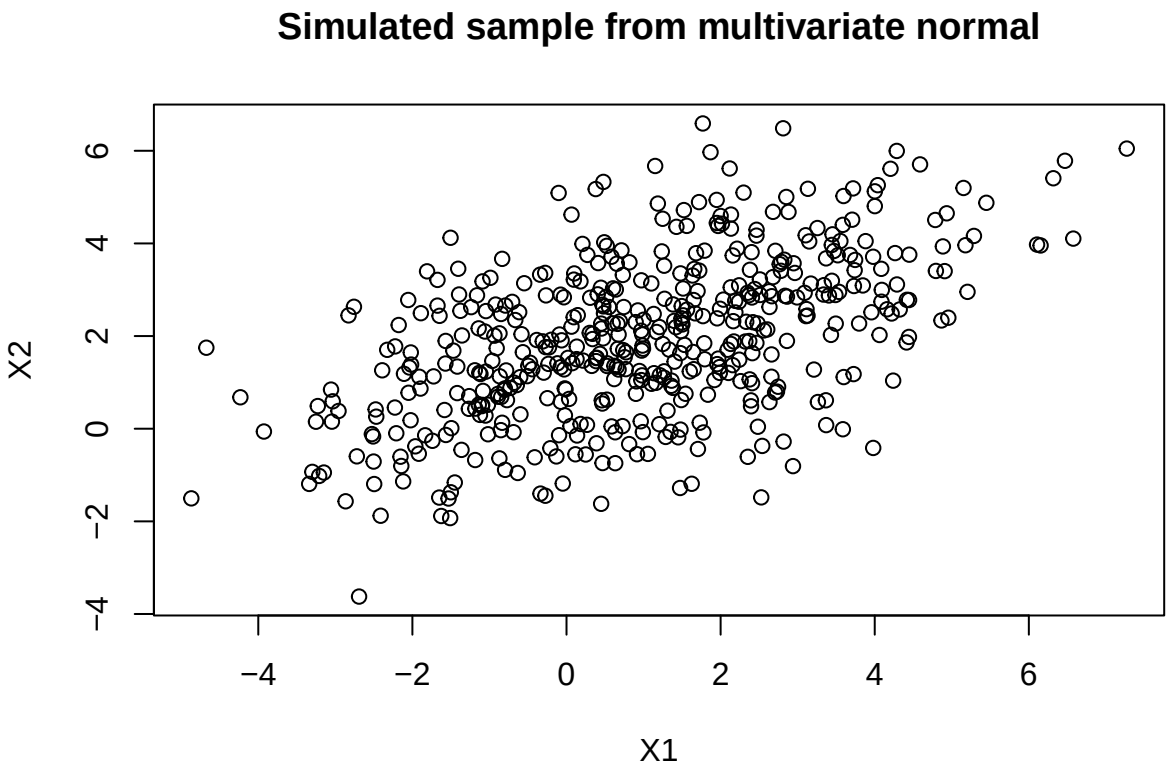
Simulation of multivariate normal models can be performed using the `rmvnorm()` function. For example,

```
x <- rmvnorm(n = 500, mean = c(1, 2), sigma = sigma)
head(x)
```

```
##            [,1]     [,2]
## [1,]  0.4651394 2.677685
## [2,]  1.2112011 1.154777
```

```
## [3,]  0.7727047 1.649591
## [4,]  4.2882523 3.114262
## [5,]  0.4101875 3.575816
## [6,] -1.1396597 2.166214
```

```
plot(x,
  main = "Simulated sample from multivariate normal",
  xlab = "X1", ylab = "X2"
)
```



Remark. The MLEs for $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are explicit and given by the vector of sample means and the sample covariance matrix, which can be computed easily as

```
colMeans(x)
```

```
## [1] 0.9603442 1.9359186
```

```
var(x) * (1 - 1 / nrow(x))
```

```
##          [,1]     [,2]
## [1,] 4.366248 1.903049
## [2,] 1.903049 2.905198
```

In the next section, we will cover the estimation of more complex models.

Let us conclude the present section by recalling the close-form expression for the joint characteristic function of $\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

$$\varphi_{\mathbf{X}}(\mathbf{t}) = \exp\left(i\mathbf{t}\boldsymbol{\mu}^\top - \frac{1}{2}\mathbf{t}\boldsymbol{\Sigma}\mathbf{t}^\top\right).$$

2.4.2 Copulas

A p-dimensional copula is a distribution function on $[0,1]^p$ with uniform marginals over $[0,1]$. In other words, for a p-dimensional random vector $\mathbf{U} = (U_1, \ldots, U_p)$ on the unit cube, a copula C is

$$C(u_1, \ldots, u_p) = \mathbb{P}(U_1 \leq u_1, \ldots, U_p \leq u_p).$$

The importance of studying copulas for multivariate modeling is summarized by the following theorem, known as Sklar's Theorem.

Theorem 2.6. *Let F be a joint distribution function with marginals $F_1, \ldots, F_p$. Then there exists a copula $C : [0,1]^p \to [0,1]$ such that, for all $x_1, \ldots, x_p$ in $\bar{\mathbb{R}} = [-\infty, \infty]$,*

$$F(x_1, \ldots, x_p) = C(F_1(x_1), \ldots, F_p(x_p)) \,. \tag{2.3}$$

Moreover, if the marginals are continuous, then C is unique. Conversely, if C is a copula and $F_1, \ldots, F_p$ are univariate distribution functions, then the function F defined in (2.3) is a joint distribution function with margins $F_1, \ldots, F_p$.

We now recall a fundamental proposition in probability theory. In particular, i) is key for performing stochastic simulation.

Proposition 2.1. *Let F be a distribution function and $F^{\leftarrow}$ its generalized inverse. Then*

i) If U is a standard uniform distributed random variable, that is, $U \sim U(0,1)$, then $\mathbb{P}(F^{\leftarrow}(U) \leq x) = F(x)$.

ii) If $Y \sim F$, with F a continuous distribution function, then $F(Y) \sim U(0,1)$.

Thus, in the case of continuous marginals, Sklar's Theorem also suggests that we can find the copula of a given joint distribution. More specifically, if a random vector $\mathbf{X}$ has joint distribution function F with continuous marginal distributions $F_1, \ldots, F_p$, then the copula of F (or $\mathbf{X}$) is the joint distribution function C of the random vector $(F_1(X_1), \ldots, F_d(X_p))$.

Copulas allow for modeling very different types of dependence structures. For instance, the independence case is retrieved by the *independence copula* given by

$$C_{ind}(u_1, \ldots, u_p) = \prod_{i=1}^{p} u_i \,.$$

Indeed, Sklar's Theorem implies that r.v.'s (with continuous distributions) are independent if and only if their copula is the independence copula.

Another important property of copulas is their invariance under strictly increasing transformations of the marginals, as stated in the following proposition.

Proposition 2.2. *Let $(X_1, \ldots, X_p)$ be a random vector with continuous marginals and copula C. Let $T_1, \ldots, T_p$ be strictly increasing functions. Then $(T_1(X_1), \ldots, T_p(X_p))$ also has copula C.*

As previously mentioned, copulas can be obtained from given distribution functions. In particular, we can derive copulas from several well-known multivariate distributions. These are commonly known as *implicit* copulas. For example, consider $\mathbf{X} \sim N(\mathbf{0}, \boldsymbol{P})$, where $\mathbf{0}$ is the p-dimensional vector of zeroes, and $\boldsymbol{P}$ is a $p \times p$ correlation matrix. Then, we define the *Gaussian* copula as the copula of $\mathbf{X}$. More specifically, the Gaussian copula is given by

$$C^{Ga}_{\boldsymbol{P}}(\mathbf{u}) = \mathbb{P}(\Phi(X_1) \leq u_1, \ldots, \Phi(X_p) \leq u_p) = \boldsymbol{\Phi}_{\boldsymbol{P}}(\Phi^{-1}(u_1), \ldots, \Phi^{-1}(u_p)) \,,$$

where Φ denotes the distribution function of a (univariate) standard normal and $\boldsymbol{\Phi}_{\boldsymbol{P}}$ is the joint distribution function of $\mathbf{X}$. Note that considering more general $\mathbf{X} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, leads to the same family of Gaussian copulas due to the property of invariance under increasing transformations.

It turns out that the Gaussian copula falls into the more general family of *elliptical copulas*, which we review next.

Elliptical copulas

Before introducing elliptic copulas, we define elliptical distributions. We say that a random vector $\mathbf{X}$ follows an elliptical distribution if its characteristic function $\varphi_{\mathbf{X}}$ is of the form

$$\varphi_{\mathbf{X}}(\mathbf{t}) = \exp(i\mathbf{t}\boldsymbol{\mu}^\top)\psi\left(\mathbf{t}\boldsymbol{\Sigma}\mathbf{t}^\top\right),$$

for some vector $\boldsymbol{\mu}$ known as the location vector, some positive definite matrix $\boldsymbol{\Sigma}$ known as the dispersion matrix, and some function ψ. Note, for instance, that the multivariate normal distribution is a particular case with $\psi(t) = \exp(-\frac{1}{2}t)$. Another important example of elliptical distributions is the multivariate t distribution. Recall that $\mathbf{X}$ is said to be multivariate t-distributed with ν degrees of freedom if its joint density function is given by

$$f(\mathbf{x}) = \frac{\Gamma(\frac{1}{2}(v+p))}{\Gamma(\frac{1}{2}v)\sqrt{(2\pi)^p|\boldsymbol{\Sigma}|}}\left(1+\frac{1}{v}(\mathbf{x}-\boldsymbol{\mu})\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})^\top\right)^{-(v+p)/2}.$$

We write $\mathbf{X} \sim t(\nu, \boldsymbol{\mu}, \boldsymbol{\Sigma})$. In this case, $\psi(t) = \hat{H}(-\frac{1}{2}t)$, where $\hat{H}$ is the Laplace transform of an (appropriate) Inverse Gamma distribution.

Definition 2.1. An *elliptical copula* is the implicit copula of an elliptical distribution.

Remark. Since copulas are invariant to increasing transformations of the marginals, elliptical copulas are typically defined in terms of the standardized dispersion matrix, or correlation matrix.

Now, let us review how to work with elliptical copulas in R. There are several packages to do so, for example, `copula` and `fCopulae`. However, we will use `copula` here.

```
library(copula)
```

Elliptical copulas can be defined in R by using the `ellipCopula()` function, which creates objects of type `copula`. Next, we present an example of how to define a Gaussian copula of dimension 2 with a correlation of 0.4:

```
gaussian_cop <- ellipCopula(
  family = "normal", # Gaussian copula
  dim = 2, # Dimension of the copula
  dispstr = "ex", # Structure of the correlation matrix
  param = 0.4 # Correlation
)
gaussian_cop
```

```
## Normal copula, dim. d = 2
## Dimension:  2
## Parameters:
##   rho.1   = 0.4
```

The argument `dispstr` characterizes the structure of the correlation matrix. The available structures are "ex" for exchangeable, "ar1" for AR(1), "toep" for Toeplitz, and "un" for unstructured. For example, for dimension 3, the corresponding matrices are:

$$\begin{pmatrix} 1 & \rho_1 & \rho_1 \\ \rho_1 & 1 & \rho_1 \\ \rho_1 & \rho_1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & \rho_1 & \rho_1^2 \\ \rho_1 & 1 & \rho_1 \\ \rho_1^2 & \rho_1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & \rho_1 & \rho_2 \\ \rho_1 & 1 & \rho_1 \\ \rho_2 & \rho_1 & 1 \end{pmatrix}, \quad \text{and} \quad \begin{pmatrix} 1 & \rho_1 & \rho_2 \\ \rho_1 & 1 & \rho_3 \\ \rho_2 & \rho_3 & 1 \end{pmatrix}.$$

To exemplify these constructions, next, we make use of the `getSigma()` function to recover the correlation matrices of Gaussian copulas defined using the different values of `dispstr`:

```
getSigma(ellipCopula(
  family = "normal",
  dim = 3, dispstr = "ex",
  param = 0.4
))
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  0.4  0.4
## [2,]  0.4  1.0  0.4
## [3,]  0.4  0.4  1.0
```

```
getSigma(ellipCopula(
  family = "normal",
  dim = 3, dispstr = "ar1",
  param = 0.4
))
```

```
##      [,1] [,2] [,3]
## [1,] 1.00  0.4 0.16
## [2,] 0.40  1.0 0.40
## [3,] 0.16  0.4 1.00
```

```
getSigma(ellipCopula(
  family = "normal",
  dim = 3, dispstr = "toep",
  param = c(0.4, 0.5)
))
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  0.4  0.5
## [2,]  0.4  1.0  0.4
## [3,]  0.5  0.4  1.0
```

```
getSigma(ellipCopula(
  family = "normal",
  dim = 3, dispstr = "un",
  param = c(0.4, 0.5, 0.2)
))
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  0.4  0.5
## [2,]  0.4  1.0  0.2
## [3,]  0.5  0.2  1.0
```

We can evaluate the density and distribution functions of a `copula` object via the `dCopula()` and `pCopula()` functions, respectively. For example,

```
dCopula(c(0.5, 0.2), gaussian_cop)
```

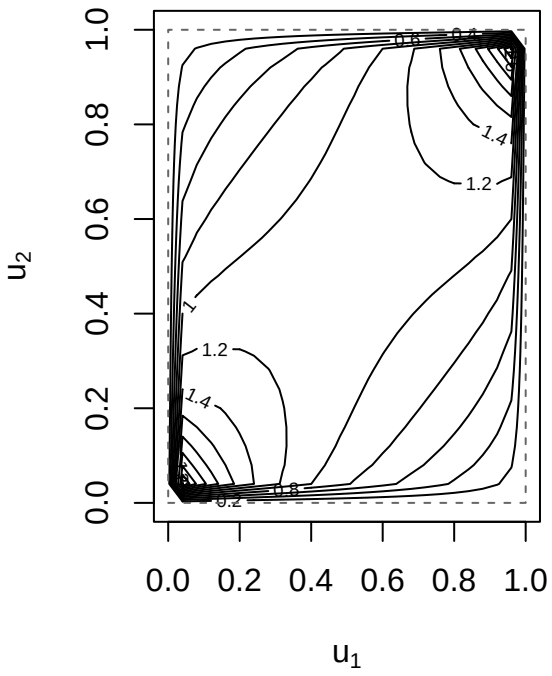
```
## [1] 1.019913
```

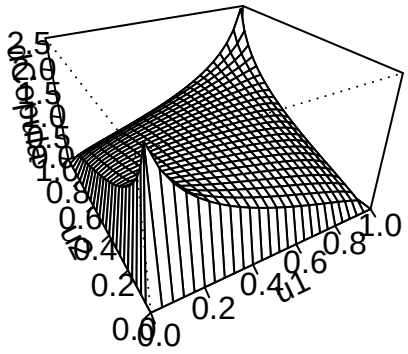
```
pCopula(c(0.5, 0.2), gaussian_cop)
```

```
## [1] 0.1449953
```

Moreover, for 2-dimensional copulas, we can visualize the joint density function by using the `contour()` and `persp()` functions:

```
par(mfrow = c(1, 2))
contour(gaussian_cop, dCopula)
persp(gaussian_cop, dCopula)
```



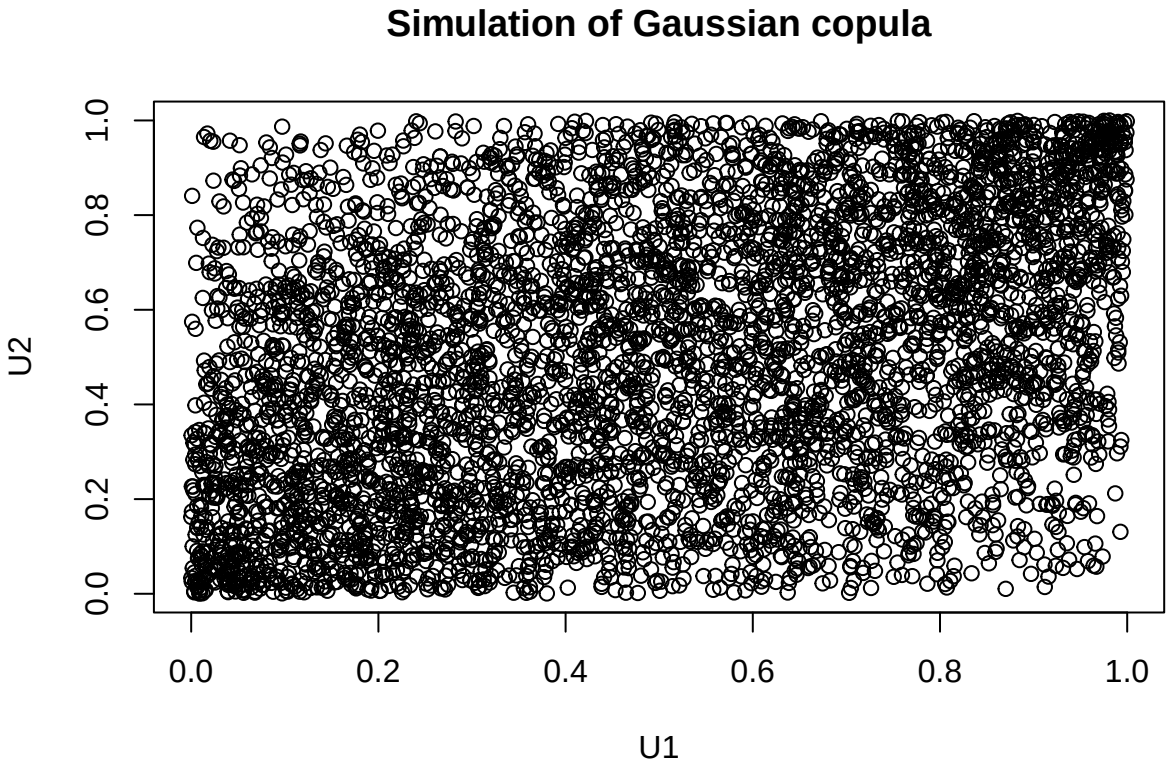


Finally, to generate random values from our copula object, we use `rCopula()`.

```
gauc_samp <- rCopula(5000, gaussian_cop)
head(gauc_samp)
```

```
##             [,1]       [,2]
## [1,] 0.54092961 0.32358013
## [2,] 0.89182178 0.95080018
## [3,] 0.21661897 0.09522707
## [4,] 0.05448803 0.59939116
## [5,] 0.41222385 0.74123338
## [6,] 0.98678731 0.94826755
```

```
plot(gauc_samp,
  main = "Simulation of Gaussian copula",
  xlab = "U1",
  ylab = "U2"
)
```



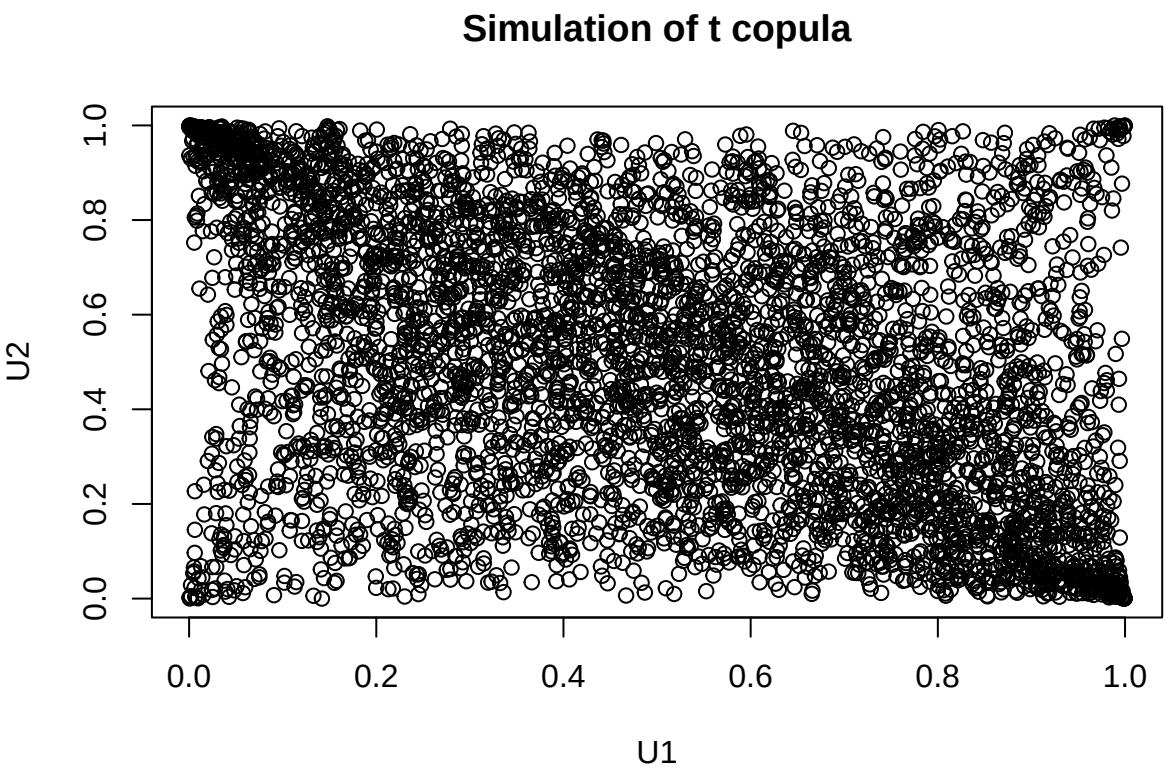
Now, let us exemplify how to define a t-copula:

```
t_cop <- ellipCopula(
  family = "t",
  dim = 2,
  dispstr = "ex",
  param = -0.4,
  df = 2
)
t_cop
```

```
## t-copula, dim. d = 2
## Dimension:  2
## Parameters:
##   rho.1   = -0.4
##   df      =  2.0
```

We can now use the methods `dCopula()`, `pCopula()`, and `rCopula()` to compute the density and distribution functions and generate random values for this new object, respectively. For example,

```
tc_samp <- rCopula(5000, t_cop)
plot(tc_samp,
  main = "Simulation of t copula",
  xlab = "U1",
  ylab = "U2"
)
```



While elliptical copulas are implied by the well-known multivariate distribution functions of elliptical distributed random vectors, the copulas themselves do not have simple closed-form expressions. However, there are families of copulas that have simple closed-form formulas. These are often referred to as *explicit* copulas. Next, we review one of the most famous families of explicit copulas: the *Archemidiean* copulas.

Archimedean copulas

An Archimedean copula is characterized by a generator function $\phi : [0,1] \to [0,\infty]$ as

$$C(\mathbf{u}) = \phi^{-1}\left(\phi(u_1) + \cdots + \phi(u_p)\right) , \tag{2.4}$$

where ϕ^{-1} is the inverse of the generator ϕ. In order for (2.4) to be a copula, a sufficient condition for the generator ϕ is that its inverse $\phi^{-1} : [0,\infty] \to [0,1]$ needs to be *completely monotonic*. In this family, the three most classical copulas are the Gumbel copula, the Frank copula, and the Clayton copula. These copulas are constructed as follows:

Gumbel copula. The generator of this copula is $\phi(t) = (-\log(t))^\alpha$, $\alpha > 1$. Then

$$C_{Gu}(\mathbf{u}) = \exp\left(-\left((-\log(u_1))^\alpha + \cdots + (-\log(u_p))^\alpha\right)^{1/\alpha}\right) .$$

Frank copula. In this case, the generator is $\phi(t) = -\log((\exp(-\alpha t) - 1)/(\exp(-\alpha) - 1))$, $\alpha \in \mathbb{R}$. Then

$$C_F(\mathbf{u}) = -\frac{1}{\alpha}\log\left(1 + \frac{(\exp(-\alpha u_1) - 1)\cdots(\exp(-\alpha u_p) - 1)}{\exp(-\alpha) - 1}\right) .$$

Clayton copula The generator of this copula is given by $\phi(t) = \alpha^{-1}(t^{-\alpha} - 1)$, $\alpha > 0$ (or $\alpha > -1$ for dimension 2). Then

$$C_C(\mathbf{u}) = \left(u_1^{-\alpha} + \cdots + u_p^{-\alpha} - p + 1\right)^{-1/\alpha} .$$

Remark. Note that the three copulas above are uniparametric.

In R, we can use the `archmCopula()` function to construct Archimedean copulas. For example, a 2-dimensional Gumbel copula of parameter $\alpha = 2.5$ is created as follows

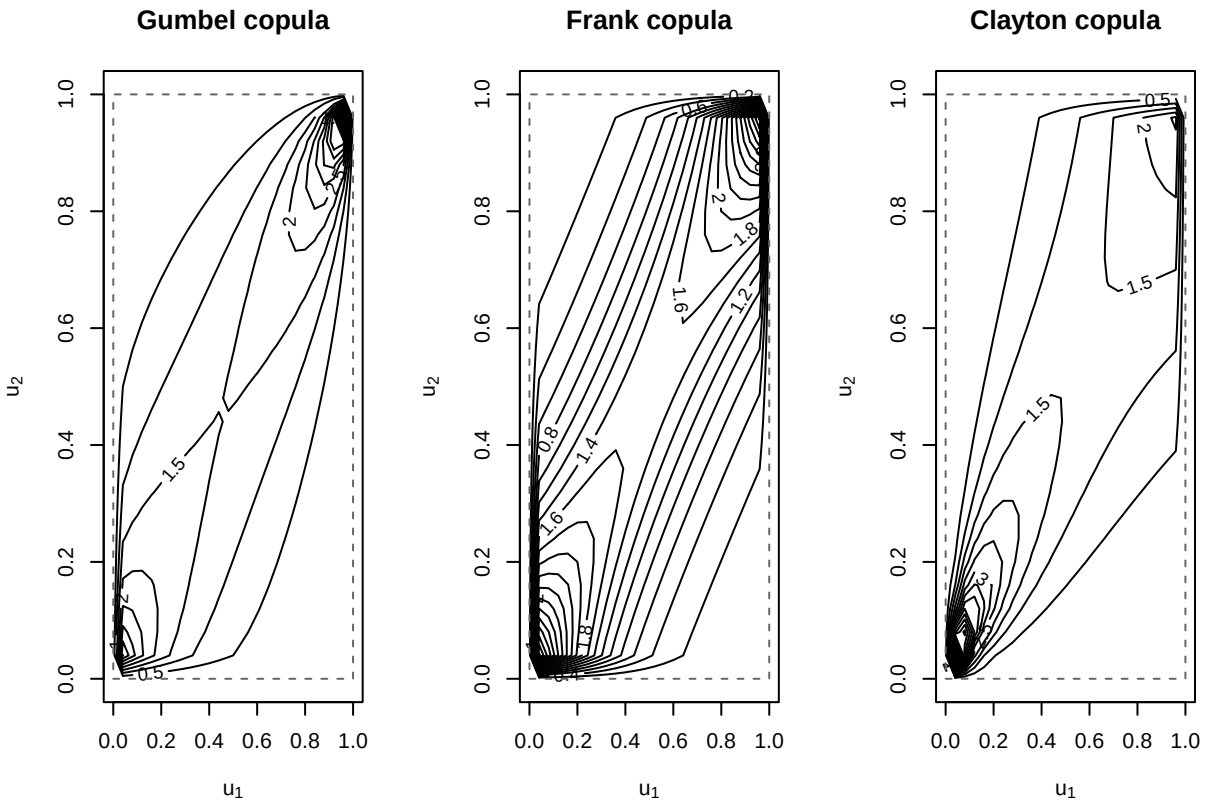
```
gum_cop <- archmCopula(family = "gumbel", dim = 2, param = 2.5)
gum_cop
```

```
## Gumbel copula, dim. d = 2
## Dimension:  2
## Parameters:
##   alpha   = 2.5
```

Again, `dCopula()`, `pCopula()`, and `rCopula()` give access to the density and distribution functions and the random values generator, respectively.

For example,

```
par(mfrow = c(1, 3))
contour(archmCopula(family = "gumbel", dim = 2, param = 2),
  dCopula,
  main = "Gumbel copula",
  nlevels = 20
)
contour(archmCopula(family = "frank", dim = 2, param = 5.5),
  dCopula,
  main = "Frank copula",
  nlevels = 20
)
contour(archmCopula(family = "clayton", dim = 2, param = 2),
  dCopula,
  main = "Clayton copula",
  nlevels = 20
)
```



We will study some properties of these Archimedean copulas when we cover dependence measures.

