Mendel University in Brno

# Probability and Statistics with Programming in R Study text

Aleš Kozubík
Žilinská univerzita v Žiline

Project: Innovative Open Source Courses for Computer Science Curriculum

24. 6. 2022

Co-funded by the Erasmus+ Programme of the European Union
West Pomeranian University of Technology Szczecin
UNIVERSITY OF ŽILINA
Mendel University in Brno

Reviewer: RNDr. Eva Rakovská, PhD., University of Economics in Bratislava, Slovakia
Project: Innovative Open Source Courses for Computer Science Curriculum
© Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic
ISBN 978-80-7509-896-2 (online ; pdf)
DOI https://doi.org/10.11118/978-80-7509-896-2

This material teaching was written as one of the outputs of the project "Innovative Open Source Courses for Computer Science Curriculum", funded by the Erasmus+ grant no. 2019-1-PL01-KA203-065564. The project is coordinated by West Pomeranian University of Technology in Szczecin (Poland) and is implemented in partnership with Mendel University in Brno (Czech Republic) and University of Žilina (Slovak Republic). The project implementation timeline is September 2019 to December 2022.

## Project information

Project was implemented under the Erasmus+.
Project name: "Innovative Open Source courses for Computer Science curriculum"
Project nr: 2019-1-PL01-KA203-065564
Key Action: KA2 – Cooperation for innovation and the exchange of good practices
Action Type: KA203 – Strategic Partnerships for higher education

### Consortium

ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY W SZCZECINIE
MENDELOVA UNIVERZITA V BRNĚ
ŽILINSKÁ UNIVERZITA V ŽILINE

### Erasmus+ Disclaimer

### Copyright Notice

Co-funded by the
Erasmus+ Programme
of the European Union

# Contents

# Introduction to R

One can characterise R as an environment for statistical computing and graphics. It represents an open source solution to data analysis, and it is free to install under the GNU general public license.From a programming point of view it is a scripting language. In this case, a simple text editor is enough for the programmer, so he does not need any special compiler or development tool.

Besides statistics, R is used in several other areas like the social sciences or biology. For example, in finance and banking, it is used to detect and identify fraud, in bioinformatics in drug development, or social media analysis when searching for potential customers for online targeting advertising campaigns. As well many renowned companies such as Facebook, Google, Linkedin, IBM £ and Twitter use R in their analytical work.

Due to its popularity, the R language has a number of advantages. Let us note at least some of them:

- it is free, most of the statistical software platforms cost thousands of dollars,
- the program has a huge collection of available packages,
- R can easily import data from a wide variety of sources,
- R contains numerous advanced statistical routines,
- R provides interactive platform for data analysis,
- R environment offers data visualisation in the form very high quality and aesthetic graphs,
- it is platform independent, it is compatible with all most frequently used operating systems,
- it is highly compatible with the programming languages like C, C++, Python, Java.

The R language is considered to be the most widespread within the statistical community. This is the main reason why it is dominant among other programming languages in the development of statistics tools. Thanks to this expansion, there is a huge community developers, users and programmers who are willing to help and share their knowledge with others.

## 1.1 Installing R

### 1.1.1 Base installation

How we have already mentioned, R is freely available from the Comprehensive R Archive Network (shortly CRAN). Its internet location is `https://cran.r-project.org/`. Here are at disposal pre-compiled binaries for all common platforms Linux, Mac OS, and

Windows. Here you can select the most suitable mirror for downloading the installation package. After downloading it follow the directions for installing the core on the OS platform you are running.

In the further text we assume working on Linux platform. Because the core of the R environment is incorporated in the Linux distributions, the Linux must not necessary download the binaries from the CRAN. They but can use the packaging systems like `apt`, `rpm`, `yum` etc. The R core is included in the package `r-base-core`. The extensions to the base we find in packages `r-cran-*`, where `*` means suitable suffix.

Once we have installed R, we can try if it works correctly. We start the R environment simply from command prompt typing:

```
username@host:~$ R
```

If you are using Win or MacOs, you start R by double-click on R icon or you launch it from Start Menu. Any of these will start the R interface. It contains short introductory note that is followed by the sign > assigning the R prompt. The R environment is ready for an interactive mode of work.

```
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

In order to finish work in R environment we simply type (let us note, that parentheses are obligatory as `q` is function):

```
> q()
```

R reacts by question:

```
Save workspace image? [y/n/c]:
```

Answering our choice `y` for yes, or `n` for no we enclose the R environment. If we select `y`, the whole working history is saved in the file `.Rhistory` saved in the working directory. The last option `c` is for cancel, and we can continue in the R environment without leaving it.

### 1.1.2 Installing packages

R comes with a huge set of packages extending its base core. Packages include reusable R functions, the documentation that describes how to use them, and sample data. They increase the power of R by improving existing base R functionalities, or by adding new ones.

To install package we use the function `install.packages()`. Entering this command without any option shows the window including the list of feasible packages. Selecting one of them will download and install it. If we know the name of the package we plan to install, we can submit this name as argument of the function. For example, package `pwr` contains functions for the power analysis. We can download and install it by the console command:

```
> install.packages("pwr")
```

Of course, if we want to make the package accessible for all users it is necessary to install them as `root`.

## 1.2 First steps, R as a calculator

We can characterise R as a case-sensitive, interpreted language. we can run it by commands at the R command prompt > or we can run a scripts, that are a sets of commands saved in the source file. At first, we turn our attention to the R console.

### 1.2.1 The R workspace and navigation

The R workspace is the current R environment, that includes any objects defined by user. how we mentioned above, this workspace can be saved when leaving the R session and is reloaded during the next R environment start. In this regime, we enter all command interactively at the command prompt. scrolling through the commands history is enabled by using the up and down arrow keys. This allows to submit a previous commands without retyping it. We only select the desired and submit it repeatedly using the Enter key.

An important part of work in any environment is communication with the operating system and navigation among the directories and files in the computer. The default working directory is the directory where R was started. In this current working directory R reads and saves files and results. The actual working directory we can fin using the `getwd()` function.

```
> getwd()
[1] "/home/user/R_programming"
```

The current working directory can be changed using the `setwd()` function. It is important that the required directory exists. In opposite case we obtain an error, how illustrates following example.

```
> setwd("new")
 Error in setwd("new") : cannot change working directory
```

In this situation we have to create the directory `new` at first. We apply the `system()` function. This function has several arguments. The first one is `command` for inserting the shell command that is put in quotation marks. Let us mention some of the other arguments:

`wait` a logical argument indicates whether the R interpreter should wait for the command to finish, or run it asynchronously,

`timeout` a limit for the elapsed time running command in a separate process, integer number of seconds,

`input` character vector is supplied, this is copied one string per line to a temporary file, and the standard input of command is redirected to the file.

In our situation, when the subdirectory `new` does not exist, we create it at first and then we set this subdirectory as the working one.

```
> system("mkdir␣new")
> setwd("new")
> getwd()
[1] "/home/user/R_programming/new"
```

### 1.2.2 Getting help

Like most Open Source projects as well includes extensive facilities for accessing documentation and searching for help. The built-in help system contains detailed references and example for any function defined in an installed package. The general function for getting help has a simple form `help()`, or shortly in the operator form `?`.

Let us assume, we want to get help about the trigonometric function `sin()`. We enter at the R console

```
> help(sin)
```

or

```
> ?sin
```

The answer we get in the unified standardised form:

- In the head we find the name of the function in the left and `package:<name>` in the middle. In our case we can see `Trig`, as this help is common for all trigonometric functions and `package:base` what indicates the trigonometric functions are contained in the R base.
- The next part of the answer is `Description`. Here is briefly described how the function works.
- The concrete syntax of the function entering is introduced in the part `Usage`.
- Part `Arguments` brings more detailed description of the arguments of the function and their data type.
- Documentation for some functions contains also part assigned as `Details`. This part provides additional information about the feasible arguments values, compatibility and similar.
- Part `Value` explains the values we get as an answer of the function.
- In `References` we see some bibliography concerned in the requested function.

- Examples of using the function are presented in the enclosing part `Examples`. It is useful especially when learning to work with the function. If we want to see only the examples of the use, we can apply the function `example()` with the name of teh function inserted in quotes.

We can use the function `help()` also to get some information about the additional packages. To get help about any package, we enter the argument `package` of the `help()` function that contains the name of the package in the quotes. Let us suppose, we need for example help to the package `survival` including the tools for the survival analysis. Then we enter our request in the form

```
> help(package="survival")
```

Some packages include also code demonstrations. Function `demo()` with given argument `package` set to the package name in quotes lists these demonstrations. For example the demos of the package `stats` we get by command

```
> demo(package="stats")
```

The concrete demonstration of linear and generalised linear modelling we get running the `demo()` function with argument `lm.glm` that we found in output from previous command

```
> demo(lm.glm)
```

It frequently happens, we need some general information to given theme. In such case we use the function `help.search()`, whose argument is the string containing the theme we are interested in. This function searches the given theme in all actually installed packages. To search information to the theme of survival analysis, we enter the command:

```
> help.search("survival")
```

Let us note, that the function `help.search()` can be substituted by double question marks. So we could equivalently search by command:

```
> ?? "survival"
```

Many packages include vignettes. These vignettes are discursive documents which illustrate and explain facilities in the package. The function `vignette()` displays the list of the vignettes in the package. For example the vignettes of the `pwr` package that is developed for the power analysis we get by:

```
> vignette(package="pwr")
Vignettes in package 'pwr':

pwr-vignette            Getting started with the pwr package
                        (source,html)
```

All introduced help function are summarised in the table 1.1

### 1.2.3 Functions for managing the workspace

Under the workspace we mean the current R working environment. The workspace refers to all the variables and functions (collectively called objects) that user creates during an

Table 1.1: The functions for getting help in the R environment

| *Function* | *Action* |
|---|---|
| `help("fun")` | Help on function `"fun"` |
| `?fun` | Help on function `"fun"` |
| `help.search("string")` | Search the help system for instances of the `string` |
| `??string` | Search the help system for instances of the `string` |
| `demo(package="name")` | Lists demos in a particular package `name` |
| `demo(demo_name)` | Runs the concrete demo `demo_name` from the demos list |
| `example("fun")` | Examples of using the function `fun` |
| `vignette()` | List of all available vignettes for currently installed packages |
| `vignette("name")` | Display specific vignettes for topic `name` |

R session, as well as any packages that are loaded. when enclosing the R session, one can save an image of the actual workspace. This image is automatically reloaded on the next R start.

When quitting R by `q()` command, we get the question like this:

```
> q()
Save workspace image? [y/n/c]:
```

Answering `y`, the workspace is saved in the file `.RData` in the current working directory. All commands are archived in the additional file `.Rhistory` we have already mentioned.

Often we need to manage the workspace. Sometimes we need to remind the names of the variables or functions we have created. We can also need to reload the history of commands from another file than `.Rhistory` and similar. The functions useful in workspace managing are listed in table 1.2

Table 1.2: The functions for managing the R workspace.

| *Function* | *Action* |
|---|---|
| `getwd()` | List the current working directory. |
| `setwd("path")` | Setting the current working directory to `path`. |
| `ls()` | List the objects in the current workspace. |
| `rm(objects)` | Removes the given objects from the workspace. |
| `history(value)` | Display last `value` commands, default `value` is 25. |
| `savehistory("file")` | Save the command history in selected `file`. |
| `loadhistory("file")` | Reload the command history from selected `file`. |
| `save.image("file")` | Save the workspace to `file`. |
| `load("file")` | Load a workspace into the current session from `file`. |

### 1.2.4 R as calculator

We turn attention to the simplest use of R console, means use R as an calculator. Console prompt enables interactively compute the operations and functions, as well as to create and use objects. The R prompt starts with the sign > and how we have mentioned, the commands run after pressing the key Enter.

So we can conduct simple calculations with numbers, for example:

```
> 5+3
[1] 8
```

The label `[1]` tell us, which component of the output we are looking at. This is not very interesting at this moment, as our output has only one component. When the command is completed, the output is followed by new prompt > telling us that it is ready for the next command. If we don't see new prompt, it may be because we entered an incomplete command. Let us see on the next case:

```
> 5-
+
```

In this case R responds by + what means we have to type the rest of the command and then press Enter. Alternatively we can press Esc in order to cancel the command and go back to the prompt. So, the previous calculation can be completed as follows:

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

List of the arithmetic operations is presenter in table 1.3. The following listing illustrates the use of some operations.

```
> 5*(-3)
[1] -15
> 23/7
[1] 3.285714
> 23%%7
[1] 2
> 23%/%7
[1] 3
> 2^10
[1] 1024
```

Table 1.3: List of the arithmetic operators.

| *Symbol* | *Operation* | *Symbol* | *Operation* |
|---|---|---|---|
| `+` | Addition. | `^` | Exponentiation. |
| `-` | Subtraction. | `%%` | Modulus (Remainder from integer division). |
| `*` | Multiplication. | `%/%` | Integer division. |
| `/` | Division. | | |

Besides the arithmetic operations R brings a set of relational operators whose results are logical values. Their list is presented in table 1.4 and their use we illustrate in the next listing.

```
> 5<3
[1] FALSE
> 10>=8
[1] TRUE
> 2*3==6
[1] TRUE
```

Table 1.4: List of the relational operators.

| *Symbol* | *Operation* | *Symbol* | *Operation* |
|---|---|---|---|
| < | Less than | > | Greater than |
| <= | Less than or equal to | >= | Greater than or equal to |
| == | Equal to | != | Not equal to |

If the calculation is composed from several arithmetic operators, they are evaluated in the usual order. That means at first is computed exponentiation followed by division, then multiplication and in the end subtraction and addition. certainly the operations order can be changed using parentheses, how is common practice. We can see it in the different results in the following example.

```
> 6^2-12/3+11
[1] 43
> (6^2-12)/3+11
[1] 19
```

In addition to the operators R has numerous built-in functions. These functions enable performing a variety tasks. Many of them are specialised to performing deep statistical analysis and we will deal with them in the later lessons. On this place, we introduce only elementary mathematical functions and computing the elementary sample characteristics.

In order to use any function, we type its name followed by parentheses. Some functions require submitting one or more arguments placed between the parentheses. As an example of the function, that does not reqire any argument is the function `date()` which gives the actual computer date and time.

```
> date()
[1] "Wed Mar  3 12:50:12 2021"
```

As an example of function with one argument, we can introduce trigonometric function `sin()` that gives the value of sine of the given angle. Let us suppose, we want to get the value of $\sin 30^\circ$. As the `sin()` function requires the argument to be expressed in the radians, we have to transform the angle to value $\pi/6$. Then we can compute the function value. To illustrate it, we can compare two results (the second one is the result we requested).

```
> sin(30)
[1] -0.9880316
> sin(pi/6)
[1] 0.5
```

Some functions can contain more arguments that are frequently voluntary and if not submitted, the default values are used. As an example we can introduce the logarithmic function `log()`. Its obligatory argument is the number whose logarithm is to be evaluated and the voluntary argument is the base of the logarithm. The default base is the Euler constant e, so the function provides the natural logarithm. Let us see the example.

```
> log(10)
[1] 2.302585
> log(10,base=10)
[1] 1
```

Another function with optional arguments is the function`round`, that rounds its argument to the nearest integer. Typing the optional argument `digits` we can state the number of the decimal places.

```
> round(pi)
[1] 3
> round(pi,digits=3)
[1] 3.142
```

Let us note, that there is no obligatory order of the arguments, if we declare which values belong to them. On the other hand, we can write the arguments without specifying their purpose, if we save the suggested order. How we can see, the following commands are equivalent.

```
> round(pi,3)
[1] 3.142
> round(digits=3,pi)
[1] 3.142
```

The list of the most frequently used mathematical functions we find in table 1.5.

Table 1.5: List of the most frequently used mathematical functions.

| *Function* | *Purpose* | *Function* | *Purpose* |
|---|---|---|---|
| `exp()` | Exponential | `log()` | Logarithm (default natural) |
| `log10()` | Logarithm with base 10 | `sqrt()` | Square root |
| `sin()` | Sine | `asin()` | Arc sine |
| `cos()` | Cosine | `acos()` | Arc cosine |
| `tan()` | Tangent | `atan()` | Arc tangent |
| `abs()` | Absolute value | `round()` | Rounding (default to integer) |

### 1.2.5 Objects

R is object-oriented language. Also, everything in R is an object and it represents some data that has been stored in memory. with given name. Objects can be given any name but there are some rules that must be respected:

- the name consists only of lower or upper case letters, numbers, underscores and dots,
- the name begins with upper or lower case letter,
- R is case sensitive (it means A and a are two different objects),

- the name must nor be any of the R's reserved words (the list of them is visible after entering `help(reserved)`),

The objects in R can contain data of different type. Besides values or strings, it can be any data type or structure like for example function or graph. R environment does not require any data type specification when initialising a new object. We create a new object simply with the assignment operator in the form of left arrow (<-). The arrow points to the object name and on its second side is the value we want to assign to the object. throughout the right arrow works in the same manner as left arrow, it is recommended to avoid using the right arrows, to keep the clarity of the code.

We can create the object `height_cm` and then print its value:

```
> height_cm<-185
> height_cm
[1] 185
```

We can equivalently create a new object also using the sign "to be equal" (=). But here are situations when this kind of assignment does not work. One of such cases is using the equality sign as an argument of the function. We see it in the following example.

```
> log(x=25,base=5)
[1] 2
> x
Error: object 'x' not found
> log(x<-25,base=5)
[1] 2
> x
[1] 25
```

In the first case, we only submitted the argument for the logarithmic function `x=25`, while in the second case we have at the same time created the object `x`. There is also third possibility how to create a new object. We can use the function `assign()`. The name of the object is to be put in the quotation marks and its value is given by the second argument of the function. However, this method is seldom used to create new objects.

```
> assign("y",100)
> y
[1] 100
```

Once we created the object, we can use its name for any calculations. For example we can recalculate the height given in cm into inches as follows.

```
> height_cm/2.54
[1] 72.83465
```

When creating a new object, it is important to keep in mind, that if we choose an object name that is already in use, we overwrite the old value without any warning. Let us note, that in addition to the reserved words, here are also some predefined constants in the R environment. One of such constants is for example the Ludolf number assigned as `pi`. This name of a new object is enabled, but it is dangerous to use it, because it overwrites its predefined value. This can later lead to hardly identifiable errors in calculations.

Sometimes we need to assign the same value into more objects. We can use the shorten syntax in such situation.

```
x<-y<-z<-50
```

The list of all created object we obtain as an answer of the `ls()` function. The names are printed as strings enclosed in the quotation marks.

```
> ls()
[1] "height_cm" "x"          "y"          "z"
```

The objects we will not use in the future can be removed from the memory using the `rm()` function. The arguments of the `rm()` functions are the names of the objects without the quotations. For example:

```
> rm(x,z)
> ls()
[1] "height_cm" "y"
```

To remove all defined objects, we can use the combination of both functions `ls()` and `rm()`.

```
> rm(list=ls())
> ls()
character(0)
```

# Data structures in R

Creating a suitable dataset plays crucial role in the statistical analysis. Datasets are usually rectangular tables of data, whose columns contains the variables and rows represent the observations. For the successful statistical analysis, it is important to keep the information in structure and format that meet our needs. In the current chapter we describe the elementary data types and how they can be joined in simple or more complex data structures.

## 2.1 Data types

The simplest objects in R environment are the vectors[1]. Each vectors is an one-dimensional array and it holds several data values, all of the same type. The possible data types are:

- numeric,
- integer,
- complex,
- logical,
- character.

### 2.1.1 Data type `numeric`

The data type `numeric` represents the real decimal numbers and it is the default type of each new object. If we assign to any variable the decimal value, it is of the numeric type. The type of the object we get by function `class()`. Let us see the example.

```
> x<-12.35
> class(x)
[1] "numeric"
```

Let's notice that inserting a whole number in the variable does not change its type, but it remains numeric.

```
> z<-100
> class(z)
[1] "numeric"
```

In order to insert the vectors of the length greater than one, we use the combine function `c()`.

[1]Let us note, that simple variable is in R implemented as a vector of the length one.

```
> v<-c(2,4,6,8,10,12)
> class(v)
[1] "numeric"
> v
[1] 2 4 6 8 10 12
```

The single components of the vector are then accessed using indices in brackets. How illustrates the following example, we cen select one or more components.

```
> v[2]
[1] 4
> v[c(1,3,5)]
[1] 2 6 10
```

Using the colon operator we can create a sequences of consecutive numbers. This construction can be also used to submit the indices of the vector components. Let us see the example:

```
> 2:10
[1]  2  3  4  5  6  7  8  9 10
> v[2:4]
[1] 4 6 8
```

If we apply the colon operator with the non-whole numbers, the vector starts with the first submitted value and its adjacent components differ by one. If the final element specified does not belong to the sequence then it is discarded. See the following code.

```
> v1<-6.6:12.8
> v1
[1]  6.6  7.6  8.6  9.6 10.6 11.6 12.6
> v2<-3.54:8.95
> v2
[1] 3.54 4.54 5.54 6.54 7.54 8.54
```

Some programming languages accept also using the negative integers in the colon operator. The R language admits the minus sign only with zero as the second operator. The components are then printed in reverse order. See the code and its output.

```
> v[5:-0]
[1] 10  8  6  4  2
```

Finally, to create a vector containing sequence with user defined increment, we have to use the function `seq()`. Let us suppose, we want to create the sequence from 4 to 8 with increment of 0.4. Again, if the length of the interval does not match with whole multiple of the increment, the sequence ends with the nearest integer, less than the upper bound. We illustrate it in the following code.

```
> v<-seq(4,8,by=0.4)
> v
 [1] 4.0 4.4 4.8 5.2 5.6 6.0 6.4 6.8 7.2 7.6 8.0
> v<-seq(4,8,by=0.7)
> v
[1] 4.0 4.7 5.4 6.1 6.8 7.5
```

### 2.1.2 Data type integer

The data type `integer` represents, as usually, the variables including the whole numbers. However, it is important to remind, that implicit data type is `numeric`. Therefore, if we submit whole number value, the resulting object is not automatically integer. We can see it in the code:

```
> n<-10
> class(n)
[1] "numeric"
```

If we want to create the object of the `integer` type, we must submit the value using the `as.integer()` function. Alternatively, the variables of the integer type can be submitted as whole numbers ended by letter `L`. So in the previous case, the code that creates integer object `n` has to be modified:

```
> n<-as.integer(10)
> class(n)
[1] "integer"
> n<-10L
> class(n)
[1] "integer"
```

The mentioned function `as.integer()` can be used also to get the value of a vector in the required data type. If the content of the has any decimal part, it is discarded and only integer part is printed. Let us see the example.

```
> v
[1] 4.0 4.7 5.4 6.1 6.8 7.5
> as.integer(v)
[1] 4 4 5 6 6 7
```

How it follows from the previous illustration, the argument of the `as.integer()` function must not be necessary the whole number. It can be any numeric or logical value. The decimal part of the numeric value is discarded and the logical value are transformed to 0 (FALSE) or 1 (TRUE). Only character strings are non-permissible and `NA` is answered instead. The next code shows possible responses of of the `as.integer()` function.

```
> as.integer(2.718)
[1] 2
> as.integer(TRUE)
[1] 1
> as.integer(FALSE)
[1] 0
> as.integer("abc")
[1] NA
Warning message:
NAs introduced by coercion
```

However, when making any computations, it is important to keep in mind that a variable may be retyped. Moreover, this change of the variable type occurs without any warning. Let us see the following example.

```
> x<-as.integer(20)
> class(x)
[1] "integer"
```

```
> x<-x/3+1
> x
[1] 7.666667
> class(x)
[1] "numeric"
```

### 2.1.3 Data type `complex`

The R environment provides also the possibility to work with complex numbers. The complex value is in R defined via the imaginary unit `i`. The following code illustrates, how to create the complex variable.

```
> z<--3+8i
> z
[1] -3+8i
> class(z)
[1] "complex"
```

In mathematics, the complex numbers are closely related to taking the square roots from the negative numbers. As the imaginary unit is defined by the relation $i^2 = -1$, it is frequently interpreted as $\sqrt{-1}$. But in the R computations we have to keep in mind, that value $-1$ is not of the complex type and therefore we do not get the result in the form of the complex number but `NaN` instead.

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

In order to receive the result as the complex number, we must enter the value as the complex type. To do so, here are two alternatives. One of them is to enter $-1$ as the complex number with zero imaginary part and the second alternative is to use definition of the variable type by the function with `as` prefix. Both alternatives are presented in the following code.

```
> sqrt(-1+0i)
[1] 0+1i
> sqrt(as.complex(-1))
[1] 0+1i
```

Let us note that `sqrt()` and `as.complex()` functions has to be entered in the given order. If we enter them in the reverse order, the code does not work correctly, how shows the example.

```
> as.complex(sqrt(-1))
[1] NaN+0i
Warning message:
In sqrt(-1) : NaNs produced
```

When entering the complex number with unit imaginary part, it is necessary to write the coefficient 1. In opposite the imaginary unit is understood as object with name `i` and R answers by warning about non existing object. We can observe it in the following code excerpt.

```
> a<-1+i
Error: object 'i' not found
> a<-1+1i
> a
[1] 1+1i
```

### 2.1.4 Data type `logical`

Data type `logical` can have two logical values `TRUE` or `FALSE`. It is frequently created via comparison between variables.

```
> x<-10;y<-20
> z<-x<y
> z
[1] TRUE
> class(z)
[1] "logical"
```

Here are defined all standard logical operations, how listed in table 2.1. Their using is illustrated by the following code.

```
> a<-TRUE;b<-FALSE
> a&b
[1] FALSE
> a|b
[1] TRUE
> !a;!b
[1] FALSE
[1] TRUE
```

Table 2.1: List of the standard logical operations.

| *Operation* | *Purpose* | *Operation* | *Purpose* |
|---|---|---|---|
| `&` | Logical `AND` | `\|` | Logical `OR` |
| `!` | Negation | | |

How we have already mention, the logical values can be transformed to the integer or numeric value using the functions `as.integer()` or `as.numeric()` respectively. The logical value `TRUE` is in both functions interpreted as number 1 and the logical value `FALSE` is transformed to 0. Similarly, any numeric value can be transformed to logical using the function `as.logical()`. Any non-zero value of its argument is transformed to logical value `TRUE` and only 0 is interpreted as `FALSE`. See the example.

```
> as.logical(1)
[1] TRUE
> as.logical(2)
[1] TRUE
> as.logical(0.5)
[1] TRUE
> as.logical(0)
[1] FALSE
> as.logical(-1)
[1] TRUE
```

### 2.1.5 Data type character

The objects of the type `character` are used to store the string values. This type can contain any alphanumeric characters. They are entered using the quotation marks or by the function `as.character()`. The numbers are in this situation interpreted as strings, so it is not allowed to make some calculations with them. We see it in the following example, where the variable `s` is of the type `character` and therefore we get its value with quotation marks.

```
> var<-"string"
> class(var)
[1] "character"
> s<-as.character(3.14)
> 2*s
Error in 2 * s : non-numeric argument to binary operator
> class(s)
[1] "character"
> s
[1] "3.14"
```

The character type objects can be concatenated using the `paste()` function.

```
> name<-"Donald"
> surname<-"Knuth"
> paste(name,surname)
[1] "Donald␣Knuth"
```

Sometimes it is required to change the default separation of the concatenated characters by space. To do this, we use the additional argument `sep` of the `paste()` function. In the previous case, we can use comma instead of space.

```
> paste(name,surname,sep=",")
[1] "Donald,Knuth"
```

Frequently it is more convenient to create a readable string with the `sprintf()` function, which has a C language syntax. This function returns a character vector containing a formatted combination of text and variable values. The string contains normal characters, which are passed through to the output string, and also conversion specifications which operate on the arguments. The allowed conversion specifications start with a `%` and end with one of the formatting letters. The most frequently used letters are summarised in table 2.2.

We illustrate the use of formatted output in the following examples.

```
> sprintf("%s␣has␣%i␣dogs","John",3)
[1] "John␣has␣3␣dogs"
> sprintf("Number␣pi␣equals␣%f",pi)
[1] "Number␣pi␣equals␣3.141593"
> sprintf("Number␣pi␣equals␣%0.12f",pi)
[1] "Number␣pi␣equals␣3.141592653590"
> sprintf("10!␣in␣exponential␣%e",factorial(10))
[1] "10!␣in␣exponential␣3.628800e+06"
sprintf("100␣in␣octal␣notation␣%o",100)
[1] "100␣in␣octal␣notation␣144"
> sprintf("1000␣in␣hexadecimal␣notation␣%X",1000)
[1] "1000␣in␣hexadecimal␣notation␣3E8"
```

Table 2.2: List of the most frequently used variable output formatting letters of the sprintf() function.

| *Letter* | *Format* |
|---|---|
| s | Character string, NA values converted to ”NA”. |
| d,i | Integer values. |
| o | Integer in octal notation. |
| x,X | Integer in hexadecimal notation using the same case for a-f as the code. |
| f | Double precision value, in fixed point decimal notation. The number of decimal places is specified by the precision, the default is 6. |
| e,E | Double precision value, in exponential decimal notation, using the same case for e as the code. |

One of the typical string operations is extracting some substring. In R is implemented the function substr(), whose arguments are the original string and the start and end positions of the substring that should be extracted. Let us see the example.