2.4.3 Constructing multivariate distributions from copulas

Given a copula C and marginals $F_1, \ldots, F_p$, we now want to construct a multivariate distribution from these components. The `copula` package allows us to do this via the `mvdc()` function, which creates objects of the type `mvdc`. There are three main arguments for this function: `copula`, which is a copula object, `margins` a character vector with the names of the marginals, and `paramMargins`, which is a list with the parameters of the marginals. The use of this function is better illustrated with an example. Let us imagine that we want to create a bivariate distribution with normally distributed marginals $X_1 \sim N(1, 4)$ and $X_2 \sim N(2, 2)$ and copula the Clayton copula with parameter $\alpha = 2$. First, we need the copula object:

```
clay_cop <- archmCopula(family = "clayton", dim = 2, param = 2)
```

We can now define our `mvdc` object:

```
my_mvd <- mvdc(
  copula = clay_cop,
  margins = c("norm", "norm"),
  paramMargins = list(list(mean = 1, sd = 2), list(mean = 2, sd = sqrt(2)))
)
my_mvd
```

```
## Multivariate Distribution Copula based ("mvdc")
##  @ copula:
## Clayton copula, dim. d = 2
## Dimension:  2
## Parameters:
##   alpha   = 2
##  @ margins:
## [1] "norm" "norm"
##    with 2 (not identical)  margins; with parameters (@ paramMargins)
## List of 2
##  $ :List of 2
##   ..$ mean: num 1
##   ..$ sd  : num 2
##  $ :List of 2
##   ..$ mean: num 2
##   ..$ sd  : num 1.414214
```

We can now use `dMvdc()` to access the joint density of our multivariate model:

```
dMvdc(c(1,2), my_mvd)
```

```
## [1] 0.08333573
```

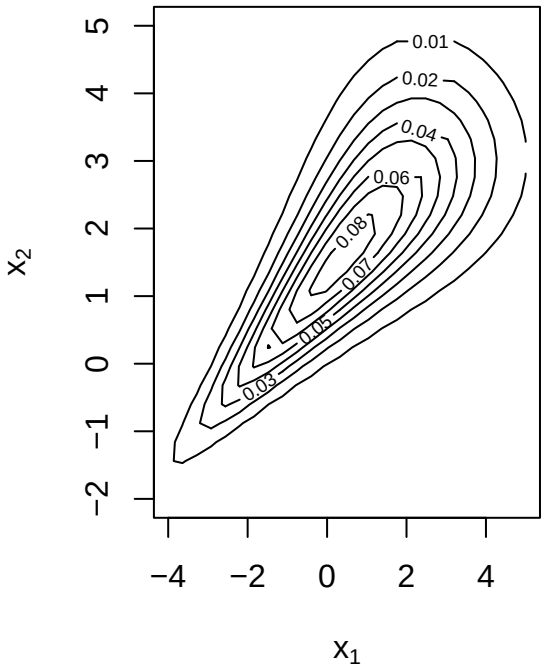
The joint distribution function can be evaluated using `pMvdc()`:

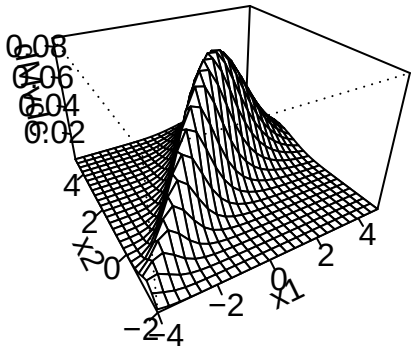
```
pMvdc(c(1,2), my_mvd)
```

```
## [1] 0.3779645
```

Again, we can use `contour()` and `persp()` to visualize our distribution:

```
par(mfrow = c(1, 2))
contour(my_mvd, dMvdc, xlim = c(-4, 5), ylim = c(-2, 5)) # xlim and ylim must be given
persp(my_mvd, dMvdc, xlim = c(-4, 5), ylim = c(-2, 5))
```



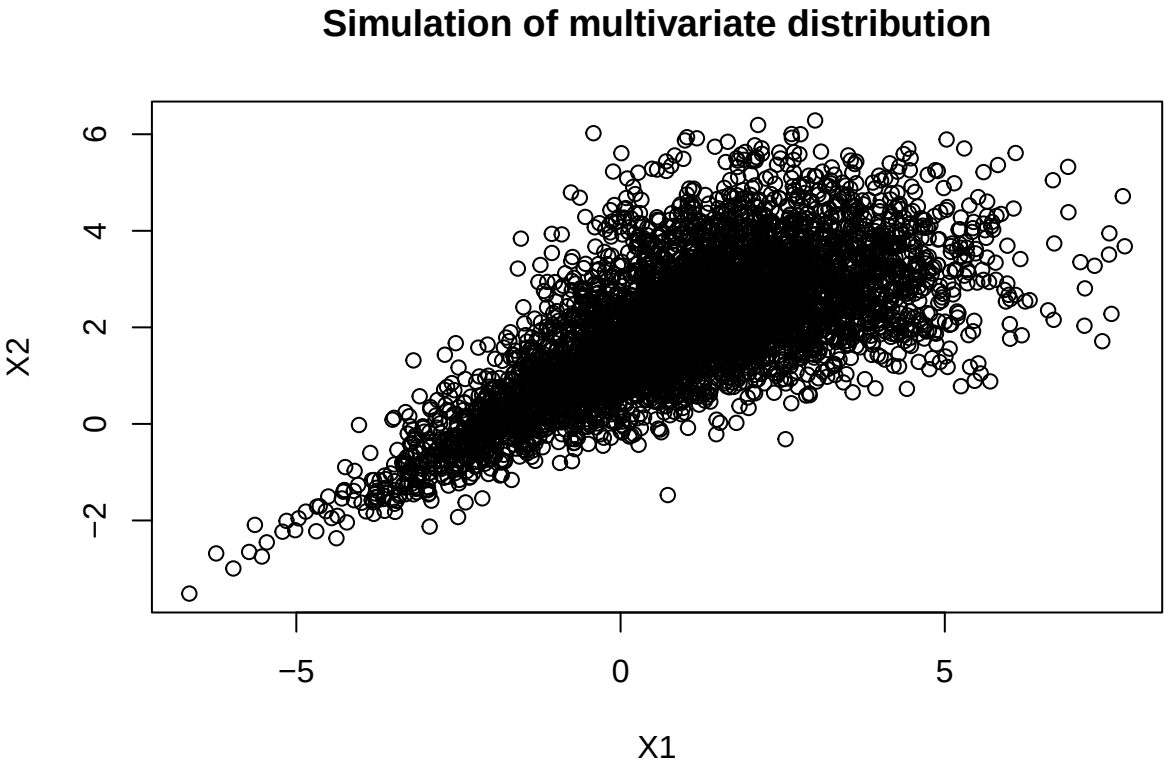


Finally, random values from our multivariate model can be generated using `rMvdc()`:

```
my_mvdc_sample <- rMvdc(5000, my_mvd)
head(my_mvdc_sample)
```

```
##            [,1]      [,2]
## [1,]  0.1009846 0.7844453
## [2,] -1.3441178 0.6030844
## [3,]  2.5885680 2.5168967
## [4,]  1.8196410 3.0904643
## [5,]  3.0041156 2.0874910
## [6,] -1.4500833 0.8198126
```

```
plot(my_mvdc_sample,
  main = "Simulation of multivariate distribution",
  xlab = "X1",
  ylab = "X2"
)
```



Remark. We can pass user-defined distributions for the marginals as long as the density (`d`), distribution (`p`), and quantile (`q`) functions are available. For example, if we name our distribution `foo`, then we need the `dfoo()`, `pfoo()`, and `qfoo()` functions.

2.4.4 Dependence measures

We now present some measures that assess the strength of dependence between random variables in different ways. We will limit ourselves to bivariate random vectors (X_1, X_2) for presentation purposes.

We first recall one of the most popular dependence measures: Pearson correlation. The correlation between X_1 and X_2, denoted by $\rho(X_1, X_2)$, is given by

$$\rho(X_1, X_2) = \frac{\mathbb{E}(X_1X_2) - \mathbb{E}(X_1)\mathbb{E}(X_2)}{\sqrt{\mathrm{Var}(X_1)\mathrm{Var}(X_2)}}.$$

We know that if X_1 and X_2 are independent, then $\rho(X_1, X_2) = 0$, but the converse is false. If $|\rho(X_1, X_2)| = 1$ is equivalent to saying that $X_2 = a + bX_1$ for some $a \in \mathbb{R}$ and $b \neq 0$.

There are several pitfalls of using the correlation. For instance, we can have perfectly dependent random variables that exhibit relatively low correlation:

```
set.seed(1)
x <- rexp(100)
y <- exp(x * 10)
cor(x, y)
```

```
## [1] 0.4163004
```

Moreover, the correlation depends on the marginal distributions and may not even exist in some cases (we require finite second moments).

Hence, we now introduce other measures of dependence that depend only on the underlying copula of our multivariate model.

Rank correlation

There are two crucial rank correlation measures: Kendall's tau and Spearman's rho. The empirical estimators of rank correlations can be calculated by looking at the ranks of the data (i.e., the positions of the data), hence the name. We begin by defining Kendall's tau, which can be understood as a measure of concordance for a bivariate random vector. Recall that two point in $\mathbb{R}^2$, lets say (x_1, x_2) and $(\tilde{x_1}, \tilde{x_2})$, are said to be *concordant* if $(x_1 - \tilde{x}_1)(x_2 - \tilde{x}_2) > 0$ and *discordant* if $(x_1 - \tilde{x}_1)(x_2 - \tilde{x}_2) < 0$. This motivates the following definition for random vectors (X_1, X_2)

Definition 2.2. For two random variables X_1 and X_2, Kendall's tau is given by

$$\rho_\tau(X_1, X_2) = \mathbb{P}((X_1 - \tilde{X}_1)(X_2 - \tilde{X}_2) > 0) - \mathbb{P}((X_1 - \tilde{X}_1)(X_2 - \tilde{X}_2) < 0)\,,$$

where $(\tilde{X}_1, \tilde{X}_2)$ is an independent copy of (X_1, X_2)

In R, we can compute the empirical Kedall's tau by using `method = "kendall"` in `cor()`:

```
set.seed(1)
x <- rexp(100)
y <- exp(x * 10)
cor(x, y, method = "kendall")
```

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

The second way to measure rank correlation is with Spearma's rho. Although this measure can also be defined in terms of concordance and discordance for random pairs, we adopt the following definition:

Definition 2.3. For two random variables X_1 and X_2 with distribution functions F_1 and F_2, respectively, Spearma's rho is given by

$$\rho_S(X_1, X_2) = \rho(F_1(X_1), F_2(X_2))\,,$$

where ρ denotes the Pearson correlation.

In R, the empirical Spearman's rho can be computed by changing `method = "spearman"` in `cor()`:

```
set.seed(1)
x <- rexp(100)
y <- exp(x * 10)
cor(x, y, method = "spearman")
```

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

As previously mentioned, an important feature of rank correlations is that they depend only on the (unique) copula of (X_1, X_2).

Proposition 2.3. *Let X_1 and X_2 random variables with continuous marginal distribution functions and unique copula C. The rank correlations are given by*

$$\rho_\tau(X_1, X_2) = 4 \int_0^1 \int_0^1 C(u_1, u_2) dC(u_1, u_2) - 1$$

$$\rho_S(X_1, X_2) = 12 \int_0^1 \int_0^1 (C(u_1, u_2) - u_1 u_2) du_1 du_2$$

The `copula` package comes with the functions `tau()` and `rho()` to compute the rank correlations for a copula object. Let us give an example:

```
clay_cop <- archmCopula(family = "clayton", dim = 2, param = 2)
tau(clay_cop)
```

```
## [1] 0.5
```

Table 2.3: Kendall's tau and coefficients of tail dependence for some copulas. D_1 is the Debye function $D_1(\theta) = \theta^{-1} \int_0^\theta t/(\exp(t) - 1)dt$ and $t_{\nu+1}$ is the cdf of a univariate t distribution with $\nu + 1$ degrees of freedom.

Copula	ρ_τ	λ_u	λ_l
Gaussian	$(2/\pi)\arcsin(\rho)$	0	0
t	$(2/\pi)\arcsin(\rho)$	$2t_{\nu+1}\left(-\sqrt{(\nu+1)(1-\rho)/(1+\rho)}\right)$	Same as λ_u
Gumbel	$1 - 1/\alpha$	$2 - 2^{1/\alpha}$	0
Frank	$1 - 4\alpha^{-1}(1 - D_1(\alpha))$	0	0
Clayton ($\alpha > 0$)	$\alpha/(\alpha + 2)$	0	$2^{-1/\alpha}$

```
rho(clay_cop)
```

```
## [1] 0.6828928
```

Coefficients of tail dependence

The coefficients of tail dependence measure the strength of dependence in the tails of a bivariate distribution. These are defined as follows:

Definition 2.4. Let X_1 and X_2 be random variables with distribution functions F_1 and F_2, respectively. The *coefficient of upper tail dependence* λ_u of X_1 and X_2 is

$$\lambda_u := \lambda_u(X_1, X_2) = \lim_{q \to 1^-} \mathbb{P}(X_2 > F_2^{\leftarrow}(q) \mid X_1 > F_1^{\leftarrow}(q)),$$

provided a limit $\lambda_u \in [0, 1]$ exists. If $\lambda_u \in (0, 1]$, then we say that X_1 and X_2 show upper tail dependence. Alternatively, if $\lambda_u = 0$, then they are said to be asymptotically independent in the upper tail. Similarly, we define the *coefficient of lower tail dependence* λ_l as

$$\lambda_l := \lambda_u(X_1, X_2) = \lim_{q \to 0^+} \mathbb{P}(X_2 \leq F_2^{\leftarrow}(q) \mid X_1 \leq F_1^{\leftarrow}(q)),$$

provided a limit $\lambda_l \in [0, 1]$ exists.

When the distribution functions of X_1 and X_2 are continuous, the upper and lower tail dependence coefficients can be written (solely) in terms of the copula of the bivariate distribution. More specifically, we have that

$$\lambda_u = \lim_{q \to 1^-} \frac{1 - 2q + C(q, q)}{1 - q} \quad \text{and} \quad \lambda_l = \lim_{q \to 0^+} \frac{C(q, q)}{q}.$$

In R, we can compute these coefficients for a copula object using `lambda()`. For example,

```
lambda(clay_cop)
```

```
##     lower     upper
## 0.7071068 0.0000000
```

We conclude this section by providing Table 2.3, which contains the closed-form formulas of some dependence measures for the copula models introduced so far.

2.4.5 Fitting

Next, we will review two estimation methods for multivariate models based on copulas.

Full maximum likelihood estimation

Consider $\mathbf{X} = (X_1, \ldots, X_p)$ a random vector with joint distribution function F which is specified by marginals with distribution functions $F_1 \ldots, F_p$ and densities $f_1 \ldots, f_p$, and a copula C with density

c. Furthermore, let $\boldsymbol{\beta}_j$ be the parameters of the jth marginal, that is, $f_j(\cdot, \boldsymbol{\beta}_j)$, $j = 1, \ldots, p$, $\boldsymbol{\alpha}$ be the parameters of the copula C, i.e., $C(\cdot, \boldsymbol{\alpha})$, and $\boldsymbol{\theta} = (\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_p, \boldsymbol{\alpha})$. Next, suppose that we have n independent realizations $\tilde{\mathbf{x}} = \{\mathbf{x}_i = (x_{i1}, \ldots, x_{ip}) : i = 1, \ldots, n\}$ from $\mathbf{X}$. Then, the loglikelihood function is given by

$$l(\boldsymbol{\theta}; \tilde{\mathbf{x}}) = \sum_{i=1}^{n} \log(c(F_1(x_{i1}; \boldsymbol{\beta}_1), \ldots, F_p(x_{ip}; \boldsymbol{\beta}_p); \boldsymbol{\alpha})) + \sum_{i=1}^{n} \sum_{j=1}^{p} \log(f_j(x_{ij}; \boldsymbol{\beta}_j)) \,.$$

Thus, the maximum likelihood estimator $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$ is given by

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} l(\boldsymbol{\theta}; \tilde{\mathbf{x}}) \,.$$

Let us now illustrate how to implement this procedure in R. Consider the following multivariate model:

```
my_mvd <- mvdc(
  copula = ellipCopula(family = "normal", param = 0.5),
  margins = c("gamma", "gamma"),
  paramMargins = list(list(shape = 2,scale = 1), list(shape = 3, scale = 2))
)
```

Then, we simulate a sample from the model above

```
n <- 200
set.seed(1)
sim_dat <- rMvdc(n, my_mvd)
```

Firstly, we can make loglikelihood evaluations for a certain set of parameters using the `loglikMvdc()` function. For example, using the original parameters of our model, we obtain:

```
loglikMvdc(param = c(2, 1, 3, 2, 0.5), sim_dat, my_mvd)
```

```
## [1] -793.0129
```

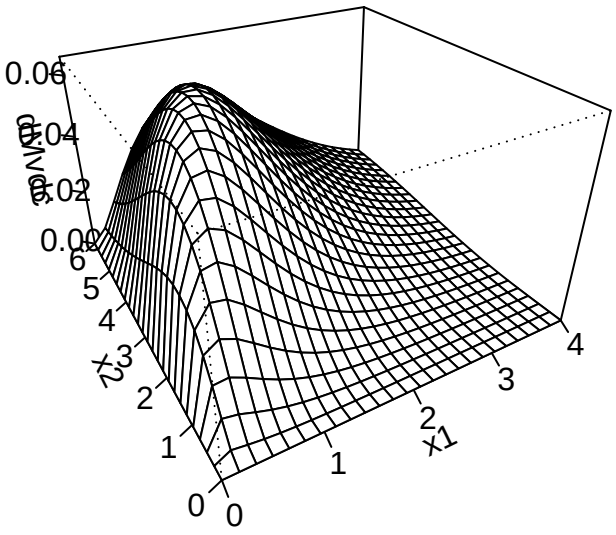
Note that the parameters must be passed in vector form and ordered: first, the parameters of the marginals and last, the copula parameters. Now, to perform MLE, we can use the `fitMvd()` function. In this case, we need to pass the data, an mvdc object with the structure that we want to fit, and the starting values for the optimization. For example,

```
mut_fit <- fitMvdc(sim_dat, my_mvd, start = c(1, 1, 1, 1, 0.1))
mut_fit
```

```
## Call: fitMvdc(data = sim_dat, mvdc = my_mvd, start = c(1, 1, 1, 1,
##     0.1))
## Maximum Likelihood estimation based on 200 2-dimensional observations.
## Copula:  normalCopula
## Margin 1 :
## m1.shape m1.scale
##   2.1431   0.9435
## Margin 2 :
## m2.shape m2.scale
##    3.308    1.859
## Copula:
##  rho.1
## 0.4879
## The maximized loglikelihood is -792.2
## Optimization converged
```

A useful characteristic of the implementation above is that we get an mvdc object with the estimated parameters as part of the output, which can be accessed using the `@` operator. Then, we can use all the methods for mvdc objects with our fitted multivariate model. For instance,

```
persp(mut_fit@mvdc, dMvdc, xlim = c(0, 4), ylim = c(0, 6))
```



Two-step estimation

When the dimension p increases, the number of parameters to be estimated increases as well, making the optimization problem harder. In such a case, separating the estimation problem into fitting the marginals and the copula separately can significantly reduce the computational burden. This approach is often referred to as a two-stage parametric maximum likelihood method. More specifically, the method consists of estimating first the parameters of the marginals, that is, we first compute

$$\hat{\boldsymbol{\beta}}_j = \arg\max_{\boldsymbol{\beta}} \sum_{i=1}^{n} \log(f_j(x_{ij}; \boldsymbol{\beta})), \quad j = 1, \ldots, p.$$

Next, we estimate the parameter $\boldsymbol{\alpha}$ of C, given the parameters of the marginals $\hat{\boldsymbol{\beta}}_j$, $j = 1, \ldots, p$, by finding

$$\hat{\boldsymbol{\alpha}} = \arg\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \log(c(F_1(x_{i1}; \hat{\boldsymbol{\beta}}_1), \ldots, F_p(x_{ip}; \hat{\boldsymbol{\beta}}_p); \boldsymbol{\alpha})).$$

In R, we can use the machinery introduced in Section 2.3 to fit the marginal distributions, and with those estimators at hand, then we can use the `fitCopula()` to find the MLE of the copula parameters. Let us now look at a specific example. First, we estimate the parameters of the marginals

```
par1 <- fitdist(sim_dat[, 1], distr = "gamma")$estimate
par1
```

```
##    shape     rate
## 2.142617 1.058976
```

```
par2 <- fitdist(sim_dat[, 2], distr = "gamma")$estimate
par2
```

```
##     shape      rate
## 3.3085663 0.5378679
```

Next, using the estimators above, we find the evaluation points $(F_1(x_{i1}; \hat{\boldsymbol{\beta}}_1), F_2(x_{ip}; \hat{\boldsymbol{\beta}}_2))$ that will be used to estimate the copula parameters.

```
u_dat <- cbind(pgamma(sim_dat[, 1], par1[1], par1[2]),
    pgamma(sim_dat[, 2], par2[1], par2[2]))
```

Finally, we estimate the copula parameters using `fitCopula()`

```
cop_fit <- fitCopula(my_mvd@copula, u_dat, start = 0.1)
cop_fit
```

```
## Call: fitCopula(my_mvd@copula, data = u_dat, ... = pairlist(start = 0.1))
## Fit based on "maximum pseudo-likelihood" and 200 2-dimensional observations.
## Copula: normalCopula
##  rho.1
## 0.4879
## The maximized loglikelihood is 27.2
## Optimization converged
```

Thus, the estimated multivariate model would be the following:

```
mut_fit_2stage <- mvdc(
  copula = cop_fit@copula,
  margins = c("gamma", "gamma"),
  paramMargins = list(
    list(shape = unname(par1[1]), rate = unname(par1[2])),
    list(shape = unname(par2[1]), rate = unname(par2[2]))
  )
)
mut_fit_2stage
```

```
## Multivariate Distribution Copula based ("mvdc")
##  @ copula:
## Normal copula, dim. d = 2
## Dimension:  2
## Parameters:
##   rho.1   = 0.4879325
##  @ margins:
## [1] "gamma" "gamma"
##    with 2 (not identical)  margins; with parameters (@ paramMargins)
## List of 2
##  $ :List of 2
##   ..$ shape: num 2.142617
##   ..$ rate : num 1.058976
##  $ :List of 2
##   ..$ shape: num 3.308566
##   ..$ rate : num 0.5378679
```

with corresponding loglikelihood

```
loglikMvdc(
  param = c(par1, par2, cop_fit@copula@parameters),
  sim_dat,
  mut_fit_2stage
)
```

```
## [1] -792.1991
```

2.5 Linear regression

Linear regression is one of the most commonly used statistical methods in practice. The main idea is to use explanatory variables (or covariates) to explain (and predict) a random variable of interest. More specifically, a linear regression model relates a response variable Y to a p-dimensional vector $X = (X_1, \ldots, X_p)$ of covariates via the identity

$$\mathbb{E}\left[Y \mid X\right] = \sum_{j=1}^{p} X_j \beta_j = X\boldsymbol{\beta}\,,$$

where $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)^\top$. It is common practice to add an *intercept* parameter β_0, so that the linear model becomes

$$\mathbb{E}[Y \mid X] = \beta_0 + X\boldsymbol{\beta}\,.$$

For notation convenience, the intercept is included among the other parameters, that is, $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^\top$. Note that $\boldsymbol{\beta}$ is now a $(p+1)$ dimensional vector. We now want to express our regression model in terms of matrix products. To that end, we need to make an extra adjustment consisting of adding the covariate $X_0 = 1$ to X. Thus, the regression model can be written as

$$\mathbb{E}[Y \mid X] = X\boldsymbol{\beta} = \beta_0 + \sum_{j=1}^{p} X_j\beta_j\,.$$

Now assume that we have a set of observations $(y_1, x_1), \ldots, (y_n, x_n)$, where $x_i = (x_{i1}, \ldots, x_{ip})$ are vectors of covariates, $i = 1, \ldots, n$, and that we want to estimate $\boldsymbol{\beta}$ from this data. The most common method to do so is the method of *least squares*, which consists of finding the coefficients $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)$ that minimize the residual sum of squares

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{n} x_{ij}\beta_j)^2\,.$$

Before showing how to minimize the above expression, we introduce further notation. Let $\mathbf{X}$ be the $n \times (p+1)$ matrix with i-th row the vector x_i (with 1 in the first position) and let $\mathbf{y} = (y_1, \ldots, y_n)^\top$. Then we can write the residual sum of squares as

$$RSS(\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\,.$$

It is easy to see that the value of $\boldsymbol{\beta}$ that minimizes the expression above is given by

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}\,.$$

With the estimated parameters $\hat{\boldsymbol{\beta}}$ at hand, we can now compute the predicted values $\hat{\mathbf{y}}$ for the input data as

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}\,.$$

Moreover, we could use the above expression to predict values for new covariate information.

In R, linear regression can be performed using the `lm()` function. Let us now look at a specific example, dealing with the *Prostate Cancer* data set available in the `genridge` R package. First, we need to load the data set into our working space:

```
library(genridge)
```

```
## Loading required package: car
```

```
## Loading required package: carData
```

```
##
## Attaching package: 'genridge'
```

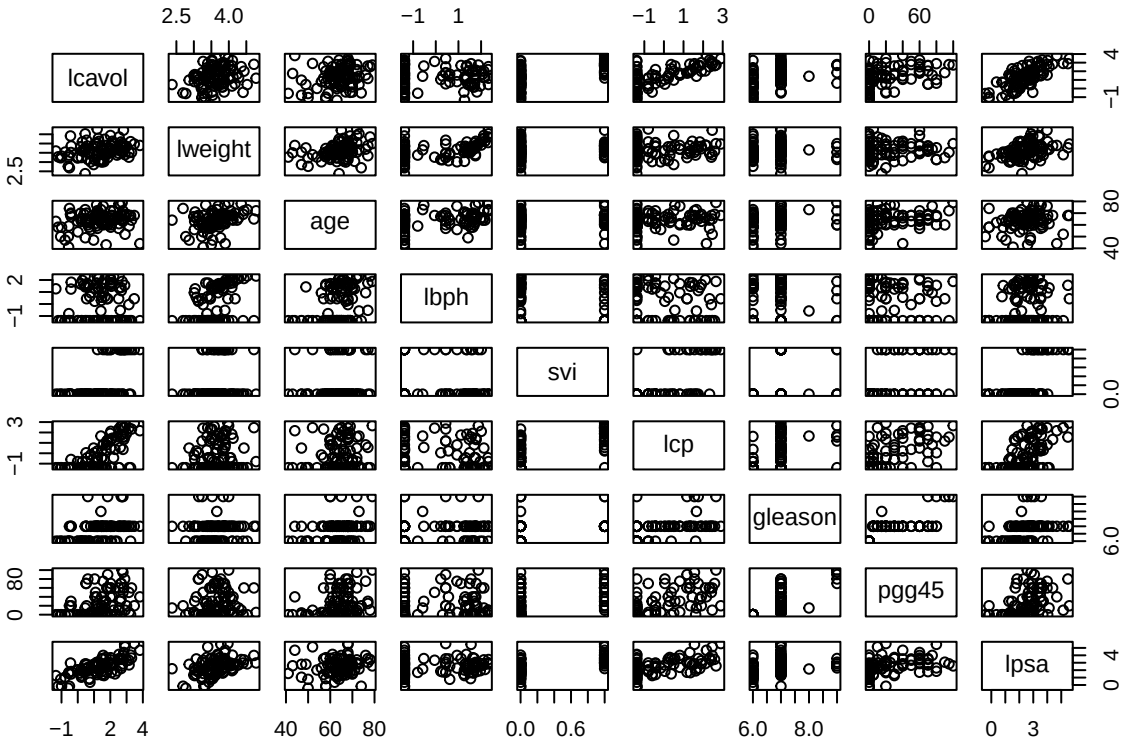
```
## The following object is masked from 'package:survival':
##
##     ridge
```

```
data(prostate)
prostate <- prostate[, -10] # No relevant for the present analysis
head(prostate)
```

```
##       lcavol  lweight age      lbph svi       lcp gleason pgg45       lpsa
## 1 -0.5798185 2.769459  50 -1.386294   0 -1.386294       6     0 -0.4307829
## 2 -0.9942523 3.319626  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 3 -0.5108256 2.691243  74 -1.386294   0 -1.386294       7    20 -0.1625189
## 4 -1.2039728 3.282789  58 -1.386294   0 -1.386294       6     0 -0.1625189
## 5  0.7514161 3.432373  62 -1.386294   0 -1.386294       6     0  0.3715636
## 6 -1.0498221 3.228826  50 -1.386294   0 -1.386294       6     0  0.7654678
```

We aim to explain the variable `lpsa` (level of prostate-specific antigen) in terms of the other variables. The following is a scatter plot matrix showing every pairwise plot between the variables:

```
plot(prostate)
```



One can observe, for example, that `lcavol` and `lpc` are positively correlated with `lpsa`. Moreover, note that `svi` is a binary variable, and `gleason` is a categorical variable.

We now fit a linear regression model using all our covariates:

```
prost_lm <- lm(formula = lpsa ~ lcavol + lweight + age + lbph
  + svi + lcp + gleason + pgg45, data = prostate)
prost_lm
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight + age + lbph + svi + lcp +
##     gleason + pgg45, data = prostate)
##
## Coefficients:
## (Intercept)       lcavol      lweight          age         lbph          svi
##    0.181561     0.564341     0.622020    -0.021248     0.096713     0.761673
##         lcp      gleason        pgg45
##   -0.106051     0.049228     0.004458
```

Using our linear model above, we can now compute the predicted values for the input data with the function `predic()`:

```
pros_pred <- predict(prost_lm)
head(pros_pred)
```

```
##         1         2         3         4         5         6
## 0.8229078 0.7612550 0.4416131 0.6199877 1.7315458 0.8434007
```

If we want to predict values of a new set of covariate values, these have to be passed into the `predict()` function in the form of a data frame with column names equal to those of the data frame used to create the linear mode. For example,

```
xn <- matrix(c(1.35, 3.6, 64, 0.1, 0, -0.18, 6, 2.5), 1)
colnames(xn) <-  names(prostate[, -9])
xn <- as.data.frame(xn) # Covariates have to be in the form of a data.frame
xn
```

```
##   lcavol lweight age lbph svi   lcp gleason pgg45
## 1   1.35     3.6  64  0.1   0 -0.18       6   2.5
```

```
predict(prost_lm, xn)
```

```
##        1
## 2.158081
```

So far, we have made minimal assumptions regarding the linear model. However, we need further assumptions to draw inference about the parameters and the model. More specifically, we assume that our model is of the form

$$Y_i = X_i\boldsymbol{\beta} + e_i\,, \quad i = 1, \ldots, n\,,$$

where the e_i, called error terms or *residuals*, are uncorrelated with $\mathbb{E}[e_i] = 0$ and $Var(e_i) = \sigma^2$. Moreover, note that by using the vector notation $\mathbf{e} = (e_1, \ldots, e_n)^\top$, we can rewrite the model as

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \mathbf{e}\,.$$

Under these assumptions, it is easy to see that the covariance matrix of the least squared estimator $\hat{\boldsymbol{\beta}}$ is given by

$$Var(\hat{\boldsymbol{\beta}}) = \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1}\,.$$

Note also that we can estimate the residuals as

$$\hat{\mathbf{e}} = \mathbf{y} - \hat{\mathbf{y}}\,.$$

In R, one can compute the residuals of a linear model using the `residuals()` function. For instance,

```
res_pros <- residuals(prost_lm)
summary(res_pros)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.76644 -0.35510 -0.00328  0.00000  0.38087  1.55770
```

It turns out that properties of the estimate residuals $\hat{\mathbf{e}}$, lead to the following unbiased estimator of the variance σ^2

$$\hat{\sigma}^2 = \frac{1}{n-p-1}(\mathbf{y} - \hat{\mathbf{y}})^\top(\mathbf{y} - \hat{\mathbf{y}}) = \frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\,.$$

If we further assume that the residuals are normally distributed with mean 0 and variance σ^2, that is, $e_i \sim N(0, \sigma^2)$, $i = 1 \ldots, n$, then it is easy to show that

$$\hat{\boldsymbol{\beta}} \sim N(\boldsymbol{\beta}, \sigma^2(\mathbf{X}^\top\mathbf{X})^{-1})\,,$$

and

$$(n-p-1)\hat{\sigma}^2 \sim \sigma^2\chi^2_{n-p-1}\,,$$

where χ^2_{n-p-1} denotes a chi-squared distribution with $n-p-1$ degrees of freedom. Moreover, $\hat{\boldsymbol{\beta}}$ and $\hat{\sigma}^2$ are independent. These properties are particularly useful for performing hypothesis testing and creating confidence intervals for the parameters β_j.