```
z<-"We have an interesting lesson in R today"
> substr(z,start=12,stop=34)
[1] "interesting lesson in R"
```

In order to replace some part of the string by another substring, we apply the function sub(). It should be noticed, that it is important to pay attention to unambiguity of the substring, because only the first occurrence is substituted. Sometimes it can lead to undesirable result, how illustrate the following example. Let us suppose, we want to replace “my sister” by “your sister” in the sentence: “Here is my brother and my sister”. The first use of the sub() function shows the incorrect substitution due to unambiguity. The correct solution brings the second case of using the sub() function.

```
> z<-"Here is my brother and my sister"
> sub("my","your",z)
[1] "Here is your brother and my sister"
> sub("my sister","your sister",z)
[1] "Here is my brother and your sister"
```

Besides the sub() function, there is implemented as well gsub() function. It differs from sub() in that gsub() substitutes all matches respectively. We illustrate it in the following code.

```
> gsub("my","your",z)
[1] "Here is your brother and your sister"
```

## 2.2 Data structures

In R ne finds a large variety of objects for storing data. They can have a form of single scalars or variables, but can be also combined in some structures. In the R we can use the following data structures:

- vector,

- matrix,
- array,
- list,
- frame.

In this section we introduce briefly all the mentioned structures and methods of working with them. The correct selection of the data structure is often essential for successful data analysis.

### 2.2.1 Vector

Vector is the simplest data structure. It can be characterised as a sequence of data elements of the same basic type. The single values contained in the vector are called components. How we mentioned in the subsection about numeric data type, as well the single variable can be considered as a vector of the length one.

The number of components of the vector is referred as its length. This value we get using the `length()` function. Let us note, that vector `v` is created by combine function, we have introduced in the subsection 2.1.1.

```
> v<-c(1,3,5,7,9)
> length(v)
[1] 5
```

The combine function`c()` does not only create the vectors, but it can be applied to combine two or more vectors. It can happen, that combined vectors have not the same data type. In such situations the types are converted to the type character. It is important to note this, as this can lead to a loss of the ability to perform calculations on numerical values. Let us see the examples.

```
> w<-c("a","b","cc")
> c(v,w)
[1] "1"  "3"  "5"  "7"  "9"  "a"  "b"  "cc"
```

We can observe, that numerical values originally stored in the vector `v` are now transformed to characters. On the other hand, if we create a logical vector `u`, its values are converted to numeric values 0 or 1. But when combining with the vector och characters, the logical values are transformed to the strings.

```
> u<-c(TRUE,FALSE,TRUE)
> c(u,v)
[1] 1 0 1 1 3 5 7 9
> c(u,w)
[1] "TRUE"  "FALSE" "TRUE"  "a"     "b"     "cc"
```

The vector arithmetic is implemented component-wise, means that arithmetic operations are performed component-by-component. We can use the following arithmetic operations:

+ addition of a number to all components or addition of vectors component-by-component,

− subtracting of a number from all components or subtracting of vectors component-by-component,

* multiplication of all components by number or multiplication of vectors component by component,

/ dividing all components by number or dividing of vectors component-by-component.

Let us see some examples of the vector arithmetic. Special attention pay to illustration, how does work the multiplication and dividing by a number and multiplication and dividing of vectors.

```
> v<-c(1,3,5,7,9)
> u<-c(10,20,30,40,50)
> u+v
[1] 11 23 35 47 59
> u-v
[1]  9 17 25 33 41
> 5*v
[1]  5 15 25 35 45
> u*v
[1]  10  60 150 280 450
> u/5
[1]  2  4  6  8 10
> u/v
[1] 10.000000  6.666667  6.000000  5.714286  5.555556
```

The operation conducted between number and vector is generalised to operate with vectors whose lengths does not match. The so called recycling rule is applied in such situations. It means, that the operations is conducted component-wise, where the shorter vector is used repeatedly. This rule is limited by the condition, that the length of the longer vector must be a multiple of the length of the shorter one. We illustrate it in the example, where the condition is fulfilled only in the first case while an error occurs in the second case.

```
> v<-c(10,20,30)
> u<-1:9
> u+v
[1] 11 22 33 14 25 36 17 28 39
> u<-1:10
> u+v
 [1] 11 22 33 14 25 36 17 28 39 20
Warning message:
In u+v:longer object length is not a multiple of shorter object
        length
```

In the subsection 2.1.1 we have mentioned accessing the vector component by their index. Let us remind the components are accessed using the numeric values enclosed in brackets. As a novelty we can introduce, that the indices can be also repeated, like shows the next code.

```
> u<-2*1:10
> u[c(2,3,5,5)]
[1]  4  6 10 10
```

Another alternative how to select some components of some vector is to use the vector of logical values. This logical vector must have the same length as the original vector. Its components are TRUE if the corresponding components in the original vector are to be included in the slice, and FALSE if otherwise.

```
> u<-2*1:6
> L<-c(FALSE,TRUE,TRUE,FALSE,FALSE,TRUE)
> u[L]
[1]  4  6 12
```

The R environment allows to assign names to the vector components. This is performed using the `names()` function. for example we can create the character vector with two components.

```
> v<-c("Donald","Knuth")
> v
[1] "Donald" "Knuth"
```

Now we assign names to the components of v.

```
> names(v)<-c("Name","Surname")
> v
    Name  Surname
"Donald"  "Knuth"
```

Now we can retrieve the second component of the vector by its name.

```
> v["Surname"]
Surname
"Knuth"
```

### 2.2.2 Matrix

We can define matrix as a two dimensional collection of data of the same type arranged in rectangular layout. We create the matrix object in memory with the `matrix()` function. The `matrix()` function can contain more arguments:

`vector` contains the elements of the matrix,

`nrow` is an integer value, it specifies number of rows in the matrix,

`ncol` is an integer value, it specifies number of columns in the matrix,

`byrow` is a logical value, it indicates, if the matrix should be filled by rows (`byrows=TRUE`) or by columns (`byrows=FALSE`), its default value is `FALSE`,

`dimnames` is a list of character vectors that contain optional row and columns labels.

Let us now create two matrices, one of them by rows and the second one by columns.

```
> A<-matrix(3:8,nrow=3,ncol=2,byrow=TRUE)
> A
     [,1] [,2]
[1,]    3    4
[2,]    5    6
[3,]    7    8
> B<-matrix(3:8,nrow=3,ncol=2,byrow=FALSE)
> B
     [,1] [,2]
[1,]    3    6
[2,]    4    7
[3,]    5    8
```

To the single elements of the matrix are accessed by pair of indices in brackets. The first index corresponds to the row number and the second to the column number, how it is commonly used in matrix algebra. So the element located in the second row and second column of the matrix A is extracted by code:

```
> A[2,2]
[1] 6
```

Omitting one of the indices leads to extracting of the row or column from the matrix. Let us see, how to extract first column of the matrix A and second row of the matrix B.

```
> A[,1]
[1] 3 5 7
> B[2,]
[1] 4 7
```

We can also extract submatrix that contains more than one rows or columns at a time.

```
> B[c(1,3),]
     [,1] [,2]
[1,]    3    6
[2,]    5    8
> C<-matrix(1:12,nrow=3)
> C
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> C[c(1,3),c(2,4)]
     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

Finally, let us show how to assign names to rows and columns of the matrix. Later, we can call the elements of the matrix by the names.

```
> dimnames(A)<-list(c("row1","row2","row3"),
+ c("col1","col2"))
> A
     col1 col2
row1    3    4
row2    5    6
row3    7    8

> A["row2","col1"]
[1] 5
```

We construct the transpose of a matrix by interchanging its columns and rows with the function t().

```
> t(B)
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
```

We can also combine matrices. It is necessary, the matrices have the same number of columns or the same number of rows. If they have the same number of rows, we can combine the columns with the cbind() function. Let us note using of the diag() function that construct the diagonal matrix with given elements on the diagonal.

```
> cbind(B,diag(c(1,2,5)))
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    6    1    0    0
[2,]    4    7    0    2    0
[3,]    5    8    0    0    5
```

Similarly, we can combine the rows of two matrices if they have the same number of columns. In such situation we apply the function `rbind()`. Let us suppose, we want to combine the matrix `C` with the second and the fourth row of the diagonal matrix that has on its diagonal numbers 1, 2, 5 and 7. We use the following code.

```
> rbind(C,diag(c(1,2,5,7))[c(2,4),])
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
[4,]    0    2    0    0
[5,]    0    0    0    7
```

### 2.2.3 Array

Arrays are generalizations of the matrix data structure. One can characterise them as more than two dimensional matrices. These data structures are created similarly like matrices by following command:

```
name<-array(vector, dimensions,dimnames)
```

In the presented code `name` represents the name of the array being created, `vector` contains data for the array[2], `dimensions` is a numeric vector that defines the length for each dimension and `dimnames` is a list of names how the dimensions are reported. The last argument `dimnames` is optional and its use depends on the need of naming the dimensions.

Now we illustrate creating of the $3 \times 4 \times 3$ array. For greater clarity of the array, we create at first the names of single dimensions.

```
> dim1<-c("A1","A2","A3")
> dim2<-c("B1","B2","B3","B4")
> dim3<-c("C1","C2","C3")
```

Now we are ready to create the array `z`, that contains the integers from 1 to 36. We use following code

```
> z<-array(1:36,c(3,4,3), dimnames=list(dim1,dim2,dim3))
```

Let us now see the structure, how is the array stored in the memory:

```
> z
, , C1

   B1 B2 B3 B4
A1  1  4  7 10
A2  2  5  8 11
```

[2]Let us remind, they must be of the same type.

```
A3   3   6   9  12

, , C2

    B1  B2  B3  B4
A1  13  16  19  22
A2  14  17  20  23
A3  15  18  21  24

, , C3

    B1  B2  B3  B4
A1  25  28  31  34
A2  26  29  32  35
A3  27  30  33  36
```

We can observe, that arrays are saved in the form of multiple matrices. The single values or subsets of the array are accessed similarly to matrices by indexing in the brackets `[` and `]`. In our example, we can access the first `C1` matrix by command:

```
> z[,,"C1"]
    B1  B2  B3  B4
A1   1   4   7  10
A2   2   5   8  11
A3   3   6   9  12
```

To extract for example value 8 from the array, we apply all three indices determining its position. Let us note, the dimensions can be called by numbers or by names. As well the combination of indices and names works correctly, how shows the example:

```
> z[2,3,1]
[1] 8
> z[2,3,"C1"]
[1] 8
> z["A2","B3",1]
[1] 8
```

Similarly like in matrices, the sub-arrays can be extracted by submitting a scope of the indices. We illustrate it on subtracting the matrix, containing numbers 17, 18, 20, 21 from the array `z`. Here is the code and result:

```
> z[2:3,2:3,2]
   B2  B3
A2 17  20
A3 18  21
```

Alternatively, we can use the name of matrix `C2`:

```
> z[2:3,2:3,"C2"]
   B2  B3
A2 17  20
A3 18  21
```

### 2.2.4 Frame

Data frame is the most common structure for storing the data. This structure is more general than matrices or arrays as it enables storing the column vectors of the different data types. Data frames are created using the function `data.frame()` and its general form is:

```
> name<-data.frame(col1,col2,col3, ...)
```

Here `name` is the user defined name of the new variable containing the data frame, and `col1`, `col2`, `col3` (or more) are the columns vectors of any type. The names of the each single columns are provided by the `names()` function.

We make it clear by the following listing. Let us suppose, we want to create the dataset for the analysis of the individual playing statistics of the basketball players. For simplicity of the illustrative example, let us suppose, we register for each player only a few data, like some `playerID`, `position`, and for scoring the number of shooting attempts `attempted` and successful attempts as variable `made`. Then we create the data frame `players`:

```
> playerID<-c(1,2,3,4)
> position<-c("forward","guard","forward","center")
> attempted<-c(12,6,10,15)
> made<-c(7,4,6,12)
> players<-data.frame(playerID,position,attempted,made)
> players
  playerID position attempted made
1        1  forward        12    7
2        2    guard         6    4
3        3  forward        10    6
4        4   center        15   12
```

In the top line, named the *header*, we see the names of the columns. Each subsequent row is the *data row*. Its first value is the row name (ordinal number if no row names are submitted) and it is followed by the actual data. Each individual data member of a row is called a *cell*.

There are several ways how to access the each cell of the data frame. One of them is to use the index notation similarly how we did it with matrices and arrays. We can demonstrate this approach with the `players` data frame created earlier.

```
> players[1:2]
  playerID position
1        1  forward
2        2    guard
3        3  forward
4        4   center
```

Another option is to use the column names (given as character vector) instead of indices:

```
> players[c("playerID","attempted","made")]
  playerID attempted made
1        1        12    7
2        2         6    4
3        3        10    6
4        4        15   12
```

The third possibility is to use the $ notation. This notation consists from the data frame name on the first place and column name on the second place, that are separated by the $ sign. We can select for example all player positions from the data frame `players` by following request:

```
> players$position
[1] forward guard   forward center
Levels: center forward guard
```

We get the answer in the form of the character vector. In the second line we obtain information about all levels of the positions recorded in the frame. It can bring some discomfort if we have to write the name of the data frame very frequently. Therefore there are the functions `attach()` and `detach()`, that makes using of the concrete data frame easier.

In order to access the single column we apply the he double square bracket `[[]]` operator. For example to get the fourth column of our `players` data frame we will write `players[[4]]`. Let us compare the results when using simple and double brackets operators.

```
> players[4]
  made
1    7
2    4
3    6
4   12
> players[[4]]
[1]  7  4  6 12
```

How we see, the results differ in their types. In the first case we get the result as new data frame, while the second alternative gives a vector as its result. This difference in the resulting structures is crucial for further computations. The double brackets operator is equivalent to use of coma in the one bracket operator, how shows the following listing (the column is called by index or by its name).

```
> players[,4]
[1]  7  4  6 12
> players[,"made"]
[1]  7  4  6 12
```

Similarly we can slice also rows from the data frame. In order to call rows by its name, let as at first to assign some names to our object `players`:

```
> row.names(players)<-c("Player1","Player2","Player3","Player4")
> players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12
```

If we need to slice for example the third row, we can do it by index od name of the row, using the single bracket operator with the extra comma in the square bracket operator.

```
> players[3,]
        playerID position attempted made
Player3        3  forward        10    6
> players["Player3",]
        playerID position attempted made
Player3        3  forward        10    6
```

To extract more than one rows, we use a numeric index vector.

```
> players[c(1,3),]
        playerID position attempted made
Player1        1  forward        12    7
Player3        3  forward        10    6
> players[2:4,]
        playerID position attempted made
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12
```

The `attach()` function adds the data frame to the search path of the R environment. It enables to write only the column names. Let us see the demonstration.

```
> attach(players)
The following objects are masked _by_ .GlobalEnv:

    attempted, made, playerID, position

> 100*made/attempted
[1] 58.33333 66.66667 60.00000 80.00000
```

After attaching the data frame `players`, we can easily compute the shooting percentages of each player, without writing the full name in the $ notation. To remove the frame from the search path, we simply use the `detach()` function, how illustrates the listing.

```
> detach(players)
```

Alternative to the attaching the frame to the search path is using the `with()` function. Our previous example can be then rewritten as:

```
> with(players, {
+ 100*made/attempted}
+ )
[1] 58.33333 66.66667 60.00000 80.00000
```

Statements within the braces {} are evaluated with reference to the data frame name submitted as the first variable of the `with()` function. The limitation of the `with()` function is, that objects created in its body will exist only within this function. If we need to create an object that will exist also outside of the `with()`, we have to create it using `<<-` instead of the common assignment by `<-`. We demonstrate it in the following listing.

```
> with(players, {
+ percent_local<-100*made/attemted
+ percent_global<<-100*made/attemted}
+ )
> percent_local
Error: object 'percent_local' not found
> percent_global
[1] 58.33333 66.66667 60.00000 80.00000
```

We often need to merge date from two or more datasets. For these purposes are in R environment implemented two functions. In order to join merge data frames horizontally, we use function `merge()`. Its arguments are two data frames that should be merged and the third argument `by="column name"` defines the key variable for joining the data.

It is clear, that this column name must be given in both merged frames. To demonstrate the merging, let us at first create second data frame, that include the statistics about offensive and defensive rebounds of the players. Then we merge these two data frames `players` and `rebounds` into the new frame `new_players`. The key variable for joining the data is the `playerID`. How we can see in the result, the new data frame does not save the row names, throughout we have assigned the same names in both merged data frames. We have to keep in mind this fact, because it can lead to errors when calling the rows of the new frame by their names.

```
> offensive<-c(5,2,3,10)
> defensive<-c(6,3,8,12)
> rebounds<-data.frame(playerID,defensive,offensive)
> row.names(rebounds)<-c("Player1","Player2","Player3","Player4")
> new_players<-merge(players,rebounds,by="playerID")
> new_players
  playerID position attempted made defensive offensive
1        1  forward        12    7         6         5
2        2    guard         6    4         3         2
3        3  forward        10    6         8         3
4        4   center        15   12        12        10
```

The second alternative of merging data frames is adding a new rows to the existing data frame. To add a new row we use the function `rbind()`. Arguments of this function are two data frames. They must have the same variables, but these variables do not have to be in the same order. If the first data frame has variables that the second data frame does not, then either:

1. Delete the extra variables in the first data frame or
2. Create the additional variables in the second data frame and set them to `NA` (missing values).

To demonstrate the `rbind()` function, we create at first the new data frame `players2` that contains a few additional players data. Then we merge both frames in a new object assigned as `more_players`. Let us see the listing.

```
> position<-c("center","guard","forward")
> attempted<-c(14,8,12)
> made<-c(10,5,8)
> players2<-data.frame(playerID,made,attempted,position)
> row.names(players2)<-c("Player5","Player6","Player7")
> more_players<-rbind(players,players2)
> more_players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12
Player5        5   center        14   10
Player6        6    guard         8    5
Player7        7  forward        12    8
```

How we can see, the row names are saved in this vertical data merging. The data frame `players2` was created with different columns order, to illustrate that joining is independent on it. On the other hand, the order of submitting the variables in the `rbind()` function is substantial. How shows the following listing, submitting them in reversal order leads to different result:

```
> rbind(players2,players)
        playerID made attempted position
Player5        5   10        14   center
Player6        6    5         8    guard
Player7        7    8        12  forward
Player1        1    7        12  forward
Player2        2    4         6    guard
Player3        3    6        10  forward
Player4        4   12        15   center
```

### 2.2.5 List

Lists represent the most complex data structure. In general we can say, that lists are ordered collections of objects. They enable to gather more objects, reported as its *components* under one name. List can be a combinations of all another data structures: vectors, matrices, data frames and even other lists. To create a list, we use the function `list()`. Its syntax is simple:

`\list(object1,object2,...)`

Its arguments are names of existing objects. Optionally we can name the object in the created list:

`\list(name1=object1,name2=object2,...)`

We will demonstrate creating the list named `NBA` from our existing data frames `players` and `players2`. Moreover, in the list we join the player statistics with some clubs. At first we create the list containing one club:

```
> NBA<-list(club="Bulls",city="Chicago",Players=players)
> NBA
$club
[1] "Bulls"

$city
[1] "Chicago"

$Players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12
```

Later we can add a next member of the list using the concatenate function `c()`, how demonstrates the listing.

```
> NBA<-c(NBA,list(club="Celtics",city="Boston",Players=players2))
> NBA
$club
[1] "Bulls"

$city
[1] "Chicago"

```

```
$Players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12

$club
[1] "Celtics"

$city
[1] "Boston"

$Players
        playerID made attempted position
Player5        5   10        14   center
Player6        6    5         8    guard
Player7        7    8        12  forward
```

Recall that this function concatenates all arguments into a single vector structure. In this case it means, that the second club has got the positions from 4 to 6 in the new list, while the element with double index `[2,1]` does not exist in the list.

```
> NBA[1]
$club
[1] "Bulls"

> NBA[4]
$club
[1] "Celtics"

> NBA[2,1]
Error in NBA[2, 1] : incorrect number of dimensions
```

When working with a list, we have to distinguish among the single and double brackets operators. Applying the single bracket operator we get a slice of the list. As an example, we can show the following listing and its output.

```
> NBA[3]
$Players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
Player4        4   center        15   12
```

Using the double brackets operator gives the answer that seems to be the same at first sight, however, they differ in the data structure. While the single bracket operator answers a list structure, using the double brackets answers the same structure that has the given component. This allows referring directly to its elements. All demonstrates the following listing.

```
> NBA[[3]]
$Players
        playerID position attempted made
Player1        1  forward        12    7
Player2        2    guard         6    4
Player3        3  forward        10    6
```

```
Player4         4    center         15    12
> NBA[3][2]
$<NA>
NULL
> NBA[[3]][2,]
        playerID position attempted made
Player2         2     guard          6    4
> class(NBA[3])
[1] "list"
> class(NBA[[3]])
[1] "data.frame"
```

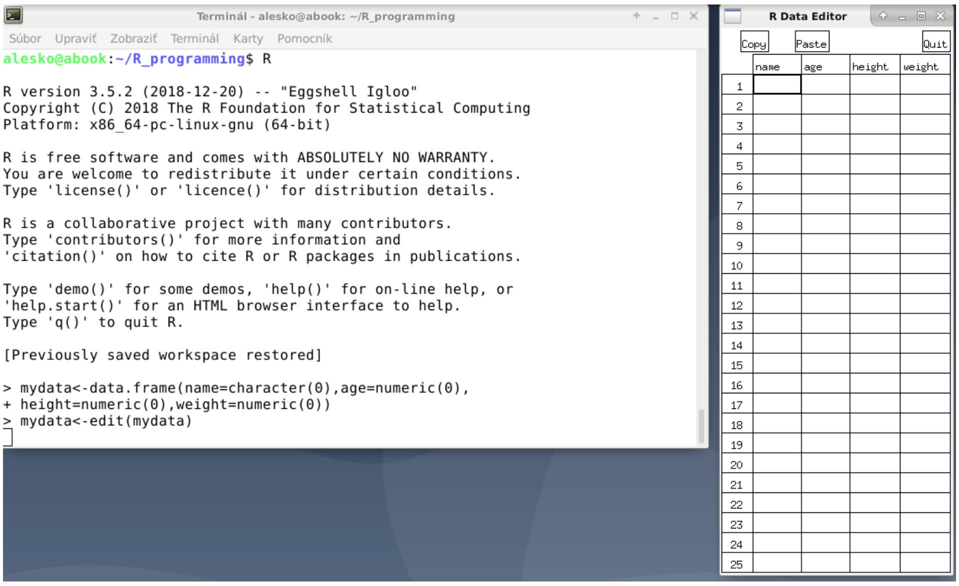
The double brackets notation allows modifying a list members directly.

```
> NBA[[3]][2,]
        playerID position attempted made
Player2         2     guard          6    4
> NBA[[3]][2,3]<-c(7)
> NBA[[3]][2,]
        playerID position attempted made
Player2         2     guard          7    4
```

Once we have named the components of the list, we can refer to them by names instead of indices. In order to access some member directly, we again use the double brackets operator. In our previously constructed examples, the component `Players` has the data frame structure, which means we can access its component again by indices or names. We demonstrate a variety of modes of accessing the same member of the list.

```
        playerID position attempted made
Player2         2     guard          7    4
> NBA[["Players"]][2,]$position
[1] guard
Levels: center forward guard
> NBA[["Players"]]["Player2",]$position
[1] guard
Levels: center forward guard
> NBA[[3]]["Player2",]$position
[1] guard
Levels: center forward guard
> NBA[[3]][2,]$position
[1] guard
Levels: center forward guard
> NBA[[3]][2,2]
[1] guard
Levels: center forward guard
```

Let us note here that in our list are two components named as `Players`. We see that calling by name respects only its first occurrence in the list, so the second component named `Players` we have to refer by its index. It is a warning for duplicate naming in the list.


Terminál - alesko@abook: ~/R_programming
Súbor Upraviť Zobraziť Terminál Karty Pomocník
R Data Editor
Copy
Paste
Quit
name
age
height
weight

```
alesko@abook:~/R_programming$ R

R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> mydata<-data.frame(name=character(0),age=numeric(0),
+ height=numeric(0),weight=numeric(0))
> mydata<-edit(mydata)
```

Figure 2.1: A built-in simple data editor on the linux platform

## 2.3 Data input and output

### 2.3.1 Entering data from the keyboard

Probably the simplest method of entering data is from the keyboard. To enter data from keyboard we work in two steps:

1. Create the empty data frame with the variable names and types we want to store in the dataset. This step can be omitted if we want to modify an existing data frame.
2. Invoke the simple data editor using the function `edit()`, whose argument is the name of the data frame we want to edit. This approach can be used to add some data from the keyboard into the existing data frame. We see the result of invoking the editor in figure 2.1.

As an example we create empty data frame named `mydata` with four variables: `name` that has type character and three numeric variables `age`, `height` and `weight`. Let us note, that assigning like `numeric(0)` and `character(0)` create a variable of the given type but without any data.

```
> mydata<-data.frame(name=character(0),age=numeric(0),
+ height=numeric(0),weight=numeric(0))
> mydata<-edit(mydata)
```

When quieting the editor, the entered data gathered in the `mydata` data frame.

### 2.3.2 Importing data from data files

The role of statistics is mass data processing. Therefore, manually entering data from the keyboard is very inconvenient and not very practical. It is much more efficient to load data from existing data files. The R environment allows to import data from a variety of file formats like `csv` fieles, `text` files, Excel files, ODBC database and many others. Most of them requires installing specialized packages, therefore me shortly mention only a few elementary methods.

### Data in `.csv` files

One of the most commonly used data sample formats are the comma separated values, usually stored in files with the `csv` extensions. Each cell inside such data file is separated by a special character, which the most frequently is a comma, although other characters can be used as well.

The first row of the data file should (but must not) contain the column names instead of the actual data. Here is a sample of the expected format.

```
Column1,Column2,Column3
A,10,0.11
B,20,0.22
C,30,0.33
```

Let us suppose these data are saved in a file named `mydata.csv`[3]. We read the data using the function `read.csv()`.

```
> mydata<-read.csv("mydata.csv")
> class(mydata)
[1] "data.frame"
> mydata
  Column1 Column2 Column3
1       A      10    0.11
2       B      20    0.22
3       C      30    0.33
```

We can observe the object `mydata` created using the `read.csv()` function has the data frame structure.

The `read.csv()` function has several optional arguments. Here we mention some of the most used. The complete list of them we get using the `help(read.csv)` command. In addition to the name of the input file (which is given in quotation marks), we can add the following options:

- `header` which is a logical value that indicates whether the input file contains the names of variables as the first line. Its default value is TRUE.
- `sep` defines the field separator character. The default value is comma, if `sep = ""`, the separator is “white space” (means one or more spaces, tabulators or newlines.
- `dec` defines the character used in the file for decimal points. Its default value is `.`. In Central European countries, it is customary to use a comma instead of decimal point. In such cases it is necessary to declare `dec =","`. In this context we mention also `read.csv2()` function, which adopts using the comma for decimal numbers and semicolon as delimiter.
- `skip = n` specify the number of lines to skip before the data starts. This option is useful for data tables with blank rows or text padding at the top of files.

[3]For example simply by copy and paste in any text editor.

- `stringsAsFactors` which is a logical value that indicates whether the strings are converted to factors.To prevent character columns being converted to factors we have to set it to `FALSE`.
- `row.names` a vector of row names. It can be a vector of actual row names or a single number stating the column of the row names. If `row.names` is missing, the rows are numbered.

### Importing from the delimited text file

We can import data using the `read.table()` function. It works similarly like the `read.csv()` function, means it reads the data from specified data file and creates a data frame object. These two functions differ only in the default options settings. The two arguments we need to be aware of are the field separator `sep` and the argument `header` indicating whether the file contains the names of the variables as its first line. Here are the differences:

- In the `read.table()` function the default separators are white spaces `sep = ""` whereas in the `read.csv()` function the default separators are commas `sep=","`.
- In the `read.table()` is no header line suggested and the default value is `header=FALSE`, while in the `read.csv()` function the default option is `header=TRUE`.

We can demonstrate it on the following example, where we read the same file by both function. Compare the structure of the objects created and the object we got in the previous illustrative example.

```
> mydata <-read.table("mydata.csv")
> mydata
                    V1
1 Column1,Column2,Column3
2              A,10,0.11
3              B,20,0.22
4              C,30,0.33
> mydata <-read.table("mydata.csv",header=TRUE,sep=",")
> mydata
  Column1 Column2 Column3
1       A      10    0.11
2       B      20    0.22
3       C      30    0.33
```

How we see, in the first case we get only one column, all variables are converted into strings as the `factor` type. In the second case, when we declared existence of the header line in the input file, only one column was created again. But it is named by concatenation of all column names. Finally, in the third alternative, with full declaration of `header` and `sep` options we got the same structure of `mydata` as we have applied the `read.csv()` function.