To test the hypothesis that a coefficient of the linear model β_j is equal to zero, i.e., $\beta_j = 0$, we consider the standardized coefficient or Z-score

$$z_j = \frac{\hat{\beta}_j}{\hat{\sigma}\sqrt{v_j}}\,,$$

where v_j is the j-th diagonal element of $(\mathbf{X}^\top\mathbf{X})^{-1}$. Under the null hypothesis that $\beta_j = 0$, z_j is t_{n-p-1} distributed, and hence large (absolute) values of z_j will lead to the rejection of this null hypothesis. In R, we can use `summary()` to access the values of these statistics (*t-statistic* column):

```
summary(prost_lm)
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight + age + lbph + svi + lcp +
##     gleason + pgg45, data = prostate)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.76644 -0.35510 -0.00328  0.38087  1.55770
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.181561   1.320568   0.137  0.89096
## lcavol       0.564341   0.087833   6.425 6.55e-09 ***
## lweight      0.622020   0.200897   3.096  0.00263 **
## age         -0.021248   0.011084  -1.917  0.05848 .
## lbph         0.096713   0.057913   1.670  0.09848 .
## svi          0.761673   0.241176   3.158  0.00218 **
## lcp         -0.106051   0.089868  -1.180  0.24115
## gleason      0.049228   0.155341   0.317  0.75207
## pgg45        0.004458   0.004365   1.021  0.31000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6995 on 88 degrees of freedom
## Multiple R-squared:  0.6634, Adjusted R-squared:  0.6328
## F-statistic: 21.68 on 8 and 88 DF,  p-value: < 2.2e-16
```

R also provides the *p-value*s associated with these tests (column *Pr(>|t|)*). Recall that a p-value is the probability of obtaining test results at least as "extreme" as the results observed, assuming that the null hypothesis is correct. In other words, the smaller the p-value, the stronger the evidence that you should reject the null hypothesis. A nice feature of R is that marks with `*` and `.` the parameters with p-values lower than commonly used significance levels.

It is often the case that we need to test for the significance of groups of coefficients simultaneously. For that purpose, we can use the F statistics, defined as

$$F = \frac{(RSS_0 - RSS_1)/(p_1 - p_0)}{RSS_1/(n - p_1 - 1)},$$

where RSS_1 is the residual sum of squares for the fit of the bigger model with $p_1 + 1$ parameters, and RSS_0 is the corresponding value for the (nested) smaller model with $p_0 + 1$ parameters. Under Gaussian assumption on the residuals, and that the smaller model is correct, the F statistics is $F_{p_1-p_0,n-p_1-1}$ distributed. In the output of the `summary()` function above, we have an F statistics (and corresponding p-value) that compares with a model with all coefficients equal to zero (i.e., test the null hypothesis: "All coefficients are zero").

Finally, note that `summary()` displays additional information, such as the *Multiple R-squared*. This measures the proportion of the variance in the response variable that the model can explain, and it varies from 0 to 1. Thus, we should aim for a model with multiple R-squared as close to 1 as possible. The last piece of information is the *Adjusted R-squared*, which is simply an adjustment to the multiple R-squared that considers the data size and the number of covariates. Hence, the adjusted R-squared is recommended to be used over the (conventional) multiple R-squared.

Considering all the information described above, let us now consider a smaller model that consists only of the covariates `lcavol`, `lweight`, `lbph`, `svi`, and without an intercept.

```
prost_lm2 <- lm(formula = lpsa ~ lcavol + lweight
  + lbph + svi - 1, data = prostate)
summary(prost_lm2)
```

```
##
## Call:
## lm(formula = lpsa ~ lcavol + lweight + lbph + svi - 1, data = prostate)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.84248 -0.39868  0.01499  0.44124  1.51028
##
## Coefficients:
##         Estimate Std. Error t value Pr(>|t|)
## lcavol   0.53274    0.07370   7.229 1.34e-10 ***
## lweight  0.44069    0.03083  14.293  < 2e-16 ***
## lbph     0.09098    0.04992   1.822 0.071619 .
## svi      0.71339    0.20651   3.455 0.000832 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7011 on 93 degrees of freedom
## Multiple R-squared:  0.9368, Adjusted R-squared:  0.9341
## F-statistic: 344.8 on 4 and 93 DF,  p-value: < 2.2e-16
```

We can see that for the covariates selected, we can reject the null hypothesis $\beta_j = 0$, and we obtain a larger adjusted R-squared. All good indicators that our model is better.

Furthermore, we can test if "the smaller model is correct" using the F statistics, which can be accessed in R via the `anova()` function:

```
anova(prost_lm, prost_lm2)
```

```
## Analysis of Variance Table
##
## Model 1: lpsa ~ lcavol + lweight + age + lbph + svi + lcp + gleason +
##     pgg45
## Model 2: lpsa ~ lcavol + lweight + lbph + svi - 1
##   Res.Df    RSS Df Sum of Sq     F Pr(>F)
## 1     88 43.058
## 2     93 45.715 -5   -2.6568 1.086 0.3738
```

Hence, we cannot reject the null hypothesis that "the smaller model is correct". We can also use information criteria to assess the quality of a model. For instance, we can compute the AIC for our two models using the `AIC()` function:

```
AIC(prost_lm)
```

```
## [1] 216.4952
```

```
AIC(prost_lm2)
```

```
## [1] 212.303
```

Confirming our previous selection.

We are often also interested in constructing confidence intervals for the coefficients β_j. Fortunately, it is easy to see that under the Gaussian assumption on the residuals, a $(1-\alpha)$ confidence interval for β_j is given by

$$\hat{\beta}_j \pm z_{1-\alpha/2} v_j^{1/2} \hat{\sigma} \,,$$

where $z_{1-\alpha/2}$ is the $(1-\alpha/2)$ quantile of a standard normal distribution. We can compute confidence intervals in R via the `confint()` function. For instance, for our last model, we have

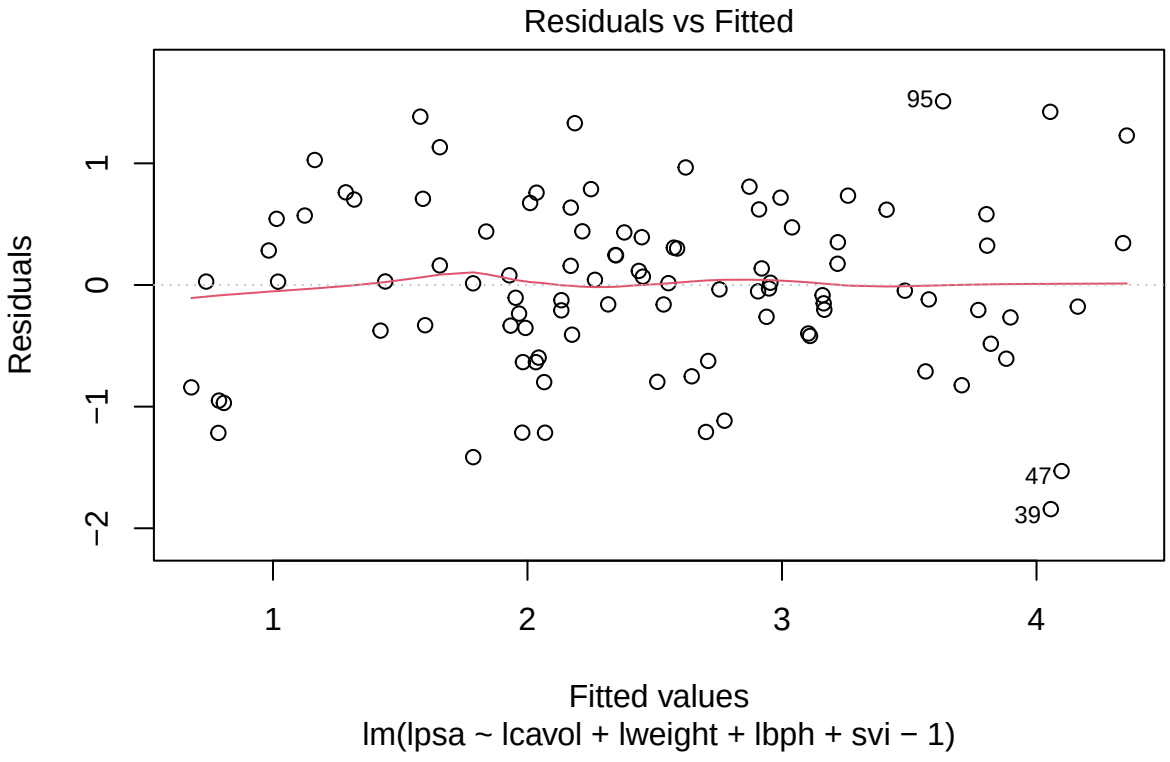
```
confint(prost_lm2)
```

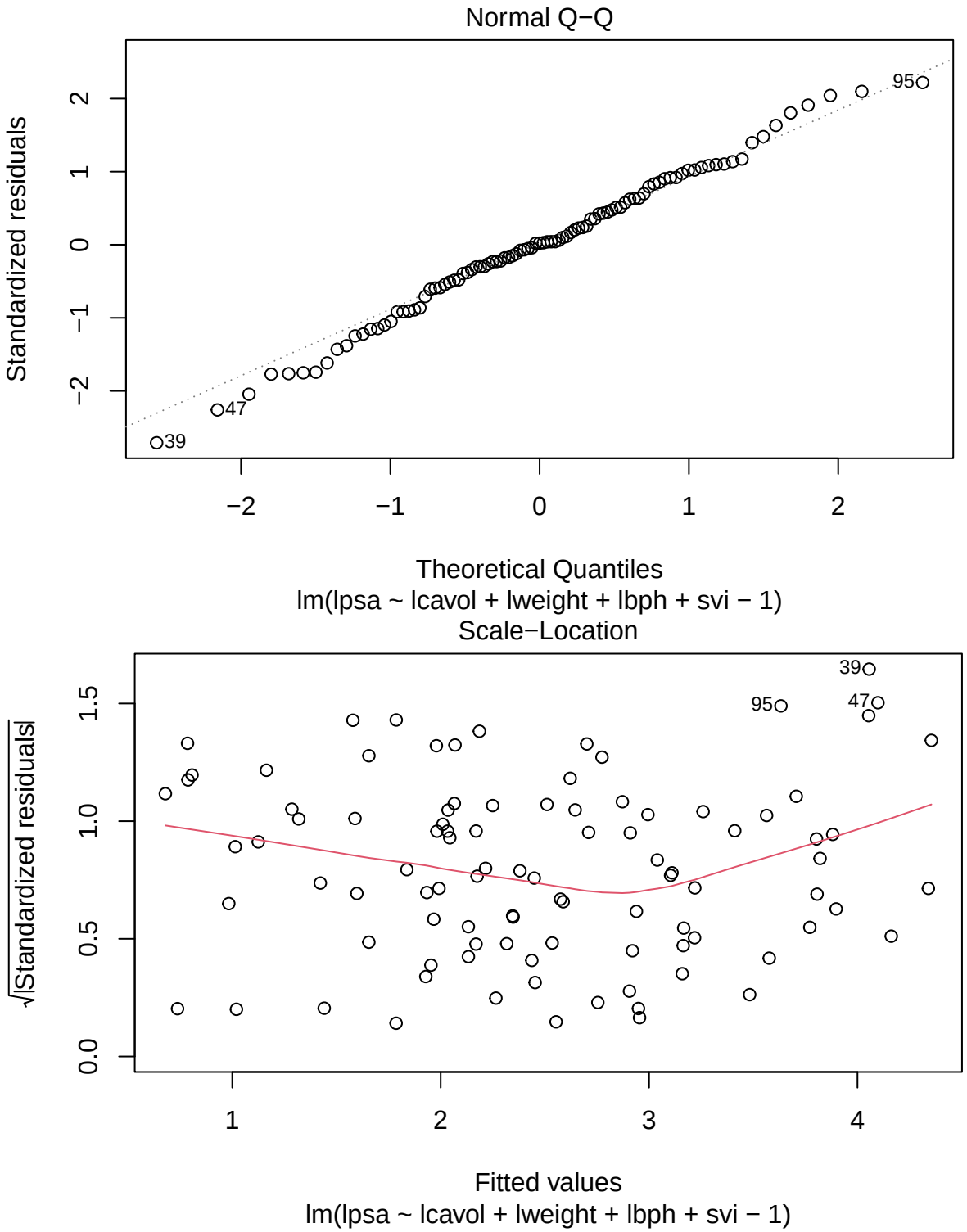
```
##                2.5 %    97.5 %
## lcavol   0.386398071 0.6790891
## lweight  0.379458771 0.5019129
## lbph    -0.008161222 0.1901136
## svi      0.303311220 1.1234695
```

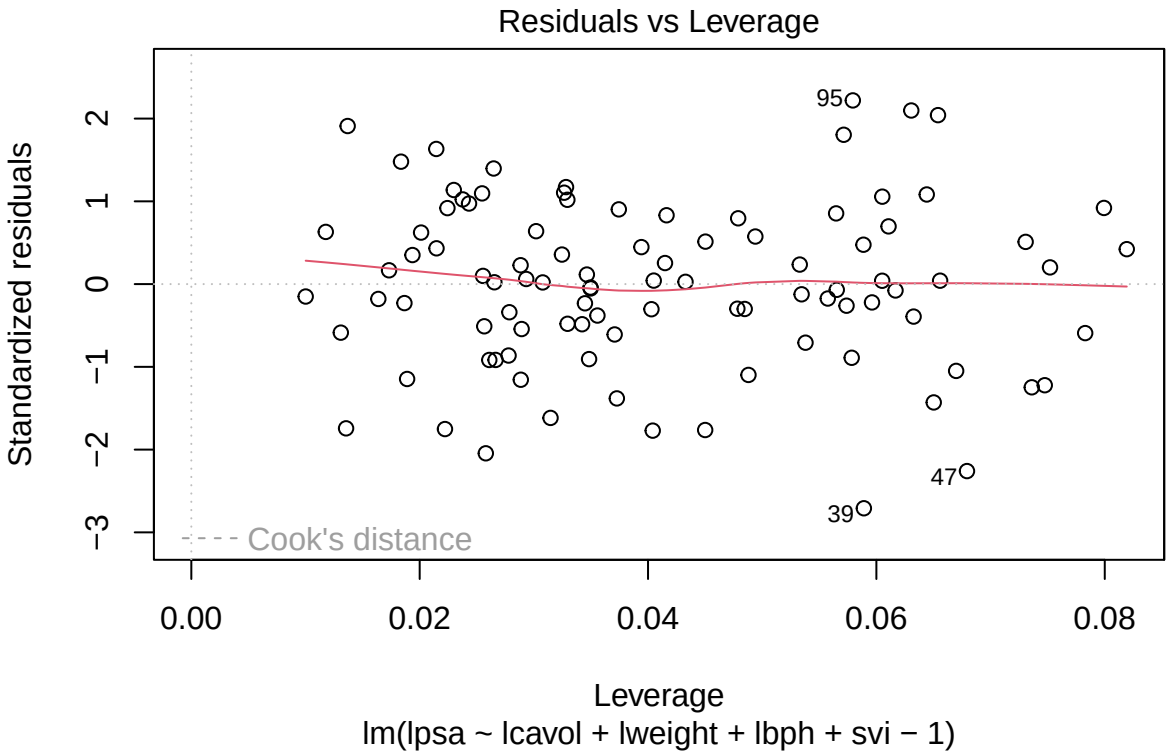
Corresponding to 95% confidence intervals for the parameters β_j.

Finally, note that all the above discussion regarding the inference of a linear model is based on the assumption that the residuals are normally distributed with mean 0 and constant variance σ^2. Hence it is important to check whether this assumption is satisfied or not. One way to do this is by using visual tools, which are readily available using the `plot()` function. For instance,

```
plot(prost_lm2)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(lpsa ~ lcavol + lweight + lbph + svi − 1)
Scale−Location
√|Standardized residuals|
Fitted values
lm(lpsa ~ lcavol + lweight + lbph + svi − 1)
39
47
95



The first plot (*Residuals vs Fitted*) helps us to assess the condition of mean zero of the residuals. Ideally, we should observe approximately a straight line around 0 (as above). The second plot (*Normal Q-Q*) deals with the normality of the residuals, and we should preferably observe points close to the identity line. The third plot (*Scale-Location*) assists us in identifying if the assumption of constant variance holds. We look for a red line close to a horizontal line (no matter the location). Finally, the fourth plot (*Residuals vs Leverage*) helps us to identify data points that are "extreme" and affect considerable the regression model. This is the case (not observed in the example above) when having points outside the dotted red lines. An in-depth analysis of those points should take place in such a situation.

Chapter 3

R for Finance

3.1 Market portfolio and CAPM

Let us imagine that we have available a certain amount of money and want to invest it by buying different stocks available in the market. A combination of stocks that we can buy is what is called a *portfolio*. The main problem that we want to address here is determining the portfolio, or equivalently, the ratio of stocks that we should buy. Ideally, we would like to pick a portfolio that will return a high profit with low risk. Modern portfolio theory gives us some solutions to this problem. The main idea of modern portfolio theory is to assume that the price movements (or returns) of stocks (or risky assets) are random variables. Then, we can define the *risk* of a particular stock as how much the individual return of that stock deviates from its mean return. This is usually quantified by the standard deviation, which is often called *volatility* in finance. Typically, a stock that might give high returns have high volatility, and stocks with low volatility often make low returns. Hence, the problem we want to address is calculating the portfolio that generates a higher profit with less volatility than individual stocks.

3.1.1 Mean-variance portfolio

We start this section by giving a simple example. Consider two stocks, A and B, with corresponding returns R_A and R_B following the joint probability mass function described in Table 3.1 below.

Now, suppose that we want to construct a portfolio with a weight w_A on stock A and w_B on stock B, with $w_A + w_B = 1$. Then, the return of the portfolio is

$$R_p = w_A R_A + w_B R_B \,,$$

which is also a random variable. Then, the expected value and variance of R_p are given by

$$\mathbb{E}[R_p] = w_A \mathbb{E}[R_A] + w_B \mathbb{E}[R_B] \,,$$

and

$$Var(R_p) = w_A^2 Var(R_A) + w_B^2 Var(R_B) + 2 w_A w_B Cov(R_A, R_B) \,,$$

respectively. Consider now $w_A = 0.6$. Then we can compute the expected value and variance of the portfolio as follows. First, we compute the expected value and variance of A and B and the covariance of A and B:

Table 3.1: Joint density for R_A and R_B

State	R_A	R_B	Probability
Depression	-20%	5%	0.25
Recession	10%	20%	0.25
Normal	30%	-12%	0.25
Boom	50%	9%	0.25

```
rA <- c(-0.2, 0.1, 0.3, 0.5)
rB <- c(0.05, 0.2, -0.12, 0.09)
prob <- rep(0.25, 4)

expA <- sum(rA * prob)
expA
```

```
## [1] 0.175
```

```
expB <- sum(rB * prob)
expB
```

```
## [1] 0.055
```

```
varA <- sum(rA^2 * prob) - expA^2
varA
```

```
## [1] 0.066875
```

```
varB <- sum(rB^2 * prob) - expB^2
varB
```

```
## [1] 0.013225
```

```
covAB <- sum(rA * rB * prob) - expA * expB
covAB
```

```
## [1] -0.004875
```

With the above quantities at hand, we can now compute the expected return and variance of our portfolio as follows:

```
w <- c(0.6, 0.4)

exp_por <- sum(w * c(expA, expB))
exp_por
```

```
## [1] 0.127
```

```
var_por <- varA * w[1]^2 + varB * w[2]^2 + 2 * covAB * w[1] * w[2]
var_por
```

```
## [1] 0.023851
```

Moreover, the standard deviation is given by

```
sqrt(var_por)
```

```
## [1] 0.1544377
```

On the other hand, note that the weighted average of the standard deviations of R_A and R_B is

```
w[1] * sqrt(varA) + w[2] * sqrt(varB)
```

```
## [1] 0.2011612
```

which is greater than the standard deviation of the portfolio. This is known as the *diversification effect*.

Diversification Effect: The standard deviation of the portfolio is less than or equal to the weighted average of the standard deviations of the individual securities. Equality holds only if the correlation of the assets is 1.

So far, we have treated w_A and w_B as fixed constants. However, if w_A and w_B change in a way such that $w_A + w_B = 1$, then $\mathbb{E}[R_p]$ and $SD(R_p)$ also change. This leads to the following definition of an *opportunity set*.

Definition 3.1 (Opportunity set)**.** The possible pairs of $\mathbb{E}[R_p]$ and $SD(R_p)$ that can be formed by varying w_A (and hence w_B) is called the *opportunity set* or *feasible set*.

Remark. Note that in the definition above, we allow $w_A < 0$ (this means we can borrow A) and $w_A > 1$ (this means w_B can be negative, and hence we can borrow B). However, it is common practice to restrict w_A to $[0, 1]$.

We now want to depict the shape of the opportunity set for stocks A and B. For computational purposes, it is convenient to write the expressions of $\mathbb{E}[R_p]$ and $Var(R_p)$ in terms of matrices operations (remember, R is quite efficient doing these types of operations). Thus, we first need to introduce further notation. Let $\mathbf{R} = (R_A, R_B)$ and $\mathbf{w} = (w_A, w_B)$, then

$$\mathbb{E}[R_p] = \mathbb{E}[\mathbf{R}]\mathbf{w}^\top ,$$

and

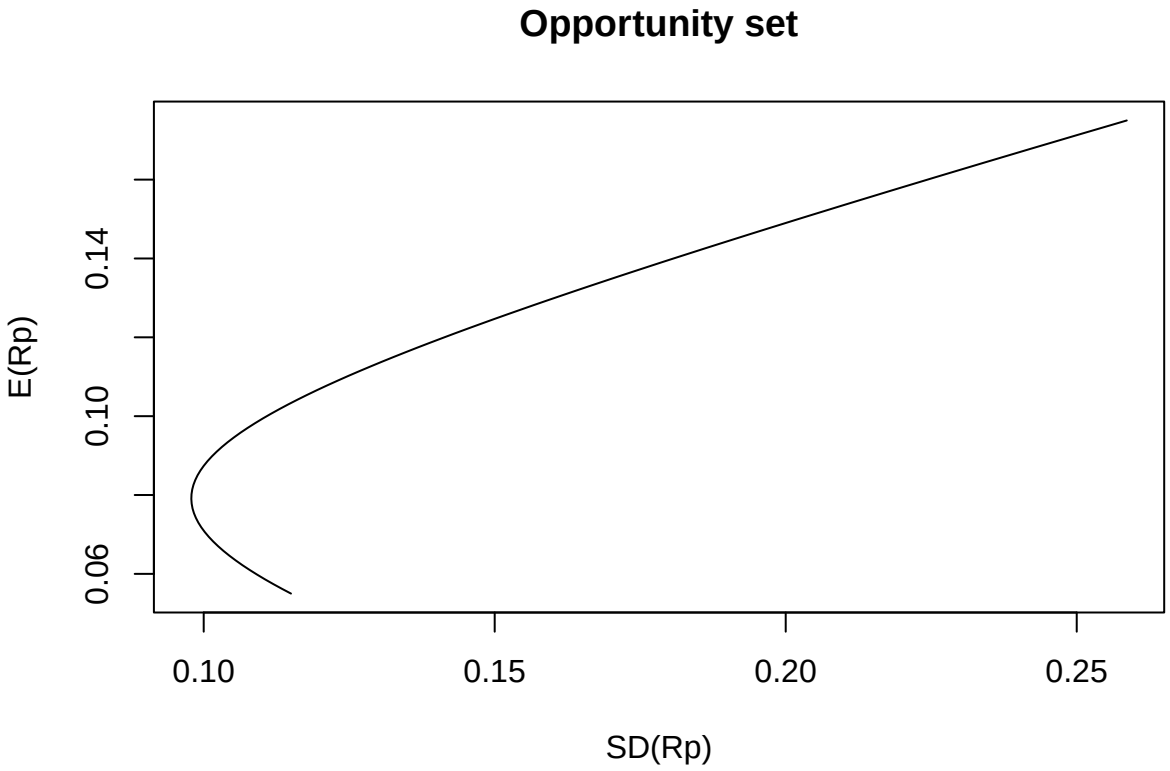
$$Var(R_p) = \mathbf{w} Var(\mathbf{R})\mathbf{w}^\top ,$$

where $\mathbb{E}[\mathbf{R}]$ is the mean vector of $\mathbf{R}$, i.e., $\mathbb{E}[\mathbf{R}] = (\mathbb{E}[R_A], \mathbb{E}[R_B])$, and $Var(\mathbf{R})$ is the covariance matrix of $\mathbf{R}$, that is,

$$Var(\mathbf{R}) = \begin{pmatrix} Var(R_A) & Cov(R_A, R_B) \\ Cov(R_A, R_B) & Var(R_B) \end{pmatrix} .$$

The following code plots the opportunity set for stocks A and B described in Table 3.1.

```
wa <- seq(0, 1, by = 0.01)
w <- cbind(wa, 1 - wa)

exp_p <- w %*% c(expA, expB)

sigma <- matrix(c(varA, covAB, covAB, varB), ncol = 2)

sd_portfolio <- function(sigma, w) {
  sg <- rep(0, nrow(w))
  for (i in 1:nrow(w)) {
    sg[i] <- sqrt(t(w[i, ]) %*% sigma %*% (w[i, ]))
  }
  sg
}

sd_p <- sd_portfolio(sigma, w)
plot(sd_p, exp_p,
  type = "l",
  main = "Opportunity set",
  xlab = "SD(Rp)",
  ylab = "E(Rp)"
)
```



In the picture above, we can observe that there is a point on the opportunity set such that $SD(R_p)$ is a minimum. This point corresponds to a portfolio that can be formed from mixing the assets and is known as the **minimum variance portfolio** (MV). In R, we can find the weights of such a portfolio as follows:

```
wmin <- w[match(min(sd_p), sd_p), ]
wmin
```

```
##  wa
## 0.2 0.8
```

The corresponding expected return and standard deviation of the minimum variance portfolio are then

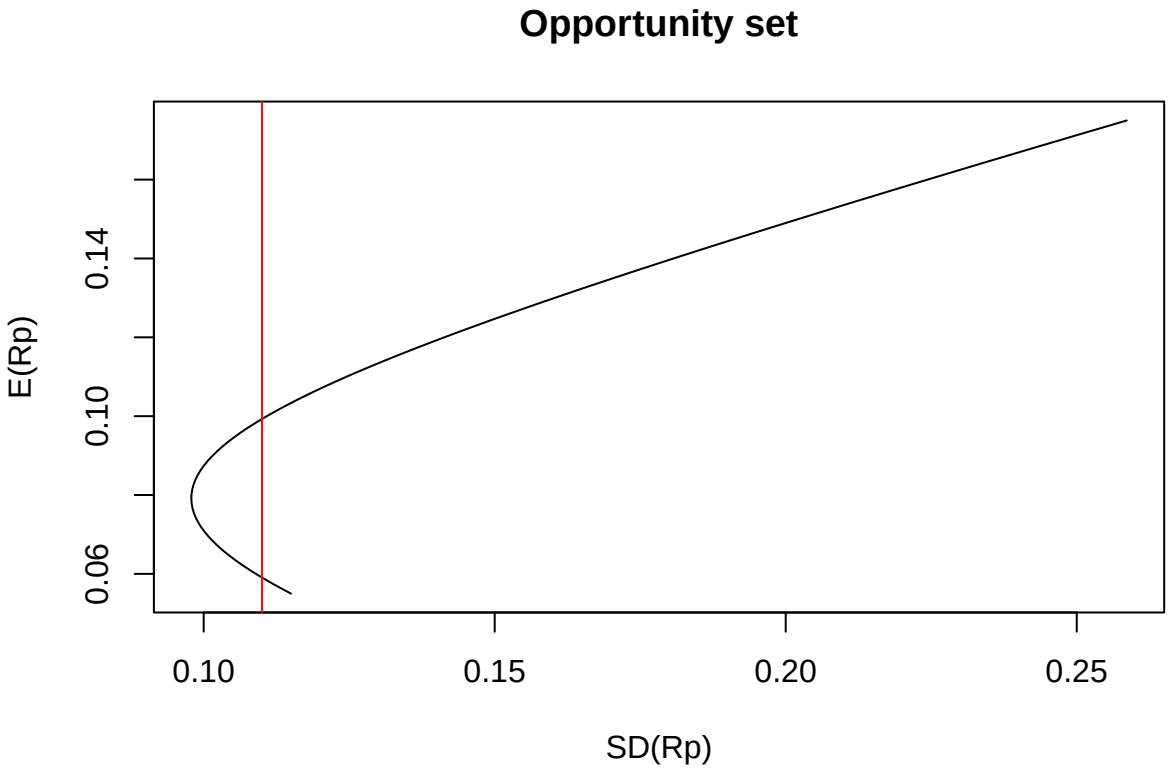
```
exp_mv <- wmin %*% c(expA, expB)
exp_mv
```