### Reading the Excel files

Perhaps the best way, how to import Excel files is to import it to a comma-separated file from within Excel and then use the functions we described earlier. However, this approach has some limitations. For example, if we work with more sheets in one document, they must be exported separately, each sheet in an extra file. Fortunately, there are several packages that allow us to import data directly from Excel files. Let us mention some of them:

- `xlsx`,
- `XLconnect`
- `readxl`

Excel 2007 and newer versions use an `xlsx` format. Therefore we introduce here the `xlsx` package that is suitable to access spreadsheets in this format. Of course, it is necessary to install this package before its first use. The `xlsx` package is a java based solution and it is available for all three platforms: Windows, Mac, and Linux.

We install the package by standard command:

```
install.packages("xlsx")
```

To use it in actual workspace, we load it by the standard way:

```
load("xlsx")
```

This package provides two functions we can use for reading the contents of an Excel worksheet into a R data.frame. These functions have similar names `read.xlsx()` and `read.xlsx2()`. The difference between these two functions is:

- `read.xlsx()` preserves the data type, the type of the variable corresponds to each column in the worksheet, but it is slow for large data sets (worksheet with more than 100 000 cells).
- `read.xlsx2()` is faster on big files.

Both functions have similar syntax:

```
read.xlsx(file, sheetIndex, header=TRUE, colClasses=NA)
read.xlsx2(file, sheetIndex, header=TRUE, colClasses="character")
```

Their arguments have the following meaning:

- `file` is the name of the file containing the spreadsheet. If the files is not in the working directory, it has to be declared with the full path.
- `sheetIndex` a number indicating the index of the sheet to read. We can replace it by the `sheetname` argument given as character string with the sheet name.
- `header` logical value. If `header=TRUE`, the first row is used as the names of the variables.
- `colClasses` a character vector that represents the class of each column.
- `startRow`, `endRow` numbers specifying the index of starting row and the last row to read.

## Reading the JSON files

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages.

To get JSON files into R, we first need to install or load the `rjson` package. Once this is done, we can use the `fromJSON()` function. It usage depends if the `json` file is stored in our computer or has to be downloaded from internet through URL. In the first case we use the function in the form:

```
data<-fromJSON(file = "filename.json")
```

while in the second

```
data<-fromJSON(file = "URL to the json file")
```

In both cases, the object `data` is stored as the list. For the further analysis we can convert the data using the `as.data.frame()` function.

### 2.3.3 Data output into the files

#### Writing into the `.csv` file

R can create `csv` file form existing data frame. To create the `.csv` file we use the `write.csv()` function, or alternatively the `write.csv2()` function, that uses a comma for the decimal point and a semicolon for the separator. The common syntax of the function is:

```
write.csv(object,file="file_name",...options)
```

where `object` is obligatory argument containing the name of the data frame we want to save and `file_name` is the name (or full path) of the file for writing that is obligatory as well. Let us note, that setting `file=""` indicates output to the console. We introduce some of the optional arguments (the full list we get by `help(write.csv)`.

- `append` which is a logical value that indicates whether the output is appended to exiting file. The default value is `FALSE` and any existing file of the given name is destroyed.
- `sep` defines the field separator character. Values within each row of `object` are separated by this character.
- `dec` the string to use for decimal points in numeric or complex columns, must be a single character. The default value is decimal point.
- `row.names` a logical value indicating whether the row names of `object` are to be written.

#### Writing data into the Excel files

Similarly like for reading, we need to load the `xlsx` package. It provides `write.xlsx()` and `write.xlsx2()` functions that one can use to export data into the Excel workbook. Note that the difference is the same as in the case of the functions for reading the spreadsheets, means `write.xlsx2()` achieves better performance for very large data. The simplified syntax for these functions is:

```
write.xlsx(x,file,sheetName="Sheet1",col.names=TRUE,row.names=TRUE,append=FALSE)
write.xlsx2(x,file,sheetName="Sheet1",col.names=TRUE,row.names=TRUE,append=FALSE)
```

Their arguments have the following meaning:

- `x` a data.frame to be written into the workbook.
- `file` the path to the output file.
- `sheetName` the character string with the sheet name.
- `col.names` logical value, it indicates if the column names of `x` are to be written along with `x` to the file.
- `row.names` logical value, it indicates if the row names of `x` are to be written along with `x` to the file.
- `append` logical value, it indicates if `x` should be appended to an existing file, if `FALSE`, it overwrites the existing file with the same path.

## Writing data into the JSON files

Writing data to `json` data file has to be done in two phases. In the frist step we must prepare the JSON object and in the second step we write it in the file.

To prepare the the JSON object we use the `toJSON()` function from the `rjson` package. Its general format is:

```
dataJSON<-toJSON(data)
```

Let us note, the object `data` we want to export in `json` format must be a list structure. Once we have prepared the object `dataJSON`, we can save it using the `write()` function in the form:

```
write(dataJSON, "filename.json")
```

# Probability distributions in R

Being an environment specialized in the statistics, R enables working with a big variety of the probability distributions. Each of the distribution that R handles has four functions. All of them have the same principles of the name construction, that is composed from the root name of the distribution and prefix which is one of the four letters:

- `p` for the cumulative distribution function,
- `d` for the density or probability function,
- `q` for the quantile function, the inverse to the cumulative distribution function,
- `r` random variable having the specified distribution (random values generator).

For example, the most commonly used normal distribution has root name `norm` an the functions are then `pnorm`, `dnorm`, `qdnorm`, and `rnorm`. Similarly, for the binomial distribution, these functions are `pbinom`, `dbinom`, `qbinom`, and `rbinom`. And so forth.

The most useful functions for working with the continuous random variables are the “p” and “q” functions, because the density calculated by the “d” function can only be used to calculate probabilities via integrals and R doesn’t integrate. on the other hand, for discrete distributions the ”d” function has probabilistic meaning $f(x) = P[X = x]$ and hence it is useful in calculating probabilities.

## 3.1 Random samples

One of the most common process done by the analysts is taking samples of the data. R offers the standard function `sample()` to take a sample from the datasets. Its simplified syntax is:

```
sample(x, size, replace, prob)
```

where

- `x` is a vector or a data set the sample is drawn from,
- `size` is a sample size,
- `replace` is logical value, states if the values are repeated in the sample or not,
- `prob` a vector of probability weights.

The simplest use of the `sample()` function is with only one the first argument, which gives randomized order of the `x` components, how illustrates the listing.

```
> sample(6)
[1] 4 3 5 1 6 2
> sample(4:10)
[1]  9  7  5  4  8 10  6
> sample(c(1,3,5,7,9))
[1] 9 5 7 3 1
```

Adding the second argument states the sample size. So we can for example randomly select five integers between 1 an 40 by the command:

```
> sample(1:40,5)
[1] 30 35 34  5 29
```

Note that the default behaviour of the `sample()` function is to replace the values. It leads to errors in the case, when the sample size exceeds the length of the data to be sampled. Let us suppose, we want to simulate rolling the dice 50 times. If we use the the `sample()` function in the default regime, we get the following error:

```
> sample(6,50)
Error in sample.int(x, size, replace, prob):
 cannot take a sample larger than the population when 'replace=FALSE'
```

Therefore we have to explicitly set the logical value `replace=TRUE`, how illustrated in the next listng.

```
> sample (6 ,50 , replace = TRUE )
 [1] 6 5 3 3 5 5 4 6 3 1 3 2 ...
[39] 2 6 6 6 4 2 2 5 1 6 1 5
```

Finally, we can set the probabilities of all possible outcomes, that must not be necessary same likely. For example, we can simulate tossing the unfair coin with higher frequencies of heads than tails. Let us suppose, that heads fall twice more than tails. Then we set the argument `prob=c(2/3,1/3)`. Let us see the result in the following listing (`h` means “head” and `t` means “tail”).

```
> sample (c("h","t"),20,replace = TRUE,prob=c(2/3,1/3))
 [1] "h" "t" "h" "h" "h" "h" "t" "h" "h" "t"
[11] "h" "t" "h" "h" "t" "h" "h" "t" "t" "h"
```

We may easily experience, that if we take samples, they will be random and they change each time we apply the `sample()` function. If we wish to avoid such changes, or if we need to reconstruct the same sample, we can use the `set.seed()` function. Setting a fix value in this function leads to producing the same sequence in each attempt. This case is illustrated in the listing below.

```
> sample(6)
[1] 2 5 6 1 4 3
> sample(6)
[1] 1 2 3 5 4 6
> set.seed(3)
> sample(6)
[1] 2 5 6 1 4 3
> sample(6)
[1] 2 5 6 1 4 3
```

Here we have seen, that setting `set.seed(3)` before calling the `sample()` function produces always the same sequence, whereas calling the `sample()` function without it causes change in the result.

## 3.2 Discrete distributions

Informally, we can characterize discrete random variables as random variables that can have discrete values as outcomes. More formally, we say that the set of values of a discrete random variable is at most countable. A discrete probability distribution is then applicable to the scenarios where the set of possible outcomes is discrete (e.g. a coin toss, a roll of a dice), and the probabilities are here encoded by a discrete list of the probabilities of the outcomes, known as the probability mass function. If we assign as $H$ the set of all possible values of the discrete random variable $X$, we can introduce the probability mass function $p(x)$ by formula

$$p(x) = \mathbb{P}\left(X = x\right), \ x \in H. \tag{3.1}$$

In the R environment we can use a variety of implemented discrete distributions. Let us mention some of them:

- Bernoulli distribution,
- binomial distribution,
- geometric distribution,
- hypergeometric distribution,
- negative binomial distribution,
- Poisson distribution.

### 3.2.1 Bernoulli distribution

Bernoulli distribution is the simplest probability distribution. This is a distribution with only two possible values. It can be interpreted as an indicator variable, if some random event occurs or not. Formally, let $A$ is a random event that occurs with probability $\mathbb{P}\left(A\right) = p$ and the random variable $X = 1$ if $A$ occurs and $X = 0$ otherwise. Then the probability mass function has the form

$$p(x) = p^x(1-p)^{1-x} \text{ for } x = 0 \text{ or } 1$$

This distribution is implemented in the R package `Rlab` as `bern` with parameter `prob`. Reading the library `Rlab` and using the usual prefixes, we have at disposal four functions:

- `rbern(n,prob)`, where `n` is a number of observations and `prob` is a probability of occurring the random event $A$ (success in the trial). It generates a vector of `0` and `1` selected from the Bernoulli distribution with given probability.
- `pbern(q, prob, lower.tail = TRUE, log.p = FALSE)`
- `dbern(x, prob, log = FALSE)`
- `qbern(p, prob, lower.tail = TRUE, log.p = FALSE)`

### 3.2.2 Binomial distribution

The binomial distribution is a discrete distribution that describes number of successes in the series of trials wit two possible outcomes: success or failure. All its trials are independent, the probability of success remains the same and the previous outcome does not affect the next outcome. The outcomes from different trials are independent. Binomial distribution helps us to find the individual probabilities as well as cumulative probabilities over a certain range.

Formally, let us assign $p$ the probability of success in one trial, $n$ the number of independent trials and $x$ the number of successes in a sequence of $n$ independent experiments. The random variable X follows the binomial distribution, if its probability mass function has the form:

$$\mathbb{P}\left(X = x\right) = \binom{n}{x} p^x q^{n-x}, \quad x = 0, 1, 2, \dots, n, \tag{3.2}$$

where $q + p = 1$.

We have four functions for handling binomial distribution in R:

- `rbinom(n,prob)`, where `n` is numbers of observations, `p` is the probability of success. This function generates $n$ random variables of a particular probability.
- `pbinom(x, n, k)`, where `n` is total number of trials, `p` is probability of success, `x` is the value at which the probability has to be found out. The function `pbinom()` is used to find the cumulative probability of a data following binomial distribution till a given value i.e. it finds $P(X \leq k)$.
- `dbinom(x, n, p)`, where `n` is total number of trials, `p` is probability of success, `x` is the value at which the probability has to be found out.This function is used to find probability at a particular value for a data that follows binomial distribution i.e. it finds $P(X = k)$.
- `qbinom(prob, n, p)`, where `prob` is the probability, `n` is the total number of trials and `p` is the probability of success in one trial. This function is used to find the $n$-th quantile, that is if $P(X \leq k)$ is given, it finds $k$.

**Example 3.2.1** *Suppose there are twenty multiple choice questions in a quiz. Each question has five possible answers, and only one of them is correct. Find the probability of having six or less correct answers if a student attempts to answer every question at random.*

**Solution**: Since only one out of five possible answers is correct, the probability of answering a question correctly by random is $1/5 = 0.2$. We can find the probability of having exactly 6 correct answers by random attempts using the `dbinom()` as follows

```
> dbinom(6,20,0.2)
[1] 0.1090997
```

To find the probability of having six or less correct answers by random attempts, we apply the function `dbinom()` with $x = 0, \dots, 6$. So we have:

```
> dbinom(0,20,0.2) + dbinom(1,20,0.2) + dbinom(2,20,0.2)+
+ dbinom(3,20,0.2) + dbinom(4,20,0.2) + dbinom(5,20,0.2)+
+ dbinom(6,20,0.2)
[1] 0.9133075
```

Alternatively, we can use the cumulative probability function for binomial distribution `pbinom()`. So we get

```
> pbinom(6,20,0.2)
[1] 0.9133075
```

**Example 3.2.2** *Let us now return to the example 3.2.1. Student pass the exam successfully, if he answers more than 10 questions in the quiz correctly. What is the probability, that student pass the exam if he answers the questions by random?*

**Solution**: In this case we will apply the function `pbinom()` but with the option `lower.tail=FALSE`. So we have

```
> pbinom(10,20,0.2,lower.tail=FALSE)
[1] 0.0005634137
```

**Example 3.2.3** *Let us assume we are in charge of quality for a factory. We make 250 widgets per day. Defective widgets must be reworked. We know that there is a 2% error rate. Let us simulate how many widgets we will need to fix each day this week.*

**Solution**: To generate random sample from the binomial distribution with number of trials $n = 250$ and probability of success $p = 0.02$, we use the `rbinom()` function. For the one week sequence we choose the sample size equal to seven. So we get:

```
> rbinom(7,250,0.02)
[1] 2 5 3 9 5 9 5
```

**Example 3.2.4** *Let us assume we make a test of the drug that has a 80% success rate. Each trial has 30 patients. How many patients is in the bottom 10% percent of positive outcome? Let us state each decile in this treatment test.*

**Solution**: Then 10% of trials will have between 0 and 21 patients respond positively to this treatment. We state this using the function `qbinom()`:

```
> qbinom(0.1,30,0.8)
[1] 21
```

Similarly, if we want the number of patients with a positive response in the bottom 20% of trials, we would enter

```
> qbinom(0.2,30,0.8)
[1] 22
```

In order to get each decile in this drug test we enter

```
> qbinom(seq(0.1,1,0.1),30,0.8)
 [1] 21 22 23 24 24 25 25 26 27 30
```

### 3.2.3 Hypergeometric distribution

The hypergeometric distribution is a discrete probability distribution that describes the probability of $x$ successes in $n$ draws without replacement, from a finite population of size $N$ that contains exactly $K$ objects with that feature. Each draw is either a success or a failure but in contrast to binomial distribution, the probability of success does not remain the same and the previous outcome affects the next outcome.

Let us denote as $N$ the population size, $K$ the number of success states in the population, $n$ the number of draws and $x$ the number of observed successes. The random variable $X$ follows the hypergeometric distribution, if its probability mass function has the form

$$\mathbb{P}(X = x) = \frac{\binom{K}{x}\binom{N-K}{n-x}}{\binom{N}{n}} \quad x = 0, 1, 2, \dots, n. \tag{3.3}$$

In R, there are 4 built-in functions to generate the hypergeometric distribution:

- `rhyper(N, m, n, k)`, generally refers to generating random numbers function by specifying a seed and sample size,
- `phyper(x, m, n, k)`, defines the cumulative distribution function of the hypergeometric distribution,
- `dhyper(x, m, n, k)`, defines the probability mass function of the hypergeometric distribution,
- `qhyper(N, m, n, k)`, is basically hypergeometric quantile function used to specify a sequence of probabilities between 0 and 1.

Here `x` represents the data set of values, `m` size of the population, `n` number of samples drawn, `k` number of items in the population, and `N` hypergeometrically distributed values.

**Example 3.2.5** *A committee of 5 people is to be selected from 10 women and 8 men. What is the probability that the committee will consist of 3 women and 2 men? What is the probability that in the committee will be a majority of women?*

**Solution**: To state the probability, that there will be 3 women in the committee we use the function `dhyper()`. By requirements $x = 3$ women in the committee, $m = 10$ total number of women in the group, $n = 8$ the total number of men in the group and $k = 5$ the number of the committee members. So we get

```
> dhyper(3,10,8,5)
[1] 0.3921569
```

Women can have the majority in the committee if there are 5, 4 or 3 women, or alternatively if there are at most 2 men. Therefore we have two alternatives, how to compute the probability. One approach is based on the `dhyper()` function when we get the answer as a sum of its three values:

```
> dhyper(5,10,8,5)+dhyper(4,10,8,5)+dhyper(3,10,8,5)
[1] 0.6176471
```

Alternatively, we can compute this probability using the `phyper()` function with arguments $x = 2$ men in the committee, $m = 8$ total number of men in the group, $n = 10$ the total number of women in the group and $k = 5$ the number of the committee members. We get the same result:

```
> phyper(2,8,10,5)
[1] 0.6176471
```

**Example** 3.2.6 *Suppose a shipment of 100 DVD players is known to have ten defective players. An inspector randomly chooses 15 for inspection. Let us simulate how many defective players will be selected in the sequence of 10 inspections.*

**Solution**: The shipment contains $m = 10$ defective DVD players and $n = 90$ non-defecive DVD players and inspector randomly selects $k = 15$. When the inspection is repeated $N = 10$ times, to simulate their results we apply the function `rhyper()` with given arguments. So we obtain:

```
> rhyper(10,10,90,15)
 [1] 4 1 1 0 2 0 1 2 3 2
```

## 3.2.4 Negative binomial distribution

The negative binomial distribution is a discrete probability distribution that models the number of successes in a sequence of independent and identically distributed Bernoulli trials before a specified (non-random) number of failures (denoted $n$) occurs. If we further denote $x$ the number of successes and the probability of the success as $p$, we can write the probability mass function of the negative binomial distribution in the form:

$$\mathbb{P}(X = x) = \binom{n+x-1}{n-1} p^x q^n, \quad x = 0, 1, 2, \ldots \tag{3.4}$$

where $q + p = 1$, $p > 0$, $q > 0$.

We have four built-in functions for handling negative binomial distribution in R:

- `rnbinom(N,n,prob)`, where `n` is numbers of trials, `N` is the sample size, `prob` is the probability of success. This function generates $N$ random variables of a particular probability.
- `pnbinom(x, n, p)`, is used to compute the value of negative binomial cumulative distribution function. Here `x` is number of failures prior to the `n`-th success, and `p` is probability of success.
- `dnbinom(x, n, p)`, is the probability of `x` failures prior to the `n`-th success (note the difference) when the probability of success is `p`.
- `qnbinom(x, n, p)`, is used to compute the value of negative binomial quantile function. Here `x` is the vector of quantile levels, `n` is the total number of trials and `p` is the probability of success in one trial.

**Example** 3.2.7 *An oil company conducts a geological study that indicates that an exploratory oil well should have a 20% chance of striking oil. What is the probability that the first strike comes on the third well drilled? What is the probability that the third strike comes on the seventh well drilled?*

**Solution**: To find the requested probability, we need to find $\mathbb{P}(X = 2)$ by formula (3.4) with $n = 14$. Note that is technically a geometric random variable, since we are only looking for one success. Since a geometric random variable is just a special case of a negative binomial random variable, we'll try finding the probability using the negative binomial probability mass function. Due to the implementation of the `dnbinom()` function, we set `x=2` failures prior `n=1` success and `p=0.2`. So we obtain the result

```
 > dnbinom(2,1,0.2)
[1] 0.128
```

To answer the second question we choose x=4 failures prior n=3 successes. Then we obtain

```
> dnbinom(4,3,0.2)
[1] 0.049152
```

### 3.2.5 Poisson distribution

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event. If random variable $X$ follows the Poisson distribution with parameter $\lambda > 0$ (the average number of events), its probability mass function has the form

$$\mathbb{P}(X = x) = \frac{e^{-\lambda}\lambda^x}{x!}, \quad x = 0, 1, 2, \dots \tag{3.5}$$

There are four Poisson functions available in R:

- `dpois(x,l)` calculates the probability mass function value $\mathbb{P}(X = x)$ of the Poisson distribution with the parameter $\lambda$ implemented as argument `l`.
- `ppois(x,l)` calculates the cumulative distribution function of a random variable that follows the Poisson distribution. It returns the probability, that the variable value is less or equal to `x`, the argument `l` is the parameter of the distribution. Stating the additional argument `lower.tail=FALSE` the right tail of the distribution is considered, means we get the probability $\mathbb{P}(X > x)$.
- `rpois(k,l)` is used for generating random numbers from a given Poisson distribution. Here `k` is number of random numbers needed and `l` is the parameter of the distribution.
- `qpois(q,l)` is used for generating quantile of a given Poisson's distribution. Here `q` is a vector of the quantile levels required and `l` is the parameter of the distribution.

**Example 3.2.8** *On a particular river, overflow floods occur once every 100 years on average. Calculate the probability of* $k = 0, 1, 2, 3, 4, 5,$ *or* $6$ *overflow floods in a 100-year interval.*

**Solution**: As the overflow occurs once in 100 years, we can consider it to be a rare event and suppose that the number of overflows follows the Poisson distribution. To find the the requested probabilities, we use the `ppois()` function for `x` being a vector of integers from 0 to 6 and parameter `l` equal to 1 overflow in 100 years. Then we enter the following code:

```
> x<-seq(0:6)
> dpois(x,1)
[1] 3.678794e-01 1.839397e-01 6.131324e-02 1.532831e-02 3.065662e-03
[6] 5.109437e-04 7.299195e-05
```

**Example 3.2.9** *A life insurance salesman sells on the average 3 life insurance policies per week. Let us calculate the probability that in a given week he will sell some policies.*

**Solution**: "Some policies" means "1 or more policies" and therefore we have to calculate the probability $\mathbb{P}(X > 0) = 1 - \mathbb{P}(X \leq 0)$. We will apply the function `ppois()` with the additional argument `lower.tail` set to `FALSE`. The parameter of the distribution is `l=3` policies sold in one week. So we obtain

```
> ppois(0,3,lower.tail=FALSE)
[1] 0.9502129
```

Alternatively we can compute the probability using the `dpois()` function as:

```
> 1-dpois(0,3)
[1] 0.9502129
```

**Example 3.2.10** *A company produces 300 electric motors daily. The probability an electric motor is defective is 0.01. Let us generate the number of defective motors made daily during one working week.*

**Solution**: The average number of defectives in daily production of 300 motors is $\lambda = 0.01 \times 300 = 3$. To generate the daily number of defectives we use the `rpois()` function with arguments `k=5` working days and `l=3`. So we get

```
> rpois(5,3)
[1] 3 3 4 2 2
```

**Example 3.2.11** *Consider a computer system with Poisson job-arrival stream at an average of 2 per minute. What is the maximum jobs that should arrive one minute with 90% certainty.*

**Solution**: To find a maximum arrivals with at least 90% certainty means to find the 90% quantile. We can apply the `qpois()` function with arguments `q=0.9` and `l=2` average arrivals per minute. So we have

```
> qpois(0.9,2)
[1] 4
```

So, here is at least 90% chance, that the number of arrivals does not exceed 4 per minute.

## 3.3 Continuous distributions

We can characterise the continuous distribution by words as a probability distribution whose support is an uncountable set, such as an interval in the real line. They are uniquely characterized by a cumulative distribution function that can be used to calculate the probability for each subset of the support. The meaning of the cumulative distribution function $F(x)$ is the probability $F(x) = \mathbb{P}(X \leq x)$.

Besides the cumulative distribution function, the continuous random variable can be described as well by the density function $f(x)$. Its value has no probabilistic meaning, but it is joined with the cumulative distribution function by the relation

$$F(x) = \int_{-\infty}^{x} f(t)\,\mathrm{d}t.$$

In the R environment are implemented many continuous distributions. In this short survey we will mention only some of them:

- uniform distribution,
- exponential distribution,
- normal distribution,
- Student *t* distribution,
- Chi square distribution,
- Fisher F distribution.

Let us note, that last three distributions are joined with confidence intervals and the statistical hypotheses testing. These distributions are implemented as they are needed in the specialized functions for performing the test and we meet them in the later chapters. Therefore we will concern here in some examples, that lead to the uniform, exponential and normal distribution.

### 3.3.1 Uniform distribution

The continuous uniform distribution describes an experiment where there is an arbitrary outcome that lies between certain bounds. More exactly, we say that random variable $X$ governs by the uniform distribution with parameters $a$ and $b$, $a < b$, if its density function has the form:

$$f(x) = \begin{cases} \frac{1}{b-a} & x \in \langle a; b\rangle \\ 0 & \text{elsewhere} \end{cases} \tag{3.6}$$

In the R environment, here are implemented the functions:

- `dunif()` that gives the density function, its arguments are vector `x` and parameters `min` and `max` of the distribution,
- `punif()` that gives the cumulative distribution function, its arguments are vector `x` and parameters `min` and `max` of the distribution,
- `qunif()` that gives the quantile function, its arguments are quantiles `q` and parameters `min` and `max` of the distribution,
- `runif()` that generates the random values of the variable, its arguments are size of the sample `n` and parameters `min` and `max` of the distribution.

**Example 3.3.1** *Let us suppose trams leave the stop at regular 5 minute intervals. We calculate what is the probability that the passenger will wait*

*a) more than 3 minutes or,*

*b) not more than 1.5 minutes,*

*if he comes to the stop in a random moment.*

**Solution**: Due to the regularity of the departure times, the waiting time is the random variable that governs by the uniform distribution with parameters $a = 0$ and $b = 5$. Therefore the probability that the passenger will wait more than 3 minutes $\mathbb{P}(X > 3) = 1 - F(3)$. Entering the code `1-punif(3,min=0,max=5)` we get the answer, that this probability equals 0.4:

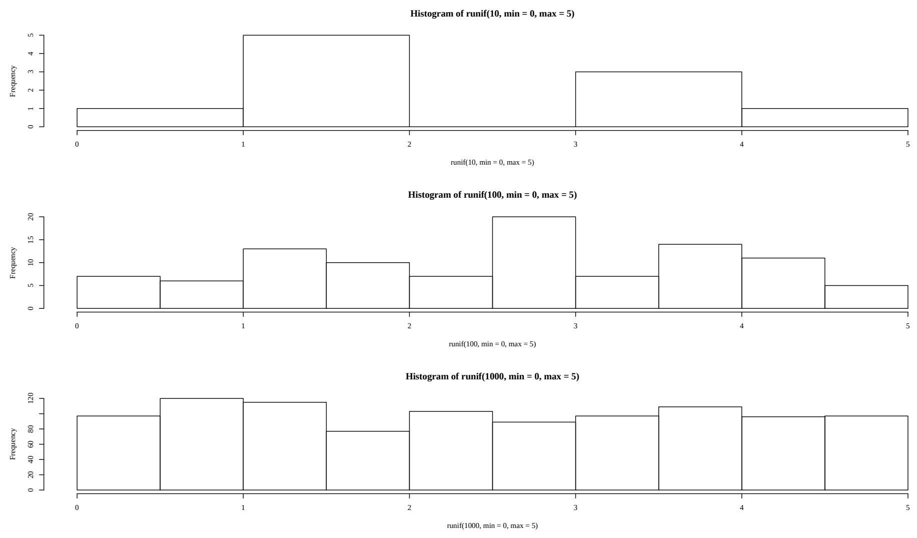


Figure 3.1: Histograms of 10, 100 and 1 000 simulations of the tram departures to the example 3.3.1. The number of simulations increases from top to bottom.

```
>  1-punif(3,min=0,max=5)
[1] 0.4
```

Similarly, the second question is about probability $\mathbb{P}(X \leq 1.5) = F(1.5)$. The requested result we get as `puinf(1.5,min=0,max=5)`, so the probability is 0.3.

```
> punif(1.5,min=0,max=5)
[1] 0.3
```

We can simulate the situation using the `runif()` function. Increasing the sample size we can also illustrate, how increasing number of the random experiments leads to better approximation of the distribution. The histograms of the samples of 10, 100 and 1 000 values are illustrated in figure 3.1. There we see, how with the increasing sample size from top to bottom the histogram is more uniformly distributed.

The source code is.

```
par(mfrow = c(3, 1))
hist(runif(10,min=0,max=5))
hist(runif(100,min=0,max=5))
hist(runif(1000,min=0,max=5))
```

### 3.3.2 Exponential distribution

The exponential distribution is the probability distribution that describes the waiting time between events in a Poisson point process, i.e., a process in which events occur continuously and independently at a constant average rate. The random variable $X$ governs by the exponential distribution with parameter $\lambda > 0$, usually reported as the rate, if its density function has the form:

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{3.7}$$

When the parameter $\lambda$ is interpreted as the rate, the mean waiting time is $1/\lambda$. In the R environment is the exponential distribution implemented with the following functions:

- `dexp()` that gives the density function, its arguments are vector `x` and parameter `rate` of the distribution,
- `pexp()` that gives the cumulative distribution function, its arguments are vector `x` and parameter `rate` of the distribution,
- `qexp()` that gives the quantile function, its arguments are quantiles `q` and parameter `rate` of the distribution,
- `rexp()` that generates the random values of the variable, its arguments are size of the sample `n` and parameter `rate` of the distribution.