```
##       [,1]
## [1,] 0.079
```

```
sd_mv <- sd_portfolio(sigma, matrix(wmin, 1))
sd_mv
```

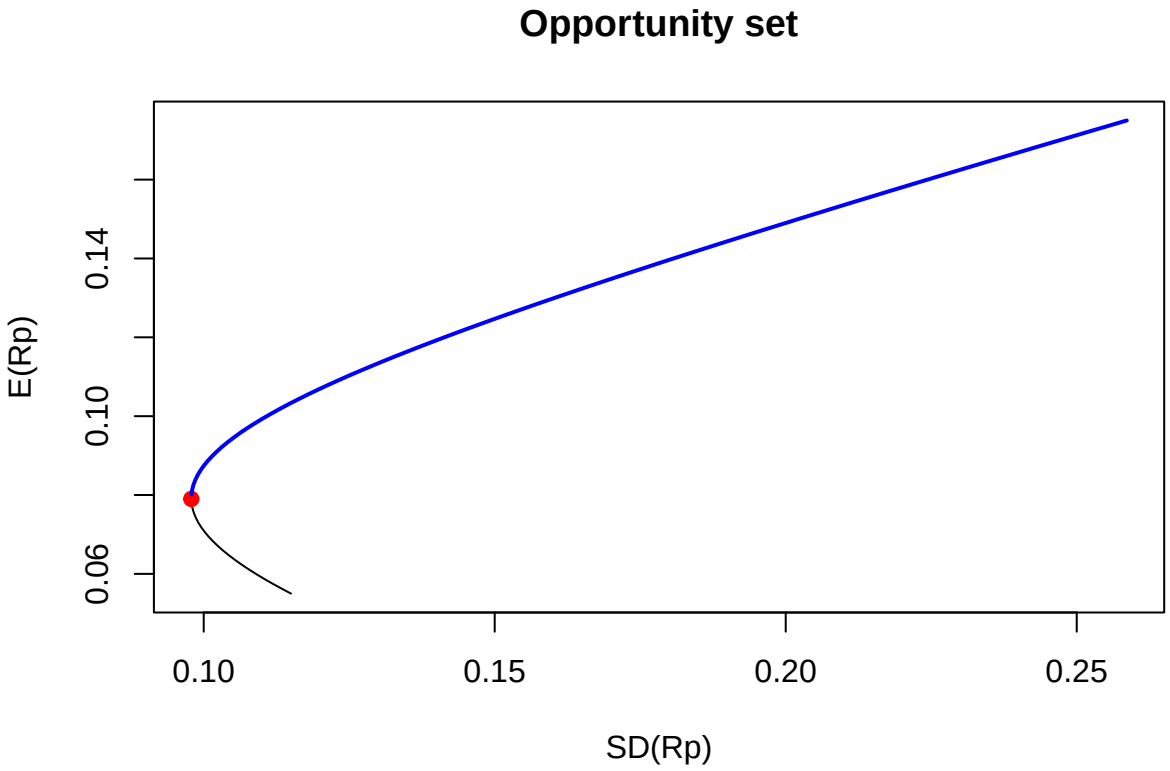
```
## [1] 0.09787237
```

Suppose now that we want to create a portfolio by mixing assets such that the standard deviation of the resulting portfolio is greater than the minimum standard deviation that can be achieved. For instance, the figure below shows that we have two options to pick a portfolio with a standard deviation of 0.11.



Among these two portfolios, for sure, we would like to pick the one with a higher $\mathbb{E}[R_p]$, meaning the portfolio above the point MV. More generally, we can conclude that the part of the opportunity set below the point MV would never be selected. The opportunity set above the point MV is called the **efficient frontier**. The following plot shows the MV point and the efficient frontier for our example.

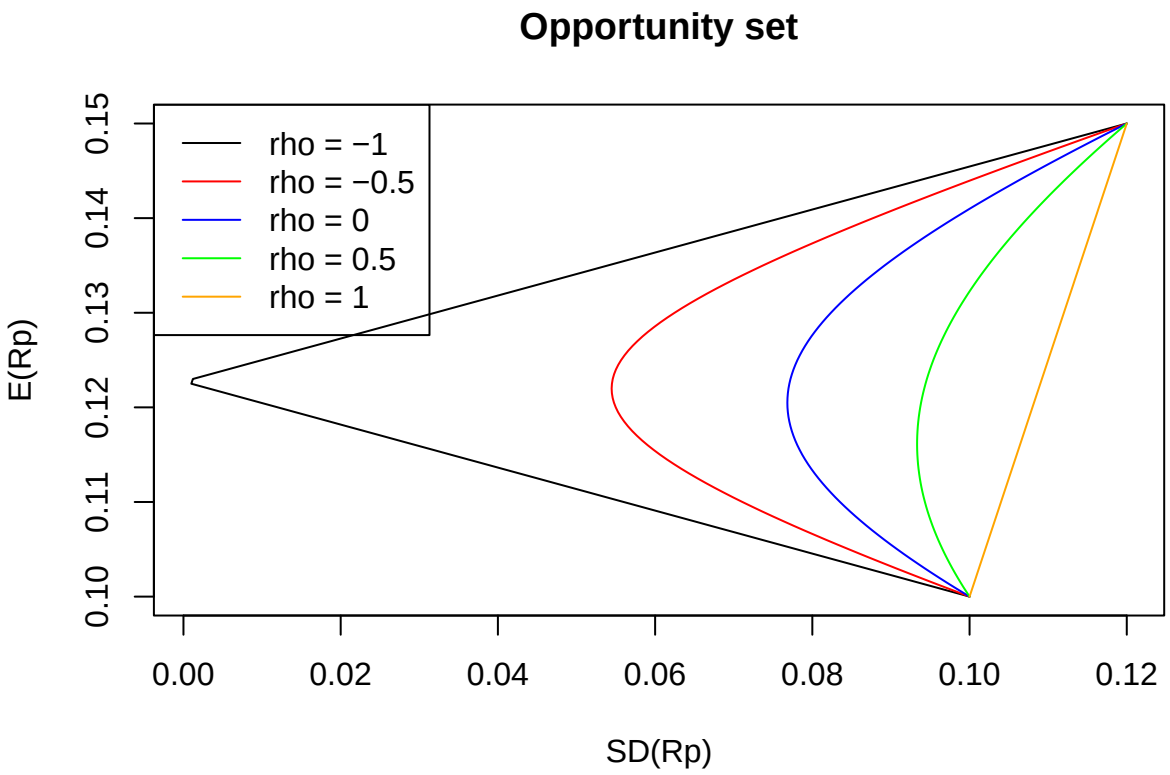
```
# Efficient frontier
exp_ef <- exp_p[exp_p > as.vector(exp_mv)]
sd_ef <- sd_p[exp_p > as.vector(exp_mv)]

plot(sd_p, exp_p,
  type = "l",
  main = "Opportunity set",
  xlab = "SD(Rp)",
  ylab = "E(Rp)"
)
points(sd_mv, exp_mv, col = "red", pch = 19) # MV
lines(sd_ef, exp_ef, col = "blue", lwd = 2) # Efficient frontier
```



Next, we want to illustrate how the shape of the opportunity changes according to the correlation of the risky assets. For that purpose, let us consider two stocks, A and B, with the following characteristic: $\mathbb{E}[R_A] = 10\%$, $\mathbb{E}[R_B] = 15\%$, $SD(R_A) = 10\%$, and $SD(R_B) = 12\%$. We also assume five different possibilities for the correlation between A and B: -1, -0.5, 0, 0.5, and 1.

```
wa <- seq(0, 1, by = 0.01)
w <- cbind(wa, 1 - wa)

exp_p <- w %*% c(0.1, 0.15)

sd_A <- 0.1
sd_B <- 0.12

sigma1 <- matrix(c(sd_A^2, -1 * sd_A * sd_B, -1 * sd_A * sd_B, sd_B^2), 2)
sigma2 <- matrix(c(sd_A^2, -0.5 * sd_A * sd_B, -0.5 * sd_A * sd_B, sd_B^2), 2)
sigma3 <- matrix(c(sd_A^2, 0 * sd_A * sd_B, 0 * sd_A * sd_B, sd_B^2), 2)
sigma4 <- matrix(c(sd_A^2, 0.5 * sd_A * sd_B, 0.5 * sd_A * sd_B, sd_B^2), 2)
sigma5 <- matrix(c(sd_A^2, 1 * sd_A * sd_B, 1 * sd_A * sd_B, sd_B^2), 2)

sd_p1 <- sd_portfolio(sigma1, w)
sd_p2 <- sd_portfolio(sigma2, w)
sd_p3 <- sd_portfolio(sigma3, w)
sd_p4 <- sd_portfolio(sigma4, w)
sd_p5 <- sd_portfolio(sigma5, w)

plot(sd_p1, exp_p,
  type = "l",
  main = "Opportunity set",
  xlab = "SD(Rp)",
  ylab = "E(Rp)"
)
lines(sd_p2, exp_p, col = "red")
lines(sd_p3, exp_p, col = "blue")
lines(sd_p4, exp_p, col = "green")
```

```
lines(sd_p5, exp_p, col = "orange")
legend("topleft",
  leg = paste("rho =", c(-1, -0.5, 0, 0.5, 1)),
  lty = 1,
  col = c("black", "red", "blue", "green", "orange")
)
```



The straight line (orange) corresponds to $\rho = 1$, representing points that would have been generated if the two assets were perfectly positively correlated. When $\rho \neq 1$, we see that the resulting opportunity set is always to the left of the straight line, and hence the standard deviation of these portfolios is less than the ones of the case when $\rho = 1$ for the same values of $\mathbb{E}[R_p]$. This illustrates the diversification effect. Since the curve bends towards the left more significantly as ρ decreases, the impact of diversification increases with decreasing ρ. Furthermore, in the particular case of negative correlations, we can think of this effect as one asset working as a safety net (or hedge) for the plunge in the price of another asset.

Multiple risky assets

We now introduce the more general setting with n stocks in a portfolio. More specifically, suppose that we have n risky assets with random returns $R_1, R_2, \ldots, R_n$. Then, if we construct a portfolio with portfolio weights $w_1, w_2, \ldots, w_n$, where $w_1 + w_2 + \cdots + w_n = 1$, the portfolio return R_p is given by

$$R_p = w_1 R_1 + w_2 R_2 + \cdots + w_n R_n = \sum_{i=1}^{n} w_i R_i \,.$$

Thus, the mean and variance of R_p are then given by

$$\mathbb{E}[R_p] = \sum_{i=1}^{n} w_i \mathbb{E}[R_i] \,,$$

$$Var(R_p) = \sum_{i=1}^{n} w_i^2 Var(R_i) + 2 \sum_{i<j} w_i w_j Cov(R_i, R_j) \,.$$

By letting $\mathbf{R} = (R_1, R_2, \ldots, R_n)$ and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$, we can rewrite the expressions above in terms of matrices as

$$\mathbb{E}[R_p] = \mathbb{E}[\mathbf{R}]\mathbf{w}^\top,$$

and

$$Var(R_p) = \mathbf{w} Var(\mathbf{R}) \mathbf{w}^\top,$$

where $\mathbb{E}[\mathbf{R}]$ and $Var(\mathbf{R})$ are the mean vector and covariance matrix of $\mathbf{R}$, receptively.

The following example shows how to construct the opportunity set when dealing with three risky assets.

Example 3.1. Consider three stocks, A, B, and C, with expected returns $\mathbb{E}[R_A] = 0.105$, $\mathbb{E}[R_B] = 0.18$, and $\mathbb{E}[R_C] = 0.02$, respectively. The covariance matrix is the following:

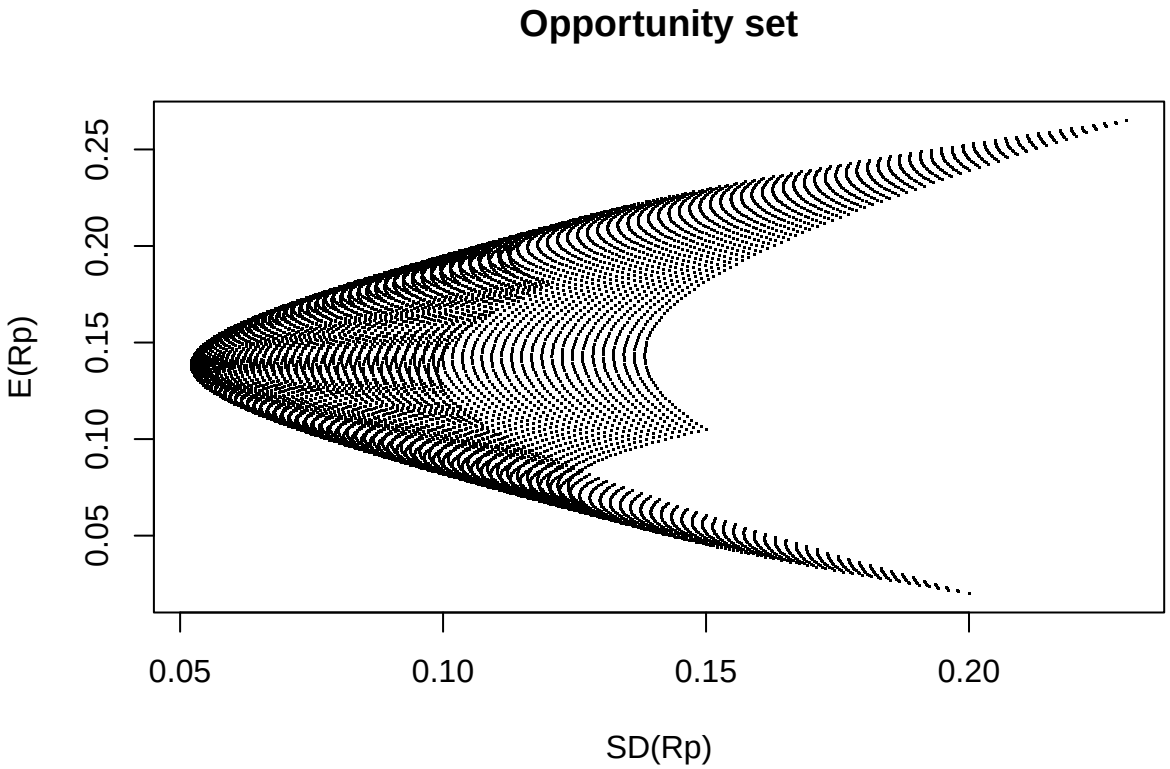
$$Var(\mathbf{R}) = \begin{pmatrix} 0.15^2 & -0.012 & 0.002 \\ -0.012 & 0.12^2 & -0.002 \\ 0.002 & -0.002 & 0.2^2 \end{pmatrix}.$$

Then, the following code plots the opportunity set available to any investor:

```
exp_r <- c(0.105, 0.18, 0.02)

sigma <- matrix(c(
  0.15^2, -0.012, 0.002,
  -0.012, 0.12^2, -0.002,
  0.002, -0.002, 0.2^2
),
ncol = 3, byrow = T
)
sigma
```

```
##         [,1]    [,2]   [,3]
## [1,]  0.0225 -0.0120  0.002
## [2,] -0.0120  0.0144 -0.002
## [3,]  0.0020 -0.0020  0.040
```

```
# Generates a grid of values for w1 and w2
w_grid <- expand.grid(
  wa = seq(0, 1, length.out = 100),
  wb = seq(0, 1, length.out = 100)
)
w_grid <- as.matrix(w_grid)

w <- cbind(w_grid, 1 - w_grid[, 1] - w_grid[, 2])

# Expected return
rp <- as.vector(w %*% exp_r)

sd_p <- sd_portfolio(sigma, w)
plot(sd_p, rp,
  pch = ".",
  main = "Opportunity set",
  xlab = "SD(Rp)",
  ylab = "E(Rp)"
)
```



Note that we have several portfolios that lead to the same expected return but with different volatilities. This leads us to the idea of the *minimum variance frontier*, which is the curve consisting of the smallest volatility for the same return. Thus, an investor will choose a portfolio corresponding to the upper-half side of the minimum variance frontier. Such an upper-half side of the minimum variance frontier is the *efficient frontier* described before.

To give an understanding of the effect of diversification in the case of n risky assets, we assume for the moment the following:

(a) All securities have the same variance, i.e., $Var(R_1) = \cdots = Var(R_n) = \text{var}$.
(b) The covariance for every pair of securities is the same, i.e., $Cov(R_i, R_j) = \text{cov}$ for all $i \neq j$.
(c) We have an equally weighted portfolio, i.e., $w_1 = \cdots = w_n = 1/n$.

Under the above assumptions, it is easy to see that

$$Var(R_p) = \frac{1}{n}(\text{var} - \text{cov}) + \text{cov}\,,$$

In particular, we have that $Var(R_p) \to \text{cov}$ as $n \to \infty$.

The essence of the equation above can be summarized into two main points:

(1) When n increases, $Var(R_p)$ decreases, and hence the diversification effect increases with n.
(2) Even if n tends to infinity, $Var(R_p)$ does not drop to zero. There is a limit to the diversification effect: A diversified portfolio can eliminate some, but not all, of the risk of the individual securities.

This leads us to the following definition of systematic and unsystematic risk.

Definition 3.2.

- **Systematic risk** (or market risk or non-diversifiable risk) is the portion of an asset's risk that cannot be eliminated through diversification. Interest rates, recessions, and catastrophes are examples of systematic risks.
- **Unsystematic risk** (or specific risk or diversifiable risk) is the portion of an asset's risk that can be eliminated by including the security as part of a well-diversified portfolio. It represents the component of a stock's return that is not correlated with general market moves.

3.1.2 Capital Asset Pricing Model (CAPM)

Previously we have been dealing with portfolios consisting of two or more risky assets. Let us now incorporate a risk-free security into our analysis. More specifically, let R_p be the return on a risky portfolio (a portfolio constituted of risky assets) and let R_f be the risk-free interest rate (i.e., the return of a risk-free asset). Next, consider a (complete) portfolio consisting of w_p parts of the risky portfolio and w_f parts of the risk-free asset, $w_p + w_f = 1$. Then, the return R of this portfolio is given by

$$R = w_p R_p + w_f R_f = w_p R_p + (1 - w_p) R_f \,.$$

Taking the expectation in the expression above, we have

$$\begin{aligned} \mathbb{E}[R] &= w_p \mathbb{E}[R_p] + (1 - w_p) R_f \\ &= w_p (\mathbb{E}[R_p] - R_f) + R_f \,. \end{aligned}$$

On the other hand, the variance of R is given by

$$Var(R) = w_p^2 Var(R_p) \,,$$

which implies that the standard deviation of R is

$$SD(R) = w_p SD(R_p) \,.$$

This last equation implies that

$$w_p = \frac{SD(R)}{SD(R_p)} \,.$$

Thus, we obtain the following expression for the opportunity set of this portfolio

$$\mathbb{E}[R] = R_f + \frac{(\mathbb{E}[R_p] - R_f)}{SD(R_p)} SD(R) \,,$$

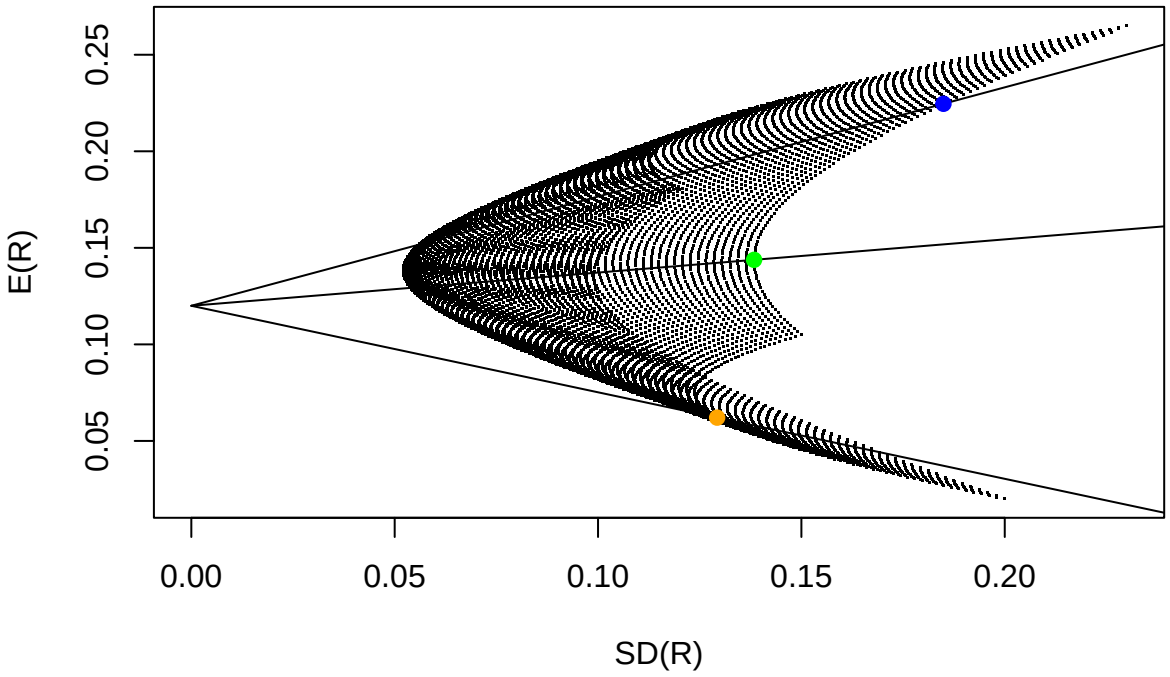
which corresponds to a straight line with intercept R_f and slope

$$\frac{(\mathbb{E}[R_p] - R_f)}{SD(R_p)} \,.$$

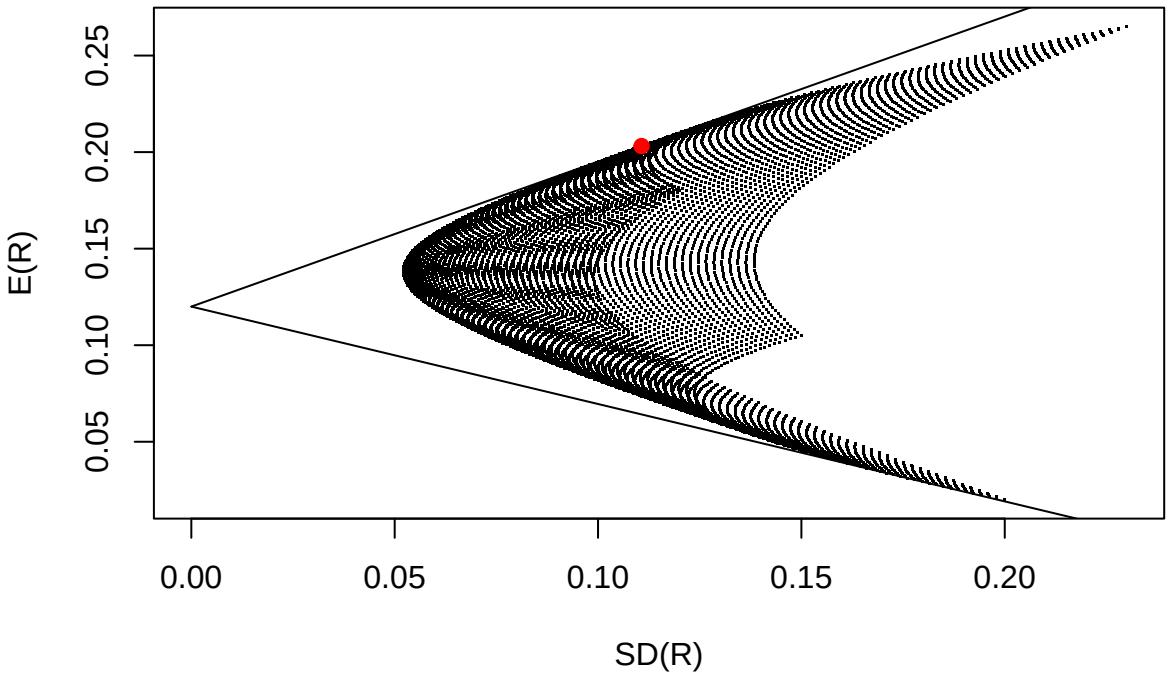
The latter is also known as the *Sharpe ratio*.

For example, let us return to Example 3.1 and take a couple of portfolios there to combine with the risk-free asset. If the risk-free interest rate is 12%, we obtain the following picture for the opportunity set of the (complete) portfolio

```
rf <- 0.12 # Risk-free rate
slope1 <- (rp[50] - rf) / sd_p[50]
slope2 <- (rp[2500] - rf) / sd_p[2500]
slope3 <- (rp[7500] - rf) / sd_p[7500]
x <- seq(0, 1, by = 0.01)
plot(sd_p, rp,
  pch = ".",
  xlab = "SD(R)",
  ylab = "E(R)",
  xlim = c(0, 0.23)
)
lines(x, slope1 * x + rf)
lines(x, slope2 * x + rf)
lines(x, slope3 * x + rf)
points(sd_p[50], rp[50], col = "orange", pch = 19)
points(sd_p[2500], rp[2500], col = "green", pch = 19)
points(sd_p[7500], rp[7500], col = "blue", pch = 19)
```



Since every risky portfolio above can be combined with the risk-free security, the opportunity set of market securities is the region between the two lines picture below:



The efficient frontier that is formed from all securities is the straight line that:

(1) passes through the risk-free rate, and
(2) is tangential to the efficient frontier formed solely from risky securities.

The efficient frontier described above is known as the **capital market line** (CML).

Every rational investor would pick a portfolio on the CML. Of course, the exact point they will pick depends on the investor's risk preference. However, notice that every point on the CML is formed by mixing the risk-free security with the portfolio market denoted by a red dot in the above picture, the portfolio with the highest Sharpe ratio. This portfolio is known as the **optimal risky portfolio** or **market portfolio**.

The CML and market portfolio in the figure above were generated using the following code:

```
rf <- 0.12 # Risk-free rate
amax <- max((rp - rf) / sd_p) # Slope of the CML
orp <- match(amax, (rp - rf) / sd_p) # Optimal risky portfolio
x <- seq(0, 1, by = 0.01)
plot(sd_p, rp,
  pch = ".",
  xlab = "SD(R)",
  ylab = "E(R)",
  xlim = c(0, 0.23)
)
lines(x, amax * x + rf)
points(sd_p[orp], rp[orp], col = "red", pch = 19)
```

Moreover, we can find the weights of the market portfolio easily by typing

```
w[orp, ]
```

```
##         wa         wb
##  0.4242424  0.9191919 -0.3434343
```

Note that in the above, we generated a series of values for the weights of the risky portfolio, then computed the mean and standard deviation, and extracted from this set of values the market portfolio. However, the weights of the market portfolio can be computed algebraically by using the following result:

Proposition 3.1. *Suppose that we have n risky assets and that the market portfolio R_M has weights w_i^M, $i = 1, \ldots, n$. Then, for some constant c,*

$$\frac{\mathbb{E}[R_i] - R_f}{Cov(R_i, R_M)} = c\,, \quad i = 1, \ldots, n\,,$$

with $Cov(R_i, R_M) = \sum_{k=1}^{n} w_k^M Cov(R_i, R_k)$.

The result suggests that to derive the weights of the market portfolio, we can proceed as follows:

(1) Find "weights" (they do not need to sum to 1) that make $\mathbb{E}[R_i] - R_f = Cov(R_i, R_M)$ hold for all $i = 1, \ldots, n$.
(2) Rescale the weights to sum to 1.

Note that this requires us to solve a system of linear equations, which can be easily done in R by using `solve()`. For our particular example, we have

```
unscaled_w <- solve(sigma, exp_r - rf)
scaled_w <- unscaled_w / sum(unscaled_w)
scaled_w
```

```
## [1]  0.4234973  0.9215102 -0.3450075
```

Earlier in this section, we pointed out that the risk (or standard deviation) of a stock can be broken down into systematic and unsystematic risks. More specifically, we saw that the unsystematic risk can be diversified away in a large portfolio, but the systematic risk cannot. Thus, a diversified investor holding the market portfolio must worry about the systematic risk, but not the unsystematic risk, of every security in a portfolio. We now introduce the concept of *beta*, which gives a way to measure the systematic risk of a security. It turns out that beta is the "best" measure of the risk of an individual security from a diversified investor's point of view.

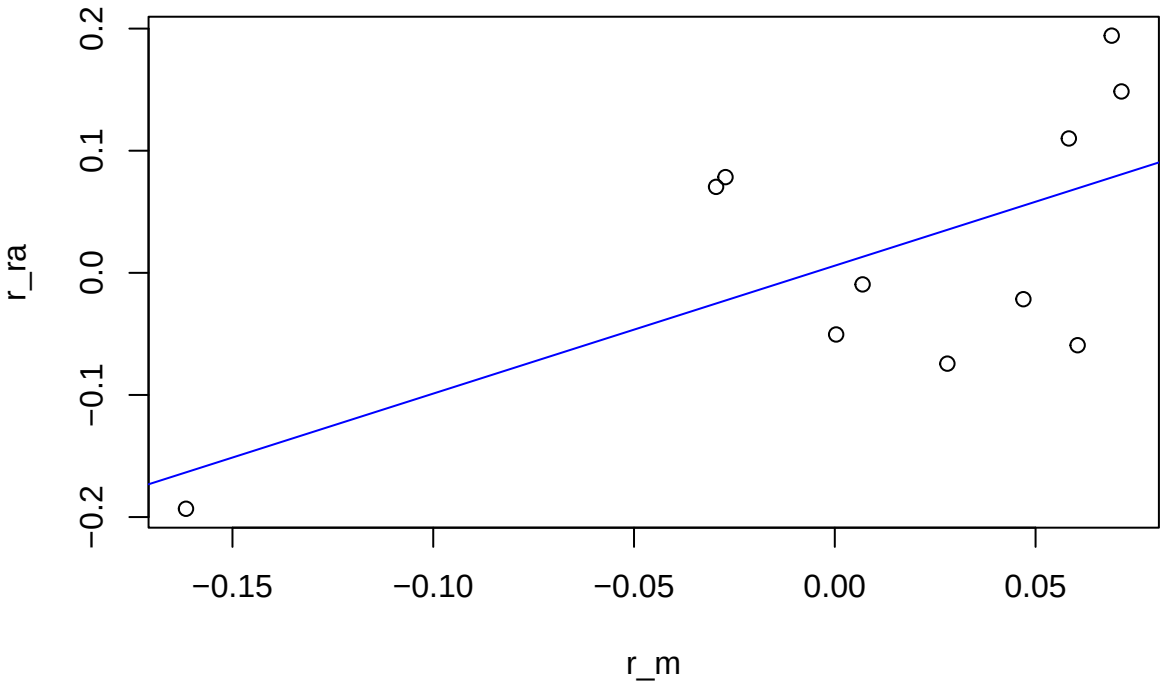
The concept of beta can be better explained with a simple example. Let us assume that the following vectors represent the historical returns of the market portfolio (`r_m`) and a risky asset (`r_ra`)

```
r_m <- c(
  0.000341, 0.068962, 0.046964, 0.006922,
  -0.029561, 0.028035, -0.027218, -0.161576,
  0.060479, 0.071397, 0.058284
)
```

```
r_ra <- c(
  -0.050484, 0.194222, -0.021584, -0.009475,
  0.070406, -0.074354, 0.078366, -0.193192,
  -0.059271, 0.148521, 0.11004
)
```

We now plot a graph of (r_m, r_ra) and draw the regression line that best fits the data

```
dat_beta <- data.frame(r_m = r_m, r_ra = r_ra)
reg_line <- lm(r_ra ~ r_m, data = dat_beta)
plot(dat_beta)
abline(reg_line, col = "blue")
```



The coefficients of this regression line are

```
reg_line
```

```
##
## Call:
## lm(formula = r_ra ~ r_m, data = dat_beta)
##
## Coefficients:
## (Intercept)          r_m
##    0.005849     1.047370
```

The slope of the regression is what is called the **beta** of the risky assets. The natural interpretation of beta is that the returns of the risky asset are magnified 1.047370 times over those of the market.

The actual definition of beta is:

$$\beta_i = \frac{Cov(R_i, R_M)}{\sigma^2(R_M)},$$

where $Cov(R_i, R_M)$ is the covariance between the return on the ith security and the return on the market portfolio, and $\sigma^2(R_M)$ is the variance of the market.

In full generality, the beta of a security can be interpreted as follows: For a security with a beta greater than 1, its movement is an amplification of the movement of the market (as represented by the market portfolio). For a security with a beta between 0 and 1, its movements tend to be in the same direction as the market, but to a less extent. For negative values of beta, the security is expected to do "against" the market, meaning that it is expected to do well when the market does poorly and vice versa.

More generally, for any portfolio of n risky assets, its beta β_p can be computed as a weighted average of individual asset betas, that is,

$$\beta_p = \frac{Cov(R_p, R_M)}{\sigma^2(R_M)} = \sum_{i=1}^{n} w_i \beta_i \,.$$

From the definition, is clear that the beta of the market portfolio is 1.

We finish this section with the most important application of beta: the capital asset pricing model (CAPM).

The CAPM states that the expected return on a security with beta β is

$$\mathbb{E}[R] = R_f + \beta(\overline{R}_M - R_f) \,,$$

where R_f is the risk-free rate, and $\overline{R}_M$ is the expected market return.

Keeping R_f and R_M fixed, we can plot a graph of $\mathbb{E}[R]$ versus β. This line is known as the **security market line** (SML).

3.2 The binomial model

This section introduces the binomial tree model and uses it to compute arbitrage-free prices for European-style options. Although the mathematical description of this model involves only simple algebra, it is a powerful tool to understand arbitrage pricing theory. The main idea of the binomial model is to break the time to maturity of an option into periods. Then, in each period, and given the underlying asset's price at the beginning of the period, it assumes that the stock price will change to one of two possible values at the end of the period. Thus, we can then determine the value of the option recursively by starting at the maturity date, evaluating the option's value under each possibility for the final prices of the stock, and then moving backward through the tree.

3.2.1 One-period binomial model

We start with the one-period version of the model. The model assumes that we have two assets: a risk-free asset (e.g., a treasury bond) and a stock. Here, we denote by B_t and S_t the bond and stock prices at time t, respectively. Typically, time $t = 0$ represents the present time, and $t = 1$ denotes some future time.

Let us begin by describing the behavior of the stock prices. If at time $t = 0$ the price is S_0, then the stock price at time $t = 1$, S_1, is given by the following random variable

$$S_1 = \begin{cases} uS_0 \,, & \text{with probability } p_u \,, \\ dS_0 & \text{with probability } p_d \,, \end{cases}$$

where u, d, p_u, and p_d are positive positive constants satisfying $d < u$, and $p_u + p_d = 1$. Figure 3.1 shows the development of the price of a stock under the above specification.

It is often convenient to write instead

$$S_1 = S_0 Z \,,$$

where Z is a random variable defined as

$$Z = \begin{cases} u \,, & \text{with probability } p_u \,, \\ d \,, & \text{with probability } p_d \,. \end{cases}$$

On the other hand, the bond price is deterministic and given by

$$\begin{aligned} B_0 &= 1 \,, \\ B_1 &= (1 + R) \,, \end{aligned}$$

where R is the (risk-free) spot rate for the period.

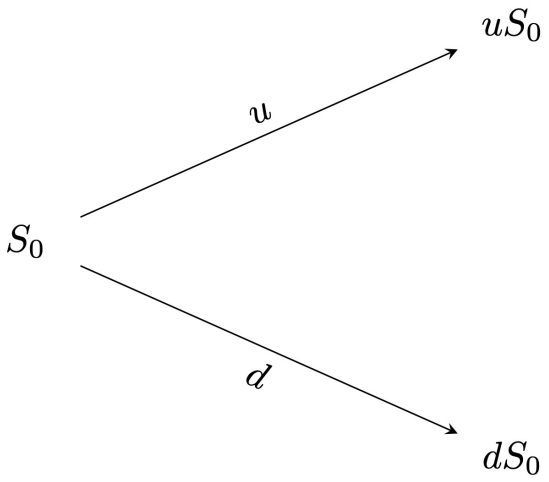


Figure 3.1: Stock prices in a one-period binomial tree.

Portfolios and arbitrage

We will consider portfolios constituted of risky assets S and bonds B. More specifically, we will denote by x the number of bonds we hold in our portfolio, y the number of stock units held, and let $h = (x, y)$. Consider now a fixed portfolio $h = (x, y)$. Then, the value of the portfolio at time t, V_t^h, is given by

$$V_t^h = xB_t + yS_t \,.$$

Note that this portfolio has a deterministic value at $t = 0$ and a stochastic value at $t = 1$.

We now introduce one of the central concepts of the theory in this section: arbitrage portfolio.

Definition 3.3. An **arbitrage** portfolio is a portfolio h with the properties

$$\begin{aligned} &V_0^h = 0\,, \\ &V_1^h > 0 \quad \text{with probability } 1\,. \end{aligned}$$

Essentially an arbitrage portfolio is a deterministic money-making machine. Hence, we can interpret the existence of an arbitrage portfolio as a severe case of mispricing on the market. Therefore, it is natural to investigate when a given market model is arbitrage-free.

It turns out that our binomial model above is free of arbitrage if and only if the following conditions hold:

$$d \leq (1 + R) \leq u\,.$$

Moreover, the above condition is equivalent to saying that $1 + R$ is a convex combination of u and d, i.e.,

$$1 + R = uq_u + dq_d\,.$$

where $q_u, q_d \geq 0$ and $q_u + q_d = 1$. In particular, the weights q_u and q_d can be interpreted as probabilities for a "new probability measure Q" with the property $Q(Z = u) = q_u$, $Q(Z = d) = q_d$. Denoting expectation w.r.t. this measure by $\mathbb{E}^Q$, it is easy to see that

$$S_0 = \frac{1}{1+R}\mathbb{E}^Q[S_1]\,.$$

Q is known as the "**risk-neutral measure**". Furthermore, q_u and q_d are explicit and given by

$$\begin{aligned} q_u &= \frac{(1+R)-d}{u-d}\,, \\ q_d &= \frac{u-(1+R)}{u-d}\,, \end{aligned}$$

These are known as the **risk-neutral probabilities** and will play an essential role in option pricing.

The above motivates the concept of **risk-neutral valuation**: The price of an asset is the expectation at time 1, calculated using the risk-neutral probabilities, and then discounted using the risk-free rate of interest.

Risk-neutral pricing

Definition 3.4. A **contingent claim** (financial derivative) is any random variable X of the form $X = \Phi(S_1)$, where Φ is some given real valued function known as the *contract function*.

The interpretation is that the contract holder receives the stochastic amount X at time $t = 1$. Two important examples are the European call and put options. For an European call option with strike K, we have that the $X = \max(S_1 - K, 0)$, while for the European put option with same strike K, the claim is $X = \max(K - S_1, 0)$. In what follows, we will denote by $\Phi(u)$ and $\Phi(d)$, the evaluations $\Phi(uS_0)$ and $\Phi(dS_0)$, respectively.

Our main problem is now determining the "fair" price for a given contingent claim X. If we denote the price of X at time t by $\Pi(t; X)$, then, in order to avoid arbitrage, we must have

$$\Pi(1; X) = X\,.$$

However, the hard part of the problem is determining $\Pi(0; X)$. The way to solve this problem is to find a portfolio h such that

$$V_1^h = X\,.$$

We call this a **hedging** portfolio or a **replicating** portfolio. Then, any price at $t = 0$ of the claim X, different to V_0^h (the price of the replicating portfolio), will lead to an arbitrage possibility. In other words, the price at $t = 0$ of the claim must be given by

$$\Pi(0; X) = V_0^h\,.$$

Let us now find the replicating portfolio $h = (x, y)$ for a fix and arbitrary contingent claim X with contract function Φ. This portfolio should satisfy that

$$\begin{aligned} V_1^h &= \Phi(u) \quad \text{if} \quad Z = u\,, \\ V_1^h &= \Phi(d) \quad \text{if} \quad Z = d\,. \end{aligned}$$

Substituting the expression for the value of the portfolio, we obtain the following system of equations:

$$\begin{aligned} x(1+R) + yuS_0 &= \Phi(u)\,, \\ x(1+R) + ydS_0 &= \Phi(d)\,. \end{aligned}$$

Solving for x and y in the above system gives

$$\begin{aligned} x &= \frac{1}{1+R}\frac{u\Phi(d) - d\Phi(u)}{u-d}\,, \\ y &= \frac{1}{S_0}\frac{\Phi(u) - \Phi(d)}{u-d}\,. \end{aligned}$$

Then, the price at $t = 0$ of the claim X, $\Pi(0; X)$, is given by

$$\begin{aligned} \Pi(0; X) &= x + yS_0 \\ &= \frac{1}{1+R}\left(\frac{(1+R)-d}{u-d}\Phi(u) + \frac{u-(1+R)}{u-d}\Phi(d)\right)\,. \end{aligned}$$

Here, we recognize the risk-neutral probabilities q_u and q_d, and we can rewrite the pricing formula above as

$$\Pi(0; X) = \frac{1}{1+R}\left(\Phi(u)q_u + \Phi(d)q_d\right)\,.$$

Thus, the right-hand side can now be interpreted as an expected value under the risk-neutral probability measure Q. More specifically, we have that

$$\Pi(0; X) = \frac{1}{1+R}\mathbb{E}^Q[X]\,.$$

Example 3.2. Consider a stock currently with current price $S_0 = 50$. The Stock's price is expected to increase to 60 or decrease to 40 during the next year. The risk-free interest rate is 5% compounded annually. Compute the price of a 1-year European call option with a strike price of 55.

Solution.

```
s0 <- 50
su <- 60
sd <- 40
rf <- 0.05
strike <- 55
u <- su / s0
d <- sd / s0

# Note that d <= (1 + R) <= u is satisfied
u
```

```
## [1] 1.2
```

```
d
```

```
## [1] 0.8
```

```
# Risk-neutral probabilities
qu <- (1 + rf - d) / (u - d)
qu
```

```
## [1] 0.625
```

```
qd <- 1 - qu
qd
```

```
## [1] 0.375
```

```
phiu <- max(su - strike, 0)
phiu
```

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

```
phid <- max(sd - strike, 0)
phid
```

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

```
# Derivative price
price <- (phiu * qu + phid * qd) / (1 + rf)
price
```

```
## [1] 2.97619
```

3.2.2 Multiperiod binomial model

We now extend the one-period binomial model to multiperiod. To do so, we now let the time index t run from $t = 0$ to $t = T$, where T is fixed. Here, T will denote the number of periods. As previously, we have a bond with price B_t and a stock with price S_t at time t.

More specifically, the bond prices are deterministic and given by

$$\begin{aligned} &B_0 = 1\,, \\ &B_{n+1} = (1+R)B_n\,, \quad n = 0, \dots, T-1 \end{aligned}$$

On the other hand, if at time $t = 0$ the price is of the stock is S_0, then the future stock prices are random and given by

$$S_{n+1} = Z_n S_n\,, \quad n = 0, \dots, T-1$$

where $Z_0, ..., Z_{T-1}$ are iid random variables, taking only the two values u and d with probabilities

$$\mathbb{P}(Z_n = u) = p_u\,, \quad \mathbb{P}(Z_n = d) = p_d\,.$$

In other words, during each time step, the stock price either moves up to u times its initial value or moves down to d times its initial value. Figure 3.2 shows the behavior of the stock price for $T = 2$.

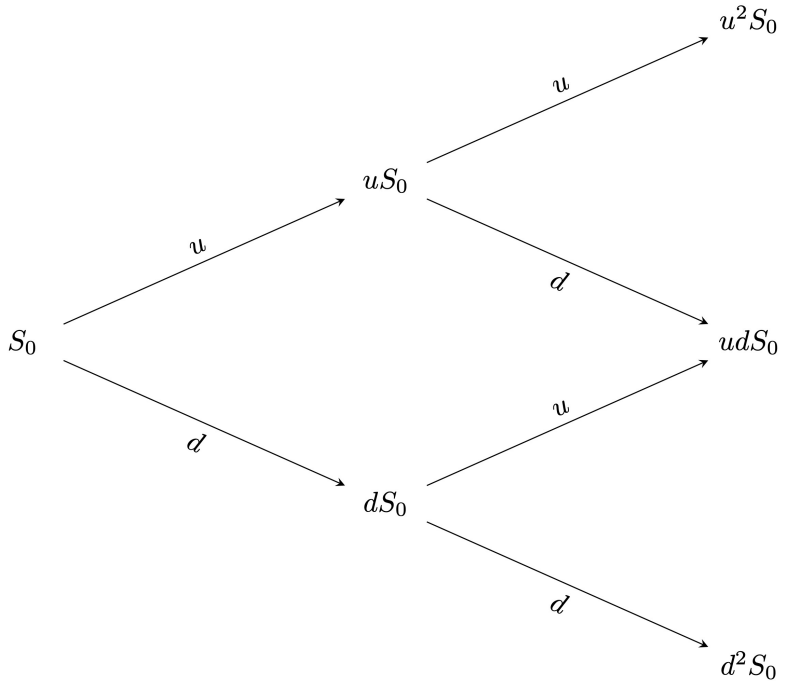


Figure 3.2: Multiperiod binomial tree.

Note that the prices of the stock at time t can be written as

$$S_t = u^k d^{t-k} S_0\,, \quad k = 0, \ldots, t\,,$$

where k denotes the number of up-moves that have occurred. Thus each node in the binomial tree can be represented by a pair (t, k) with $k = 0, \ldots, t$. Figure 3.3 shows the representation of of the nodes of a binomial tree for $T = 2$.

Recall that the main aim is to find the arbitrage-free price of a given contingent claim (financial derivative). In particular, we will only work with contingent claims of the form

$$X = \Phi(S_T)\,,$$

that is, claims whose value only depends on the stock price at the final time T, S_T. This type of contingent claims is typically called *simple*. Note that the European Call and Put options are examples of simple claims.

As in the one-period model, the problem is solved by finding a portfolio h that replicates the final value of the contingent claim, that is, if V_t^h denotes the value of the portfolio at time t, we require that

$$V_T^h = X\,.$$

Then, the price of the derivative $\Pi(0; X)$ at time $t = 0$ must be the price of the portfolio V_0^h to avoid arbitrage opportunities. More specifically,

$$\Pi(0; X) = V_0^h\,.$$

It turns out that to find the price of this portfolio (and hence of the claim), we only need to repeatedly and recursively apply the principles of the one-step binomial model. We illustrate this in a binomial

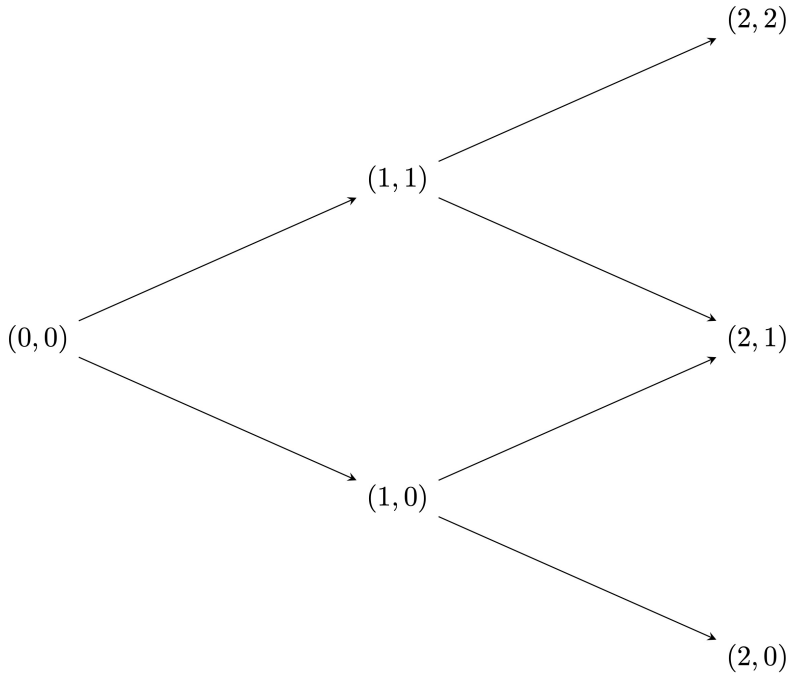


Figure 3.3: Nodes of a multiperiod binomial tree.

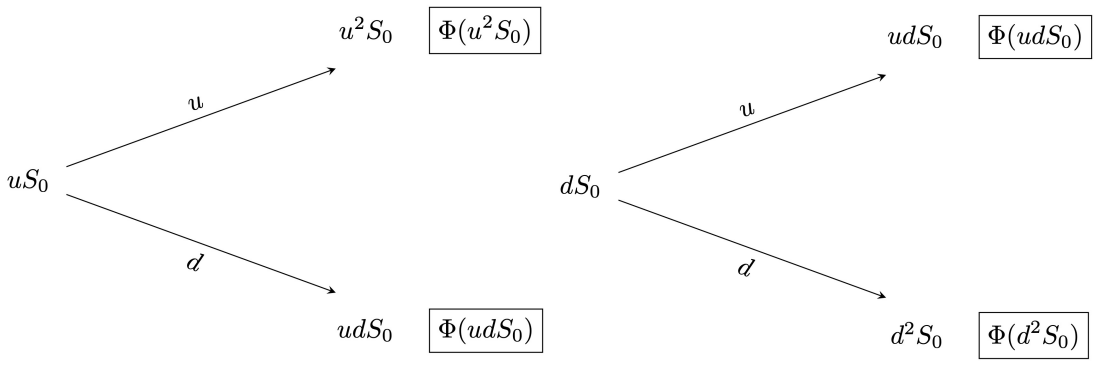


Figure 3.4: One-period subtrees.

model with $T = 2$. First, we need to introduce further notation. We denote by $V_t(k)$ the value of the replicating portfolio of the claim at the node (t, k). Next, we need to split the tree in Figure 3.3 into two (one-period) subtrees as shown in Figure 3.4.

Then, we can apply the principles of the one-period binomial model to compute $V_1(1)$ and $V_1(0)$ as follows

$$V_1(1) = \frac{1}{(1+R)}(q_u\Phi(u^2S_0) + q_d\Phi(udS_0)),$$
$$V_1(0) = \frac{1}{(1+R)}(q_u\Phi(udS_0) + q_d\Phi(d^2S_0)),$$

where q_u and q_d are the risk neutral probabilities given by

$$q_u = \frac{(1+R)-d}{u-d},$$
$$q_d = \frac{u-(1+R)}{u-d}.$$

Thus, we now obtain the one-period binomial tree given in Figure 3.5.

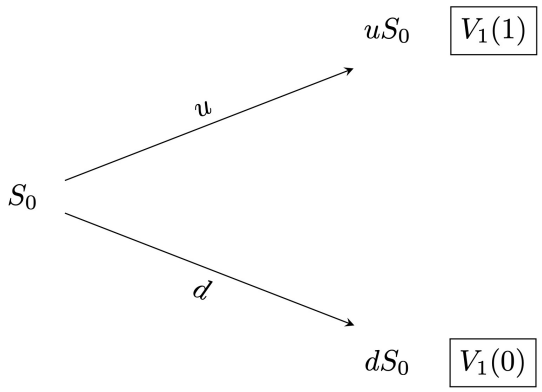


Figure 3.5: One-period subtree.

Finally, using the same idea, we compute the initial price of the portfolio $V_0(0)$ as shown below

$$V_0(0) = \frac{1}{(1+R)}(q_uV_1(1) + q_dV_1(0))$$

Example 3.3. Consider a stock currently with current price $S_0 = 50$. The Stock's price is expected to increase to 10% or decrease 8% during the next two six-month periods. The risk-free interest rate is 5% compounded annually. Compute the price of a 1-year European put option with a strike price of 55.

Solution.

```
s0 <- 50
u <- 1.1
d <- 0.92
strike <- 55

# Risk-free rate for the period of six months
rf <- (1 + 0.05)^(6 / 12) - 1

# Number of periods
n <- 2

# Future stock prices
k <- 0:n
s <- u^k * d^(n - k) * s0
s
```

```
## [1] 42.32 50.60 60.50
```

```
# Value of the contingent claim at maturity
phi <- pmax(strike - s, 0)
phi
```

```
## [1] 12.68  4.40  0.00
```

```
# Risk-neutral probability qu
qu <- (1 + rf - d) / (u - d)
qu
```

```
## [1] 0.5816393
```

```
# Price of the option
vu <- (qu * phi[3] + (1 - qu) * phi[2]) / (1 + rf)
vu
```

```
## [1] 1.796424
```

```
vd <- (qu * phi[2] + (1 - qu) * phi[1]) / (1 + rf)
vd
```

```
## [1] 7.674504
```

```
v0 <- (qu * vu + (1 - qu) * vd) / (1 + rf)
v0
```

```
## [1] 4.153022
```

We now give the general binomial algorithm.

Proposition 3.2. *Consider a claim $X = \Phi(S_T)$. Then $V_t(k)$ can be computed recursively as*

$$V_t(k) = \frac{1}{(1+R)}(q_u V_{t+1}(k+1) + q_d V_{t+1}(k)),$$
$$V_T(k) = \Phi(u^k d^{T-k} S_0).$$

In other words, the binomial algorithm consists of the following steps:

1. Generate the tree of stock prices.
2. Calculate the value of the claim at each final node.
3. Calculate the claim values sequentially at each preceding node.

To generate the tree of stock prices (1.), we can employ the following code:

```
build_tree <- function(s0, u, d, n) {
  tree <- matrix(0, nrow = n + 1, ncol = n + 1)
  for (t in 1:(n + 1)) {
    k <- 0:(t - 1)
    tree[t, 1:t] <- u^k * d^(t - 1 - k) * s0
  }
  tree
}
```

If we consider the same input data as in Example 3.3, we obtain the following tree

```
tree <- build_tree(50, 1.1, 0.92, 2)
tree
```

```
##       [,1] [,2] [,3]
## [1,] 50.00  0.0  0.0
## [2,] 46.00 55.0  0.0
## [3,] 42.32 50.6 60.5
```

With the above tree at hand, we can evaluate the claim at the final nodes (2.) by simply taking the last row of the matrix. For instance,

```
pmax(strike - tree[nrow(tree), ], 0)
```

```
## [1] 12.68  4.40  0.00
```

Finally, for the recursive computations (3.) we can use the following code:

```
value_bin_mod <- function(qu, rf, tree, strike) {
  val_tree <- matrix(0, nrow = nrow(tree), ncol = ncol(tree))
  val_tree[nrow(tree), ] <- pmax(strike - tree[nrow(tree), ], 0) # European put
  for (t in (nrow(tree) - 1):1) {
    for (k in 1:t) {
      val_tree[t, k] <- ((1 - qu) * val_tree[t + 1, k]
        + qu * val_tree[t + 1, k + 1]) / (1 + rf)
    }
  }
  val_tree
}
```

Applying the above code to our example we obtain:

```
opt_price <- value_bin_mod(qu, rf, tree, strike)
opt_price
```

```
##           [,1]     [,2] [,3]
## [1,]  4.153022 0.000000    0
## [2,]  7.674504 1.796424    0
## [3,] 12.680000 4.400000    0
```

Note that the price at time $t = 0$ is given by the entry `[1, 1]`, that is,

```
opt_price[1, 1]
```

```
## [1] 4.153022
```

Remark. The above code for performing the binomial algorithm was presented using a European put option as an example. However, it can be easily modified to price any other simple derivative.

It turns out that Proposition 3.2, implies the following risk-neutral valuation formula.

Proposition 3.3. *The arbitrage-free price at $t = 0$ of a claim X with maturity T is given by*

$$\Pi(0; X) = \frac{1}{(1+R)^T} \sum_{k=0}^{T} \binom{T}{k} q_u^k q_d^{T-k} \Phi(S_0 u^k d^{T-k}) \,.$$

Thus, an alternative solution to Example 3.3 using the above result is the following:

```
pi0 <- sum(choose(n, k) * qu^k * (1 - qu)^(n - k) * phi) / (1 + rf)^n
pi0
```

```
## [1] 4.153022
```

Sometimes, instead of u and d a volatility of the stock σ is provided. This stock's volatility is defined as the annualized standard deviation of the stock return. Luckily, given the volatility σ, we can calculate u and d by using

$$u = \exp(\sigma/n) \quad \text{and} \quad d = \exp(-\sigma/n) = 1/u \,,$$

where n is the number of intervals over one year. One can show that when n tends to infinity, the binomial model converges to the Black-Scholes model.

R packages for the binomial model

There are several R packages that have implementations for the binomial model. For instance, `derivmkts` and `fOptions`. However, only important examples such as call and put options are available. Thus, when dealing with less standard derivatives, the previous analysis is highly relevant. We will come back to more complex examples shortly, but first let us illustrate the use of the `derivmkts` package.

```
library(derivmkts)
```

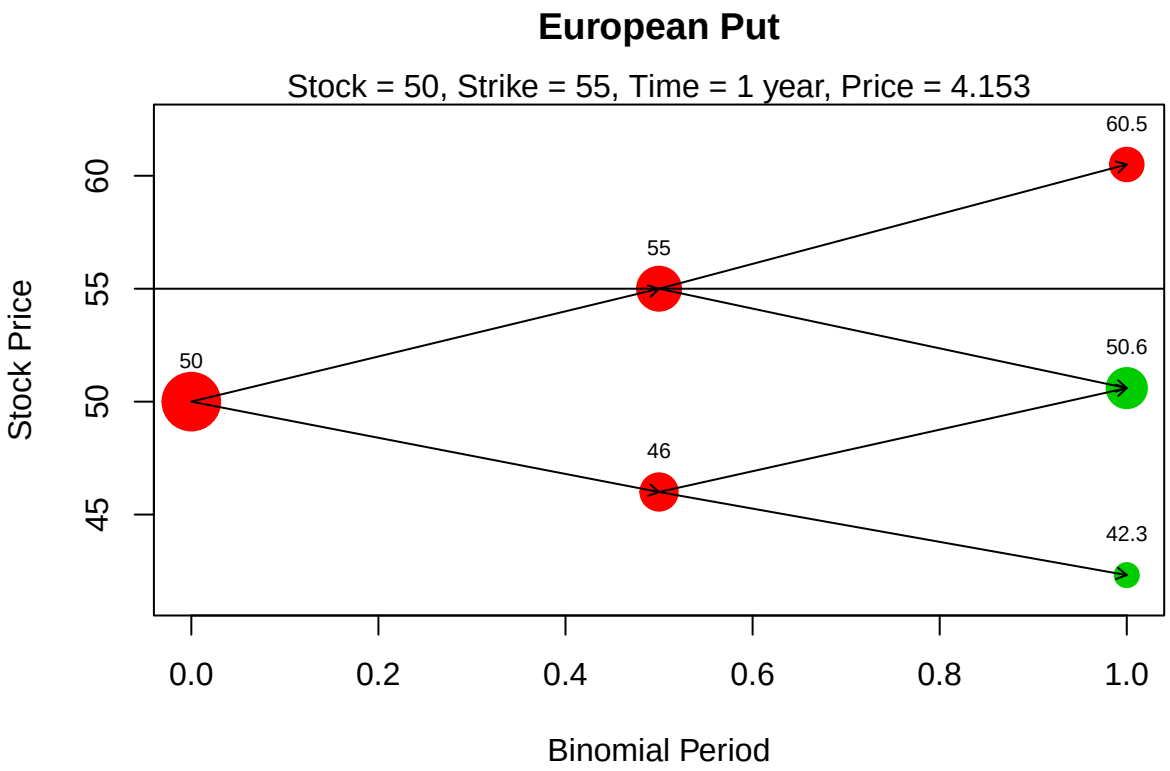
To compute prices of European (and American) call and put options we make use of the function `binomopt()`. Let us solve Example 3.3 using this function. We need to note the following about `binomopt()`: the risk-free rate (`r`) must be an annual continuously compounded rate. To price an European option we need to change the default value of the argument `american = TRUE` to `FALSE`. By default, the function works with a volatility, hence we need to use `specifyupdn = TRUE` to indicate that we will provide u and d through the arguments `up` and `dn`. We describe the rest of the arguments needed in the following code:

```
binomopt(
  s = 50, # Initial stock price
  k = 55, # Strike price
  r = log(1 + 0.05), # Continuously-compounded risk-free rate
  tt = 1, # Time to maturity
  d = 0, # Dividends, in our case, we do not work with dividends, hence 0.
  nstep = 2, # Number of periods
  american = FALSE, # To indicate European
  putopt = TRUE, # To indicate a Put option
  specifyupdn = TRUE, # Tells the function to use u and d
  up = 1.1, # Value of u
  dn = 0.92 # Value of d
)
```

```
##    price
## 4.153022
```

Moreover, `derivmkts` contains the function `binomplot()` which plots the development of the stock price and shows graphically the probability of being at each node (represented as the area of the circle at that price). The following is the plot for our example:

```
binomplot(
  s = 50, k = 55, r = log(1 + 0.05), tt = 1, d = 0, nstep = 2,
  american = FALSE, putopt = TRUE, specifyupdn = TRUE, up = 1.1, dn = 0.92,
  v = 0, # A value of the volatility must be provided, although not used
  plotarrows = TRUE, # Plots arrows that connect the nodes of the tree
  plotvalues = TRUE # Plots the values of the stock prices at each node
)
```



Note that a horizontal line with the strike value is included in the plot. This can also be omitted by using `drawstrike = FALSE`. Finally, the green and red colors indicate whether or not the option is optimally exercised there (green if yes, red if no). However, this is mainly relevant when working with American options.

We finish our discussion of the binomial model with the following more complicated example, which cannot be solved directly with the functions in `derivmkts`. Hence the importance of our early discussion.

Example 3.4. A stock price is currently 40. Over each of the next three 5-month periods, it is expected to go up by 10% or down by 5%. The risk-free interest rate is 6% per annum with continuous compounding.

1. Use a three-step binomial model to find the value of a European style derivative that pays off $X = [\max(K - S_T, 0)]^3$, where S_T is the stock price in 15 months and $K = 44$.
2. Use a three-step binomial model to find the value of a European-style derivative that pays off $X = \min(S_T^2, K)$, where S_T is the stock price in 15 months and $K = 2000$.

Solution. We will solve 1. using Proposition 3.2, and 2. using Proposition 3.3:

1.

```
s0 <- 40
kk <- 44
u <- 1.1
d <- 0.95
rf <- exp(0.06 * 5 / 12) - 1

tree <- build_tree(s0, u, d, 3)
tree
```

```
##         [,1]  [,2]  [,3]  [,4]
## [1,] 40.000  0.00  0.00  0.00
## [2,] 38.000 44.00  0.00  0.00
## [3,] 36.100 41.80 48.40  0.00
## [4,] 34.295 39.71 45.98 53.24
```

```
qu <- (1 + rf - d) / (u - d)
qu
```

```
## [1] 0.5021008
```

```
value_bin_mod <- function(qu, rf, tree, kk) {
  val_tree <- matrix(0, nrow = nrow(tree), ncol = ncol(tree))
  val_tree[nrow(tree), ] <- (pmax(kk - tree[nrow(tree), ], 0))^3 # Given claim
  for (t in (nrow(tree) - 1):1) {
    for (k in 1:t) {
      val_tree[t, k] <- ((1 - qu) * val_tree[t + 1, k]
        + qu * val_tree[t + 1, k + 1]) / (1 + rf)
    }
  }
  val_tree
}

opt_price <- value_bin_mod(qu, rf, tree, kk)
opt_price
```

```
##          [,1]     [,2] [,3] [,4]
## [1,] 132.0264  0.00000    0    0
## [2,] 253.1042 18.61830    0    0
## [3,] 482.5491 38.34034    0    0
## [4,] 914.0851 78.95359    0    0
```

Thus, the price of the option is

```
opt_price[1, 1]
```

```
## [1] 132.0264
```

2.