**Example 3.3.2** *Suppose the mean checkout time of a supermarket cashier is three minutes. Find the probability of a customer checkout being completed by the cashier in:*

*a) less than two minutes,*

*b) more than five minutes.*

**Solution**: As we know the mean completing time, the checkout processing rate equals to its inverted value, one divided by the mean checkout completion time. Hence the processing rate is $1/3$ checkouts per minute. So the question a) is answered as the probability $\mathbb{P}(X < 2)$ that is computed using the `pexp()` and equals 0.4866.

```
> pexp(1/3,2)
[1] 0.4865829
```

Similarly, the question b| we answer by probability $\mathbb{P}(X > 2)$ that we can compute in two alternative approaches

```
>  1-pexp(1/3,5)
[1] 0.1888756
>  pexp(1/3,5,lower.tail=FALSE)
[1] 0.1888756
```

**Example 3.3.3** *Malfunction in a particular type of electronic device are known to follow an exponential distribution with a mean time of 30 months until the device malfunctions. Let us find the probability that.*

*a) a randomly selected device will malfunction within the first year (12 months),*

*b) a randomly selected device will last more than 6 years (72 months).*

**Solution**: Let us denote as $X$ the random variable that represents the time to the malfunction of the device. In the case a) we need to answer the question, what is the probability $\mathbb{P}(X < 12)$ if the the random variable $X$ follows the exponential distribution with parameter $\lambda = 1/30$. This probability equals 0.32968. In R we get the result by the command:

```
> pexp(1/30,12)
[1] 0.32968
```

In order to answer question b), we have to find the probability $\mathbb{P}(X \geq 70)$ that equals 0.0907. To get the answer using the `pexp()` function, we have to set the argument `lower.tail=FALSE`, how shows the code below:

```
> pexp(1/30,72,lower.tail=FALSE)
[1] 0.09071795
```

To illustrate the meaning of the quantiles we find the the length of time within which 60 percent of devices will have malfunctioned. We use the `qexp()` function how shows the following command:

```
> qexp(0.6,1/30)
[1] 27.48872
```

So 60 percent of devices will malfunction within approx. 27.5 months.

### 3.3.3 Normal distribution

A normal (or Gauss) distribution is a type of continuous probability distribution for a real-valued random variable. The general form of its probability density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-1/2\left(\frac{x-\mu}{\sigma}\right)^2}. \tag{3.8}$$

The parameter $\mu$ is the mean the distribution (and also its median and mode), while the parameter $\sigma$ is its standard deviation.The normal distribution is important because of the Central Limit Theorem, which states that the population of all possible samples of size $n$ from a population with mean $\mu$ and variance $\sigma^2$ approaches a normal distribution with mean $\mu$ and $\sigma^2/n$ when $n$ approaches infinity.

In the R environment, here are implemented the functions:

- `dnormf()` that gives the density function, its arguments are vector `x` and parameters `mean` and `sd` of the distribution,
- `pnorm()` that gives the cumulative distribution function, its arguments are vector `x` and parameters `mean` and `sd` of the distribution,
- `qnorm()` that gives the quantile function, its arguments are quantiles `q` and parameters `mean` and `sd` of the distribution,
- `rnorm()` that generates the random values of the variable, its arguments are size of the sample `n` and parameters `mean` and `sd` of the distribution.

**Example 3.3.4** *Assume that the test scores of a college entrance exam fits a normal distribution. Furthermore, the mean test score is 70, and the standard deviation is 10. What is the percentage of students*

*a) scoring 85 or more in the exam,*

*b) scoring 60 or less in the exam.*

**Solution**: We apply the function pnorm() of the normal distribution with mean 70 and standard deviation 10. Since we are looking for the percentage of students scoring higher than 85, we are interested in $\mathbb{P}(X \geq 85)$, means the upper tail of the normal distribution. Therefore we use the logical parameter lower.tail=FALSE.

```
> pnorm(85, mean=70, sd=10, lower.tail=FALSE)
[1] 0.0668072
```

Therefore, the percentage of students scoring 85 or more in the college entrance exam is 6.68%.

In order to answer the question b), we need to calculate the probability $\mathbb{P}(X < 60)$. We use the pnorm() function again:

```
> pnorm(60, mean=70, sd=10)
[1] 0.1586553
```

Therefore, the percentage of students scoring 60 or less in the college entrance exam is 15.87%.

**Example 3.3.5** *According to the data from www.uvzsr.sk, the average height of 18 years old boys in Slovakia was 179 cm wit the standard deviation of 6.68 cm in the year 2011. If we suppose that the height is normally distributed, let us find the probability, that randomly selected boy in age of 18 years would be*

*a) more than 200 cm tall,*
*b) less than 160 cm tall.*

**Solution**: Let us denote the random variable that describes the height as $X$, To answer the question a) we have to compute the probability $\mathbb{P}(X \geq 200)$. It equals 0.0008 and we find it using the function pnorm(), where we set the argument lower.tail=FALSE, how illustrated in the following command:

```
> pnorm(200,179,6.68,lower.tail=FALSE)
[1] 0.000834096
```

Similarly, to answer the question b) we use the function pnorm but we must not set the argument lower.tail in this case. So to find the probability $\mathbb{P}(X < 160)$ we use the following command.

```
> pnorm(160,179,6.68)
[1] 0.002225376
```

It means that the probability we were searching for is 0.002.

## 3.3.4 $\chi^2$ distribution

The chi-squared distribution (also chi-square ) or $\chi^2(n)$-distribution with $n$ degrees of freedom arises as the distribution of a sum of the squares of k independent standard normal random variables. Formally, if $X_i \sim N(0,1)$, $i = 1, \dots, n$ are independent random variables, then random variable $Y = X_1^2 + \cdots X_n^2$ follows the $\chi^2(n)$ distribution. Its density has the form

$$f(x) = \begin{cases} \frac{e^{-x/2} x^{n/2-1}}{2^{n/2}\Gamma(n/2)} & x > 0, \\ 0 & x \leq 0, \end{cases} \tag{3.9}$$

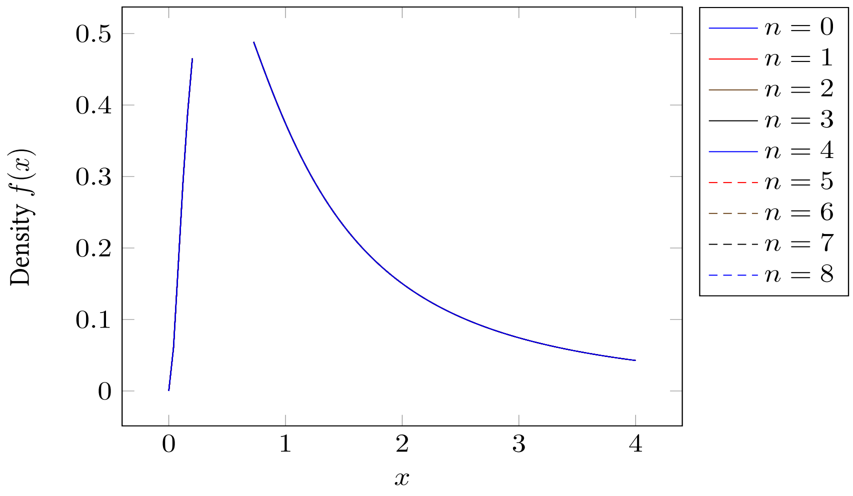


Figure 3.2: Density functions of the $\chi^2(n)$ distribution with different numbers of degrees of freedom $n$.

The parameter $n$ of the $\chi^2$ distribution is referred as the number of degrees of freedom. The graphs of the densities with different numbers of freedom are depicted in the figure 3.2. This distribution is used to construct the confidence interval a tests for the variance of the normal distribution and in the goodness of fit and independence tests. In the R language the chi-squared distribution is assigned as `chisq`. That means, here are implemented functions `pchisq()`, `dchisq()`, `qchisq()` and `rchisq()`.

### 3.3.5 Student's *t* distribution

The Student's $t(n)$-distribution with $n$ degrees of freedom arises as the distribution of the ratio of two random variables: $X$ that follows the standardized normal distribution $N(0, 1)$ and $\sqrt{Y/n}$, where $y$ follows the $\chi^2(n)$ distribution. Its density can be expressed in the form

$$f(x) = \frac{\Gamma\left((n+1)/2\right)}{\sqrt{n\pi}\,\Gamma\left(n/2\right)\left(1 + x^2/n\right)^{(n+1)/2}}, \quad x \in \mathbb{R}, \tag{3.10}$$

where the parameter $n$ represents the number of degrees of freedom. The graphs of the densities with different numbers of freedom are depicted in the figure 3.3. The $t$-distribution plays a role in a number of widely used statistical analyses, including Student's $t$-test for assessing the statistical significance of the difference between two sample means, the construction of confidence intervals for the difference between two population means, and in linear regression analysis.

In the R environment, the Student's distribution is implemented simply as `t`. Using the prefixes, we have at disposal functions `pt()`, `dt`, `qt()` and `rt()`.

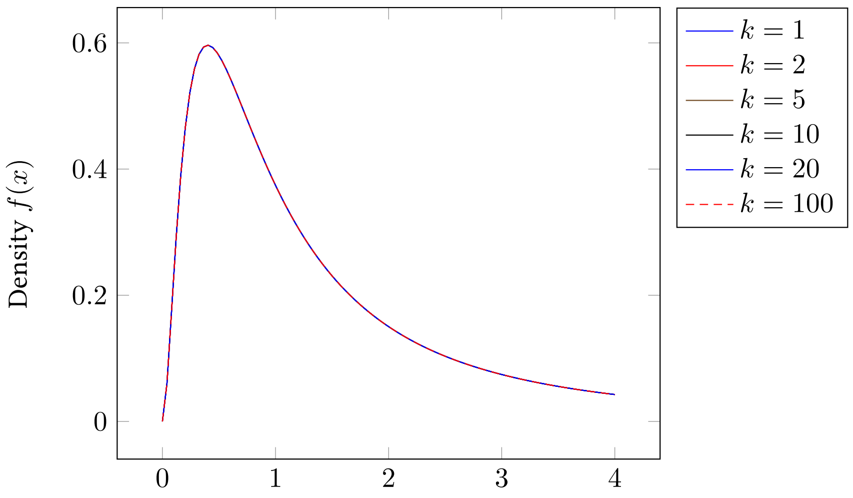


Figure 3.3: Denstity of the Student's distribution $t(n)$ with different numbers of degrees of freedom $n$.

## 3.3.6 *F*-distribution

The $F$-distribution, also known as Snedecor's $F$-distribution or the Fisher-Snedecor distribution arises as the distribution of the random variable $Y$ that is the ratio of two chi-square distributed random variables. More precisely let random variable $X_1$ follows the distribution $\chi^2(n_1)$ and $X_2$ follows the distribution $\chi^2(n_2)$. Than the random variable

$$Y = \frac{X_1/n_1}{X_2/n_2}$$

follows the Fisher-Snedecor distribution $F(n_1, n_2)$ with $n_1$ and $n_2$ degrees of freedom. We can express its density with use of the Beta function in the form

$$f(x) = \begin{cases} \left(\frac{n_1}{n_2}\right)^{n_1/2} \cdot \frac{x^{n_1/2-1}}{B(n_1/2, n_2/2)} \left(1 + n_1/n_2 x\right)^{-(n_1+n_2)/2} & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{3.11}$$

How the numbers of freedom influence the shape of the density we can observe on figures 3.4 and 3.5. The Fisher-Snedecor distribution is of great importance in mathematical statistics. It is used in hypothesis tests for equality of variances of two sets (called F-tests) or in analysis of variance. In the R environment, the Fisher-Snedecor distribution is implemented simply as `f`. Using the usual prefixes, we have at disposal functions `pf()`, `df`, `qf()` and `rf()`.

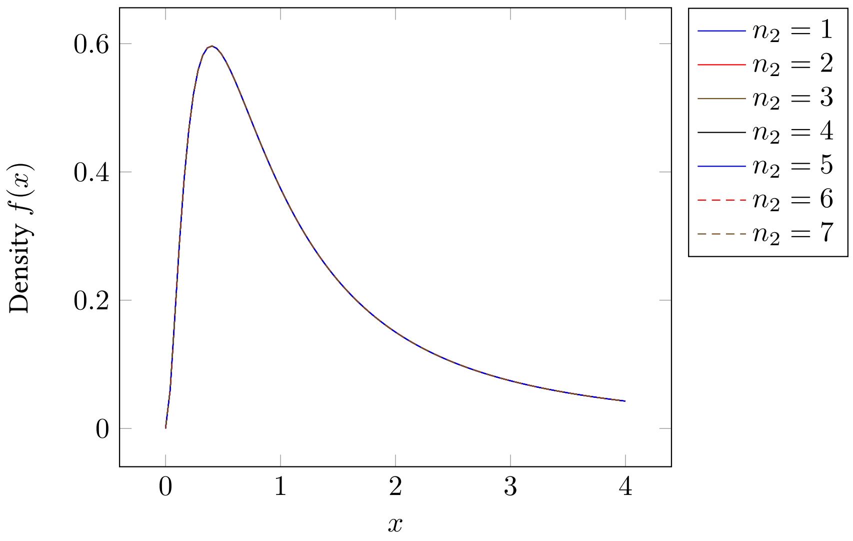


Figure 3.4: Density of the Fisher-Snedecor distribution $F(3, n_2)$ for different numbers of degrees of freedom $n_2$.

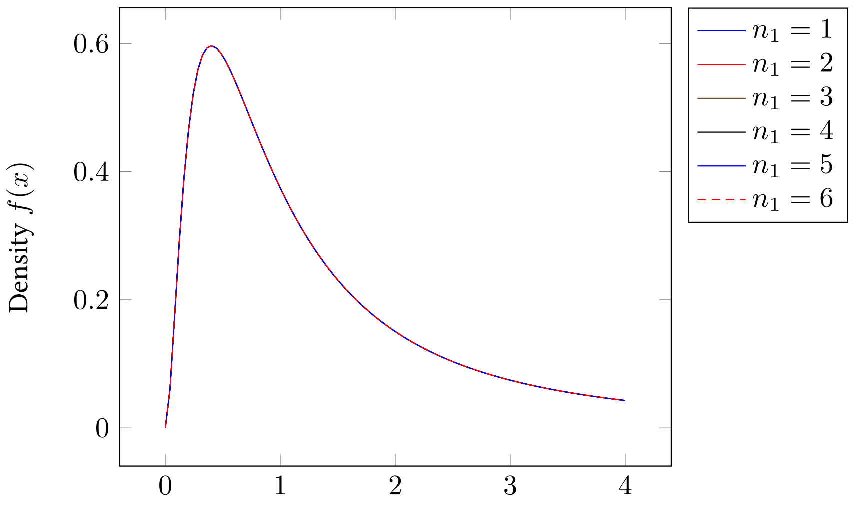


Figure 3.5: Density of the Fisher-Snedecor distribution $F(n_1, 3)$ for different numbers of degrees of freedom $n_1$.

# Programming in R

## 4.1 Built-in functions

Almost all actions in R are made through functions. The built-in functions are functions that are already defined and implemented in the programming framework. The R environment includes a rich set of the built-in functions that enable to perform almost every task. These built-in functions can be divided into following categories with respect to their functionality.

### 4.1.1 Math functions

This category of functions provides a large variety of mathematical functions to perform the mathematical calculation. Some of them we have already mention in the table 1.5. Here we introduce some additional details to using these functions.

We start with the trigonometric functions. All three mentioned functions `sin(x)`, `cos(x)` and `tan(x)` work with argument given in radians. It means, that in the case of using the grades, we have to reshape the value. Knowing that $\pi = 180^\circ$, we can recalculate the argument as $r = \pi\alpha/180$, where $r$ is the new measure in radians and $\alpha$ is the old value given in grades. Alternatively we can also use the function `deg2rad()` from the `REdaS` package. We illustrate the result in the following listing.

```
> library(REdaS)
> sin(90)
[1] 0.8939967
> sin(deg2rad(90))
[1] 1
> tan(45)
[1] 1.619775
> tan(deg2rad(45))
[1] 1
```

The logarithmic function `log()` computes as the default value the natural logarithm. To get logarithm with any base, we must declare the base value as the second argument of the function. Let us see the listing.

```
> log(4)
[1] 1.386294
> log(4,base=2)
[1] 2
```

One of the frequently used function in mathematics is the `sqrt()` function. We have already mention its ability to work with the complex numbers, but it is necessary to declare in the function argument, that we require this kind of answer. We can do it by using the imaginary unit `i` or the function `as.complex()`. Let us see the listing.

Table 4.1: List of the functions defined for the complex numbers.

| *Function* | *Purpose* | *Function* | *Purpose* |
|---|---|---|---|
| `Re(z)` | Real part of $z$. | `Im(z)` | Imaginary part of $z$. |
| `Mod(z)` | Modulus of $z$. | `Arg(z)` | Argument of $z$. |
| `Conj(z)` | Conjugate complex number $\overline{z}$. | | |

```
> sqrt(-25)
[1] NaN
Warning message:
In sqrt(-25) : NaNs produced
> sqrt(-25+0i)
[1] 0+5i
> sqrt(as.complex(-25))
[1] 0+5i
```

Some other useful functions for computing with the complex numbers are listed in the table 4.1. Their using illustrates the following listing:

```
> Mod(1+1i)
[1] 1.414214
> Arg(1+1i)
[1] 0.7853982
> Re(1+1i)
[1] 1
> Im(1+1i)
[1] 1
> Conj(1+1i)
[1] 1-1i
```

### 4.1.2 String functions

Besides the numerical data we frequently need to extract information from textual data or reformat textual data. For example we may want to concatenate the first and last name of the persons and ensure that first letter of each is capitalized. In such situations are useful the string functions, that are listed in the table 4.2.

Function `nchar()` determines the size of each elements of an character vector. Its answer is an integer vector giving the sizes of each element. In general, it has four arguments. The first is the string vector, the second argument is `type` that declares a type of chars. It can be `chars`, `bytes` or `width`. The third of the function argument is logical variable `allowNA` that determinates if should NA be returned for invalid multibyte strings or "bytes"-encoded strings. Finally, the last argument `keepNA` is also logical and it states if `NA` should be returned where ever `x` is `NA`. If `FALSE` the function `nchar()` returns 2, as that is the number of printing characters used when strings are written to output. We illustrate its using in the next listing:

```
> z<-c("yellow","black","white")
> nchar(z)
[1] 6 5 5
> str<-"This␣is␣a␣long␣string"
> nchar(str)
```

Table 4.2: List of the functions defined for the string operations.

| *Function* | *Purpose* | *Function* | *Purpose* |
|---|---|---|---|
| `nchar()` | Number of characters in string. | `substr()` | Extract or replace substrings. |
| `grep()` | Search for a pattern in string. | `strsplit()` | Splits the string at given split point. |
| `sub()` | Search for a pattern in string and substitute it. | `paste()` | Concatenates the strings using submitted separate therm. |
| `toupper()` | Converts the string into the upper case. | `tolower()` | Converts the string into the lower case. |

```
[1] 21
> z<-c("",NULL,"black",NA)
> nchar(z,keepNA=TRUE)
[1]  0  5 NA
> nchar(z,keepNA=FALSE)
[1] 0 5 2
```

Related feature is a function `nzchar()` that represents a fast way to find out if elements of a character vector are non-empty strings. How we can see in the following listing, the answer is a logical vector.

```
> z<-c("yellow","black","white")
> nzchar(z)
[1] TRUE TRUE TRUE
> z<-c("",NULL,"black",NA)
> nzchar(z)
[1] FALSE  TRUE  TRUE
```

To find a subscribed pattern in the string we use the `grep()` function. This function has formally four arguments. Its first argument is `pattern` submitted as a string we are searching for. The second argument is the string vector `x` we are searching in. The optional arguments are logical values `ignore.case` and `fixed`. The default value of `ignore.case` is set `FALSE` and the upper and lower cases are distinguished in the search. If we set `fixed=FALSE`, which is the default value of the argument, then `pattern` is a regular expression. If `fixed=TRUE`, then `pattern` is a text string. The function returns matching indices. The next source code illustrates using of the function and its answers.

```
> str <- c('abcd','bdcd','abcdabcd')
> pattern<- 'abc'
> grep(pattern, str)
[1] 1 3
> pattern<- 'Abc'
> grep(pattern, str)
integer(0)
> grep(pattern, str,ignore.case=TRUE)
[1] 1 3
> pattern<- 'a*'
> grep(pattern, str)
[1] 1 2 3
```

```
> grep(pattern, str,fixed=TRUE)
integer(0)
```

To substitute the found pattern by another string we use the function `sub()`. It has in general five arguments. Obligatory arguments are `pattern` and vector of strings `x` that have the same meaning as in the `grep()` function. The third, but written on the second position, obligatory argument is `replacement` that defines the text to replace the `pattern`. Two optional arguments `ignore.case` and `fixed` are the same as in the `grep()` function. How the function works we see in the following listing.

```
> str<-"Bohemia␣does␣not␣use␣EURO␣currency"
> str<-sub("Bohemia","Czechia",str)
> str
[1] "Czechia␣does␣not␣use␣EURO␣currency"
```

Another function to manipulate the text string is `substr()`. It has three arguments: the text string `x` and `start` and `stop` to declare position of the first and last character to be selected or replaced. The previous substitution in the string we can be alternatively done using the `substr()` function,.The following code illustrates the alternatively solution.

```
> str<-"Bohemia␣does␣not␣use␣EURO␣currency"
> substr(str,1,7)
[1] "Bohemia"
> substr(str, 1, 5)<-"Czech"
> str
[1] "Czechia␣does␣not␣use␣EURO␣currency"
```

Function `strsplit()` splits the elements of the character vector `x` at positions defined by the second `split` argument. We illustrate, how the sentence can be split into single word or letters. The third example shows split by any pattern.

```
> strsplit(str,"")
[[1]]
[1]"C" "z" "e" "c" "h" "i" "a" "␣" "d" "o" "e" "s" "␣" "n" "o" "t"
    "␣" "u" "s"
[20] "e" "␣" "E" "U" "R" "O" "␣" "c" "u" "r" "r" "e" "n" "c" "y"
> strsplit(str, "␣")
[[1]]
[1] "Czechia"  "does"     "not"      "use"      "EURO"     "currency"
> strsplit(str,"e")
[[1]]
[1] "Cz"         "chia␣do"    "s␣not␣us"   "␣EURO␣curr" "ncy"
```

To concatenate the strings we use function `paste()`, whose arguments are the strings to be concatenated and `sep` that defines the string to separate the concatenated elements. How the function works we see in the following code.

```
> paste("x",1:4,sep="")
[1] "x1" "x2" "x3" "x4"
> paste("Today␣is",date(),sep="␣")
[1] "Today␣is␣Tue␣Apr␣27␣10:39:55␣2021"
> paste(c("a","b"),1:4,sep="/")
[1] "a/1" "b/2" "a/3" "b/4"
```

Two related functions `toupper()` and `tolower` transform given string into the upper case and lower case letters respectively. Let us see the code.

```
> toupper(str)
[1] "CZECHIA␣DOES␣NOT␣USE␣EURO␣CURRENCY"
> tolower(str)
[1] "czechia␣does␣not␣use␣euro␣currency"
```

### 4.1.3 Statistical and probability functions

Many built-in functions related to the probabilities we have already discuss in the chapter 3. There are summarized all prefixes for the probabilistic functions and as well most of the implemented probability distribution. In this subsection we introduce some functions used in statistics, especially the sample characteristics. Common statistical function are presented in table 4.3.

Similarly as all functions, as well statistical functions have optional arguments that affect their outcomes. To state the sample mean of the given vector `x`, we use the function `mean(x)`. It has also optional argument `trim` that state the percentage of the highest and lowest values being dropped from the computation and so it returns the trimmed mean. Second optional argument `na.rm` is a logical value indicating whether 'NA' values should be stripped before the computation proceeds. Let us compare the results in the listing.

```
> x<-c(1,3,5,10,12)
> mean(x)
[1] 6.2
> mean(x,trim=0.2)
[1] 6
> x<-c(1,5,2,12,NA,3,6)
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 4.833333
> mean(x,na.rm=TRUE,trim=0.17)
[1] 4
```

Let us suppose, we want to analyse the the delays of the train Nr. 172 Hungaria in the station Brno hl.n. (main station) during the week from April the 21-st till April the 27-th. The reported delays in single days were 0,9,0,42,14,0, and 11 minutes. Using the statistical functions we find that the average delay is

```
> delay<-c(0,9,0,42,14,0,11)
> mean(delay)
[1] 10.85714
```

and its standard deviation resp. variance is:

```
> sd(delay)
[1] 14.92681
> var(delay)
[1] 222.8095
```

For the median delay we get

```
> median(delay)
[1] 9
```

Besides median, we can find other quantlies using the `quantile()` function. Its default outcome are the quartiles, how illustrates the following listing.

Table 4.3: List of the statistical functions .

| *Function* | *Purpose* | *Function* | *Purpose* |
|---|---|---|---|
| `mean()` | Sample mean. | `median()` | Sample median. |
| `sd()` | Standard deviation. | `var()` | Sample variance. |
| `mad()` | Median absolute deviation. | `quantile()` | Quantiles, default returns quartiles. |
| `range()` | Range of the values. | `sum()` | Sum of the vector elements. |
| `min()` | Minimum. | `max()` | Maximum. |

```
> quantile(delay)
  0%  25%  50%  75% 100%
 0.0  0.0  9.0 12.5 42.0
```

To specify the probability levels for the quantiles, we must set the optional argument `prob` of the `quantile` function. This argument requires the probabilities in the form of the numeric vector, how we can see in the listing.

```
> quantile(delay,prob=c(0,0.33,0.67,1))
   0%    33%    67%   100%
 0.00   0.00  11.06  42.00
```

In addition to the standard deviation and variance there is a robust measure of the variability of a univariate sample of quantitative data. This measure is called median absolute deviation and for the sample $X_1, \dots, X_n$ it is defined by formula:

$$\mathrm{MAD}(X) = \mathrm{median}\{|X_i - \overline{X}|\}$$

where $\overline{X}$ is median of the data. This measure is implemented in R as the function `mad()`. So we get the median absolute deviation of the train departure delays using the following code:

```
> mad(delay)
[1] 13.3434
```

The range of the recorded delays we obtain using the function `range()` whose outcome is a vector including the start and end points of the range of the sample values. How we can see from the source code, alternatively we can state the range using the functions `max()` and `min()` that provide the maximum and minimum of the sample respectively.

```
> range(delay)
[1]  0 42
> c(min(delay),max(delay))
[1]  0 42
```

Computing the sum of the delays does not give a real sense, but we can illustrate how to state the sample mean alternatively using the `sum()` function.

```
> sum(delay)/length(delay)
[1] 10.85714
```

Table 4.4: List of the other useful functions .

| *Function* | *Purpose* | *Function* | *Purpose* |
|---|---|---|---|
| `seq()` | Generate a sequence | `rep()` | Repeat x n-times. |
| `sort()` | Sort the vector x | `order()` | List sorted element numbers. |
| `ls()` | List objects in current environment | `rev()` | List the elements of x in reverse order |

### 4.1.4 Other useful functions

List (very incomplete) of some other useful functions is presented in table 4.4. The function `seq()` has three optional arguments `from`, `to` and `by` and it generates the sequence of numbers starting by `from` value and ends in `to` value. The last argument `by` defines the step of the sequence. We can illustrate the use in the listing:

```
> seq(10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(5,15)
 [1]  5  6  7  8  9 10 11 12 13 14 15
> seq(5,15,2)
[1]  5  7  9 11 13 15
```

Function `rep()` has two arguments, the vector x to be repeated and number n of the repeating cycles. We illustrate the use in the following listing:

```
> rep(1,10)
 [1] 1 1 1 1 1 1 1 1 1 1
> rep(c(1,3),4)
[1] 1 3 1 3 1 3 1 3
> rep("hello",3)
[1] "hello" "hello" "hello"
```

The functions `sort()` and `order` are closely related. They are both joined with sorting the vectorx, but `sort()` gives sorted values while `order()` gives the indices of ordered the components in the original vector. Let us see the difference in the listing:

```
> x<-c(5,2,10,3,7,8)
> sort(x)
[1]  2  3  5  7  8 10
> order(x)
[1] 2 4 1 5 6 3
```

Both functions have logical argument `decreasing` with default value `FALSE`. Setting it to `TRUE`, we get sorted sequence in descending order.

Function `rev()` gives the vector x in reverse order. If we apply it on the vector x from the previous listing, we get:

```
> rev(x)
[1]  8  7  3 10  2  5
```

Combining the functions `rev()` and `sort()` we can get the same result like the function `sort()` with option `decreasing=TRUE`:

Table 4.5: List of the relation operators used in conditional statements.

| *operator* | *meaning* | *operator* | *meaning* |
|---|---|---|---|
| `==` | equals | `!=` | not equal. |
| `>` | greater than | `<` | less than |
| `>=` | greater than or equals | `<=` | less than or equal |

```
> rev(sort(x))
[1] 10  8  7  5  3  2
```

The complete list of the built in function we get typing the command `builtins()`. Further, using the command `help(function_name)` we obtain the full description of each from the built in functions.

## 4.2 Program flow controls

### 4.2.1 Conditional statements

A condition is, in general, understood as an expression that can be either true or false. The conditional statement then allows to perform a command or set of commands only under certain conditions. Conditional statements include `if()`, the combination `if()/esle()`, `ifelse()`, and `switch()`. Each statement supports source code branching by altering the control flow.

#### If statement

The `if()` statement is common in all programming languages and it is the simplest conditional statement. When R runs, the `if()` statement performs operations based on a simple condition. The general syntax of the `if()` statement is:

```
if (condition) {commands to be performed if condition holds}
```

The commands need to be braced only when more than one command is specified. Otherwise we can write one conditionally performed command without braces. In the condition we can use the infix operators listed in table 4.5.

As a simple example we illustarte using the `if` statement to detect if given integer number is odd.

```
> x<-5
> if(x%%2){print("Odd␣number")}
[1] "Odd␣number"
> x<-6
> if(x%%2){print("Odd␣number")}
>
```

#### `If ...` else statement

This extension of the `if` statement has general syntax in the form:

```
if (test_expression) {
statement1
} else {
statement2
}
```

How we can see, this enables to execute different commands depending on the fulfilment of the condition. If the `test_expression` is true, then the `statement1` is performed and if it is false the `statement2` is executed. We can illustrate it in generalisation of the previous code to detecting the parity of the number.

```
> x<-5
> if(x%%2){print("Odd␣number")} else {print ("Even␣number")}
[1] "Odd␣number"
> x<-10
> if(x%%2){print("Odd␣number")} else {print ("Even␣number")}
[1] "Even␣number"
```

We can further customize the control level with nesting the else if statement. With else if, we can add as many conditions as we want. The syntax is:

```
if (condition1) {
    statement1
    } else if (condition2) {
    statement2
    } else if  (condition3) {
    statement3
    } else {
    statement4
}
```

Using of the nested `if` statements we can illustrate on the example with three different VAT levels.

**Example 4.2.1** *VAT has different rate according to the product purchased. Imagine we have three different kind of products with different VAT applied (actually valid in Slovakia):*

| Category | Products | VAT |
|---|---|---|
| *A* | *Masks, respirators (actually freed from VAT)* | *0%* |
| *B* | *Selected foods, books, magazines, medicaments* | *10%* |
| *C* | *All others* | *20%* |

*Let us write a chain to apply the correct VAT rate to the product a customer bought.*

```
> category <- 'B'
> price<-50
> if (category =='A'){
cat('A␣vat␣rate␣of␣0%␣is␣applied.','The␣total␣price␣is',price *1.00)
} else if (category =='B'){
cat('A␣vat␣rate␣of␣10%␣is␣applied.','The␣total␣price␣is',price *1.10)
} else {
cat('A␣vat␣rate␣of␣20%␣is␣applied.','The␣total␣price␣is',price *1.20)
}
A vat rate of 10% is applied. The total price is 55
```

### The `ifelse` statement

The `if` and `if ... else` statements should not be applied when the condition being evaluated is a vector. It is best used only when meeting a single element condition. In the case of vector condition, the `if` statement evaluates the condition only for the first element of the vector. If we enter the code:

```
> x<-c(5,4,3,2,1)
> if(x>3){x*2}
```

one can expect the result to be `10,8,3,2,1`. But the real outcome is:

```
[1] 10  8  6  4  2
Warning message:
In if (x > 3) { :
 the condition has length > 1 and only the first element will be used
```

To get the expected outcome we have to apply the `elseif` statement with general syntax:

`ifelse(condition, expression1, expression2)`

The `ifelse()` function evaluates both expression1 and expression2 and then returns the appropriate values from each based on the element-by-element value of condition. If an element passes condition as TRUE, `ifelse()` returns the corresponding value of `expression1`; otherwise, it returns `expression2`. To get the expected result, our previous code should me modified into the following form:

```
> ifelse(x>3,2*x,x)
[1] 10  8  3  2  1
```

### The `switch()` function

The `switch()` is useful when we want the function to do different things in different circumstances. This function tests an expression against elements of a list. Each value in the list is called case. If the value evaluated from the expression matches item from the list, the corresponding value is returned. Here is the syntax of the `switch()` function:

`switch (expression, list)`

Here, the expression is matched with the list of values and the corresponding value is returned. If there is more than one match for a specific value, then the switch statement will return the first match found of the value matched with the expression.

We illustrate the use of the `switch()` function in decision if the given integer is even or odd. Let us note we have to add `+1` in the expression as the cases are numbered starting from 1.

```
> x<-10
> switch(x%%2+1,"even","odd")
[1] "even"
> x<-9
> switch(x%%2+1,"even","odd")
[1] "odd"
```

The `switch()` function can be used as well with the string expression. If the expression is a character string, switch() will return the value based on the name of the element. As an example, let us see the following code.

```
> x <- "a"
> switch(x, "a"="apple", "b"="banana", "c"="cherry")
[1] "apple"
> x <- "c"
> switch(x, "a"="apple", "b"="banana", "c"="cherry")
[1] "cherry"
```

How we mentioned above, in case of multiple matches, the value of first matching element is returned.

```
> x <- "a"
> switch(x, "a"="apple", "a"="apricot", "a"="avocado")
[1] "apple"
```

In the `switch()` we can define the default value which is returned if no match is present. In such situations is returned the unnamed case element. If more than one unnamed elements are present in the list an error occurs. We illustrate the default case in the following code:

```
> x <- "x"
> switch(x, "a"="apple", "b"="banana", "c"="cherry","some␣fruit")
[1] "some␣fruit"
```

### 4.2.2 Loops

All modern programming languages provide special constructs that allow for the repetition of instructions or blocks of instructions. The R language is no exception. According to the R base manual, among the control flow commands, the loop constructs are `for`, `while` and `repeat`, with the additional clauses `break` and `next`.The next sections will take a closer look at each of these structures.

#### `for` loops in R

The `for` loop allows us to repeat a command or a block of commands a fixed number of times. The number of repeating is given by scope of values or by a predefined sequence of permissible values. The general syntax of the `for` loop is like this:

```
for (val in sequence)
{
statement
}
```

where `sequence` is a vector and `val` takes on each of its value during the loop. In each iteration, `statement` is evaluated. Below is an example to count the number of numbers in a vector that exceeds its average value.

```
> x<-c(2,5,10,8,6,3,12)
> limit<-mean(x)
> count<-0
> for(i in x){
if (i>limit) count<-count+1
  }
> count
[1] 3
```

We can stop the loop before it has looped through all the items applying the `break` statement. Let us see the following source code:

```
> x<-c(2,4,6,5,8,10,11,12,14,20)
> for (i in x){
   if(i%%2==1) {break}
  print(i/2)
  }
[1] 1
[1] 2
[1] 3
```

The loop stops on the first odd number because we select to finish the loop by `break` statement. With the `next` statement, we can skip an iteration without terminating the loop. When the loop in the modified source code below passes odd value, it will skip it and continue to loop.

```
> x<-c(2,4,6,5,8,10,11,12,14,20)
> for (i in x){
  if(i%%2==1) {next}
  print(i/2)
  }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 10
```

### `while` loops

the `while` loop is suitable when we want to repeat a command or block of commands until a given condition is satisfied however we do not know in advance how many repetitions are necessary to achieve this. the syntax of the `while` loop is.

```
while (condition){
 commands
}
```

Here, `condition` is evaluated and the body of the loop is entered if the result is `TRUE`.

We will illustrate the use of the `while` loop in simulating the die rolls. The source code represents rolling the imaginary dice until the first roll of six.

```
> roll<-0
> while(roll!=6){
  roll<-sample(1:6,1)
  print(roll)
  }
```

```
[1]  1
[1]  4
[1]  3
[1]  4
[1]  6
```

### repeat loops

The `repeat` loop is similar to the `while` loop, but it is made so that the block of commands is executed at least once, no matter what the result of the condition. The general syntax of the `repeat` loop is like this:

```
repeat {
statement
}
```

There is no condition check in repeat loop to exit the loop. We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. We can modify the previous code using the `repeat` loop into the form:

```
> repeat{
  roll<-sample(1:6,1)
  print(roll)
  if(roll==6){break}
  }
[1]  5
[1]  2
[1]  1
[1]  5
[1]  6
```

## 4.3 User defined functions

The users' ability to add functions is one of the great strengths of the R environment. The general structure of a function is given below.

```
myfunction_name <- function(arg1, arg2, ... ){
statements
return(object)
}
```

The different components of a function are:

- **Function Name** which is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** which are placeholders. When a function is invoked, we pass values to the arguments. Arguments are optional, that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** which contains a collection of statements that defines what the function does. The body of the function goes inside the curly brackets `{}`.

- **Return Value** which is the last expression in the function body to be evaluated.

We will illustrate the user definition of the function in the following example.

**Example 4.3.1** *Let us suppose, we want to define function* `cubes()` *that prints the third powers of numbers in sequence.*

We define the function by the following source code:

```
cubes <- function(a) {
   for(i in 1:a) {
      b <- i^3
      print(b)
   }
}
```

Now we are ready to call the function and get the answer:

```
> cubes(6)
[1] 1
[1] 8
[1] 27
[1] 64
[1] 125
[1] 216
```

We can define this function as well without arguments. In such circumstances it produces the sequence of the cube powers of the constant length. The definition of the function `cube()` without arguments is as follows (in this case it produces the sequence of the first five cubic powers) :

```
cubes <- function() {
   for(i in 1:5) {
      b <- i^3
      print(b)
   }
}
```

Let us emphasize that when calling the function without arguments we cannot omit the parentheses, so to get the answer we have to call the function as follows:

```
> cubes()
[1] 1
[1] 8
[1] 27
[1] 64
[1] 125
```

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments. Let us rewrite the `cubes()` function with two arguments, that allow to submit the starting and ending value of the cubes sequence. So we have the source code:

```
cubes <- function(start,end) {
   for(i in start:end) {
      b <- i^3
      print(b)
   }
}
```

Now we can call the function by position of the arguments (let us note the decreasing sequence of cubes due to larger first argument):

```
> cubes(12,10)
[1] 1728
[1] 1331
[1] 1000
```

Alternatively we can call the function by names of the arguments (let us note, when calling by names we must not save the order of arguments in the function definition):

```
> cubes(end=12,start=10)
[1] 1000
[1] 1331
[1] 1728
```

We can define the default values of the arguments in the function definition. Then we can call the function without submitting the arguments. The arguments values are submitted only to describe their values. Let us note, that in R it is not obligatory to submit all arguments but is good enough to call the function by names and declare only the values of the arguments to be changed. So we can redefine the `cubes()` function with default arguments:

```
cubes <- function(start=1,end=10) {
   for(i in start:end) {
      b <- i^3
      print(b)
   }
}
```

Then we call the function with new the ending value:

```
> cubes(end=4)
[1] 1
[1] 8
[1] 27
[1] 64
```

Until now we have been printing the value on the console. To explicitly return values, we use the `return()` function. This allows to store the function output for the later work with it. if we try to store the result of previously defined function `cubes()` in variable `z`, we get:

```
> z<-cubes(2,2)
[1] 8
> z
NULL
```

It means the variable `z` does not include any value. We must redefine the function in the following manner:

```
cubes <- function(start=1,end=10) {
   for(i in start:end) {
      b <- i^3
      return(b)
   }
}
```

Then we can run the commands:

```
> z<-cubes(2,2)
> z
[1]  8
```

The function `cubes()` actually returns only one value that is equal to the cube of the starting value and ignore the rest of the declared scope. To extend the result to whole scope, we must define the output variable as vector how illustrates the following source code:

```
cubes <- function(start=1,end=10) {
    b<-vector()              # initializing the vector
   for(i in start:end) {
        b[i-start+1]<-i^3  # adjusting the index to start form 1
   }
   return(b)
}
```

Now we obtain the complete sequence of third powers in the submitted extent:

```
> z<-cubes(4,8)
> z
[1]  64 125 216 343 512
```

In R programming, functions do not return multiple values. However, we can create a list that contains multiple objects that we need a function to return. For example:

```
powers<-function(start=1,end=10) {
        b<-vector()
        c<-vector()
        for(i in start:end) {
        b[i-start+1]<-i^2
        c[i-start+1]<-i^3
   }
   out<-list(b,c)
   return(out)
}
```

gives the output

```
> powers(1,5)
[[1]]
[1]  1  4  9 16 25

[[2]]
[1]   1   8  27  64 125
```

When defining the user function we can use ellipsis. Ellipsis are not something which is specific to R programming as it is known from many other programming languages. An ellipsis is denoted in R programming by ... (exactly three dots). Its role is to enable the function to take any number of named or unnamed arguments. It is especially useful for creating customized versions of existing functions or in providing additional options to end-users. Ellipsis ... include three consecutive periods at the end of a function declaration. Also include three periods in at the end of the argument list for the function to pass arguments to. The ellipsis is just a special argument, so be sure to include a comma between any other arguments and the ellipsis. Following source code illustrates implementation of the ellipsis. The function `ellipsisDemo()` only returns the names and values of all submitted variables.

```
> ellipisDemo <- function(...) {
    cat("I␣got␣the␣following␣arguments:\n")
    print(list(...))
  }
> ellipisDemo(a = 5, by = "orange", c = c(2,3))
I got the following arguments:
$a
[1] 5
$by
[1] "orange"
$c
[1] 2 3
```

Combination of the named arguments we illustrate by defining the function `my_histogram()` that creates histogram of the given random sample.[1]

```
myhistogram <- function(x, border="blue", ..){
 hist(x,border=border,...)
 }
> set.seed(5)
> myhistogram(rnorm(1000), breaks=30)
> myhistogram(rnorm(1000),border="red")
```

In the source we have defined the function containing the named arguments `x` for the sample and `border` that states the colour of the histogram with default value `blue`. In the first calling the function we have used the additional argument `breakes`, whichg specifies number of columns in the histogram and the resulting graph we see in figure 4.1. The second call of the function dose not used the ellipsis arguments, but changes the default colour to red. The result we see in figure 4.2.

## 4.4 Running scripts in R

While entering and running our code at the R command line is effective and simple. However, this technique has its limitations. Each time we want to execute a set of commands, we have to re-enter them at the R command line. Moreover, complex commands are potentially subject to typographical errors, necessitating that they be re-entered correctly. Repeating a set of operations requires re-entering the code stream. Fortunately, R provides a method to mitigate these issues. R scripts are that solution. An R script is simply a text file containing (almost) the same commands that we would enter on the command line of R. The word “almost” refers to the fact that if you are using `sink()` to send the output to a file, you will have to enclose some commands in `print()` to get the same output as on the command line.

It is easy to create a new R script. We can open a new empty script by clicking selecting `New File` in any text editor. Alternatively, we can open an existing file if we want to modify or extend some R script we have already created sooner. Now we are ready to edit the script by one command per line. We remind that comments are edited with the # sign in the begin of the line.

[1]For details about the function `hist()` see the section 5.3

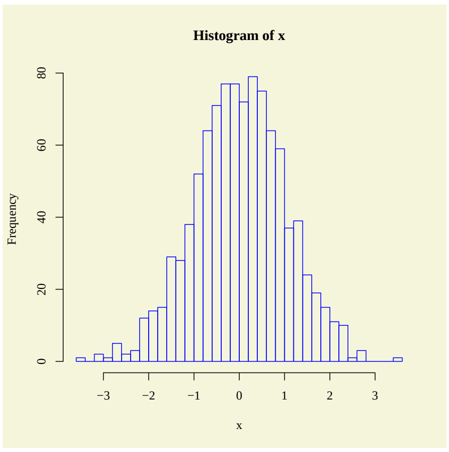


Figure 4.1: Histogram created using the user defined function `myhistogram` with ellipsis argument `break`

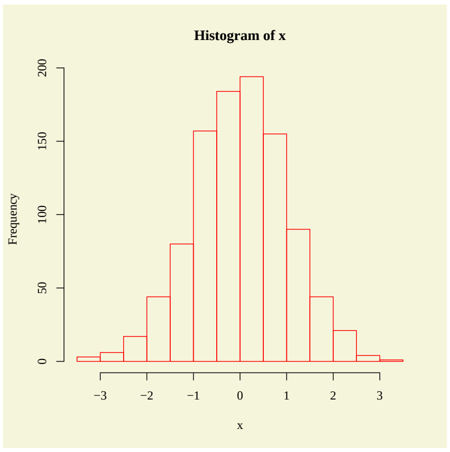


Figure 4.2: Output of the `myhistogram` with changed `border` argument and no call of ellipsis arguments.

We can save our script by selecting the `Save` or `Save as` function from the editor menu. The R script we save with the `.R` extension.

Running R scripts from the command line can be a powerful way to:

- automate our R scripts,
- integrate R into production,
- call R through other tools or systems.

There are basically two Linux commands that are used. The first is the command

```
 Rscript filename.R
```

and is preferred. The older command is

```
R CMD BATCH filename.R
```

We can call these directly from the command line or integrate them into a bash script. Wou can also call these from any job scheduler.

# Elementary graphics

## 5.1 Scatter plots

A Scatter Plot uses dots to represent values for two different numeric variables. The position of each dot on the horizontal and vertical axis indicates values for an individual data point. The most common use of the scatter plot is to display the relationship between two variables and observe the nature of such a relationship. The relationships observed can either be positive or negative, non-linear or linear, and/or, strong or weak.

The most common applications and uses of the scatter plots are:

1. Demonstration of the relationship between two variables.
2. Identification of correlation relationships.
3. Identification of data patterns.

### 5.1.1 Creating a scatter plots

The simplest way to create a scatter plot is to use the function `plot()` with two arguments `x` and `y` that contain the values we want to plot. These variables must be numeric vectors of the same length.

**Example 5.1.1** *Let us suppose, that the local ice cream shop keeps track of how much ice cream they sell versus the noon temperature on that day. Here are their figures for the last 10 days:*

| **Temperature** | *28* | *30.2* | *32* | *31* | *29.5* | *26* | *31.5* | *30* | *29* | *34* |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sales (€)** | *540* | *560* | *530* | *570* | *525* | *490* | *530* | *530* | *500* | *580* |

*Draw the scatter plot of the daily sales.*

**Solution**: At first we define two numeric vectors: `x` that contains the temperatures a,d `y` that will represent the daily sales. Requested scatter plot we get using the function `plot`. The whole procedure is recorded in the source code:

```
> x<-c(28,30.2,32,31,29.5,26,31.5,30,29,34)
> y<-c(540,560,530,570,525,490,530,530,500,580)
> plot(x,y)
```

The corresponding scatter plot we present in the figure 5.1. One can see this scatter plot is too simple and does not look very nice. It means some customizations are necessary. Before we explain the customization of the graph, we describe how we can save the resulting graph into a graphics file.

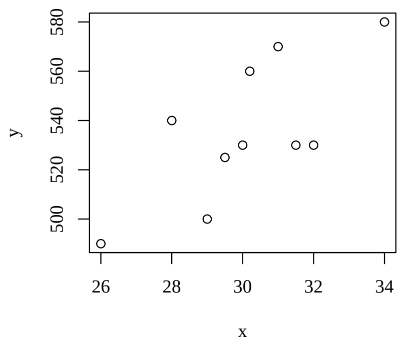


Figure 5.1: Scatter plot to the example 5.1.1 with default dot characters

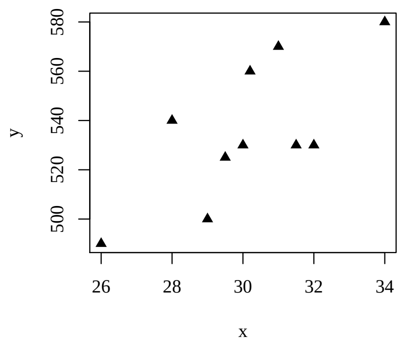


Figure 5.2: Scatter plot to the example 5.1.1 with modified dot characters

Table 5.1: List of the available formats to save the graphical outputs in the R environment.

| *Function* | *Format* |
|---|---|
| `pdf()` | Vector pdf format, best choice when used with `pdflatex`, easily resizable. |
| `svg()` | Vector `svg` format+, easily resizable. |
| `postscript()` | Vector postscript format `ps`, easily resizable. |
| `png()` | Bitmap format with high resolution, does not resize. |
| `jpeg()` | Compressed bitmap format, does not resize. |
| `bmp()` | High resolution bitmap format, does not resize. |
| `tiff()` | High resolution bitmap format, does not resize. |

In order to save the outcomes of the graphic functions, we have to at first decide about the output format that we want to use. The list of available saving formats is given in the table 5.1.

The only obligatory argument of the saving functions is the name of the file that we will use to save your graph. If necessary, this file name has tu be submitted with the full path to the directory, where we plan to save it. We may want to make adjustments to the size of the plot, resolution and others, before saving it. These adjustments are submitted as additional arguments of the `plot()` function, and the are listed in table 5.2. In order to enclose the file that actually works as an active graphical device, we use the function `dev.off()`. So, if we want to save the scatter plot in the `pdf` format, how presented on the figure 5.1, we adjust the source code in the following form:

```
> pdf("file.pdf",width=4,height=4,family="Times")
> plot(x,y)
> dev.off()
```

Table 5.2: List of the additional arguments of the graph saving function.

| *Argument* | *meaning* |
|---|---|
| `filename` | Name of the saved file, with full path if necessary. |
| `width` | Width of the resulting graph, default value 7 in. |
| `height` | Heigth of the resulting graph, default value 7 in. |
| `res` | resolution of the picture, applicable for bitmap formats, the default value 72 dpi. |
| `units` | Units of measure, can be `px` pixels, `in` inches, `cm` centimetres or `mm` millimetres. |
| `bg` | Background colour. |
| `fg` | Foreground colour. |
| `family` | The fonts used (default Helvetica). |

Table 5.3: List of the available dot characters in scatter plots.

| Dot | pch= | Bod | pch= | Dot | pch= | Bod | pch= | Dot | pch= |
|---|---|---|---|---|---|---|---|---|---|
| □ | 0 | ○ | 1 | △ | 2 | + | 3 | × | 4 |
| ◇ | 5 | ▽ | 6 | ⊠ | 7 | ✳ | 8 | ⟐ | 9 |
| ⊕ | 10 | ✡ | 11 | ⊞ | 12 | ⊗ | 13 | ⧅ | 14 |
| ■ | 15 | ● | 16 | ▲ | 17 | ◆ | 18 | ● | 19 |
| • | 20 | ○ | 21 | □ | 22 | ◇ | 23 | △ | 24 |

Working with graphs, we frequently make a plot to the screen at first. To save the graph in the e, we must re-enter the commands. R also provides the `dev.copy()` command, to copy the contents of the graph window to a file without having to re-enter the commands. To use this approach, we first produce our graph in the usual way. When we are satisfied with the way it looks, we call `dev.copy()`, passing it the driver we want to use and the file name to store it in. For example, to create a png file called `newplot.png` from a graph that is displayed by R, we type

```
> dev.copy(png,'newplot.png')
> dev.off()
```

### 5.1.2 Specifying the dot characters and lines

Function `plot()` with only two numeric vectors as its arguments plots simple scatter plot with default empty circles dots characters , how illustrated on figure 5.1. Besides these two numeric arguments, the function can contain more arguments that enable to modify the resulting graph. To manage the dot characters we use `pch` argument of the function `plot`. All accessible values of the `pch` variable are listed in the table 5.3.

For example, if we want to change the dot characters from the empty circles to the filled triangles, we simply use the additional argument `pch=17`, how shows the source code. The result we see on figure 5.2.

Table 5.4: List of the available types of graph entered as the `type` argument of the `plot` function.

| *Type* | *Meaning* |
|---|---|
| `p` | Point graph, the default value. |
| `l` | Continuous line. |
| `b` | Continuous line with the points. |
| `c` | Parts of the continuous lines, with the points omitted. |
| `o` | Parts of the continuous lines, with the points over-plotted. |
| `h` | Histogram-like graph. |
| `s` | stair steps graph. |

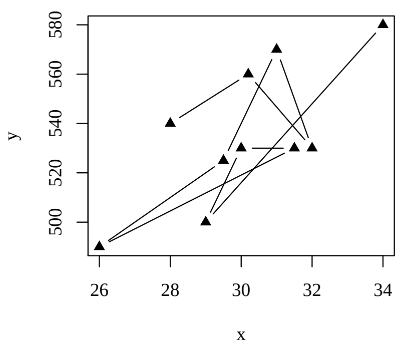


Figure 5.3: Scatter plot to the example 5.1.1 with dots and lines.

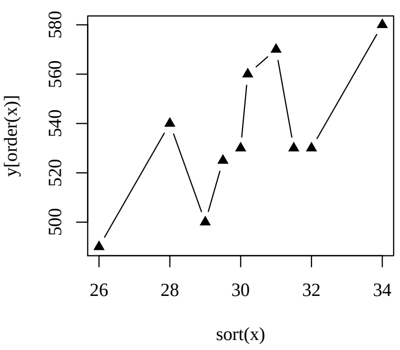


Figure 5.4: Scatter plot to the example 5.1.1 with with sorted values.

```
> plot(x,y,pch=17)
```

Similarly like the dot characters, we can modify the type of the scatter plot and as well its lines. The type of the graph is defined by the argument `type` of the function `plot()`. The available graph types are summarized in table 5.4.

We illustrate the scatter plots to the example 5.1.1 with graph type given as `type="b"` in figures 5.3 and 5.4. Let us note, that in figure 5.3 are the dots joined by line in the same order how they was entered into the numeric vector `x`, while in figure 5.4 the values are sorted increasingly according to the temperature. This does not correspond with increasing daily temperature. In order to graph the the daily sales in dependence on increasing temperature, we have to sort the data at first. We apply apply the functions `sort(x)` to sort the temperatures and `order(x)` to get the corresponding indices of the increasing `x` values. In the same order we must then sort the `y` vector. In the following listing we plot at first the scatter plot with lines in order given by entering an later we sort the values increasingly by temperature.

Table 5.5: List of the applicable line types of the `plot` function.

| *lty=* | *Line type* | *lty=* | *Line type* |
|---|---|---|---|
| 1 | Solid line (default). | 2 | Dashed line. |
| 3 | Dotted line. | 4 | Dot-dashed line. |
| 5 | Long dashed line. | 6 | Long and short double dashed line. |

```
> plot(x,y,pch=17,type="b")
> dev.off()
> plot(sort(x),y[order(x)],pch=17,type="b")
> dev.off()
```

The R language enables modification of the line type and as well the line width. Both are defined by the additional arguments of the `plot()` function. The line width is entered simply as number assigned tu the `lwd` parameter. The line thickness is specified as a multiplying factor, so line-width `lwd = n` gives a line width of `n * defaultwidth`. The default width depends on the device used for plotting. The general rules are:

- a point is 1/72 of an inch,
- a pixel is standard 1/96 of an inch, or 0.75 points.

This can depend however on the settings of our device. The `pdf()` and `postscript()` devices work with standard point as 1/72 inch, so `lwd=1` refers to a line-width of 1/96 inch or 0.75 points. But when using the bitmap devices we have to take in account also the resolution `res` argument. If we set `lwd=1`, the line width is equal to 1/96 inch, but setting `res=96` gives a line with a thickness of 1pt. The type of the line we set by argument `lty`. The applicable options are listed in table 5.5.

### 5.1.3 Colouring the graph

By default, the points and lines in the plot are black. But we can change this colour specifying the option `col` of the function `plot()` the by names (e.g `col=red`), fur-

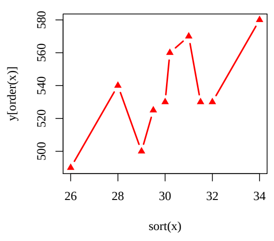


Figure 5.5: Scatter plot to the example 5.1.1 with plot colour set to red colour.

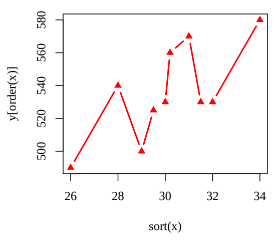


Figure 5.6: Scatter plot to the example 5.1.1 with plot colour set as number 562

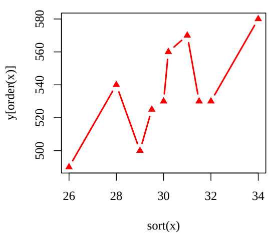


Figure 5.7: Scatter plot to the example 5.1.1 with colour set as hexadecimal number #FF0000

Table 5.6: List of the graph colouring options of the `plot` function.

| *Option* | *Meaning* | *Option* | *Meaning* |
|---|---|---|---|
| `col.axis` | Axis annotation colour. | `col.lab` | Axes labels colour. |
| `col.main` | Main title colour. | `col.sub` | Subtitle colour. |
| `bg` | Character filling color. | `fg` | Foreground colour. |

ther by the number of the colour (e.g. `col=636`) or with RGB hexadecimal code (e.g `col = "#FFCC00"`). The list of colour names we get as an answer of the function `colors()`.

Figure 5.7 illustrates three possible colour settings for the graph. Besiddes the plotted curve, one can colour other ellements of the plot. The options for the colouring are summarized in the table 5.6. Let us note, that the option `bg` for the background colour changes the colour of the filling in the dot characters (potion `pch` set from 21 to 25) and not the backround color of the whole plotting area. In order to change the backround of the whole graph we have to apply the `par()` function.