```
n <- 3
k <- 0:n

# Prices after 15 months
s <- u^k * d^(n - k) * s0
s
```

```
## [1] 34.295 39.710 45.980 53.240
```

```
# Value of the contingent claim at maturity
phi <- pmin(s^2, 2000)
phi
```

```
## [1] 1176.147 1576.884 2000.000 2000.000
```

```
pi0 <- sum(choose(n, k) * qu^k * (1 - qu)^(n - k) * phi) / (1 + rf)^n
pi0
```

```
## [1] 1614.563
```

3.3 The Black and Scholes model

The binomial model discussed in the previous section is a discrete-time model: the stock price changes at the end of each time period. Another commonly used model for option pricing is the Black-Scholes model, which assumes that the stock price moves continuously on time. The assumptions behind the Black-Scholes model are deep, and in fact, an entire course can be dedicated to the development of this model. In this course, it is sufficient for you to know how to price the options using simulations and the closed-form Black-Scholes formula. We start with a review of the Brownian motion, which will be used to describe the price movements of a stock.

3.3.1 Preliminars: Brownian motion

Definition 3.5 (Standard Brownian motion)**.** A *standard Brownian motion* is an stochastic process $W = (W(t))_{t\geq 0}$ satisfying:

1. $W(0) = 0$.
2. The process has *independent increments*, i.e., for any $0 \leq t_1 < t_2 < \cdots < t_n$, $W(t_2) - W(t_1), \ldots, W(t_n) - W(t_{n-1})$ are independent random variables.
3. For $s < t$, $W(t) - W(s) \sim N(0, t - s)$.
4. W has continuous trajectories.

To generate trajectories of a standard Brownian motion over a time period $[0, T]$, we consider a "small" time increment $\delta t > 0$. Then, consider the independent increments

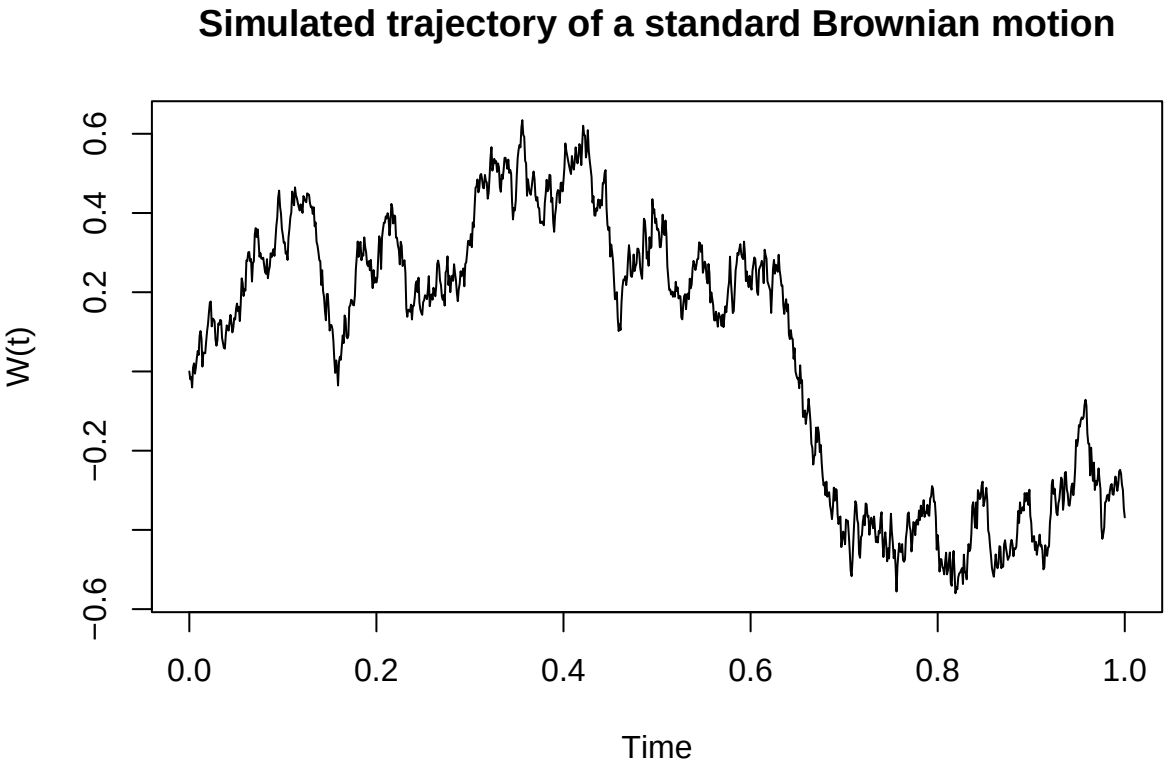
$$W(\delta t) - W(0)\,,$$
$$W(2\delta t) - W(\delta t)\,.$$

Note that these increments are $N(0, \delta t)$ distributed. Moreover, we can compute $W(2\delta t)$ as

$$W(2\delta t) = (W(2\delta t) - W(\delta t)) + (W(\delta t) - W(0))\,.$$

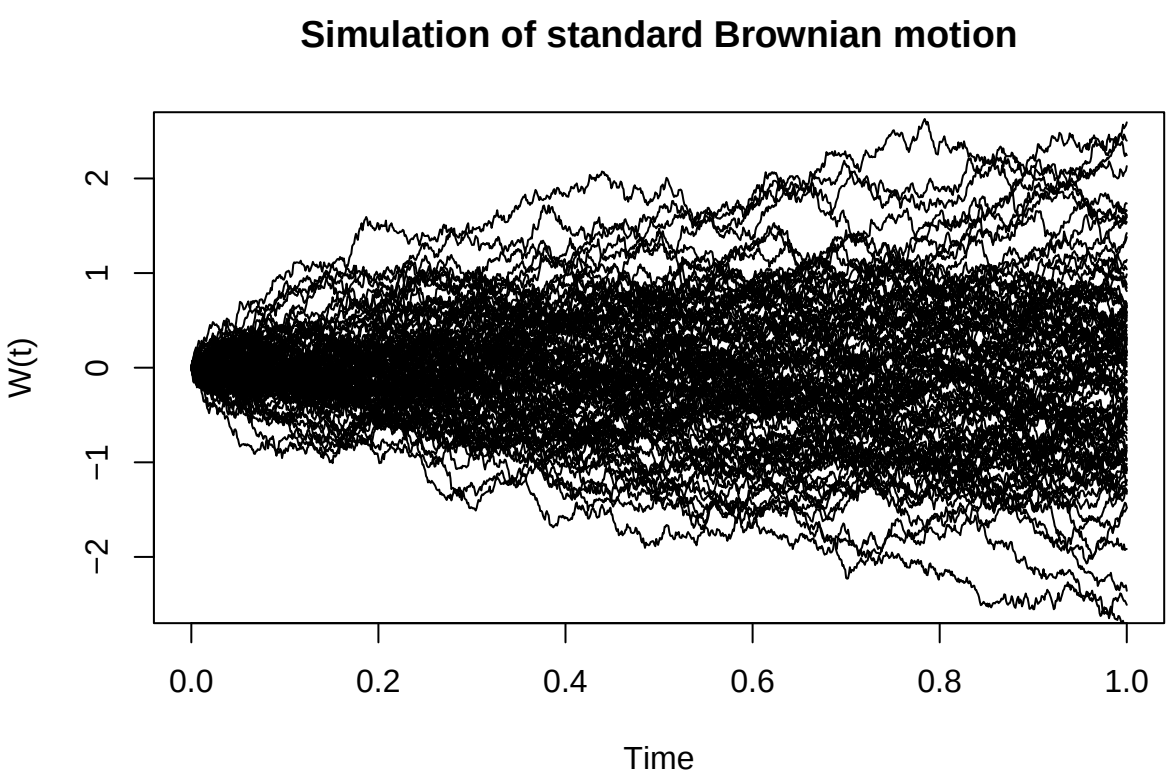
Thus, we can repeat the same logic as above to simulate a whole process trajectory. The following code implements this idea:

```
set.seed(1)
delta <- 0.001 # Increment
t <- seq(0, 1, by = delta) # Time interval
w <- rnorm(n = length(t) - 1, sd = sqrt(delta)) # iid normal distributed r.v.s
w <- c(0, cumsum(w)) # Cumulative sum - 0 is the initial value
plot(t, w,
  type = "l",
  xlab = "Time",
  ylab = "W(t)",
  main = "Simulated trajectory of a standard Brownian motion",
)
```



We now modify the code above to generate multiple trajectories:

```
nsim <- 100 # Number of simulated trajectories
w <- matrix(
  rnorm(n = nsim * (length(t) - 1), sd = sqrt(delta)),
  nsim, # Each row is one simulated trajectory
  length(t) - 1
)
w <- cbind(rep(0, nsim), t(apply(w, 1, cumsum))) # matrix with trajectories
plot(t, w[1, ], # Plots the first trajectory
  type = "l",
  ylim = c(-2.5, 2.5),
  xlab = "Time",
  ylab = "W(t)",
  main = "Simulation of standard Brownian motion"
)
apply(w[-1, ], 1, function(x, t) lines(t, x), t = t) # Plots the remaining trajectories
```




```
## NULL
```

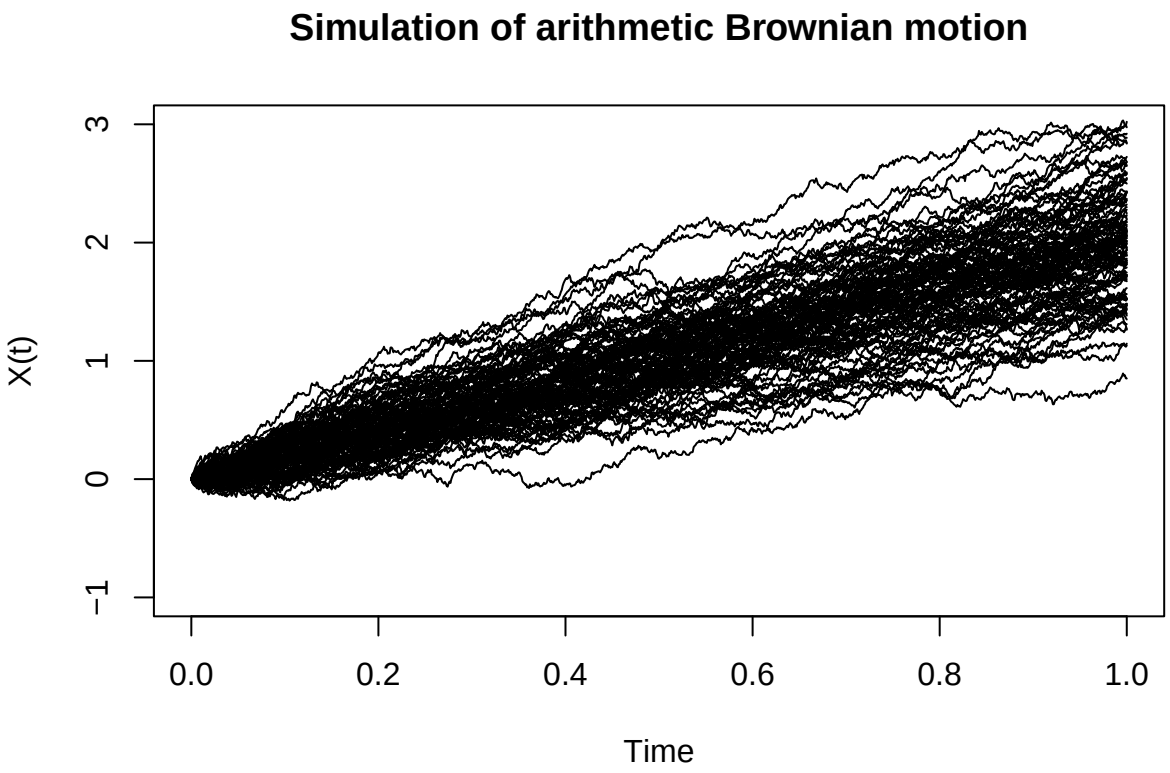
The standard Brownian motion can be generalized to the *arithmetic Brownian motion*, which scales and shifts the former. More specifically, $X(t)$ is an arithmetic Brownian motion if

$$X(t) = \mu t + \sigma W(t)$$

where $W(t)$ is a standard Brownian motion and $\mu \in \mathbb{R}$ and $\sigma > 0$. Here, μ is called the *drift*, and σ is called the *volatility* of the process. Moreover, note that for $s < t$, $X(t) - X(s) \sim N(\mu(t-s), \sigma^2(t-s))$. There are two straightforward ways to simulate from the arithmetic Brownian motion. One would be to simulate from a standard Brownian motion and then use the relationship above. The second, and perhaps easier to modify in our code above, is to generate independent $N(\mu\delta t, \sigma^2\delta t)$ random variables. The next code implements the later:

```
mu <- 2
sigma2 <- 0.2
nsim <- 100
x <- matrix(
  rnorm(n = nsim * (length(t) - 1), mean = mu * delta, sd = sqrt(delta * sigma2)),
```

```
  nsim,
  length(t) - 1
)
x <- cbind(rep(0, nsim), t(apply(x, 1, cumsum)))
plot(t, x[1, ],
  type = "l",
  ylim = c(-1, 3),
  xlab = "Time",
  ylab = "X(t)",
  main = "Simulation of arithmetic Brownian motion"
)
apply(x[-1, ], 1, function(x, t) lines(t, x), t = t)
```



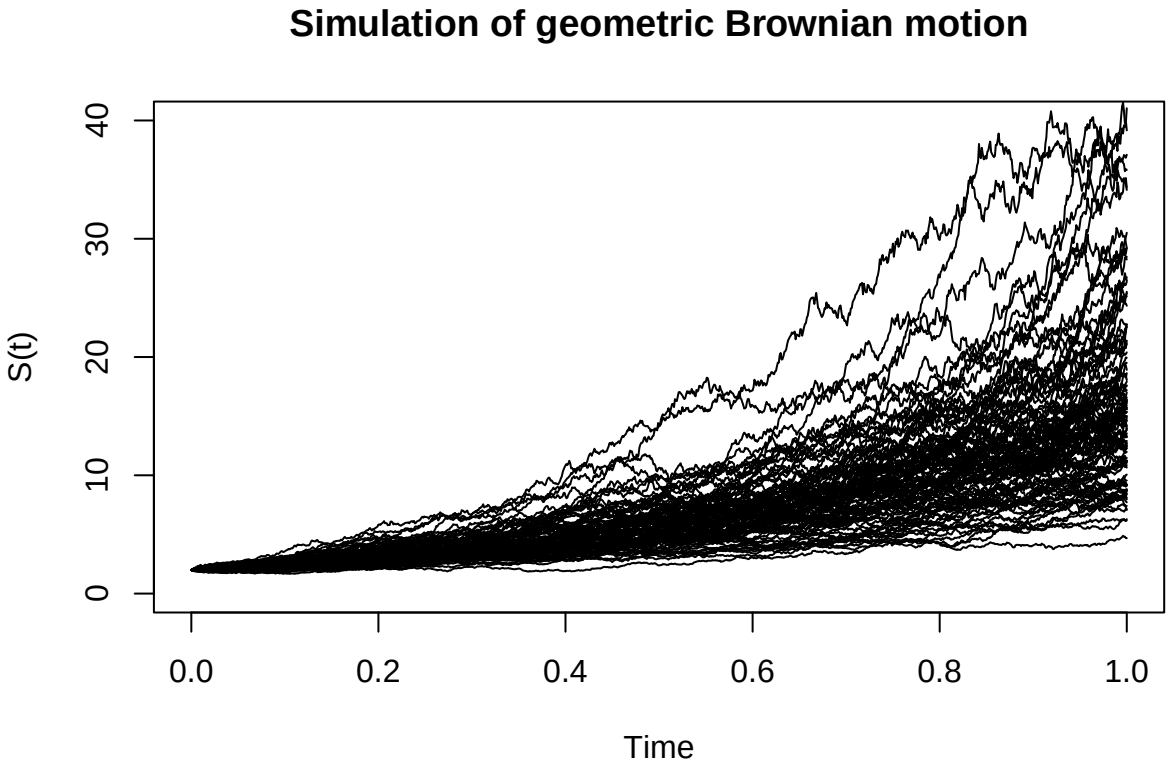

```
## NULL
```

Note that the arithmetic Brownian motion can take negative values. Hence, using it for modeling stock prices is questionable. Instead, we now introduce the so-called called *geometric Brownian motion*, which can only take nonnegative values. More precisely, a geometric Brownian motion $S(t)$ is a stochastic process of the form

$$S(t) = S(0) \exp\left(X(t)\right) = S(0) \exp\left(\mu t + \sigma W(t)\right) ,$$

where $S(0) > 0$ is the starting value, and $X(t)$ is an arithmetic Brownian motion with drift μ and volatility σ. The following code generates some trajectories of this process using the previous implementation:

```
s0 <- 2
s <- s0 * exp(x)
plot(t, s[1, ],
  type = "l",
  ylim = c(0, 40),
  xlab = "Time",
  ylab = "S(t)",
  main = "Simulation of geometric Brownian motion"
)
apply(s[-1, ], 1, function(x, t) lines(t, x), t = t)
```




```
## NULL
```

Note that in particular, $S(t)/S(0) \sim LN(\mu t, \sigma^2 t)$, and more generally, for $s < t$, $S(t)/S(s) \sim LN(\mu(t-s), \sigma^2(t-s))$.

Finally, one can show that a geometric Brownian motion of the form

$$S(t) = S(0) \exp\left(\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W(t)\right) ,$$

satisfies the *stochastic differential equation* (SDE)

$$dS(t) = \mu S(t)dt + \sigma dW(t) .$$

3.3.2 The Black and Scholes formula

As in the binomial model, we consider two assets: a risk-free asset with prices $B(t)$ and a stock with prices $S(t)$. The Black–Scholes model assumes that the prices of these two assets satisfy

$$\begin{aligned} dB(t) &= rB(t)dt , \\ dS(t) &= \alpha S(t)dt + \sigma d\tilde{W}(t) , \end{aligned}$$

where r, α and σ are deterministic constants and $\tilde{W}$ is a standard Brownian motion. Next, we consider a simple contingent claim of the form

$$X = \Phi(S(T)) .$$

It can be shown that the arbitrage-free price $\Pi(0; X)$ of this simple claim X is given by the risk-neutral evaluation formula

$$\Pi(0; X) = e^{-rT} \mathbb{E}^Q \left[\Phi(S(T))\right] , \tag{3.1}$$

where the behavior of S under Q is described by the SDE

$$dS(t) = rS(t)dt + \sigma dW(t) ,$$

whose solution is the geometric Brownian motion. In particular, we have that $S(T)$ is explicit and given by

$$S(T) = S(0) \exp\left(\left(r - \frac{\sigma^2}{2}\right) T + \sigma W(T)\right) . \tag{3.2}$$

When considering a European call option, that is, $\Phi(x) = \max(x - K, 0)$, (3.1) and (3.2) and lead to the famous Black–Scholes formula.

Proposition 3.4. *The price of a European call option $C(0)$ with strike price K and time of maturity T is given by*

$$C(0) = S(0)N(d_1) - Ke^{-T}N(d_2)\,, \tag{3.3}$$

where N is the cumulative distribution function for the $N(0,1)$ distribution and

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left(\log\left(\frac{S(0)}{K}\right) + \left(r + \frac{1}{2}\sigma^2\right) T\right) ,$$
$$d_2 = d_1 - \sigma\sqrt{T}\,.$$

Similarly, the price of a European put option $P(0)$ with strike price K and time of maturity T is given by

$$P(0) = Ke^{-T}N(-d_2) - S(0)N(-d_1)\,,$$

with d_1 and d_2 as above.

In R, the Black–Scholes formula to price European call and put option is available in the `derivmkts` package under the functions `bscall()` and `bsput()`, receptively. We illustrate its use with an example.

Example 3.5. Consider a European call option over a stock with a current price S(0) = 50 and volatility 0.25. Moreover, the risk-free interest rate with continuous compounding is 6% per annum, the strike price is 45, and the option's time to maturity is 6 months. Find the price of the option.

Solution.

```
s0 <- 50
sigma <- 0.25
rf <- 0.06
maturity <- 6 / 12
strike <- 45
d <- 0 # No dividends
bscall(s0, strike, sigma, rf, maturity, d)
```

```
## [1] 7.381939
```

Note that (3.1) suggest that we can approximate the price of an option by simulating values of $S(T)$. Moreover, the simulation can be done easily by using the function `simprice()` in `derivmkts` For example, for the previous example, we can approximate the price as follows:

```
set.seed(1)
st <- simprice(s0, sigma, rf, maturity, d, trials = 10000, periods = 1)
exp(-rf * maturity) * mean(pmax(st[st$period == 1, ]$price - strike, 0))
```

```
## [1] 7.377834
```

3.3.3 Greeks

Greeks represent sensitivities of a derivative value to changes in the underlying parameters used to determine its price. More specifically, if we denote by V be the price of a derivative, which depends on the underlying stock price S, a risk-free rate r, a volatility σ and with maturity T, then the Greeks are defined as follows:

Definition 3.6 (Greeks)**.**

Delta (Δ): Measures the rate of change of the option price with respect to changes in the underlying stock price.

$$\Delta = \frac{\partial V}{\partial S}\,.$$

Gamma (Γ): Measures the rate of change in Δ with respect to changes in the underlying stock price.

$$\Gamma = \frac{\partial^2 V}{\partial S^2} = \frac{\partial \Delta}{\partial S}\,.$$

Vega (ν): Measures sensitivity with respect to changes in the volatility σ.

$$\nu = \frac{\partial V}{\partial \sigma}\,.$$

Rho (ρ): Measures sensitivity with respect to changes in the risk-free interest rate.

$$\rho = \frac{\partial V}{\partial r}\,.$$

Theta (Θ): Measures the sensitivity of the value of the derivative to the passage of time.

$$\Theta = -\frac{\partial V}{\partial T}\,.$$

Elasticity or Lambda (λ): Is the percentage change in option value per percentage change in the underlying price (a measure of leverage).

$$\lambda = \frac{\partial V}{\partial S}\frac{S}{V} = \Delta \times \frac{S}{V}\,.$$

Psi (ψ): Is the percentage change in option value per percentage change in the underlying dividend yield (q).

$$\psi = \frac{\partial V}{\partial q}\,.$$

In particular, for a European call option with strike price K and time of maturity T we have the following explicit expressions:

$$\begin{aligned}
\Delta &= N(d_1)\,,\\
\Gamma &= \frac{N'(d_1)}{S\sigma\sqrt{T}}\,,\\
\nu &= S\sqrt{T}N'(d_1)\,,\\
\rho &= KTe^{-rT}N(d_2)\,,\\
\Theta &= -\frac{S(0)N'(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}N(d_2)\,.
\end{aligned}$$

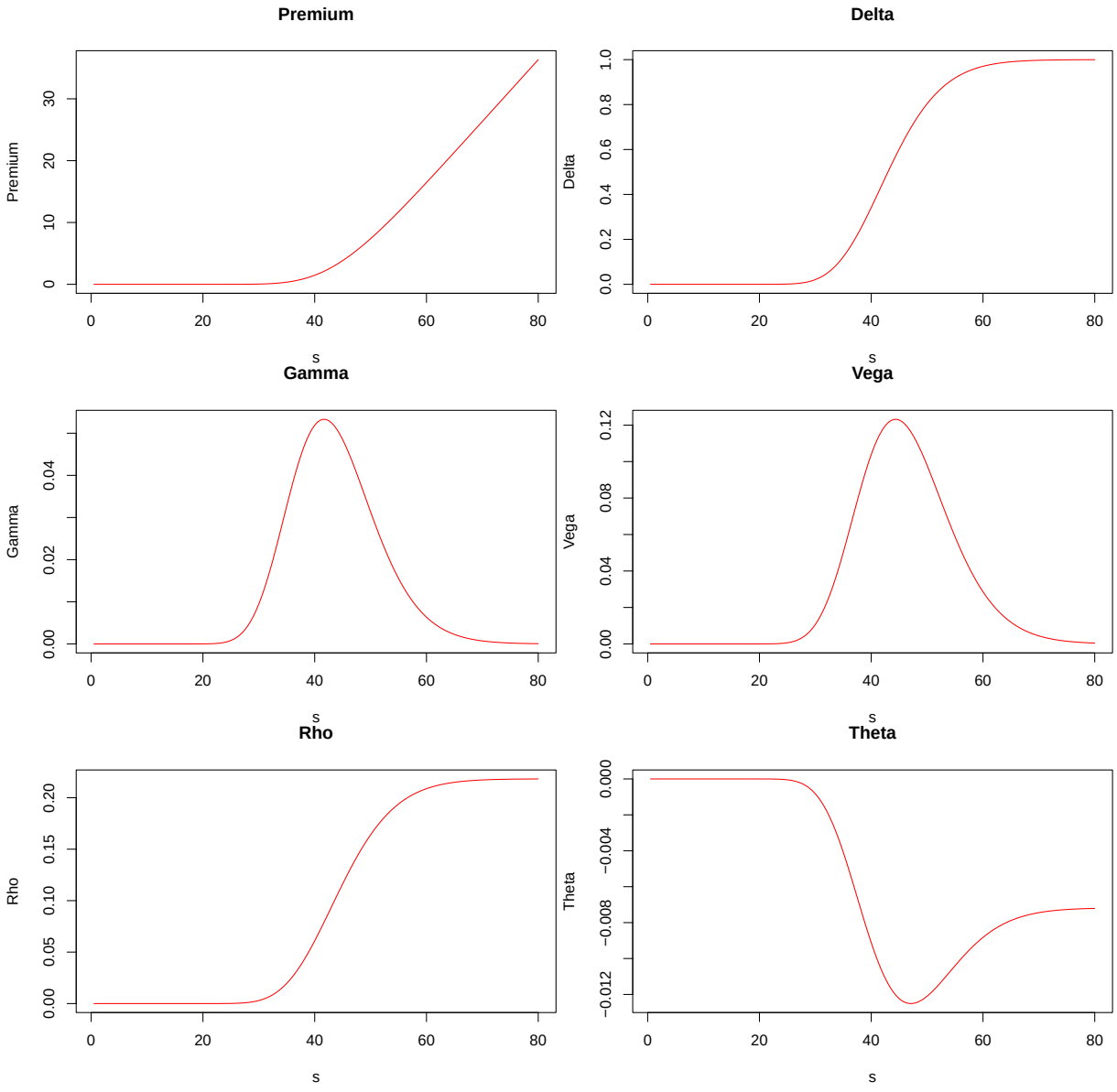
The `derivmkts` R package allows us to compute the Greeks for a financial derivative via the function `greeks()`. For instance, the Greeks for the option in Exercise 3.5 can be computed as follows:

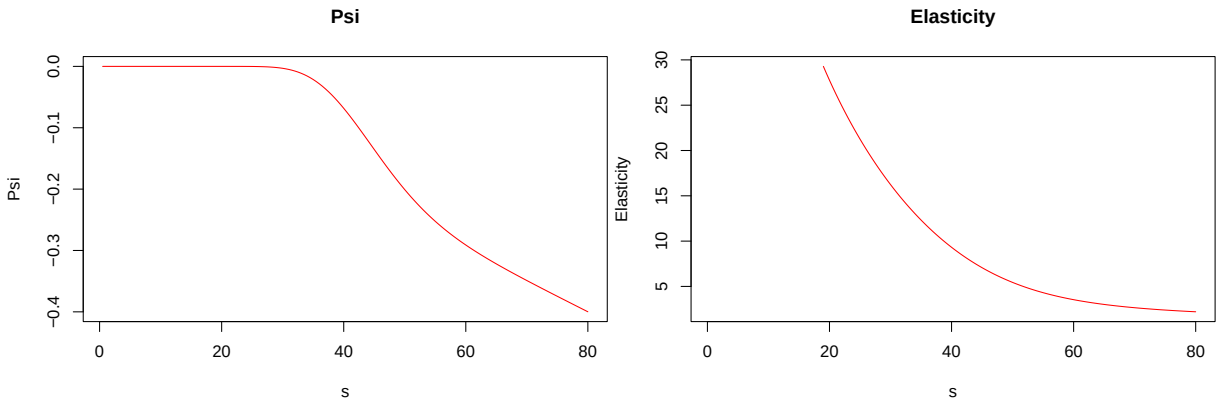
```
greeks(bscall(s0, strike, sigma, rf, maturity, d = 0))
```

```
##                 bscall
## Premium     7.38193911
## Delta       0.80347605
## Gamma       0.03134062
## Vega        0.09793950
## Rho         0.16395932
## Theta      -0.01209863
## Psi        -0.20086901
## Elasticity  5.44217476
```

Furthermore, `greeks()` accepts vector inputs, which in particular allow us to visualize the Greeks.

```
s <- seq(.5, 80, by = .5)
call_greeks <- greeks(bscall(s, strike, sigma, rf, maturity, d = 0))
for (i in rownames(call_greeks)) {
  plot(s, call_greeks[i, ], main = paste(i), ylab = i, type = "l", col = "red")
}
```




Psi
Psi
0.0
−0.1
−0.2
−0.3
−0.4
0
20
40
60
80
s
Elasticity
Elasticity
30
25
20
15
10
5
0
20
40
60
80
s

Chapter 4

R for Insurance

4.1 The collective risk model

Let S be a random variable representing the aggregate claim amount (or the total amount of claims) of an insurance portfolio of independent risks over a fixed period. Let N be a random variable representing the number of claims (or frequency) in the portfolio over that period, and $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables denoting the claim sizes (or severities), with common element $X \geq 0$. Then, S is given as the random sum

$$S = \begin{cases} X_1 + X_2 + \cdots + X_N\,, & N > 0\,, \\ 0\,, & N = 0\,. \end{cases}$$

In order to derive several properties of the model above, the collective risk model assumes that the number of claims and the claim sizes are independent, that is, N and $X_1, X_2, \ldots$ are mutually independent.

In the following, the probability mass function (pf) of N is denoted by $p_n = \mathbb{P}(N = n)$, and the probability density function (pdf) and cumulative distribution function (cdf) of X are denoted by $f_X(x)$ and $F_X(x) = \mathbb{P}(X \leq x)$, respectively.

We now present some properties of the aggregate loss S. We start with the moment generating function (mgf) $M_S(t)$, which is given by

$$M_S(t) = \mathbb{E}[\exp(tS)] = M_N\left(\ln M_X(t)\right)\,,$$

where $M_N(t)$ and $M_X(t)$ are the mgf's of N and X, respectively.

Next, the mean and variance of S are

$$\mathbb{E}[S] = \mathbb{E}[X]\mathbb{E}[N]\,,$$

and

$$\mathrm{Var}(S) = \mathbb{E}[N]\mathrm{Var}(X) + \mathrm{Var}(N)\mathbb{E}^2[X]\,.$$

Finally,the distribution function F_S of S is given by

$$F_S(x) = \mathbb{P}(S \leq x) = \sum_{n=0}^{\infty} p_n F_X^{\star n}(x)\,, \quad x \geq 0\,, \tag{4.1}$$

where $F_X^{\star n}$ is the cdf of the n-fold convolution of the r.v. X defined as

$$F_X^{\star n}(x) = \begin{cases} 1_{(x \geq 0)}\,, & n = 0\,, \\ \mathbb{P}(X_1 + X_2 + \cdots + X_n \leq x)\,, & n = 1, 2, \ldots\,, \end{cases}$$

and 1_A denotes the indicator function over a set A.

When X is continuous, differentiating F_S with respect to x yields the following expression for the density function f_S of S

$$f_S(x) = \sum_{n=0}^{\infty} p_n f_X^{\star n}(x)\,, \quad x \geq 0\,,$$

where $f_X^{\star n}$ is the pdf of the n-fold convolution of the random variable X. An analogous expression for the pf of S can be obtained in the case of X discrete.

Both the cdf and the pf (pdf) of the n-fold convolution of X can be obtained recursively as given in the following proposition.

Proposition 4.1.

(i) If X is a discrete random variable, then the cdf of the n-fold convolution of X can be evaluated recursively by the following formula

$$F_X^{\star n}(x) = \begin{cases} 1_{(x\geq 0)}\,, & n=0\,, \\ F_X(x)\,, & n=1\,, \\ \sum_{y=0}^{x} f_X(y) F_X^{\star(n-1)}(x-y), & n=2,3,\ldots\,, \end{cases} \tag{4.2}$$

while the pf of the n-fold convolution of X can be evaluated recursively by

$$f_X^{\star n}(x) = \sum_{y=0}^{x} f_X(y) f_X^{\star(n-1)}(x-y)\,.$$

(ii) If X is a continuous random variable, then the cdf of the n-fold convolution of X can be evaluated recursively by the following formula

$$F_X^{\star n}(x) = \int_0^x f_X(y) F_X^{\star(n-1)}(x-y)dy\,,$$

while the pdf of the n-fold convolution of X can be evaluated recursively by

$$f_X^{\star n}(x) = \int_0^x f_X(y) f_X^{\star(n-1)}(x-y)dy\,.$$

4.1.1 Discretization of claim amount distributions

Some numerical techniques to compute the aggregate claim amount distribution require a discrete arithmetic claim amount distribution, that is, a distribution defined on $0, h, 2h, \ldots$ for some step (or span, or lag) h. The `actuar` package provides the function `discretize()` to discretize continuous distributions. More specifically, given $F(x)$, the cdf of the distribution to discretize on some interval (a, b), the `discretize()` function supports the following four discretization methods (we denote by f_x the pf at x of the discretized distribution).

1. Upper discretization, or forward difference of $F(x)$:

$$f_x = F(x+h) - F(x)\,, \quad x = a, a+h, \ldots, b-h\,.$$

The discretized cdf is always above the true cdf.

2. Lower discretization, or backward difference of $F(x)$:

$$f_x = \begin{cases} F(a)\,, & x = a\,, \\ F(x) - F(x-h)\,, & x = a+h, \ldots, b\,. \end{cases}$$

The discretized cdf is always under the true cdf.

3. Rounding of the random variable, or the midpoint method:

$$f_x = \begin{cases} F(a+h/2)\,, & x = a\,, \\ F(x+h/2) - F(x-h/2)\,, & x = a+h, \ldots, b-h\,. \end{cases}$$

The true cdf passes exactly midway through the steps of the discretized cdf.

4. Unbiased, or local matching of the first moment method:

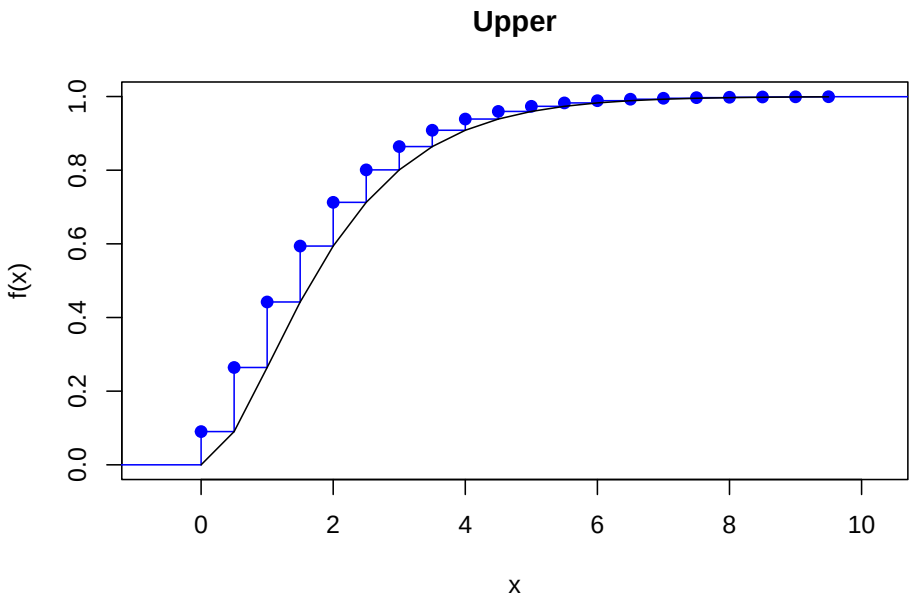
$$f_x = \begin{cases} \frac{E[X\wedge a] - E[X\wedge a+h]}{h} + 1 - F(a)\,, & x = a\,, \\ \frac{2E[X\wedge x] - E[X\wedge x-h] - E[X\wedge x+h]}{h}\,, & a < x < b\,, \\ \frac{E[X\wedge b] - E[X\wedge b-h]}{h} - 1 + F(b)\,, & x = b\,, \end{cases}$$

where $c \wedge d$ denotes $\min(c, d)$. The discretized and the true distributions have the same total probability and expected value on (a, b).

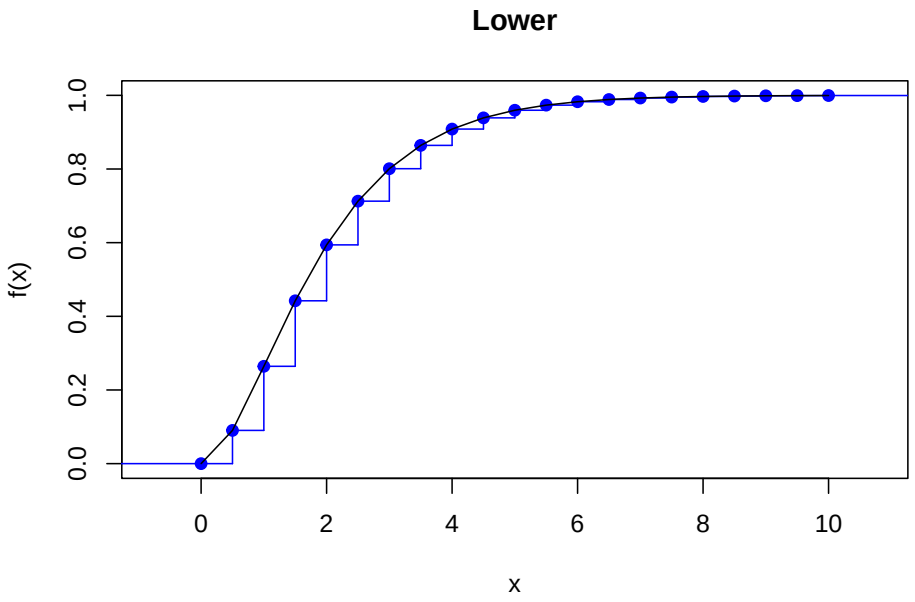
Example 4.1. Find the four discretizations of a $Gamma(2, 1)$ distribution on $(0, 10)$ with a step of 0.5 and plot their cdf against the original cdf.

Solution.

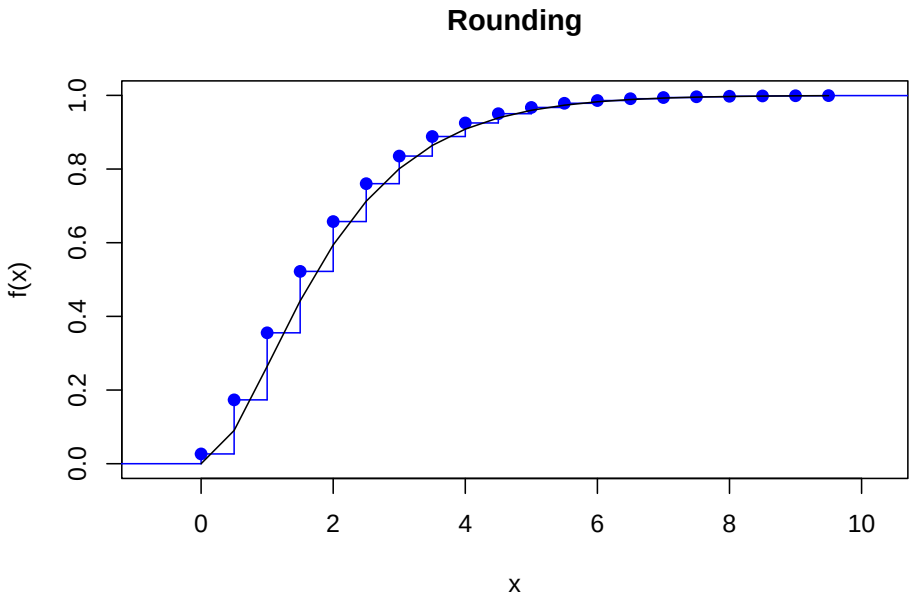
```
library(actuar)

fx_upper <- discretize(pgamma(x, 2, 1),
  method = "upper",
  from = 0,
  to = 10,
  step = 0.5
)

x <- seq(0, 10 - 0.5, 0.5)
plot(stepfun(x, diffinv(fx_upper)), pch = 19, col = "blue", main = "Upper")
lines(x, pgamma(x, 2, 1))
```




```
fx_lower <- discretize(pgamma(x, 2, 1),
  method = "lower",
  from = 0,
  to = 10,
  step = 0.5
)

x <- seq(0, 10, 0.5)
plot(stepfun(x, diffinv(fx_lower)), pch = 19, col = "blue", main = "Lower")
lines(x, pgamma(x, 2, 1))
```




```
fx_round <- discretize(pgamma(x, 2, 1),
  method = "rounding",
  from = 0,
  to = 10,
  step = 0.5
)

x <- seq(0, 10 - 0.5, 0.5)
plot(stepfun(x, diffinv(fx_round)), pch = 19, col = "blue", main = "Rounding")
lines(x, pgamma(x, 2, 1))
```

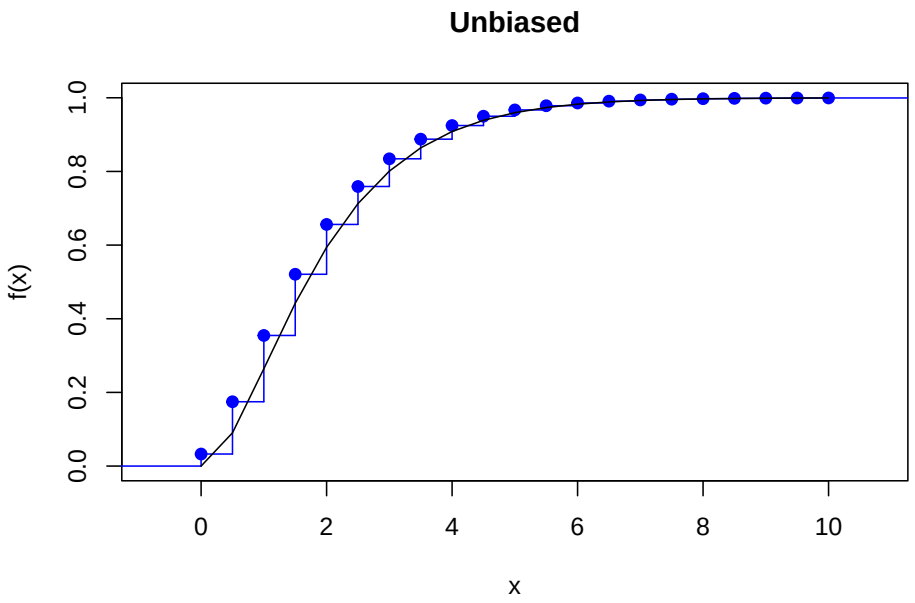



```
fx_unbi <- discretize(pgamma(x, 2, 1),
  method = "unbiased",
  from = 0,
  to = 10,
  step = 0.5,
  lev = levgamma(x, 2, 1) # Computes E[min(X, a)]
)
x <- seq(0, 10, 0.5)
plot(stepfun(x, diffinv(fx_unbi)), pch = 19, col = "blue", main = "Unbiased")
```

Table 4.1: Distribution in the $(a, b, 0)$ class

Distribution of N	Paremeters	a	b	p_0
Poisson	$\lambda > 0$	0	λ	$\exp(-\lambda)$
Binomial	$n \in \mathbb{N}$, $p \in (0,1)$	$-p/(1-p)$	$(n+1)p/(1-p)$	0
Geometric	$p \in (0,1)$	$1-p$	0	p
Negative Binomial	$r \in \mathbb{N}$, $p \in (0,1)$	$1-p$	$(r-1)(1-p)$	p^r

```
lines(x, pgamma(x, 2, 1))
```



4.1.2 Calculation of the aggregate claim amount distribution

In this section, we present various methods to compute or approximate the cdf of the aggregate claims amount S. These methods are implemented in the `actuar` package through the function `aggregateDist()`.

Panjer's recursion

The first method that we present is the well known Panjer's recursion. This method requires the severity distribution to be discrete arithmetic on $0, 1, 2, \ldots, m$ for some monetary unit and the frequency distribution to be in the $(a, b, 0)$ or $(a, b, 1)$ class of distributions. Recall that N is in the $(a, b, 0)$ class (resp. in the $(a, b, 1)$ class) if its pf p_n can be written recursively in the following form

$$p_n = \left(a + \frac{b}{n}\right) p_{n-1}, \quad n = 1, 2, 3, \ldots ,$$

resp.

$$p_n = \left(a + \frac{b}{n}\right) p_{n-1}, \quad n = 2, 3, \ldots .$$

Many known distributions belong to the $(a, b, 0)$ and $(a, b, 1)$ classes. For instance, Table 4.1 shows the most important examples in the $(a, b, 0)$ class.

Under the above setting, the pf of the aggregate claims can be calculated recursively based on the following theorem.

Theorem 4.1. *Let N be a random variable in the $(a, b, 0)$ (resp. $(a, b, 1)$) family of distributions. Then, for $x = 0, 1, \ldots$ the pf f_S of the aggregate claims is given by*

$$f_S(x) = \begin{cases} P_N(f_X(0)) , & x = 0 , \\ \frac{1}{1-af_X(0)} \sum_{y=1}^{x \wedge m} \left(a + \frac{by}{x}\right) f_X(y) f_S(x-y) , & x = 1, 2, \ldots , \end{cases}$$

resp.

$$f_S(x) = \begin{cases} P_N(f_X(0)), & x = 0 \\ \frac{1}{1-af_X(0)}\left((p_1-(a+b)p_0)f_X(x) + \sum_{y=1}^{x\wedge m}\left(a+\frac{by}{x}\right)f_X(y)f_S(x-y)\right), & x = 1,2,\dots, \end{cases}$$

where $P_N(t) = \mathbb{E}(t^N)$ *is the probability generating function (pgf) of* N*.*

To perform Panjer's recursion in R, we can use the function `aggregateDist()` with argument `method = "recursive"`. Let us give a concrete example. We consider S such that N is Poisson distributed with mean 10 and $X \sim Gamma(2,1)$. Then, we approximate the cdf of S by first discretizing the gamma distribution on (0, 22) with the unbiased method and a step of 0.5, and then using the recursive method in aggregateDist.

```
fx <- discretize(pgamma(x, 2, 1),
  method = "unbiased",
  from = 0, to = 22, step = 0.5,
  lev = levgamma(x, 2, 1)
)
Fs <- aggregateDist("recursive",
  model.freq = "poisson",
  model.sev = fx, lambda = 10,
  x.scale = 0.5
)
Fs
```

```
##
## Aggregate Claim Amount Distribution
##   Recursive method approximation
##
## Call:
## aggregateDist(method = "recursive", model.freq = "poisson", model.sev = fx,
##     x.scale = 0.5, lambda = 10)
##
## Data:  ( 143 obs. )
##  x[1:143] =     0,   0.5,     1,  ...,  70.5,    71
```

The above code returns an object of the class `aggregateDist` from which we can obtain information regarding S. Firstly, we can use directly such an object to evaluate the cdf of S:

```
Fs(20)
```

```
## [1] 0.5470771
```

Note that the support can be obtained with the `knots()` function:

```
head(knots(Fs))
```

```
## [1] 0.0 0.5 1.0 1.5 2.0 2.5
```

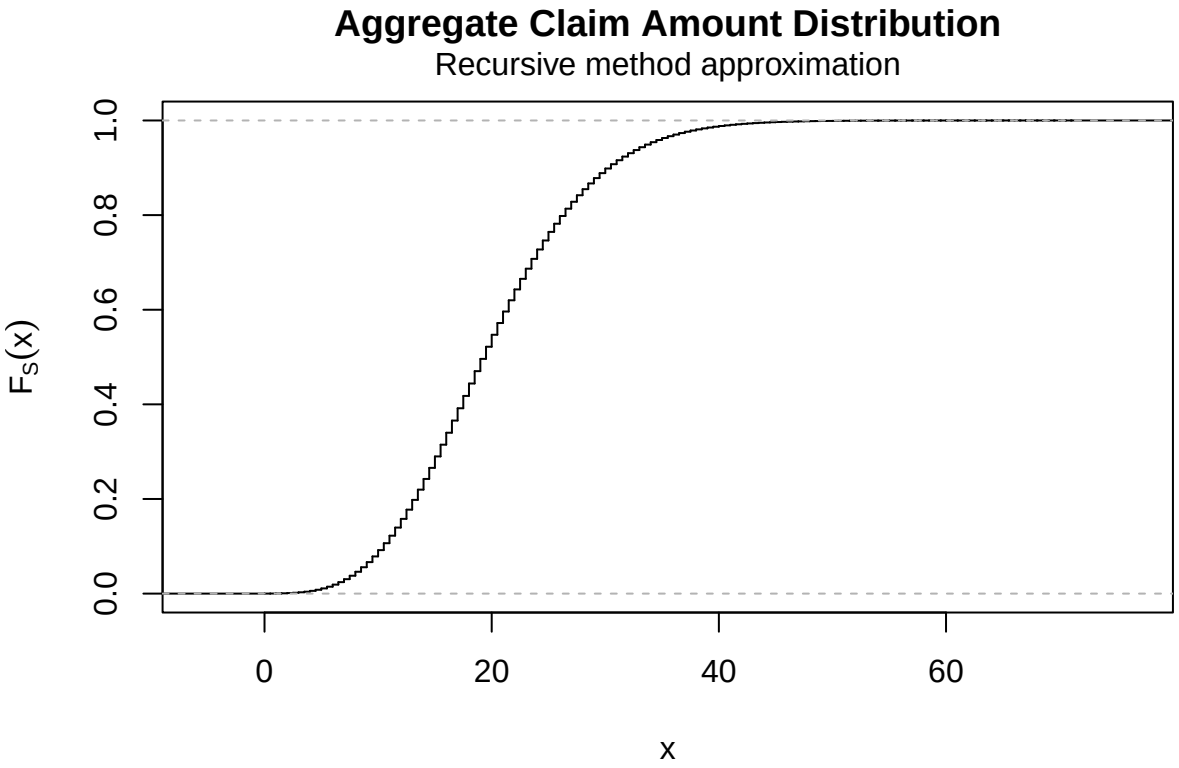
Moreover, we can apply other functions to the above object to obtain different quantities or even plots. For instance, `summary()` provides some basic information on S.

```
summary(Fs)
```

```
## Aggregate Claim Amount Empirical CDF:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.00000 14.50000 19.50000 19.99994 25.00000 71.00000
```

To plot the approximated cdf of S, we can use the `plot()` function:

```
plot(Fs, do.points = FALSE, verticals = TRUE)
```



We will illustrate additional useful functions with the other approximation method presented next.

Exact calculation by numerical convolutions

This method consists of using directly (4.1), and it can be accessed using the argument `method = "convolution"` in the `aggregateDist()` function. However, it only works with discrete severity models and computes the distribution of the convolution using (4.2). This method is computationally demanding, and hence, it only works well with relatively small problems. Nevertheless, an advantage is that any discrete distribution can be employed for the claim number. We now present a concrete example: As before, we consider a $Gamma(2, 1)$ distribution for the severities, but now consider a $Bin(10, 0.4)$ distribution for the number of claims.

```
fx <- discretize(pgamma(x, 2, 1),
  method = "unbiased",
  from = 0, to = 22, step = 0.5,
  lev = levgamma(x, 2, 1)
)
fn <- dbinom(0:10, 10, 0.4) # We require a vector of claim number probabilities
Fs <- aggregateDist("convolution",
  model.freq = fn,
  model.sev = fx,
  x.scale = 0.5
)
Fs(10)
```

```
## [1] 0.7282019
```

Once the object above was created, we can again find different quantities of interest. For instance, the density can be found using the `diff()` function:

```
fs <- diff(Fs)
head(fs)
```

```
## [1] 0.007499618 0.006946915 0.011704075 0.016188003 0.020890637 0.025716230
```

Normal approximation

Using the CLT, the cdf of S can be approximated as

$$F_S(x) \approx \Phi\left(\frac{x-\mu_S}{\sigma_S}\right),$$

where $\mu_S = \mathbb{E}[S]$, $\sigma_S^2 = \text{Var}(S)$, and Φ denotes the cdf of a standard normal. However, this approximation ignores the tail behavior of the distribution, which may be problematic when we are interested in the tails. Moreover, we require the existence of at least the second moment. To use this method in the `aggregateDist()` function, we need to pass the argument `method = "normal"` and a vector with the moments of S in the order (μ_S, σ_S^2). For instance,

```
Fs <- aggregateDist("normal", moments = c(6, 2))
Fs(5)
```

```
## [1] 0.2397501
```

Note that the above cdf evaluation can also be computed simply as

```
pnorm(5, mean = 6, sd = sqrt(2))
```

```
## [1] 0.2397501
```

Normal Power II approximation

This method approximates the cdf of S by

$$F_S(x) \approx \Phi\left(-\frac{3}{\gamma_S} + \sqrt{\frac{9}{\gamma_S^2} + 1 + \frac{6}{\gamma_S}\frac{x-\mu_S}{\sigma_S}}\right),$$

where $\gamma_S = \mathbb{E}[(S-\mu_S)^3]/\sigma_S^3$. The approximation is valid for $x > \mu_S$ only and performs reasonably well when $\gamma_S < 1$. This method is implemented in the `aggregateDist()` function and accessible through the argument `method = "npower"`. We require to give a vector with $(\mu_S, \sigma_S^2, \gamma_S)$, in that order. For example,

```
Fs <- aggregateDist("npower", moments = c(6, 2, 0.5))
Fs(7) # Accesible only for x > mu_S
```

```
## [1] 0.7716457
```

Simulation

The last method we introduce here consists of simulating a sample from S and then approximating F_S by the empirical cdf

$$F_n(x) = \frac{1}{n}\sum_{j=1}^{n} 1_{(x_j \leq x)}.$$

The above can be performed using the `aggregateDist()` function with the argument `method = "simulation"`. For instance, for S with N Poisson distributed with mean 10 and $X \sim Gamma(2,1)$ we have

```
set.seed(1)
model_freq <- expression(data = rpois(10))
model_sev <- expression(data = rgamma(2, 1))
Fs <- aggregateDist("simulation",
  nb.simul = 2500,
  model_freq, model_sev
)
```

We can then computer different quantities related to S. For instance, the mean and quantiles can be computed using the `mean()` and `quantile()` functions:

```
mean(Fs)
```

```
## [1] 20.13251
```

```
quantile(Fs)
```

```
##      25%      50%      75%      90%      95%    97.5%      99%    99.5%
## 14.38965 19.37523 25.25919 30.98962 33.85464 36.24842 39.92677 43.65289
```

Other relevant quantities in insurance applications are the Value at Risk (VaR) and the Conditional Tail Expectation. Recall that the VaR of level $\alpha \in (0,1)$, VaR_α, is the quantity satisfying

$$\mathbb{P}(S \leq VaR_\alpha) = \alpha\,,$$

and that the conditional tail expectation of level α, CTE_α, is given by

$$CTE_\alpha = \mathbb{E}[S \mid S > VaR_\alpha]\,.$$

The above quantities can be computed easily using the `VaR()` and `CTE()` function in the `actuar` package.

```
VaR(Fs)
```

```
##      90%      95%      99%
## 30.98962 33.85464 39.92677
```

```
CTE(Fs)
```

```
##      90%      95%      99%
## 34.84512 37.52703 44.26966
```

In fact, `aggregateDist()` implicitly calls the function `simul()` to perform the simulation. We can use the later directly as follows:

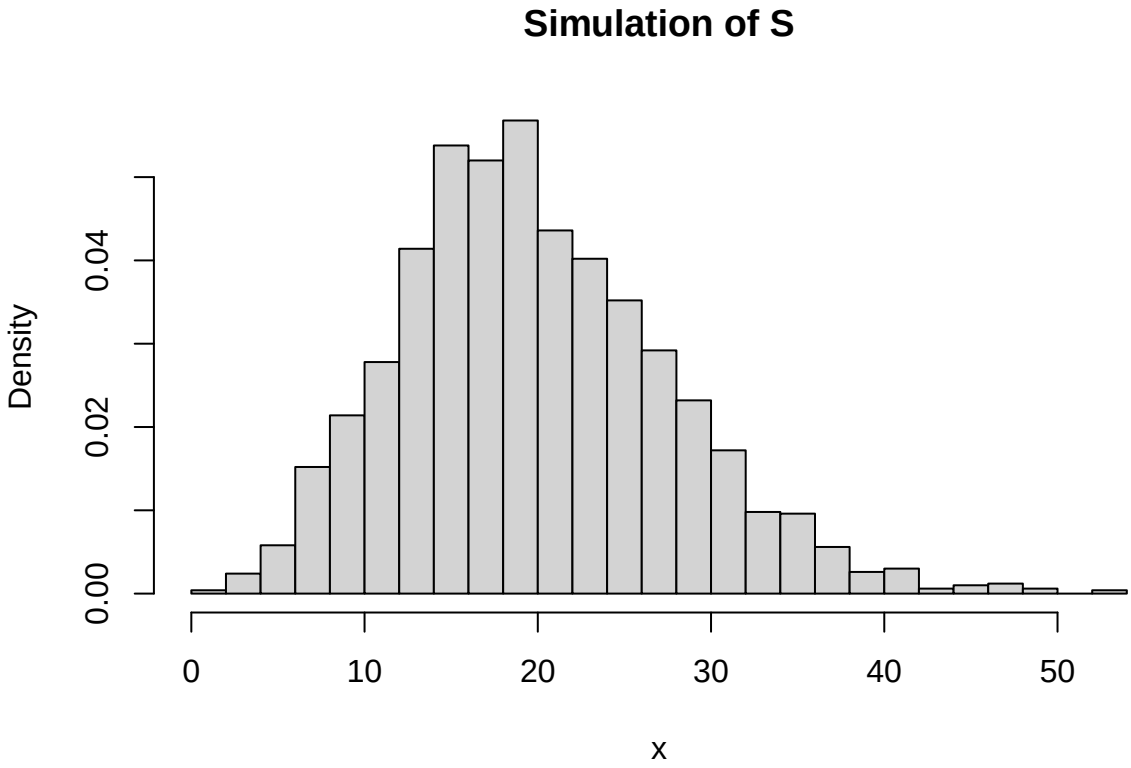
```
sim_s <- simul(2500,
  model.freq = expression(rpois(10)),
  model.sev = expression(rgamma(2, 1))
)
```

These creates an object containing the severities, frequencies and also the aggregate claim amounts. The latter, representing S, can be obtained using the `aggreate()` function.