Figure 5.8 illustrates graph to our icecream example 5.1.1 with some colouring, how we can see in the following source code.

```
>plot (sort(x),y[order(x)],lty =1, type ="b",col="aquamarine",lwd =2,
  col.axis ="violet",col.main ="green",main ="Main␣title",fg="red",
  col.lab="coral3",pch=17)
> dev.off()
```

As we mentionen above, to colour the background of the whole plot, we have to use the `par()` function., how illustrates the next listing:

```
>par(bg="beige")
>plot (sort(x),y[order(x)],lty =1, type ="b",col=30,lwd =2,
 col.axis ="darkmagenta", col.main ="blue3",col.sub="blue2",
 main="Main␣title",sub="Subtitle",fg="red",col.lab="coral4",pch=17)
> dev.off()
```

The result is presented on the figure 5.9.

An important aspect of R's use of the `col` argument is the notion of vector recycling. R expects the col argument to have the same length as the number of things its plotting (in this case the number of points). Of course, these colours are not substantively meaningful. Our data are not organized in an alternating fashion. An example of two alternating dot colours we can see on figure 5.10. Setting the colours is achieved by the source code:

```
>par(bg="beige")
> plot(sort(x),y[order(x)],pch=17,type="b",col=c("red","blue"))
> dev.off()
```

We can also produce "rainbows" of colour. For example, we could use the `rainbow()` function to get a rainbow of five different colors and use it on our plot. Let us try the source code like this:

```
>par(bg="grey")
plot(sort(x),y[order(x)],pch=17,type="b",col=rainbow(5))
> dev.off()
```

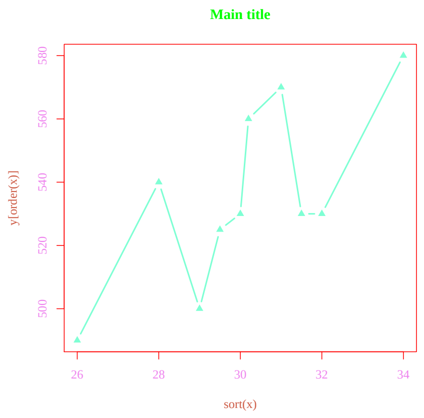


Figure 5.8: Scatter plot to the example 5.1.1 with more colour options.

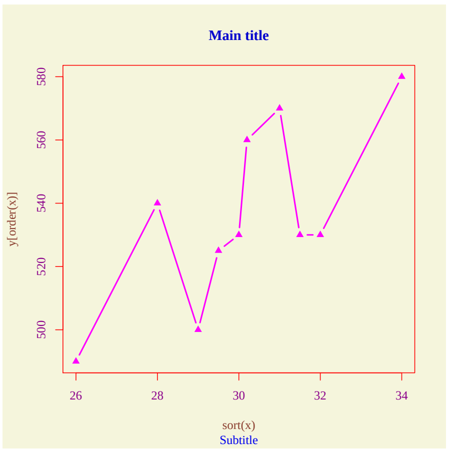


Figure 5.9: Scatter plot to the example 5.1.1 with plot colour options and backckgound set by `par()` function.

We get the result that is illustrated in figure 5.11. The `rainbow()` takes additional arguments, such as `start` and `end` that specify where on the rainbow (as measured from 0 to 1) the colours should come from. So, specifying low values for `start` and `end` will make a red/yellow plot, middling values will produce a green/blue plot, and high values will produce a blue/purple plot.

### 5.1.4 Titles and subtitles

Base R plotting functions come with an argument named `main` that allows adding a title to the plot. One can also add a subtitle, that will be positioned under the plot making use of the `sub` argument. Using these arguments we can see on graphs in figures 5.8 and 5.9.

Alternative way, how to add the title and subtitle to the graph is using the function `title()`. The difference between using this function instead of the arguments is that the arguments passed to the `title()` function only affect the texts we are adding. In addition, we can customize every single texts using the title function several times. Managing the title and subtitle is illustrated in figure 5.12. How we can see from the source code, after plotting the graph, we add the main title and later the subtitle.

```
> par(bg="beige")
> plot(sort(x),y[order(x)],pch=17,type="b",col=rainbow(4))
> title(main="Icecream␣sales",col.main="red")
> title(sub="Temperature",col.sub="blue",adj=1,line=2)
> dev.off()
```

### 5.1.5 Adding text into the plot

We can add some texts into the plotted graph using the functions `text()` and `mtext()`. The difference is, that function `text()` places the given text on any position in the plotting area while the `mtext()` function places the text into the margins.

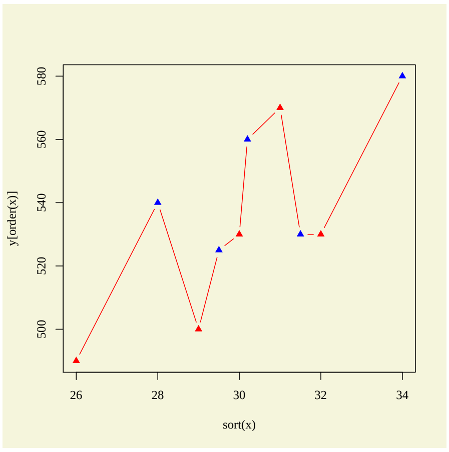


Figure 5.10: Scatter plot to the example 5.1.1 with `col` option set as a vector.

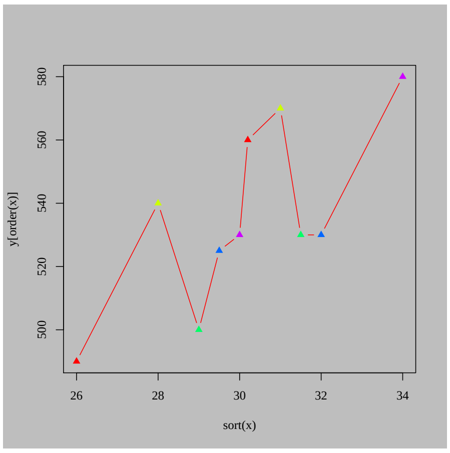


Figure 5.11: Scatter plot to the example 5.1.1 with plot option `col` set as `rainbow()` function.

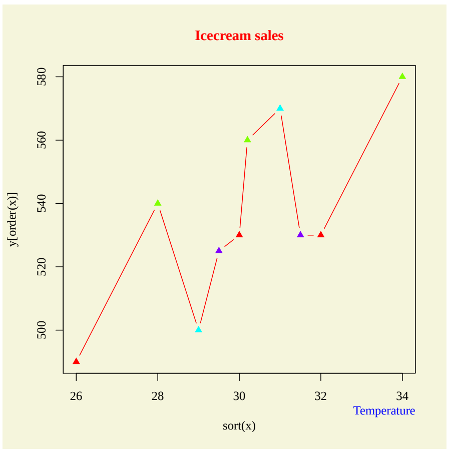


Figure 5.12: Scatter plot to the example 5.1.1 with main title and subtitle set by `title()` function.

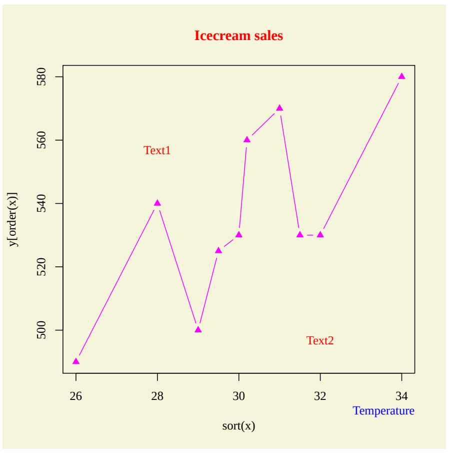


Figure 5.13: Scatter plot to the example 5.1.1 with text labels placed in graph by `text()` function.

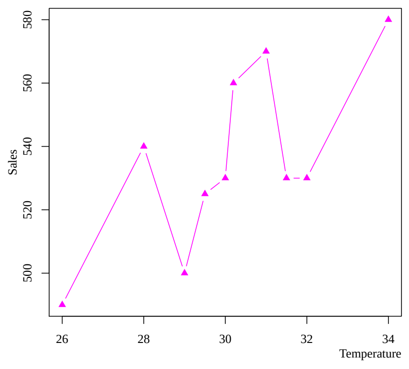


Figure 5.14: Scatter plot to the example 5.1.1 with axes labels set by `mtext()` function.

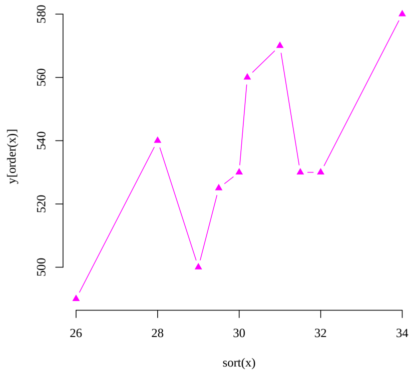


Figure 5.15: Scatter plot to the example 5.1.1 with customized axes `x` and `y`.

The function `text()` has two additional arguments:

- `location` defines the `x` and `y` coordinates, where the text will be placed. The coordinates must be submitted as the first two arguments of the function.
- `pos` defines the position according to the actual place, 1=bottom, 2=left, 3=top and 4=right. Defining position as `locator(1)` enables placing the text using the mouse.

Of course, we can also use the common arguments for plots like `col`, `font` etc. The location for the texts can be submitted as a vector to place the text on more than one position. The length off the `x` and `y` vector should be the same, otherwise the shorter vector is recycled. Placing the labels in the graph demonstrates the source code:

```
> par(bg="beige")
> plot(sort(x),y[order(x)],pch=17,type="b",col=30)
> title(main="Icecream␣sales",col.main="red")
> title(sub="Temperature",col.sub="blue",adj=1,line=2)
> text(c(28,32),c(560,500),c("Text1","Text2"),pos=1,col="red")
```

The resulting plot we see on the figure 5.13.

Similarly, we can add the labels into the margins of the graph using the function `mtext()`. As well this function has two additional arguments:

- `side` defines the the side of the plot area, where we put the text label, 1=bottom, 2=left, 3=top and 4=right.
- `line` defines the line number, where the label will be placed. Let us note the lines are numbered starting from 0.

Figure 5.14 illustrates the scatter plot with axis descriptions set by `mtext()` function in the margins of the plotting area. Here is the corresponding source code:

```
> par(bg="beige")
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,xlab="",ylab="")
> mtext("Temperature",side=1,line=2,adj=1)
> mtext("Sales",side=2,line=2)
```

### 5.1.6 Axes customization

The default axis labels depend on the function we are using, e.g. `plot()` function will use the names of the input data. The default axes are pictured as the plot box with ticks marks on the down and right sides. In this subsection we will illustrate the possibilities how to customize the axes appearance.

In order to remove the plot box we set the option `axes=FALSE` inside the plotting function. New axes we can add using the `axes()` function. Argument of the `axis()` function defines the side of the plot, where the axis will be added. As usually, the numbers define the sides 1=bottom, 2=left, 3=top and 4=right. An example if the graph with user defined axes we see in figure 5.15. And the source code that produce it:

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1)
> axis(2)
```

If necessary, we can add the box again using the function `box()`. It differs from the default plot box by tick marks, that are visible as well on the top and right side of the box.

Another customization is changing the axes colours. We can do it by setting the aptional arguments of the `axis()` function:

- `col` defines the axes line colour,
- `col.ticks` defines the ticks colour,
- `col.axis` defines the labels colour.

As an example of the axes colours customization we can show the source code:

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1,col="blue",col.ticks="red",col.axis=555)
> axis(2,col="deepskyblue2",col.ticks=444,col.axis="red")
```

The result we see in figure 5.16. As well the tick marks can be customized in different ways. We are able to:

- set the number of the tick marks with specified the start end end values,
- modify the length and orientation of the tick marks,
- rotate the tick marks labels,
- custom the tick mark labels,
- remove tick marks,
- add the minor ticks using the `Hmisc` .

Optional arguments `xaxp` and `yaxp` allow customizing the positions of the tick marks on the x-axis and y-axis respectively. Their values we set as vectors `c(start,end, regions)`, where the values `start` and `end` define the start and end value on each axis,

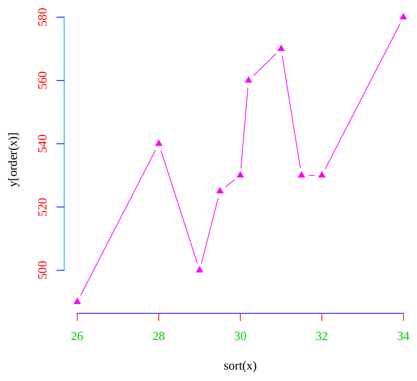


Figure 5.16: Scatter plot to the example 5.1.1 with customized axes colours.

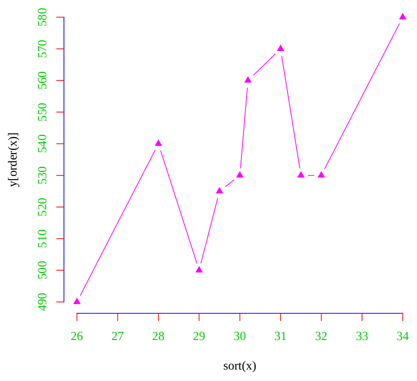


Figure 5.17: Scatter plot to the example 5.1.1 with customized axes `x` and `y`.

and the value of `regions` defines the number of regions to divide the axis. Running the source code:

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1,col="blue",col.ticks="red",col.axis=555,xaxp=c(0,8,8))
> axis(2,col="blue",col.ticks="red",col.axis=555,yaxp=c(490,580,9))
```

we get the plot, which is presented in firgure 5.17.

The argument `tck` allows to modify the length and orientation of the tick marks. Its positive value sets the marks inside the plotting area while the negative values define the marks outside from the plotting area. The greater the absolute value, the longer the ticks. the default value is `tck=-0.05`.

It is possible to rotate the tick mark labels in several ways. The rotation is enabled using the`las` argument that can take one of four values:

- `las=0` the labels are parallel to axis (default),
- `las=1` all labels are horizontal,
- `las=2` the labels are perpendicular to axis,
- `las=3` all labels are vertical.

We can demonstrate the modification of the tick marks by the following code:

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1,col="blue",xaxp=c(26,34,8),tck=0.02,las=3)
> axis(2,col="blue",yaxp=c(490,580,9),tck=0.02,las=2)
```

It produces the plot presented in figure 5.18. Let us note, we can remove the tick mark by setting the arguments `xaxt="n"` for the `x`-axis or `yaxt="n"` for the `y`-axis.

The labels of the tick marks can be changed using the argument `labels` of the `axis()` function. In order to place the labels correctly, we have to set their positions by `at` argument, how illustrates the source code.

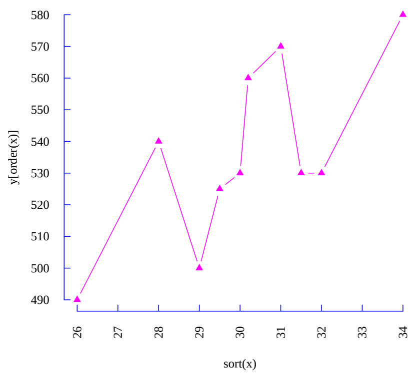


Figure 5.18: Scatter plot to the example 5.1.1 with customized tick marks and tick labels.

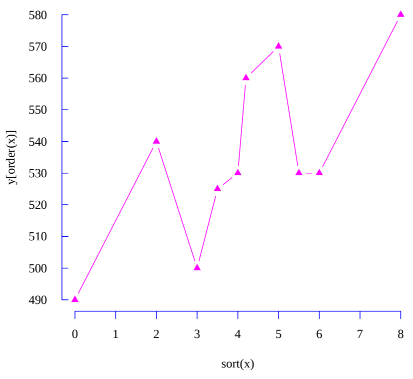


Figure 5.19: Scatter plot to the example 5.1.1 with user defined tick marks labels.

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30,axes=FALSE)
> axis(1,col="blue",at=seq(round(min(x)),round(max(x)),by=1),
 labels=0:8)
> axis(2,col="blue",yaxp=c(490,580,9),tck=0.02,las=2)
```

Graph with labels set by user we see in the figure 5.19. We can also set texts instead of numbers to the tick marks. We attain this effect defining the argument `labels` as the vector of the text strings but its length must correspond to the number of the ticks.

It is possible to add minor ticks to the axes with the minor.tick function of the Hmisc library. The function will allow you to specify the tick density, the size and addition arguments to each axis. It is illustrated in the figure 5.20 that we get using the source code:

```
> plot(sort(x),y[order(x)],pch=17,type="b",col=30)
> minor.tick(nx = 2, ny = 2,tick.ratio = 0.5)
```

Further possible customizations of the axes are axis limits and scaling. The limits for the axis we can define using the optional arguments `xlim` and `ylim` for x-axis and y-axis respectively. The limits are to be submitted as vectors in the form `c(start,end)`. We can also transform the axes into the logarithmic scale by setting the argument `log` to be equal to axis that we plan to scale. So `log="x"` sets the logarithmic scale to the x-axis, `log="y"` sets the logaritmic scale to the y-axis and `log="xy"` transforms both axis into the logarithmic scale.

Finally let us illustrate the possibility to set two dual axes. For example let us suppose, we want to plot in the same graph two characteristics of the patients' health condition, the temperature and the blood pressure. We have data of 100 patients stored in variables `y` and `z`, while the variable `x` contains the sequence of the patient identifiers, numbers from 1 to 100. At first we modify the margins of the plotting area using `par(mar=c(3,4,2,4))`. Then we plot the scatter plot of temperatures. Important step is setting the new plot by `par(new=TRUE)`. Now we are ready to plot the second dataset

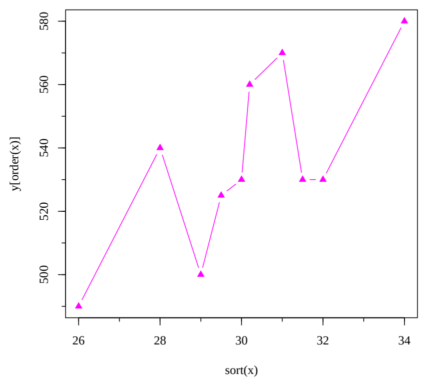


Figure 5.20: Scatter plot to the example 5.1.1 with customized tick marks and tick labels.

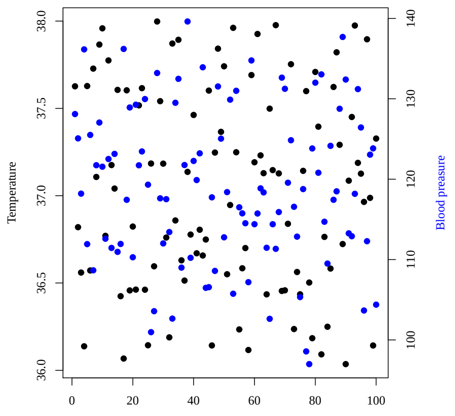


Figure 5.21: Scatter plot with dual axes for the temperature (left) and blood pressure (right).

in blue colour, with no boxes and no axes. The dual y-axis is drawn using the `axis(4)` function on the right-hand side of the graph. Its description we add using the `mtext()`. The result we see on the figure 5.21. And here is the complete source code for plotting the graph with dual axes.

```
par(mar = c(3, 4, 2, 4))
plot(x, y, pch = 19, ylab = "Temperature")
par(new=TRUE)
plot(x, z, col = 4, pch = 19,
     axes = FALSE, # No axes
     bty = "n",    # No box
     xlab = "", ylab = "")
axis(4)
mtext("Blood␣preasure", side = 4, line = 3, col = 4)
```

### 5.1.7 Plotting the curves

One of the many handy functions in R is `curve()`. It is a neat little function that provides mathematical plotting, e.g., to plot functions. The `curve()` function takes, as its first argument, an R expression. That expression should be a mathematical function in terms of x. For example, if we wanted to plot the parabola $y = x^2$, we would simply type:

```
curve(x^2)
```

We get the graph of the continuous function how depicted in figure 5.22.

One can also specify an optional add parameter to indicate whether to draw the curve on a new plotting device or add to a previous plot. For example, if we wanted to overlay the function $y = \sqrt{x}$ on top of $y = x^2$ we could type:

```
curve(x^2)
curve(sqrt(x),add=TRUE)
```

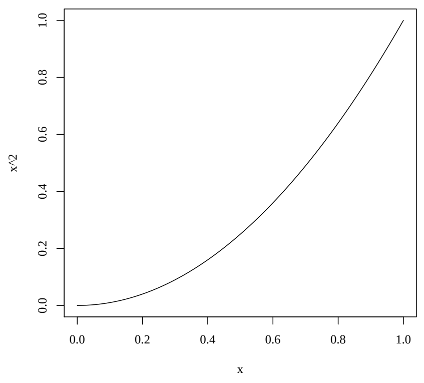


Figure 5.22: Continuous curve $y = x^2$ obtained using the curve function.

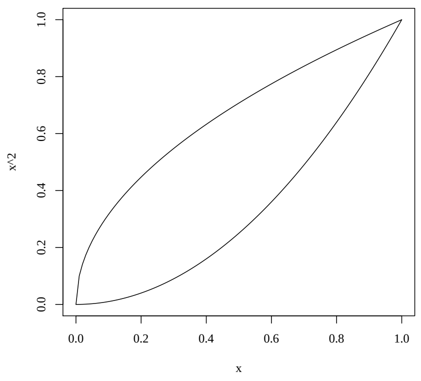


Figure 5.23: Two curves plotted in the same plotting area setting the parameter add=TRUE.

So we get the graph that contains two curves, as illustrated in figure 5.23.

Using the curve() function is not restricted to use it itself either. One can plot some data and then use curve() to draw any line on top of it. We illustrate it in figure 5.24 that we obtain using the following source code:

```
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
plot(y ~ x)
curve(x^2,add=TRUE)
```

Similarly like the plot() function, curve() accepts optional graphical parameters. So we can redraw the graph in the figure 5.24, with some colouring and points shape modification. If we submit the source code:

```
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
plot(y ~ x,col="blue",pch=17)
curve(x^2,add=TRUE,col="red")
```

we get the result that we see in figure 5.25.

We could also call these in the opposite order, but such situation we have to replace the plot() function by points(). so, the corresponding source looks like this:

```
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
curve(x^2,add=TRUE,col="red")
points(y ~ x,col="blue",pch=17)
```

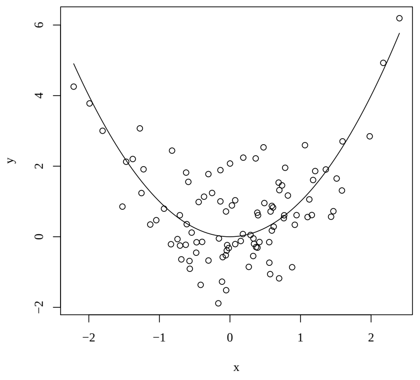


Figure 5.24: Combination the `plot` and `curve` functions in the same graph.

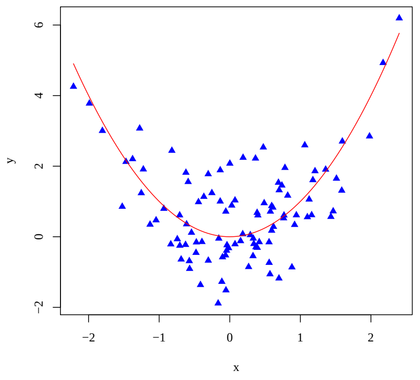


Figure 5.25: The `plot` and `curve` functions modified by optional arguments.

### 5.1.8 Adding a legend

The function `legend()` enables adding a legend to the plots in R. This function has more arguments, that allow managing the position and outlook of the legend. Let us note some of them:

- `x,y` position in the plotting area defined by coordinates in the graph,
- `legend` vector of strings for description in the legend,
- `col` vector of colours used in the graph,,
- `pch` vector of the mark shapes used in the graph,
- `lty` vector of the line types used in the graph,
- `ncol` number of columns used in the legend, default value is one column.

We will use the following user defined function that plots the graphs of the goniometric functions $\sin x$ and $\cos x$ in one graph. In the resulting plot we will illustrate adding the legend to the graph and working with the arguments of the `legend()` function. The user function `gonplot()` is given by the source code:

```
gonplot <- function() {
    curve(sin(x),xlim=c(-10,10),col="red",lwd=2,type="l",
    ylab="sin␣x",xlab="",ylim=c(-1,2))
    curve(cos(x),xlim=c(-10,10),col="blue",lwd=2,type="l",lty=2,
    ylab="sin␣x",xlab="",add=TRUE)
}
```

As the first option we introduce the position of the legend. The position argument `x` can be set to one of the values `top`, `topleft`, `topright`, `bottom`, `bottomleft`, `bottomright`, `left`, `right` or `center`. This scenario does not require to set the argument `y` as the legend position is clearly defined by word.

The text of the legend is then set with `legend` argument and the line type, width and colour with `lty`, `lwd` and `col` arguments, respectively. using the source

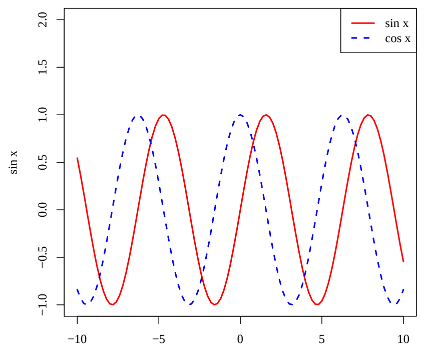


Figure 5.26: Placing the legend in top right corner of the graph.

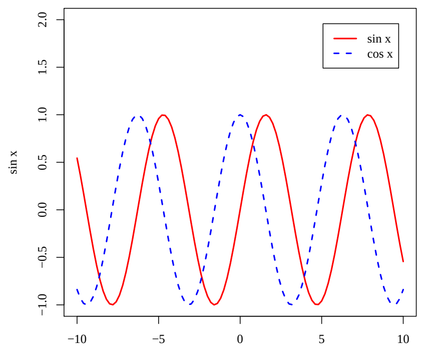


Figure 5.27: Placing the legend in top right corner of the graph with `inset=0.05`.

```
gonplot()
legend(x = "topright",                    # Position
       legend = c("sin␣x", "cos␣x"),  # Legend texts
       lty = c(1, 2),                     # Line types
       col = c("red", "blue"),            # Line colors
       lwd = 2)
```

we obtain the plot in figure 5.26. Moreover, setting the argument `inset`,we can state the distance from the margin as a fraction of the plot region. In figure 5.27 we see th eplot with legend, where we set argument `inset=0.05`.

### 5.1.9 Multiple graphs

Sometimes we need to put two or more graphs in a single plot. We can achieve it by setting some graphical parameters using the `par()` function. The number of required subplots we specify by setting the `mfrow` parameter. This parameter has a vector of form `c(row, col)` which divides the given plot into array of subplots that contains `row` rows and `col` columns. For example, if we need to plot two graphs side by side, we would have `row=1` and `col=2`.

For example we can place the plots from figures 5.24 and 5.25 in one plotting area divided in two columns. We apply the code:

```
 par(mfrow=c(1,2))
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
plot(y ~ x)
curve(x^2,add=TRUE)
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
plot(y ~ x,col="blue",pch=17)
curve(x^2,add=TRUE,col="red")
```

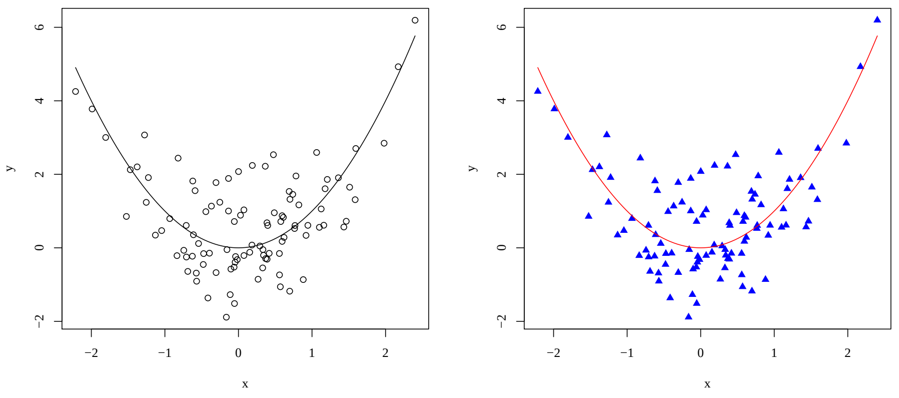


Figure 5.28: Two plots in one plotting area with two columns.

The result we see in figure 5.28.

## 5.2 Bar graphs

Bar graphs is a plot that summarizes categorical data. In its simplest form it summarizes one categorical variable by the numbers of observations in each category. Bar graph presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally.

To produce the bar graphs, R uses the function `barplot()`. The basic syntax to create a bar-chart in R is:

```
barplot(H,xlab,ylab,title, names.arg,col)
```

The parameters used in the function are as follows:

- `H` is a vector or matrix containing numeric values used in bar chart,
- `xlab`is the label for x axis,
- `ylab`is the label for y axis,
- `title` is the title of the bar char,
- `names.arg` is a vector of names appearing under each bar,
- `col` is used to give colors to the bars in the graph.

For example, let us suppose the vector `x` contains the daily sales of some products. The sales volumes can be graphically presented in the form of the bar chart. The simplest form of the bar plot we see in figure 5.29. The corresponding source code is very simple:

```
x<-c(2000,2400,1400,2600)
barplot(x)
```

For a horizontal bar chart, which is illustrated in figure 5.30 we set the argument `horiz` to `T`. The source code then looks like this:

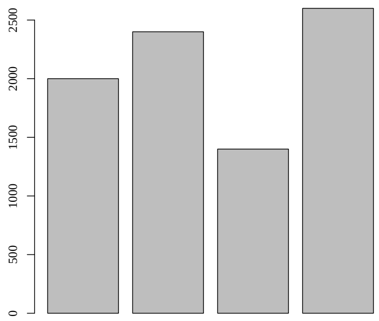


Figure 5.29: The bar chart of the daily sales.

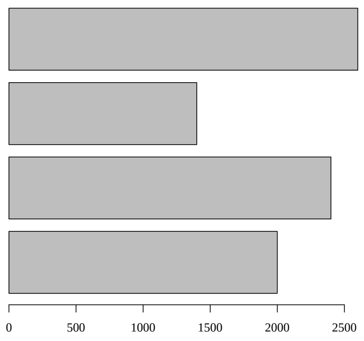


Figure 5.30: Horizontal bar chart of daily sales.