```
s_sample <- aggregate(sim_s)
summary(s_sample[1, -1])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   14.45   19.03   19.87   24.65   52.91
```

```
hist(s_sample[1, -1], freq = F, breaks = 20, main = "Simulation of S", xlab = "x")
```

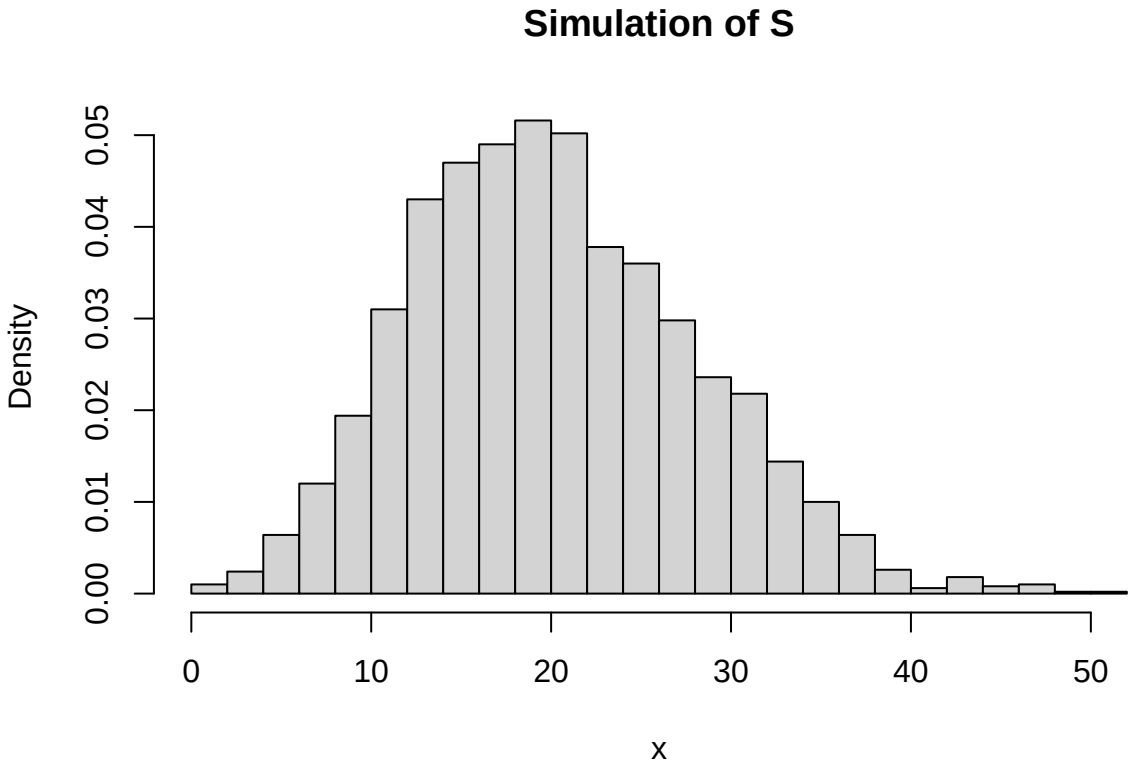


Alternatively, simulation of S can be performed with the following code:

```
s_sim <- function(n, distr, ...) {
  sum(distr(n, ...))
}
set.seed(1)
N <- rpois(2500, 10)
s_sample <- sapply(N, s_sim, distr = rgamma, shape = 2, rate = 1)
summary(s_sample)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.8999 14.3967 19.3932 20.1325 25.2531 51.6455
```

```
hist(s_sample, freq = F, breaks = 20, main = "Simulation of S", xlab = "x")
```



4.2 Ruin theory

In the previous section, we modeled the total claims over a fixed period in an insurance portfolio. In this section, we are interested in describing the evolution of the insurance company's surplus over many periods of time. In particular, we want to find the probability that the surplus becomes negative, in other words, the probability that the ruin of the insurance company occurs. Finding this ruin probability is the main topic of *ruin theory*.

4.2.1 The surplus process

In classical ruin theory, an insurer's surplus at a fixed time $t > 0$ is determined by three quantities: the amount of the surplus at time $t = 0$ (also called initial surplus), the amount of premium income received up to time t, and the amount paid out in claims up to time t. More specifically, let $U(t)$ denote the surplus of an insurance company at time t, $c(t)$ denote the premium received up to time t, and $S(t)$ denote the aggregate claims paid up to time t. If u is the initial surplus at time $t = 0$, then $U(t)$ is given by

$$U(t) = u + c(t) - S(t)\,.$$

As previously mentioned, the ruin of the insurance company occurs when the surplus becomes negative. Thus, the *probability of ruin in infinite time* is defined as

$$\psi(u) = \mathbb{P}(U(t) < 0 \text{ for some } t \geq 0)\,.$$

We now make further assumptions in the surplus process that will allow us to compute (or approximate) the ruin probability. First, we assume that the premium income is continuous over time and that the premium income in any time interval is proportional to the interval length. More specifically, $c(t)$ is given by

$$c(t) = ct\,,$$

where $c > 0$ is a constant rate. Next, we assume that the total amount claimed at time $t \geq 0$, $S(t)$, is of the form

$$S(t) = \sum_{j=1}^{N(t)} X_j\,,$$

where $N(t)$ denotes the number of claims up to time t and X_j denotes the amount of claim j. Note that we follow the convention that $S(t) = 0$ if $N(t) = 0$. Furthermore, we assume that $X_1, X_2, \ldots$ are iid random variables, which are also independent of $N(t)$. We now require a specification for the claim number process $N(t)$. We start by recalling the definition of *counting process*.

Definition 4.1. A stochastic process $\{N(t)\}_{t\geq 0}$ is called a counting process if it satisfies the following properties:

i) $N(0) = 0$.
ii) If $s \leq t$ then $N(s) \leq N(t)$.
iii) $N(t) \in \mathbb{N}$ for all $t > 0$.

Intuitively, a counting process $N(t)$ counts the number of the events that appear in the time interval $(0, t]$. In what follows we denote by $T_1, T_2, \ldots$ the jump times of $N(t)$, that is, $T_k = \inf\{t \geq 0 : N(t) \geq k\}$ for all $k \in \mathbb{N}$.

In classical ruin theory, it is common to assume that $\{N(t)\}_{t\geq 0}$ is a *Poisson process*. In fact, in such a case, the model is often referred to as the Cramer-Lundberg model or classical risk model. We now present the definition of the Poisson process.

Definition 4.2. A counting process $\{N(t)\}_{t\geq 0}$ is called a Poisson process with rate $\lambda > 0$ if:

i) $\{N(t)\}$ has stationary and independent increments.
ii) $\mathbb{P}(N(h) = 0) = 1 - \lambda h + o(h)$ as $h \to 0$.
iii) $\mathbb{P}(N(h) = 1) = \lambda h + o(h)$ as $h \to 0$.

We now give some alternative definitions of the Poisson process.

Proposition 4.2. *Let $\{N(t)\}_{t\geq 0}$ be a counting process. The following are equivalent:*

i) $\{N(t)\}$ is a Poisson process with rate $\lambda > 0$.
ii) $\{N(t)\}$ has independent increments and $N(t) \sim Pois(\lambda t)$ for all $t \geq 0$.
iii) The interarrival times $\{T_k - T_{k-1} : k \geq 1\}$ are independent and exponentially distributed with mean $1/\lambda$.

In what follows we denote the interarrival times by $W_k = T_k - T_{k-1}$. Note that, in particular, iii. implies that a Poisson process is a particular case of a *renewal process*.

Definition 4.3. A counting process $\{N(t)\}_{t\geq 0}$ is called *ordinary renewal process* if its interarrival times $\{W_k : k \geq 1\}$ are iid.

When $\{N(t)\}_{t\geq 0}$ is a renewal process, the model is called *renewal risk model* or *Sparre Andersen model*.

To study the ruin probability $\psi(u)$, we require further assumptions regarding the premium rate $c > 0$. First, one can show that

$$\psi(u) = 1 \quad \iff \mathbb{E}[cW - X] < 0\,.$$

Thus, in order to avoid ruin with probability one, we assume that

$$c > \mathbb{E}[X]/\mathbb{E}[W]\,,$$

which is known as the *net profit condition*. Note that in the classical risk model (i.e., $W \sim Exp(\lambda)$), the net profit condition reads as

$$c > \lambda\mathbb{E}[X]\,,$$

which implies that

$$ct > \mathbb{E}[S(t)]\,,$$

that is, the premium income is above the expected value of aggregated losses for any time $t \geq 0$. In such a case, c is typically described in terms of a *security loading* $\theta \geq 0$ such that

$$c = (1+\theta)\lambda\mathbb{E}[X]\,.$$

Finally, one can show that under the net profit condition

$$\psi(u) \to 0 \quad \text{as} \quad u \to \infty\,.$$

In what follows, we assume that the net profit condition is satisfied.

4.2.2 The adjustment coefficient

The quantity known as the *adjustment coefficient* is a valuable tool in risk theory that allows us to find a bound for the ruin probability. More specifically, the adjustment coefficient R is defined as the smallest strictly positive solution (if it exists) to the *Lundberg equation*

$$h(t) = \mathbb{E}\left[\exp(tX - tcW)\right] = 1\,,$$

where $c > 0$ satisfies the net profit condition. Under the assumption of independence between X and W, as in the most common models, the equation can be rewritten as

$$h(t) = M_X(t)M_W(-tc) = 1\,,$$

In general, it is not possible to explicitly solve the above equation for R. Usually, one must resort to numerical methods, many of which require an initial guess about the value of R. An upper bound of the adjustment coefficient in the classical risk model (there are many) is given by

$$R \leq \frac{2(c-\lambda\mathbb{E}[X])}{\lambda\mathbb{E}[X^2]} = \frac{2\theta\mathbb{E}[X]}{\mathbb{E}[X^2]}\,.$$

In R, we can use the `adjCoef()` function from the `actuar` package to compute the adjustment coefficient. We require the following arguments: the two moment generating functions $M_X(t)$ and $M_W(t)$ (thereby assuming independence), the premium rate c, and the upper bound of the support of $M_X(t)$ or any other upper bound for R. For example, if $W \sim Exp(2)$, $X \sim Exp(1)$ and the premium rate is $c = 2.4$, then the adjustment coefficient is

```
adjCoef(
  mgf.claim = mgfexp(x), mgf.wait = mgfexp(x, 2),
  premium.rate = 2.4, upper.bound = 1
)
```

```
## [1] 0.1666667
```

In the above solution, we passed the upper bound for the support of $M_X(t)$. However, since we are in the classical risk model setup, we could have also used the upper bound of R described previously

```
exp_aux <- function(x) {
  x * dexp(x, 1)
}
snd_aux <- function(x) {
  x^2 * dexp(x, 1)
}
exp_x <- integrate(exp_aux, 0, Inf)$value
exp_x
```

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

```
snd_x <- integrate(snd_aux, 0, Inf)$value
snd_x
```

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

```
c <- 2.4
lambda <- 2
bound <- 2 * (c - lambda * exp_x) / (lambda * snd_x)
bound
```

```
## [1] 0.2
```

```
R <- adjCoef(
  mgf.claim = mgfexp(x), mgf.wait = mgfexp(x, 2),
  premium.rate = 2.4, upper.bound = bound
)
R
```

```
## [1] 0.1666667
```

As previously mentioned, knowledge of the adjustment coefficient allows computing a bound for the ruin probability as described in the following result.

Theorem 4.2 (Lundberg's inequality)**.** *For $u \geq 0$, the ruin probability, $\psi(u)$, has an exponential upper bound, given by*

$$\psi(u) \leq \exp(-Ru)\,,$$

where R is the adjustment coefficient.

In our last example, we can then compute a bound for the ruin probability as (assuming $u = 3$)

```
u <- 3
exp(-R * u)
```

```
## [1] 0.6065307
```

4.2.3 Probability of ruin

Explicit calculation of the infinite time probability of ruin is a difficult task except for the most simple models. For example, if the interarrival times W are $Exp(\lambda)$ distributed (i.e., the claim numbers are described using a Poisson process) and the claim amounts X are $Exp(\beta)$ distributed, then

$$\psi(u) = \frac{\lambda}{c\beta} \exp((\beta - \lambda/c)u)\,, \quad u \geq 0\,.$$

In the model above, although the frequency assumption can be justified, the severity assumption can hardly be used to describe real-life problems making the model mainly an illustration tool. Fortunately, other generalizations of the exponential distribution, such as mixtures of exponentials and Erlang (and more generally, phase-type distributions), also allow for a closed-form solution for the ruin probability. The function `ruin()` of the `actuar` package allows computing the ruin probability for such cases. First, one needs to specify the claim amount and interarrival times models with any combination of "exponential", "Erlang" (and "phase-type"). Then, one passes the parameters of each model using lists with components named after the corresponding parameters of `dexp`, `dgamma` (and `dphtype`). If an argument `weights` is provided, the model is a mixture of exponential or Erlang. Let us illustrate first an exponential/exponential model with premium rate $c = 1$ (default):

```
psi <- ruin(
  claims = "e", par.claims = list(rate = 5),
  wait = "e", par.wait = list(rate = 3)
)
psi(0:10) # Evaluates the ruin probability for initial surplus from 0 to 10.
```

```
##  [1] 6.000000e-01 8.120117e-02 1.098938e-02 1.487251e-03 2.012776e-04
##  [6] 2.723996e-05 3.686527e-06 4.989172e-07 6.752110e-08 9.137988e-09
## [11] 1.236692e-09
```

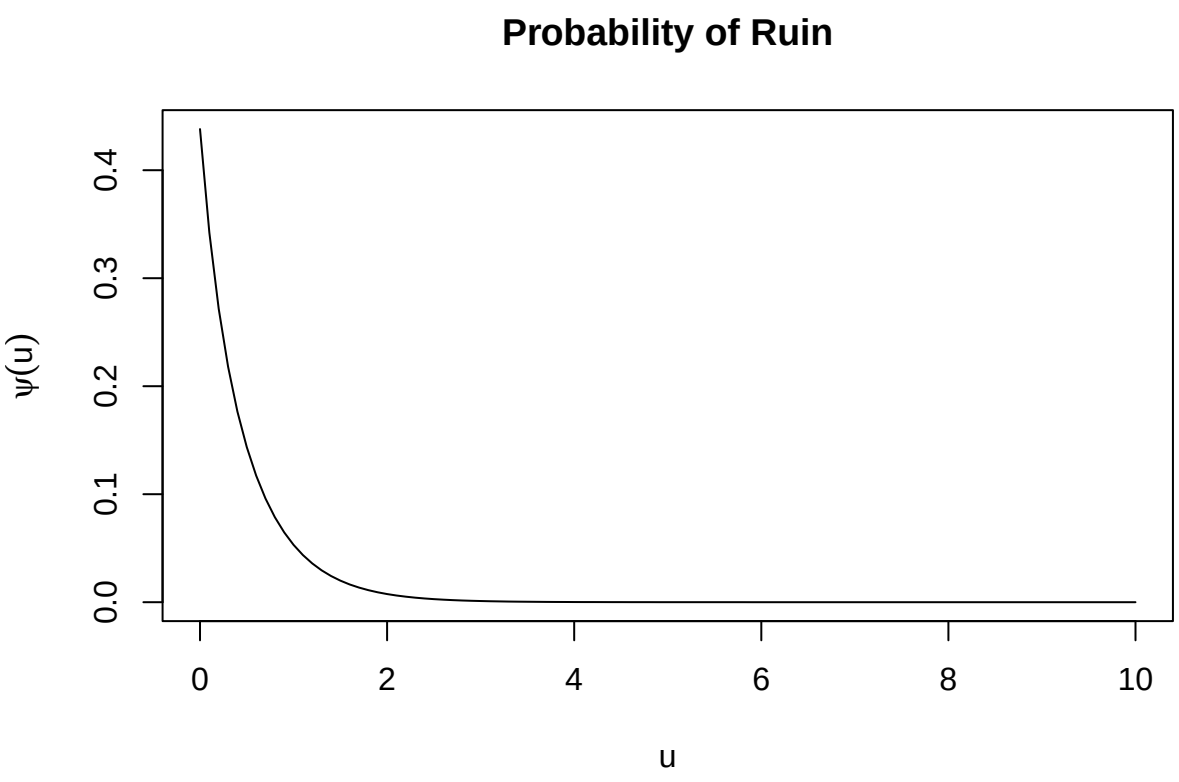
Next, we consider a model with mixture of two exponentials for the claim amount, exponential interarrival times, and premium rate $c = 1.5$.

```
psi <- ruin(
  claims = "e", par.claims = list(rate = c(3, 7), w = c(0.4, 0.6)),
  wait = "e", par.wait = list(rate = 3),
  pre = 1.5
)
psi(0:10)
```

```
##  [1] 4.380952e-01 5.310603e-02 7.529333e-03 1.070525e-03 1.522149e-04
##  [6] 2.164303e-05 3.077365e-06 4.375623e-07 6.221582e-08 8.846302e-09
## [11] 1.257832e-09
```

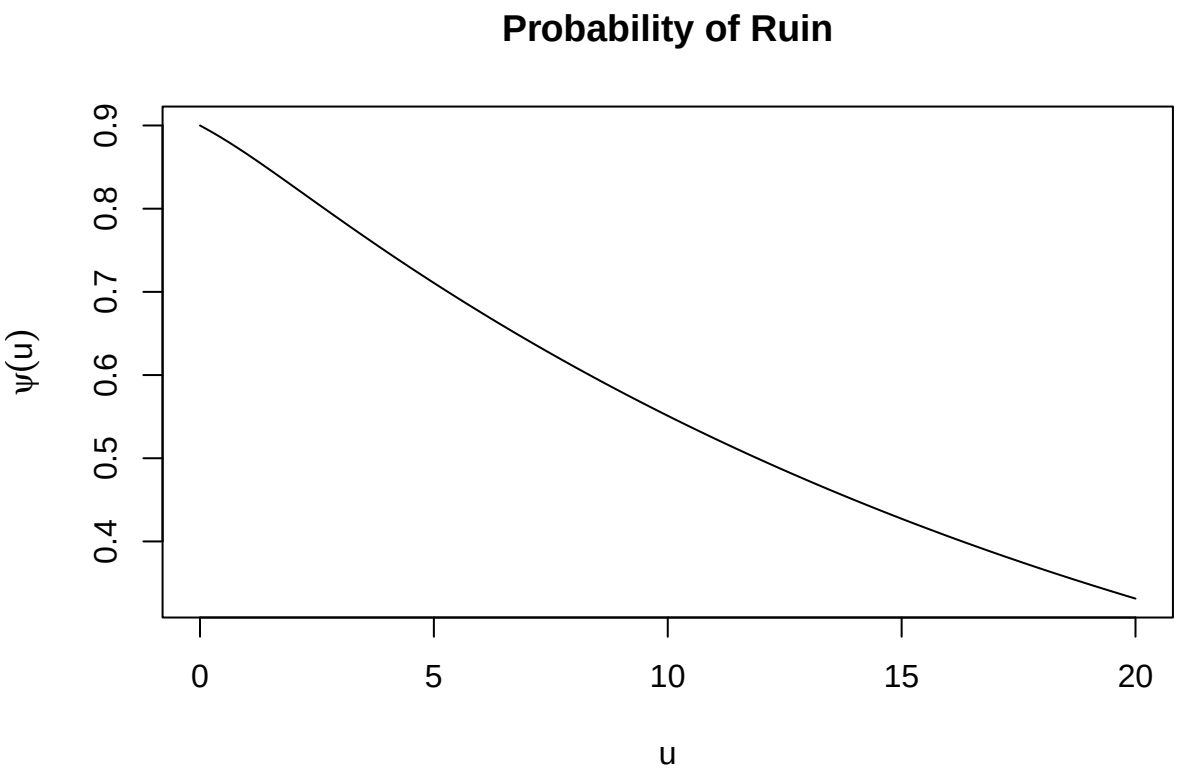
Moreover, we can plot the ruin probability straightforwardly as a function of the initial surplus using the `plot()` function as follows:

```
plot(psi, from = 0, to = 10)
```



Finally, we consider a model with Erlang claim amounts and exponentials interarrival times:

```
psi <- ruin(
  claims = "E", par.claims = list(shape = 3, rate = 1),
  wait = "e", par.wait = list(rate = 3),
  pre = 10
)
plot(psi, from = 0, to = 20)
```



4.2.4 Reinsurance

Reinsurance means that the company (the cedent) insures a part of the risk at another insurance company (the reinsurer). In this subsection, we study how different types of reinsurance can affect the adjustment coefficient and the probability of ruin. In what follows, we consider the classical risk model unless it states otherwise.

Let X (≥ 0) denote the claim amount under a reinsurance-free environment. A reinsurance arrangement is then defined in terms of a function $h(x)$ with the property $0 \leq h(x) \leq x$. Here $h(x)$ is the amount of the claim x to be paid by the cedent and $x - h(x)$ the amount to be paid by the reinsurer. The most common examples are the following two:

- **Proportional reinsurance**. $h(x) = ax$ for some $a \in [0, 1]$. Also called quota share reinsurance.
- **Excess-of-loss reinsurance**. $h(x) = \min(x, m)$ for some $m \in (0, \infty)$, referred to as the retention limit.

Next, assume that in a reinsurance-free setting, there is a security loading $\theta \geq 0$, such that the premium rate is

$$(1 + \theta)\lambda\mathbb{E}(X)\,.$$

If we assume that under the reinsurance agreement h, the reinsurer receives a premium rate determined by a security loading $\theta_h \geq 0$, that is,

$$(1 + \theta_h)\lambda\mathbb{E}\big(X - h(X)\big)\,.$$

Then, the premium rate c^* for the cedent is

$$c^* = (1 + \theta)\lambda\mathbb{E}[X] - (1 + \theta_h)\lambda\mathbb{E}[X - h(X)]\,.$$

Furthermore, we require

$$c^* > \lambda\mathbb{E}\big[h(X)\big]\,,$$

which plays the role of our new net profit condition. Thus, the surplus process of the cedent, $\{U^*(t)\}_{t\geq 0}$, is given by

$$U^*(t) = u + c^*t - \sum_{j=1}^{N(t)} h(X_j)\,, \quad t \geq 0\,.$$

Now, let R_h denote the adjustment coefficient under the h reinsurance agreement, meaning that R_h is the unique positive solution to the equation

$$1 = M_{h(X)}(t) M_W(-c^* t)\,,$$

provided that $M_{h(X)}(t) = \mathbb{E}[\exp(th(X))]$, exists.

Thus, the insurer's ultimate ruin probability upper bound is given by

$$\psi_*(u) \leq \exp(-R_h u)\,, \quad u \geq 0\,,$$

where ψ_* is the probability of ruin corresponding to the surplus process $\{U^*(t)\}_{t\geq 0}$.

We now study in more detail the proportional and excess-of-loss reinsurance agreements, which are implemented in the `actuar` package.

Proportional reinsurance

Under the proportional reinsurance, h is given by

$$h(X) = aX\,, \qquad 0 \leq a \leq 1\,.$$

Thus, the relative security loading θ_h satisfies the equation

$$c^* = \Big((1+\theta) - (1+\theta_h)(1-a)\Big)\lambda \mathbb{E}[X]\,.$$

It follows from the above equation and the net profit condition that

$$a > 1 - \frac{\theta}{\theta_h}\,.$$

In other words, the insurer must retain at least the proportion $1 - \theta/\theta_h$ in order to avoid ultimate ruin with probability one.

In this case, the adjustment coefficient R_h satisfies the equation

$$1 = M_{aX}(t) M_W(-c^* t) = M_X(at) M_W(-c^* t)\,.$$

Moreover, an upper bound of the adjustment coefficient is given by:

$$R_h \leq \frac{2(c^* - a\lambda \mathbb{E}[X])}{\lambda a^2 \mathbb{E}[X^2]}\,,$$

We now provide an example, which can be solved using the `adjCoef()` function.

Example 4.2. Consider the following classical risk model under proportional reinsurance: The claim amounts are exponentially distributed with mean 1, the Poisson rate is $\lambda = 2$, and the safety loadings are $\theta = 0.2$ and $\theta_h = 0.3$.

a) Find the adjustment coefficient R_h if $a = 0.75, 0.8, 0.9, 1$.
b) Plot the adjustment coefficient as a function of the proportion a.
c) Find an upper bound for the ruin probability if $a = 0.5$ and $u = 2$

Solution.

a)

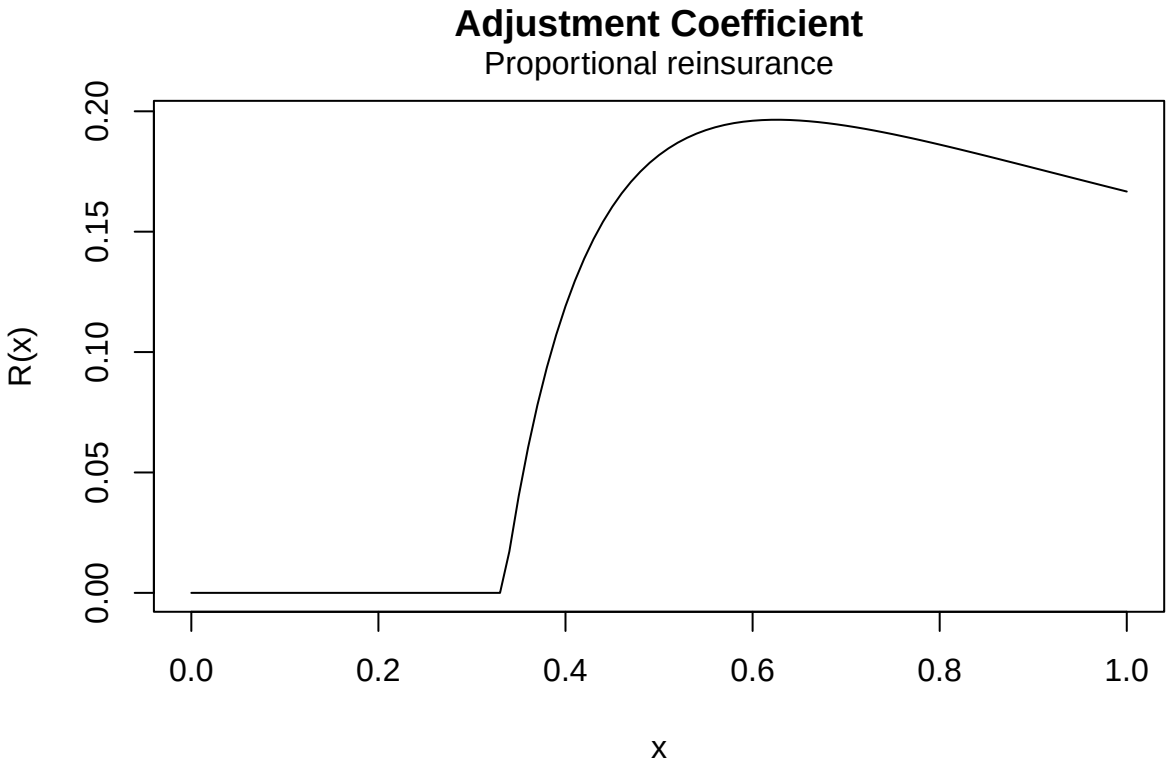
```
lambda <- 2
theta <- 0.2
thetah <- 0.3

# We require a function to compute the premium rate for different values of a
prem <- function(x) {
```

```
  ((1 + theta) - (1 + thetah) * (1 - x)) * lambda
}

# We require need a function to compututе the mgf of aX for different values of a
mgf_ax <- function(x, y) {
  mgfexp(x * y)
}

adj_prop <- adjCoef(mgf_ax,
  mgf.wait = mgfexp(x, 2),
  premium.rate = prem,
  upper = 1,
  reins = "prop", from = 0, to = 1
)
adj_prop(c(0.75, 0.8, 0.9, 1))
```

```
## [1] 0.1904762 0.1861702 0.1765317 0.1666667
```

b)

```
plot(adj_prop)
```



c)

```
exp(-adj_prop(0.5) * 2)
```

```
## [1] 0.6951439
```

Excess-of-loss reinsurance

Under the excess-of-loss reinsurance with retention level m, h is given by

$$h(X) = \min(X, m)\,.$$

Thus, the relative security loading, θ_h, satisfies the equation

$$\begin{aligned} c^* &= (1+\theta)\lambda\mathbb{E}[X] - (1+\theta_h)\lambda\mathbb{E}[X - \min(X,m)] \\ &= (1+\theta_h)\lambda\mathbb{E}[\min(X,m)] - (\theta_h - \theta)\lambda\mathbb{E}[X]\,, \end{aligned}$$

with

$$c^* > \lambda\mathbb{E}[\min(X,m)]\,.$$

Further, the adjustment coefficient, R_h, is the solution to the following equation

$$1 = M_{\min(X,m)}(t)M_W(-tc^*)\,.$$

Note that

$$\begin{aligned} M_{\min(X,m)}(t) &= \mathbb{E}[\exp(t\min(X,m)] \\ &= \int_0^m \exp(tx)f_X(x)dx + \exp(tm)(1 - F_X(m))\,. \end{aligned}$$

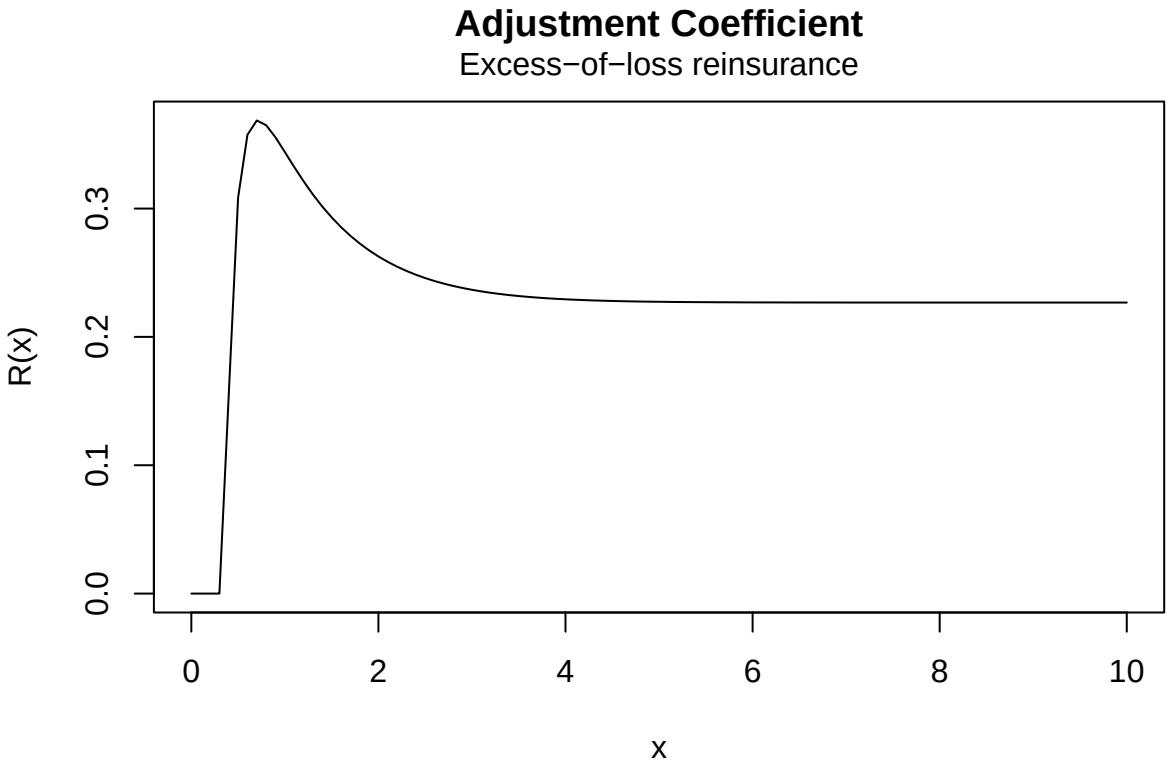
Example 4.3. Consider the following classical risk model under excess-of-loss reinsurance: The claim amounts are Gamma distributed with shape parameter 2 and rate parameter 2, the Poisson rate is $\lambda = 1$, and the safety loadings are $\theta = 0.2$ and $\theta_h = 0.3$. Plot the adjustment coefficient as a function of the retention limit m varying from 0 to 10.

Solution. We first note that for $X \sim Gamma(\alpha, \lambda)$

$$\int_0^m \exp(tx)f_X(x)dx = \left(\frac{\lambda}{\lambda - t}\right)^\alpha F_{\tilde{X}}(m)\,,$$

where $\tilde{X} \sim Gamma(\alpha, \lambda - t)$. Thus,

```
prem <- function(x) {
  1.3 * levgamma(x, 2, 2) - 0.1
}
mgfx <- function(x, l) {
  mgfgamma(x, 2, 2) * pgamma(l, 2, 2 - x) +
    exp(x * l) * pgamma(l, 2, 2, lower = FALSE)
}
adj_eol <- adjCoef(mgfx,
  premium = prem,
  upper = 1,
  reins = "excess-of-loss",
  from = 0, to = 10
)
plot(adj_eol)
```


Adjustment Coefficient
Excess−of−loss reinsurance
R(x)
0.0
0.1
0.2
0.3
0
2
4
6
8
10
x