```r
x<-c(2000,2400,1400,2600)
barplot(x,horiz=T)
```

Similarly like in the case of the `plot()` function, we can further modify the outlook of the resulting bar chart. In the source code below, we at first define the names of the fruits sold and we enter them in the `goods` vector. We use these names as the `names.arg` parameter of the bar plot to assign these names to the columns. Further we define the values of the parameters `xlab` and `ylab` for the axes names, `col` and `border` for colouring the bars and `main` to define the title of the graph.

```r
goods<-c("orange","banana","apple","plum")
barplot(x,names.arg=goods,xlab="Fruit",ylab="Sales",col="cyan",
 main="Monthly␣sale",border="black")
```

When we execute above code, it produces the graph shown in figure 5.31. We can further modify the bars to have different colours for single fruits. To get the result shown in figure 5.32, we define the vector `colors` that we use as value of the argument `col`.

```r
colours<-c("orange","yellow","red","blue")
barplot(x,names.arg=goods,xlab="Fruit",ylab="Sales",col=colours,
 main="Monthly␣sale",border="black")
```

In addition, we can create bar chart with groups of bars and stacks in each bar by using a matrix as input values. We illustrate it by modifying the previous source code. We extend the sales period for more months, and present graphically the sales volumes. We can see the graph in figure 5.33. The source code that produces this graph follows below. let us note adding the legend using the `legend()` function, similarly like in the scatter plots.

```r
months<-c("Jan","Feb","Mar","Apr")
x<-matrix(c(2000,2400,1400,2600,1800,2200,1600,2400,2100,2300,1500,
 2400,2400,1800,1200,2200),nrow=4,ncol=4)
barplot(x, main = "Sale␣volumes", names.arg = months, xlab = "Month",
```

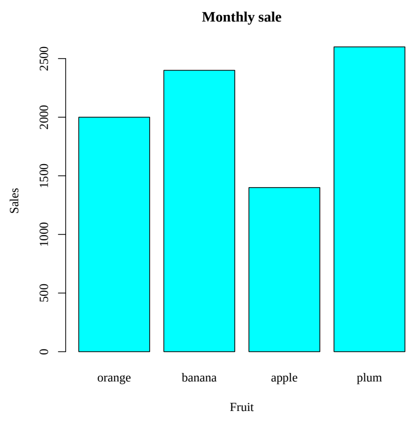


Figure 5.31: The bar chart of the daily sales with modified parameters.

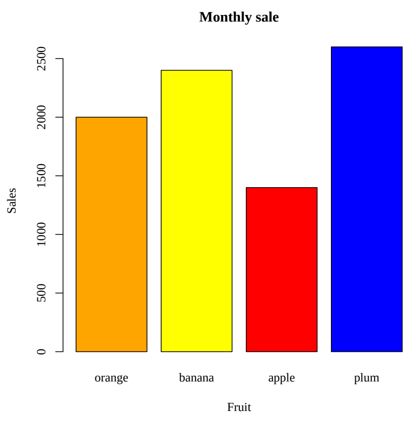


Figure 5.32: Bar chart of daily sales with different bar colours.

```
  ylab = "Sales", col = colours, ylim=c(0,11000))
legend("topright", goods, fill = colours,ncol=2)
```

The same information we can present alternatively by the grouped bar chart as shown in figure 5.34. In the source code we add the argument `beside=TRUE` in the `barplot()` function. So the corresponding line in the source is

```
barplot(x,beside = TRUE,main="Sale␣volumes", col = colours,
names.arg=months, xlab = "Month", ylab = "Sales",ylim=c(0,3000))
```

Other lines remain unchanged. Pay attention to defining the `ylim` arguments in both graphs. Extending the limits for the vertical axis assures sufficient space for adding the legend in the top right corner.

Besides filling the bars by colour, we can fill them by textures, how it is shown in figure 5.35. We can change the density of the lines setting the `density` argument whose value is vector with the length equal to the number of bars. Similarly, setting the argument `angle` as a vector of the length that equals to the number of bars we can state the angle of the filling lines. In order to create the graph in figure 5.35, we return to the vector value `x`, so the source code takes the form:

```
x<-c(2000,2400,1400,2600)
barplot(x,density=c(5,10,20,30), angle=c(0,30,60,90),col="blue",
names.arg=goods,main="Sale␣volumes",xlab="Fruit",ylab="Sales")
```

If we want to use the for example crossed lines textures it is necessary to plot the bar chart twice, with different angles. As well adding the legend has to be made in two steps, with different angles set in user defined variables `angle1` and `angle2`. We see it in the source code below. We returned in this illustration to the grouped bar chart, therefore we entered the variable `x` as matrix. Executing this code, we obtain the graph shown in figure 5.36. We illustrate here as well changing the colours of the textures.

```r
angle1<-c(0,30,60,90)
angle2<-c(90,120,150,0)
colours<-c("orange","yellow","red","blue")
months<-c("Jan","Feb","Mar","Apr")
x<-matrix(c(2000,2400,1400,2600,1800,2200,1600,2400,2100,2300,1500,
 2400,2400,1800,1200,2200),nrow=4,ncol=4)
barplot(x,density=c(10,15,20,25), angle=angle1,beside = TRUE,
 main="Sale␣volumes", col = colours,names.arg=months, xlab = "Month",
 ylab = "Sales",ylim=c(0,3000))
barplot(x,density=c(10,15,20,25),angle=angle2,beside=TRUE,
 col = colours,add=TRUE)
legend("topright", goods, ncol=2, fill=colours, angle=angle1,
 density=c(10,15,20,25))
legend("topright", goods, ncol=2, fill=colours, angle=angle2,
 density=c(10,15,20,25))
```

Further extension of the filling possibilities brings the package `patternplot`. It can fill pie charts, ring charts, bar charts and box plots with colours or textures or any external images in png or `jpeg` formats.

Sometimes we have to plot the count of each item as bar plots from categorical data. For example, let us suppose our data, including some players shooting statistics are kept in the structure of data frame `players`. We create the data set by commands:

```r
playerID<-c(1,2,3,4,5,6,7,8,9,10)
position<-c("forward","guard","forward","center","guard","center",
 "forward","guard","forward","forward")
attempted<-c(12,6,10,15,12,8,10,9,14,12)
made<-c(7,4,6,12,8,3,7,4,9,8)
players<-data.frame(playerID,position,attempted,made)
```

Simply doing `barplot(players$position)` will not give us the required plot. It will answer by error announcement as it expects the numerical vector or matrix on the

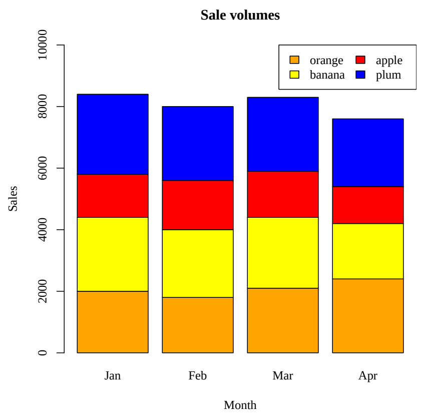


Figure 5.33: The stacked bar chart of the monthly sales.

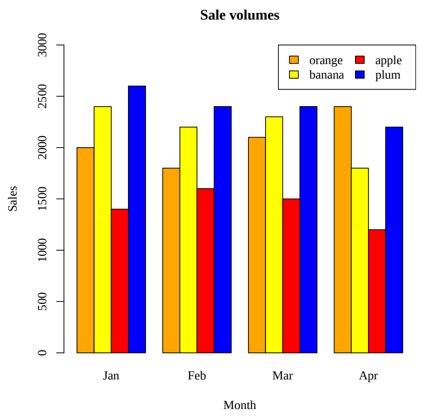


Figure 5.34: Grouped bar chart of daily sales.

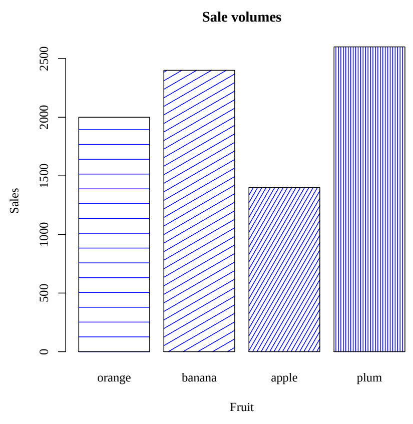


Figure 5.35: The bar chart of the monthly sales with different textures.

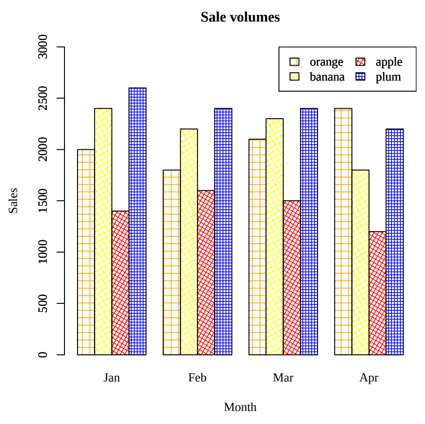


Figure 5.36: The bar chart of the monthly sales with combined textures.

entry. But we want to know the number of players on each position. This count can be quickly found using the `table()` function, as shown below.

```
barplot(table(players$position),
 main="Players␣positions",
 xlab="Position",
 ylab="Count",
 border="red",
 col="blue",
 density=10)
```

Now plotting this data will give our required bar plot. The result is shown on figure 5.37.

As well when working with data frames, we can make multiple comparisons in a bar plot. In the example below, we have a matrix of 3 vectors, each representing a set of 5 data points. We compare the 3 sets using bar plots. The three vectors represent three basketball players. In each vector, the 5 numbers represent their scoring made in 5 games of the tournament. See the source code here and the figure 5.38.

```
col1 <- c(8,10,6,12,15)
col2 <- c(10,5,9,9,12)
col3 <- c(6,10,8,11,13)
data <- data.frame(col1,col2,col3)
names(data) <- c("player-1","player-2","player-3")
barplot(as.matrix(data), main="Tournament-1", ylab="Points␣gained",
 beside=TRUE,col=rainbow(5))
legend("topleft",c("game1","game2","game3","game4","game5"),cex=1.0,
 bty="n",fill=rainbow(5))
```

## 5.3 Histograms

A histogram is an approximate representation of the distribution of numerical data. Histograms give a rough sense of the density of the underlying distribution of the data, and

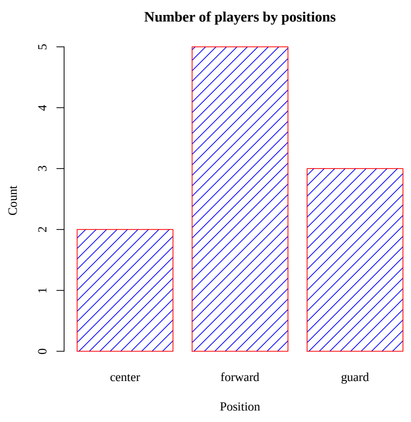


Figure 5.37: The bar chart of the counts of players on single positions.

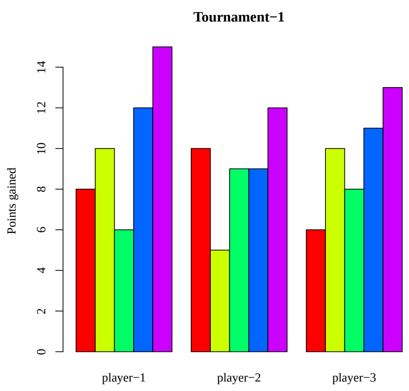


Figure 5.38: The bar of the three players scorings in five games.

often for density estimation: estimating the probability density function of the underlying variable. They represent the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

Histogram can be created using the `hist()` function in R programming language. This function takes in a vector of values for which the histogram is plotted. The basic syntax for creating a histogram using R is:

```
hist(data,main,xlab,xlim,ylim,breaks,col,border)
```

The meaning if the arguments is as follows:

- `data` is a vector containing numeric values used in histogram,
- `main` indicates title of the chart,
- `col` is used to set colour of the bars,
- `border` is used to set border colour of each bar,
- `xlab` is used to give description of x-axis,
- `xlim` is used to specify the range of values on the x-axis,
- `ylim` is used to specify the range of values on the y-axis,
- `breaks` is used to mention the width of each bar.

We can illustrate plotting the histograms on the case of rolling the dice. Let us suppose, we will roll two dice for 10 000 times and we are interested in the sum of the points thrown. We simulate it using the source code that follows:

```
dice1<-sample(1:6,replace=T,10000)
dice2<-sample(1:6,replace=T,10000)
c<-dice1+dice2
```

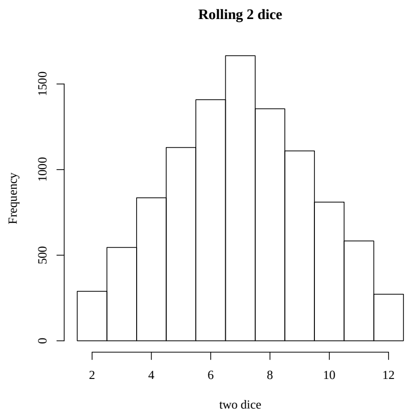


Figure 5.39: The histogram of the points gained when rolling 2 dice.

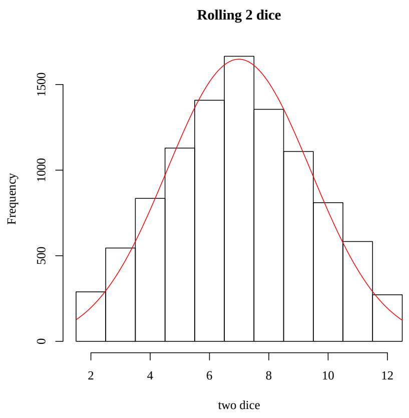


Figure 5.40: The histogram of the scores of rolling 2 dice with the density of normal distribution.

Vector c then contains the scores obtained on both dice. Now we can plot the histogram of the scores using the `hist()` function:

```
hist(c,breaks=1.5:12.5, main="Rolling␣2␣dice",xlab="two␣dice",
 ylab="Frequency")
```

Let us note, that the resulting scores are integers from 2 to 12, we set the breaks of the histogram as `breaks=1.5:12:5`. So we have covered all possible values by bars of the histogram. The result is show on figure 5.39.

The Central limit theorem known from the probability theory establishes that, in many situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution (informally a bell curve) even if the original variables themselves are not normally distributed. This can be documented in the histogram by plotting the density curve of the normal distribution in the same pot as the histogram. In order to do so, we extend the source code into the form:

```
hist(c,breaks=1.5:12.5, main="Rolling␣2␣dice",xlab="two␣dice",
 ylab="Frequency")
curve(dnorm(x,mean(c),sd(c))*10000,col="red",add=T)
```

In figure 5.40 we see the result, where the bell shaped graph of the normal density is plotted by the red colour. Let us note the multiplying the values of the `dnorm()` by 10 000, what corresponds to the number of the rolls repeating.

## 5.4 Pie graphs

A pie chart is a plot for a single categorical variable and it is an alternative to bar chart. A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion. In a pie chart, the arc length of each slice (and consequently its central angle and area), is proportional to the quantity it represents. Pie charts are not recommended in the R documentation, and their features are somewhat limited. The authors recommend bar or dot plots over pie charts because people are able to judge length more accurately than volume.

In R the pie chart is created using the `pie()` function which takes positive numbers as a vector input. The additional parameters are used to control labels, colours, title etc. The basic syntax for creating a pie-chart using the R is:

```
pie(data, labels, radius, main, col, clockwise)
```

The meaning if the arguments is as follows:

- `data` is a vector containing the numeric values used in the pie chart,
- `labels` is used to give description to the slices,
- `radius` indicates the radius of the circle of the pie chart, (value between $-1$ and $+1$),
- `main` indicates the title of the chart,
- `col` indicates the colour palette,
- `clockwise` is a logical value indicating if the slices are drawn clockwise or anti clockwise.

Let us suppose, we want to represent the shares of monthly expenses of the household by pie chart. We take in account the following expenses categories: housing, foods, clothing, entertainment and other. The expenses in the single categories and their descriptions we submit as vectors. These values we use as parameters of the pie chart, how illustrates the following source code.

```
data<-c(200,300,100,80,150)
labels<-c("housing","food","clothing","entertainment","other")
pie(data,labels,main="Monthly expenses")
```

The resulting pie chart is shown in figure 5.41. We can further modify this graph, for example by changing the colours and adding the corresponding numbers to the descriptions. So, at first we have to prepare new descriptions in the graph that we save in the variable `descriptions`. To change the colours in the graph we apply the function `rainbow()`, which defines the colour palette. Its arguments are:

- `n` the number of colours ($\geq 1$) to be in the palette,
- `s,v` the "saturation" and "value" to be used to complete the colour descriptions
- `start` the (corrected) hue in $\langle 0;1\rangle$ at which the rainbow begins,
- `end` the (corrected) hue in $\langle 0;1\rangle$ at which the rainbow ends,
- `gamma` the gamma correction, for each colour, `(r,g,b)` in RGB space (with all values in $\langle 0;1\rangle$), the final colour corresponds to $(r^\gamma, g^\gamma, b^\gamma)$,
- `alpha` the alpha transparency, a number in $\langle 0;1\rangle$, (0 means transparent and 1 means opaque).

We can use also alternative colour palettes like `heat.colors()`, `terrain.colors()`, `topo.colors()` or `cm.colors()`. The source code is then like follows:

```
description<-paste(labels,"\n",data,sep="")
pie(data,description,main="Monthly expenses",
 col=rainbow(length(data)))
```

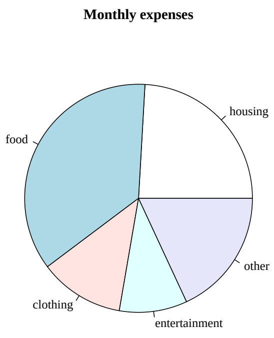


Figure 5.41: The pie chart of the monthly expenses of the household.

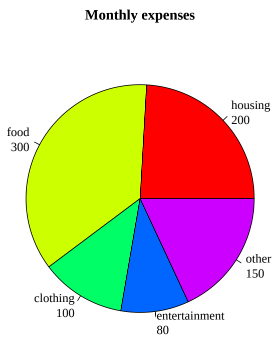


Figure 5.42: The pie chart of the monthly expenses of the household with colours customized by `rainbow()` function.

It produces the graph shown in figure 5.42. As further improvements, we may require descriptions with percentages and a graph display with a 3D effect. At first we must recalculate the percentages and add the results into the descriptions. In order to get the percentages in the integers, we apply the `trunc()` function. Then we can produce the pie chart, this time with the `heat.colors()` palette. So we have the source code

```
description<-paste(labels,"\n",trunc(100*data/sum(data)),"%", sep="")
pie(data,description,main="Monthly␣expenses",
 col=heat.colors(length(data)))
```

The result we present in figure 5.43, where we can observe that all percentages do not give the sum of 100 %. It is caused by the rounding the numbers. We remove this defect allowing more decimal digits using the `round()` function instead of `trunc()`. So we modify the chart descriptions by changing the command in the code to:

```
description<-paste(labels,"\n",round(100*data/sum(data),digits=2),
 "%", sep="")
```

The pie chart in figure 5.44 we produce by the same form of the `pie()` function as in previous case. We have apply the `heat.colors()` palette again.

In order to get the 3D-effect in the chart, we must use the package `plotrix`. We call this package by standard command `library("plotrix")`.[1] This package defines the function `pie3D()` we will use to create the charts with 3D-effect.

Our source code then has the form

```
library("plotrix")
pie3D(data,labels=description,main="Monthly␣expenses",
 col=rainbow(length(data)))
```

The function `pie3D()` has additional argument `explode` that enables plotting the the pie sectors exploding outward from the centre. If we adapt the source code to

[1]If the package `plotrix` is not installed, we have to install it using `install.packages("plotrix")`.

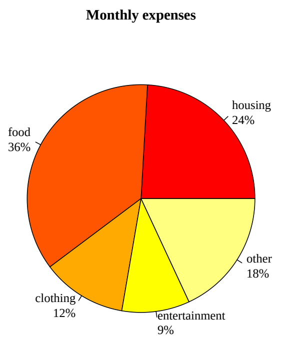


Figure 5.43: The pie chart of the monthly expenses of the household and colours defined by `heat` palette.

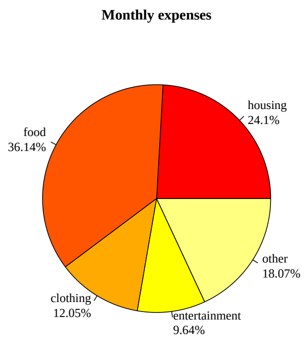


Figure 5.44: The pie chart of the monthly expenses of the household and colours defined by `heat` palette and higher precision.

```
pie3D(data,labels=description,main="Monthly␣expenses",
 col=rainbow(length(data)),explode=0.1)
```

we get the pie chart shown in figure 5.46. We can further customize the chart appearance using the parameters `height` that states the height of the 3D pie (the default value is 0.1) and `theta` that changes the viewing angle (the default angle is $\pi/6$).

An example of the 3D pie chart with customised height and viewing angle we see in the figure 5.47. And here is the source code that produces it.

```
pie3D(data,labels=description,main="Monthly␣expenses",
 col=terrain.colors(length(data)),height=0.2,theta=1.5,explode=0.1)
```

Let us note that in order to illustrate more colour palettes we used the `terrain.colors()` instead of `rainbow()` in this chart. Useful alternative to the pie charts is `fun.plot()` defined in the `plotrix` packages. It allows to compare visually the pie sectors of the chart. We can customize the fun plot setting the additional arguments:

- `max.span` the angle of the maximal sector in radians. The default is to scale `data` so that it sums to $2\pi$.
- `ticks` the number of ticks that would appear if the sectors were on a pie chart. Default is no ticks.

Illustration of the fun plot we see on the figure 5.48 and here is the source code that creates this fun plot as an outcome.

```
fan.plot(data,labels=description,main="Monthly␣expenses",
 col=rainbow(length(data)),max.span=pi,ticks=max(data))
```

How we can see from the chart in figure 5.48, the disadvantage of the fun plot is a large white space above the chart. We can remove this space by setting the new graphical

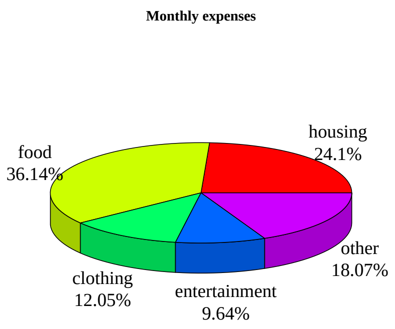


Figure 5.45: The pie chart of the monthly expenses of the household with 3D effect.

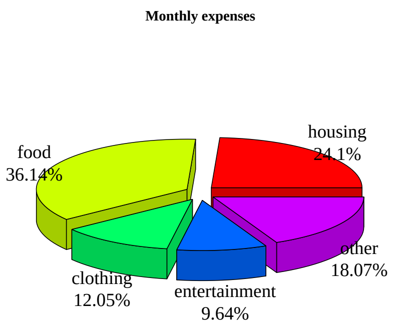


Figure 5.46: The pie chart of the monthly expenses of the household with 3D effect and exploded parts.

device with user defined height and width. A new graphical window we open by function `new.dev()`. The required size of the window we define in arguments `height` and `width`. The measure units we then submit as the `unit` argument. In our case we can set graphical window by command:

```
dev.new(width=10,height=5,unit="cm")
```

The fan chart then completely fills whole plotting area. The function `dev.new()` allows to open more graphical devices. The currently active graphical device we recognize using the function `dev.cur()` and we can switch among the devices by function `dev.set()` whose argument is an integer associated with the graphics device we want to switch to.

## 5.5 Box plots

In descriptive statistics, a box plot or boxplot (also known as box and whisker plot) is a type of chart often used in explanatory data analysis. Box plots visually show the distribution of numerical data and skewness through displaying the data quartiles (or percentiles) and averages. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.

A boxplot is constructed of two parts, a box and a set of whiskers shown in figure 5.49. The box is drawn from the lower quartile to the upper quartile with a horizontal line drawn inside the box to denote the median. However, the whiskers can represent several alternative values, among the observed data:

- the minimum and maximum of all of the data,
- one standard deviation above and below the mean of the data,
- the 9-th percentile and the 91-st percentile,
- the 2-nd percentile and the 98-th percentile.

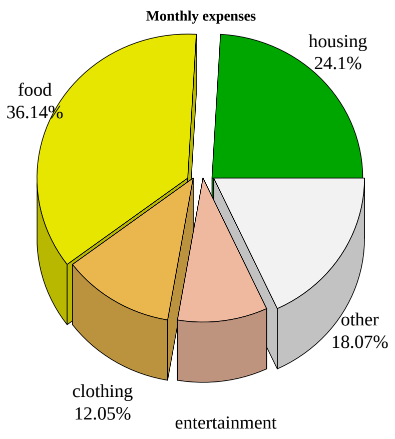


Figure 5.47: The 3D pie chart with customized appearance with the `terrain` palette.

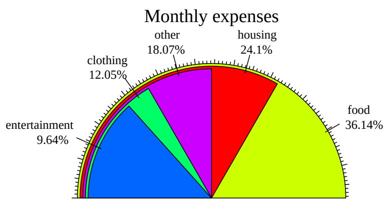


Figure 5.48: The fan plot of the monthly expenses.

Any data not included between the whiskers should be plotted as an outlier with a dot, small circle, or star, but occasionally this is not done.

Boxplots are created in R by using the `boxplot()` function. The basic syntax to create a boxplot in R is:

```
boxplot(x, data, notch, varwidth, names, main)
```

The meaning of the parameters is as follows:

- `x` is a vector or a formula,
- `data` is the data frame.
- `notch` is a logical value. Set as TRUE to draw a notch.

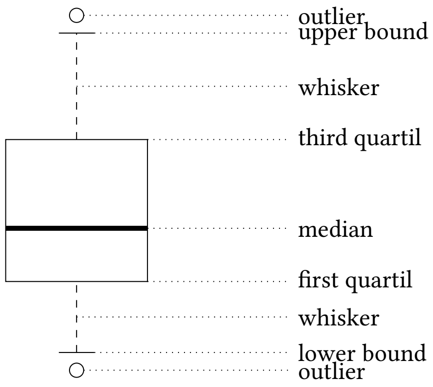


Figure 5.49: Scheme of the box plot construction.

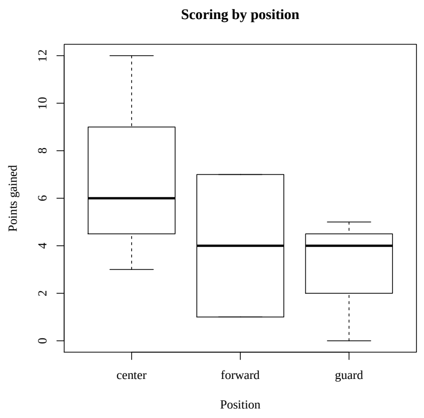


Figure 5.50: The boxplot comparing the points gained in basketball game by player position.

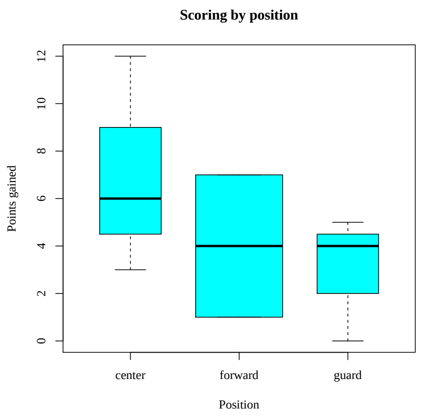


Figure 5.51: The boxplot comparing the points gained in basketball game by player position with user modified color and proportional widths.

- `varwidth` is a logical value. Set as true to draw width of the box proportionate to the sample size,
- `names` are the group labels which will be printed under each boxplot,
- `main` is used to give a title to the graph.

Let the statistical data from the basketball game are saved in file `players.csv`. This file contains the players identification, position, and number of attempted and made shoots. The boxplots in figure 5.50 compare the points gained by position. The corresponding source code is:

```
players<-read.csv("players.csv")
boxplot(made~position,data=players,
xlab="Position",ylab="Points␣gained",
main="Scoring␣by␣position")
```

Here we read the data from the file at first, and in the second step we drew the boxplot. Similarly like the other types of the plots, we can modify the outlook of the plot. How illustrates the next source code, we color the plot and setting the value `varwidth=TRUE` we arrange the width of the boxes to be proportional to the sample size.

```
boxplot(made~position,data=players,
xlab="Position",ylab="Points␣gained",
main="Scoring␣by␣position",col="cyan",varwidth=TRUE)
```

The boxplot obtained by applying this source code we see in figure 5.51. Setting the logical variable `horizontal` to `TRUE` we can rotate the boxes in the boxplot. Moreover, the colors can vary from box to box, how illustrates the boxplot in figure 5.52. We extend the arguments of the `boxplot()` function, how we see in the following listing.

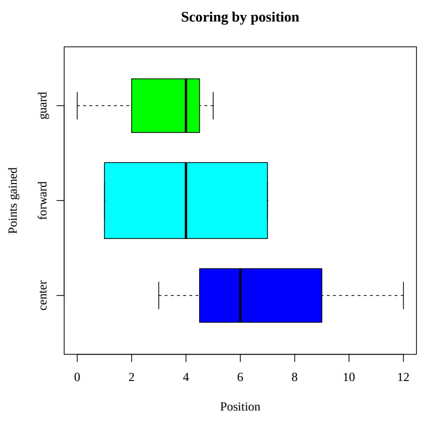


Figure 5.52: The horizontal boxplot comparing the points gained in basketball game by player position.

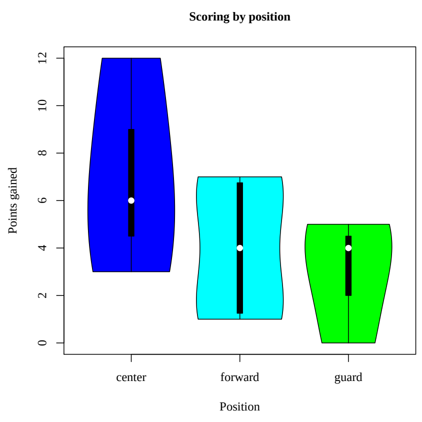


Figure 5.53: The violin plot comparing the points gained in basketball game by player position.

```
boxplot(made~position,data=players,
xlab="Position",ylab="Points␣gained",
main="Scoring␣by␣position",col="col=c("blue","cyan","green"),
varwidth=TRUE,horizontal=TRUE)
```

An alternative to the boxplot is the violin plot. The violin plot solves the issues regarding displaying the underlying distribution of the observations, as these plots show a kernel density estimate of the data. In order to construct the violin plot, we have to input the `vioplot` library to the base R. This library defines the function `vioplot()` that enables constructing the violin plots. Its use is analogical to the `boxplot()` function, how we see in the following source code:

```
library("vioplot")
vioplot(made~position,data=players,
xlab="Position",ylab="Points␣gained"
,main="Scoring␣by␣position",col=c("blue","cyan","green"))
```

The corresponding output we see on the figure 5.53. Here we can observe, how shaping of the boxes illustrates the density of the empirical distribution. Figure 5.54 shows, how we can add jittered data points to the previous violin plot. We use the `stripchart()` function with logical variable add set to `TRUE`. The source code is then extended to the following form:

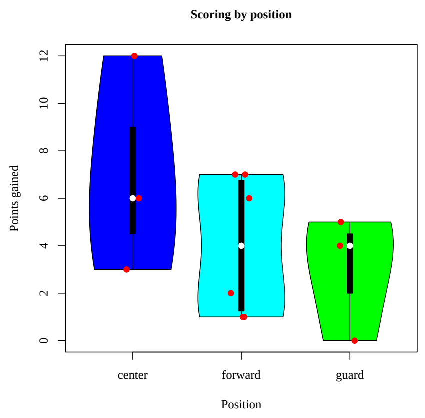


Figure 5.54: The violin plot comparing the points gained in basketball game by player position with jittered data.

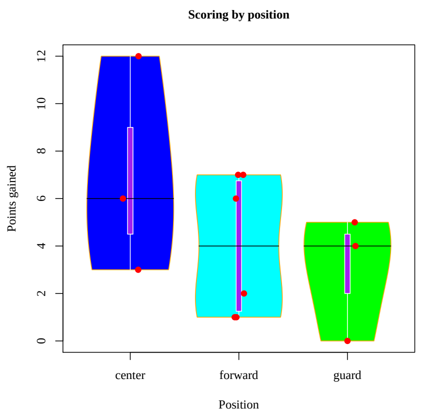


Figure 5.55: The customized violin plot comparing the points gained in basketball game by player position with jittered data added.

```r
vioplot(made~position,data=players,
xlab="Position",ylab="Points␣gained"
,main="Scoring␣by␣position",col=c("blue","cyan","green"))
stripchart(made ~ position, data=players, vertical = TRUE,
 method = "jitter",pch = 19,col="red", add = TRUE)
```

The violin plots can be further customized by the following variables:

- `col` that defines color of the area,
- `rectCol` that defines color of the rectangle,
- `lineCol` that defines color of the line,
- `colMed` that defines Pch symbol color,
- `border` that defines color of the border of the violin,
- `pchMed` that defines Pch symbol for the median,
- `plotCentre` that defines how will be plotted a median line (`points` or `line`).

The outlook of our violin plot can be customized by the following source code

```r
vioplot(made~position,data=players,
xlab="Position",ylab="Points␣gained",main="Scoring␣by␣position",
col=c("blue","cyan","green"),
rectCol="purple",lineCol="white",colMed="yellow",
border="orange",plotCentre="line")
stripchart(made ~ position, data=players, vertical = TRUE,
 method = "jitter",pch = 19,col="red", add = TRUE)
```

The result we see in figure 5.55.

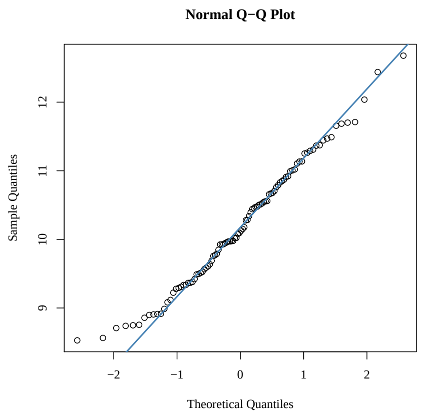


Figure 5.56: The Q-Q plot confirms graphically normality of the sample.

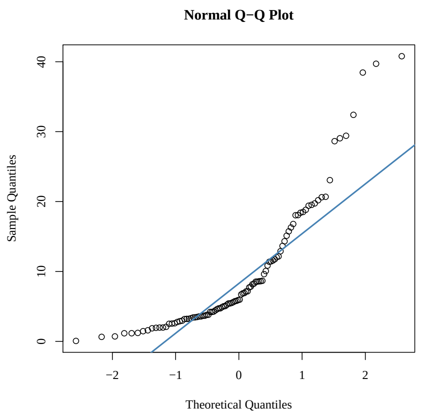


Figure 5.57: The Q-Q pot of the exponential sample against the normal theoretical distribution.

## 5.6 Q-Q plots

The quantile-quantile plot (or shortly Q-Q plot), is a graphical tool to help us assess if a set of data plausibly came from some theoretical distribution such as a normal or exponential. For example, if we run a statistical analysis that assumes our dependent variable is normally distributed, we can use a normal Q-Q plot to check that assumption. It is just a visual check, not an exact proof, but it allows us to see at-a-glance if our assumption is plausible, and if not, how the assumption is violated and what data points contribute to the violation.

A Q-Q plot is essentially a scatterplot created by plotting two sets of quantiles against one another. If both sets of quantiles came from the same distribution, the points form a roughly straight line. Q-Q plots take our sample data, sort it in ascending order, and then plot them against quantiles of the suggested theoretical distribution. The number of quantiles is selected to match the size of our sample data.

In R, we have two functions to create Q-Q plots: `qqnorm()` and `qqplot()`. While `qqnorm()` creates normal Q-Q plot (means the suggested theoretical distribution to be normal), the `qqplot()` function allows us to create a Q-Q plot to compare two datasets.

To illustrate how does the `qqnorm()` function work, let us at first generate the random sample from the normal distribution. Then we create the Q-Q plot, and in the same picture we add the 45-degree reference straight line to check graphically the normality. The source code we have just described is as follows. As a result we obtain the Q-Q plot presented in figure 5.56.

```
x<-rnorm(100,mean=10,sd=1)
qqnorm(x)
qqline(x, col = "steelblue", lwd = 2)
```

In figure 5.57 we illustrate the situation, when the sample does not come from the normal distribution. Using the function`rexp()` we generate the exponential sample and

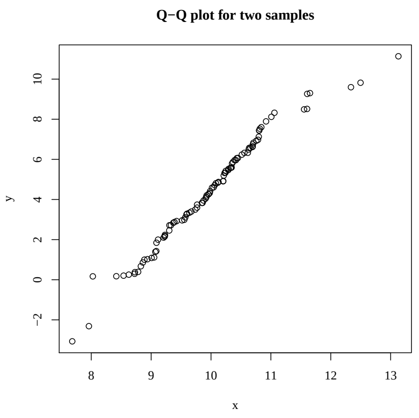


Figure 5.58: The Q-Q plot confirms graphically the samples come from the same distribution type.

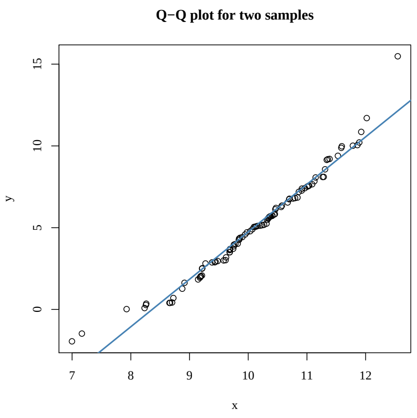


Figure 5.59: The Q-Q pot of the exponential sample against the normal theoretical distribution.

later we draw the Q-Q plot against the normal distribution. the corresponding source code is in the following listing:

```
x<-rexp(100,rate=1/10)
qqnorm(x)
qqline(x, col = "steelblue", lwd = 2)
```

From the graph in figure 5.57 it is clear, that the plot of the sample points against the theoretical quantiles violates the straight line.

In order to compare, if two random samples come from the same distribution type, we will generate two vectors x and y and then apply the function qqplot() on these samples. So we get the plot as illustrated in figure 5.58. And here is the source code:

```
x<-rnorm(100,mean=10,sd=1)
y<-rnorm(100,mean=5,sd=3)
qqplot(x,y,main="Q-Q␣plot␣for␣two␣samples")
```

Unfortunately, the qqplot() function does not cooperate with the qqline() function that is joined with the qqnorm() function. Before proceeding to get the helping straight line in the plot, let us not, that qqplot() function is equivalent with using the plot() function combined with sort() function. So, qqplot(x,y) is equivalent to plot(sort(x),sort(y)). To add the auxiliary straight line, we use abline() function together with the sort() function. So, the plot in figure 5.59 we obtain by submiting the source code

```
x<-rnorm(100,mean=10,sd=1)
y<-rnorm(100,mean=5,sd=3)
qqplot(x,y,main="Q-Q␣plot␣for␣two␣samples")
abline(lm(sort(y) ~ sort(x)), col = "steelblue", lwd = 2)
```

The qqplot() function can be used to compare the sample with any theoretical distribution. We generate the vector of the quantiles of the theoretical distribution of

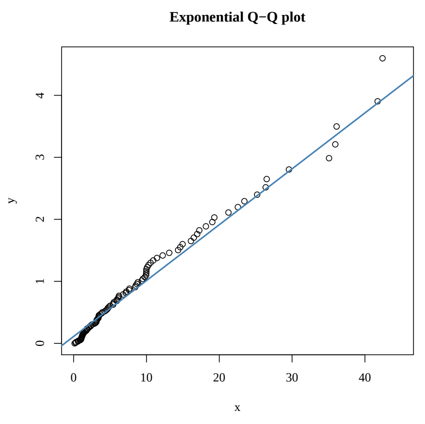


Figure 5.60: The Q-Q plot confirms graphically the sample comes from the exponential distribution.

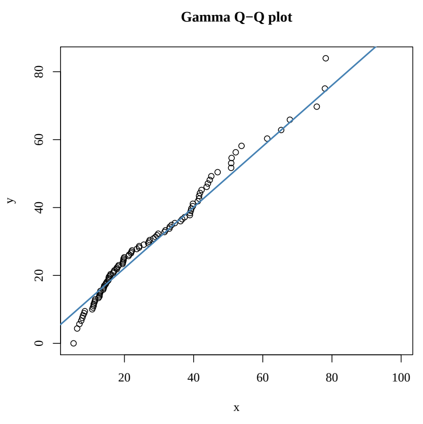


Figure 5.61: The Q-Q pot of the exponential sample against the normal theoretical distribution.

the same length as the given sample and then we use this vector as the second dataset entering into the qqplot() function. Let us suppose for certainty, we have the sample from the exponential distribution in the vector x. To compare it graphicaly with the theoretical exponential distribution, we will generate its quantiles in vector y at first and further we apply the qqploy() on these two vectors. So the source looks like this:

```r
x<-rexp(100,rate=1/10)
y<-qexp(seq(0,1,by=0.01),rate=1)
qqplot(x,y,main="exponential␣Q-Q␣plot")
abline(lm(sort(y[1:100]) ~ sort(x)), col = "steelblue", lwd = 2)
```

As a result we get the plot illustrated in figure 5.60. The auxiliary line shows, the sample comes from the exponential distribution. Let us note, to draw the auxiliary line we need to take two vectors of the same length. Because our sample contains 100 values, we take first 100 percentiles and omit the last value, that equals to $\infty$.

It is known from the probability theory, that by summing the exponential random variables with the same rate parameter $\lambda$ we get a new random variable, that has Erlang distribution with the same rate parameter and shaping parameter that equals to the number of summands. The Erlang distribution is a special case of the Gamma distribution with whole number as the shaping parameter. We can graphically illustrate it by Q-Q plot in figure 5.61 . At first, we generate the vector x as the sum of three exponential variables and as the second vector y we generate the quantiles of the Gamma distribution the same rate parameter and shaping parameter equal to 3. The source code is like this:

```r
x<-rexp(100,rate=1/10)+rexp(100,rate=1/10)+rexp(100,rate=1/10)
y<-qgamma(seq(0,1,by=0.01),rate=1/10,shape=3)
qqplot(x,y,main="Gamma␣Q-Q␣plot")
abline(lm(sort(y[1:100]) ~ sort(x)), col = "steelblue", lwd = 2)
```

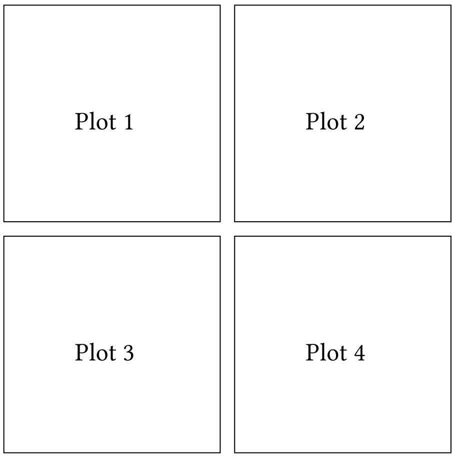


Figure 5.62: Multiple plots in one graph combined by parameter `par(mfrow=c(2,2))`.

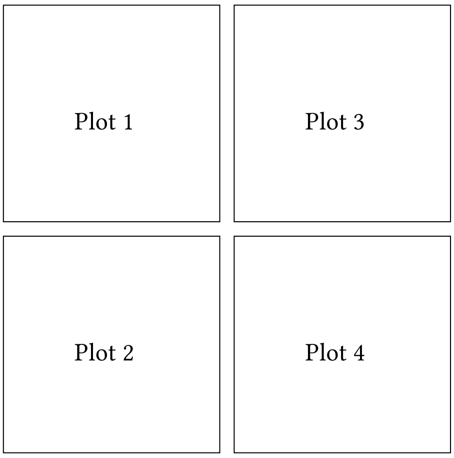


Figure 5.63: Multiple plots in one graph combined by parameter `par(mcol=c(2,2))`.

## 5.7 Simple plot combination

It is quiet simple to combine plots in base R with `mfrow` and `mfcol` graphical parameters. We just need to specify a vector that determines the number of rows and the number of columns we plan to create. The decision of which graphical parameter we should use depends on how do we want our plots to be arranged:

- `mfrow` the plots will be arranged by rows,
- `mfcol` the plots will be arranged by columns.

Figures 5.62 and 5.63 illustrate, how the multiple plots are arranged according the used specification by `mfrow` or `mcol`. A simple use of two plots side by side we have already shown in figure 5.28. Now we will illustrate, that we can place different types of plots in the same graph.

As the fisrt step we generate the sample, in this case from the exponential distribution. Let us note, that using the `set.seed()` function we assure to get same result in all repeated runs of the source code. Further we set four graphs to be placed in the picture and arranged by rows.[2] Then we gradually display the histogram in top left position, box plot in the right top position, the scatter plot in the bottom left position and finally the pie graph in the bottom right position. The result we see in figure 5.64.

The source code used to produce the figure 5.64 is as follows.

```r
set.seed(5)
x <- rexp(80)
# Two rows, two columns
par(mfrow = c(2, 2))

# Plots
```

[2]If you prefer arranging by columns, replace `mfrow=c(2,2)` by `mfcol=c(2,2)`.

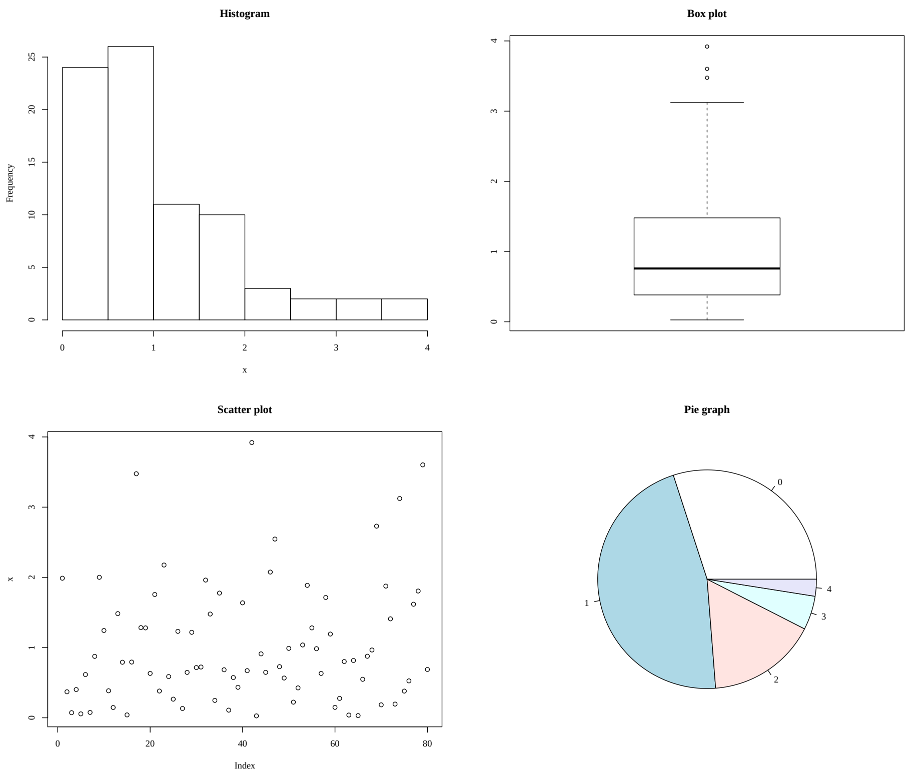


Figure 5.64: Combination of the histogram, boxplot, scatter plot, and pie graph of the data in the same figure.

```
hist(x, main = "Histogram")                       # Top left
boxplot(x, main = "Box␣plot")                   # Top right
plot(x, main = "Scatter␣plot")                    # Bottom left
pie(table(round(x)), main = "Pie␣graph")  # Bottom right

# Back to the original graphics device
par(mfrow = c(1, 1))
```

Frequently it can happen, we need to create the picture with more complex structure. In such situations we have to use the `layout()` function. This function has four important arguments:

- `mat` a matrix where each value represents the location of the figures.
- `widths` a vector for the widths of the columns. You can also specify them in centimeters with `lcm()` function.
- `heights` a vector for the height of the columns. You can also specify them in centimeters with `lcm()` function.
- `respect` Boolean or a matrix filled with 0 and 1 of the same dimensions as `mat` to indicate whether to respect relations between widths and heights or not.

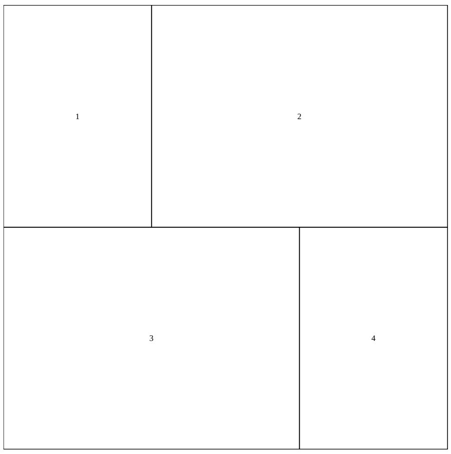


Figure 5.65: Scheme of the more complex structure of the plots combination obtained by the `layout()` function.

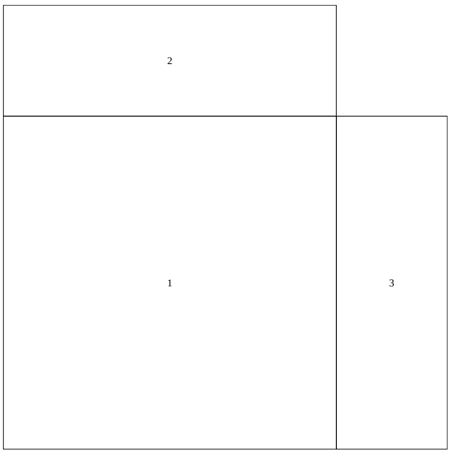


Figure 5.66: Scheme of the more complex structure of the plots combination with one empty position.

If we are not sure, we can preview a layout making use of the `layout.show()` function before adding the plots. Using the code

```r
l <- layout(matrix(c(1, 2, 2,   # First, second,
                     3, 3, 4), # third and fourth plot
            nrow = 2,
            ncol = 3,
            byrow = TRUE))
layout.show(l)
```

we get the result illustrated in the figure 5.65. In case we want to omit some graph in the scheme, we put `0` on the corresponding position in the `mat` matrix. An example we see in figure 5.66. The adequate source code is modified to

```r
l <- layout(matrix(c(2, 0, 1, 3),
              nrow = 2, ncol = 2,
              byrow = TRUE),
       widths = c(3, 1),
       heights  = c(1, 3), respect = TRUE)
layout.show(l)
```

We illustrate this method on the scatter plot accomplished with the marginals in the form of histogram and box plot. Once we have generated the vector `x` in the previous demonstrations, we run the following source code:

```r
 l <- layout(matrix(c(2, 0, 1, 3),
              nrow = 2, ncol = 2,
              byrow = TRUE),
       widths = c(9, 3),
       heights  = c(3, 9), respect = TRUE)
plot(x, main = "Scatter␣plot")
hist(x, main = "Histogram")
boxplot(x, main = "Box␣plot")
```

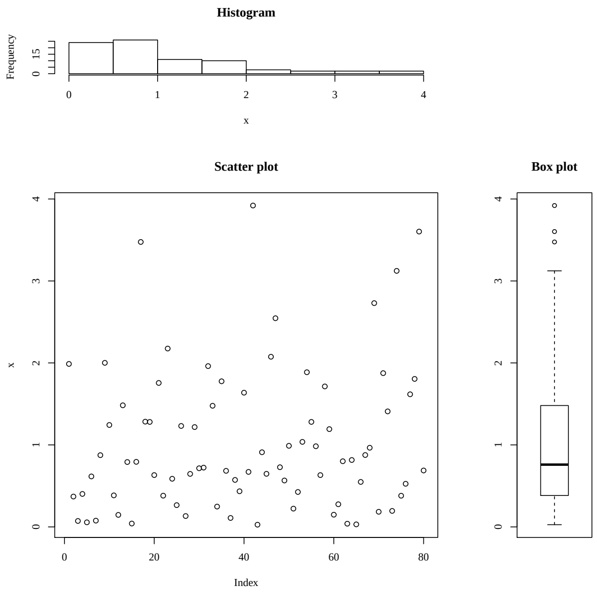


Figure 5.67: Combination of the scatter plot and corresponding histogram and boxplot as its marginals.

The result we see in figure 5.67. This combination of plots allows also better illustration of the violin plots mentioned in the previous section. We generate a sample from bimodal distribution. Two modes are then also nicely visible on the violin plot, what traditional boxplot does not. The combination of the histogram an corresponding violin plot above is shown in figure 5.68.

Here is the source code, that creates the combination of the histogram and marginal violin plot on firgure 5.68. Let us mention adjusting the range of axes setting the parameters `xlim` and `ylim`. Here it is necessary to realize that the violin plot shows the coordinates of y, it is only oriented horizontally, therefore we set there `ylim` argument. It is as well necessary to manage the margins of both plots, to set the axis in correct positions. The marginal frame of the violin plot is removed by setting `bty="n"` in the `par()` function.

```
 # Generating the bimodal data
 n <- 10000
ii <- rbinom(n, 1, 0.5)
data<-rnorm(n, mean=130, sd=10) * ii +rnorm(n,mean=80,sd=5) * (1-ii)
# Setting layout of the plot
l <- layout(matrix(c(2, 1),
              nrow = 2, ncol = 1,
              byrow = TRUE),
       widths = c(9, 3),
       heights  = c(3, 9), respect = TRUE)

# Histogram
hist(data, probability = TRUE, col = "grey", axes = FALSE,
 main = "", xlab = "",  ylab = "",xlim=c(50,160))
```

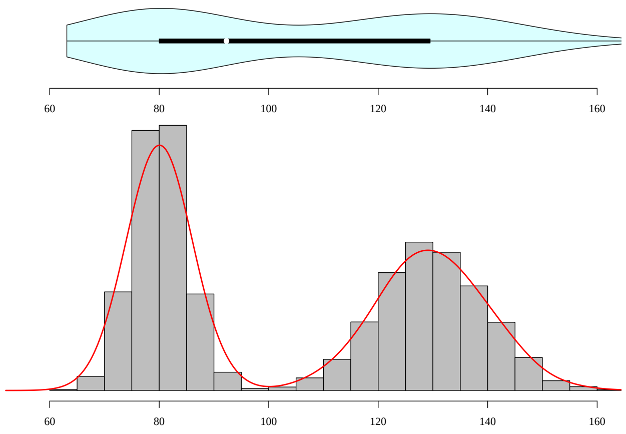


Figure 5.68: Combination of the histogram and corresponding violin plot.

```

# X-axis
axis(1)

# Density
lines(density(data), lwd = 2, col = "red")

# Add violin plot
par(mar = c(0, 4.1, 0, 0),bty="n")
vioplot(data, horizontal = TRUE, yaxt = "n", axes = FALSE,
col = rgb(0, 1, 1, alpha = 0.15),ylim=c(50,160))
```

# Basics of the statistics

In everyday life, we encounter a large number of facts collected in the form of data. Statistics provides us with methods to organise and summarise this data and procedures to draw conclusions based on the information contained in this data. The typical target of a statistical investigation is some well-defined set of objects, which we refer to as the population.

When carrying out a statistical enquiry, two situations may arise. Sometimes we have the required information on all elements of the population. In this case, we have a data set that we call a *census*. As an example, consider a periodic population census. However, collecting information on all elements belonging to a population is very time-consuming and costly, which frequently makes it impossible to carry out a census. Instead of analysing the whole population, we only analyse data from a subset called a *sample*.

## 6.1 Descriptive characteristics of the sample

In the previous chapters, we have shown an illustrative graphical interpretation of the obtained data. However, the formal analysis requires performing some numerical calculations. We will focus on the numerical data in particular. We will discuss the approach to categorical data at the end of the chapter.

We will suppose, that our sample is in the form of the numerical values $x_1, x_2, \dots, x_n$. Essential characteristics of such a set of numbers are its location (particularly the centre) and its variability. At first, we will present some location characteristics and their computation, and later in the section, the variability characteristics.

### 6.1.1 Measures of location

#### The mean

The *sample mean* of the given set of numbers $x_1, x_2, \dots, x_n$ is defined by formula:

$$\overline{x} = \frac{x_1 + x_2 + \cdots + x_n}{n} = \frac{\sum_{i=1}^{n} x_i}{n}. \tag{6.1}$$

The mean, or the arithmetic average, given by equation (6.1) is the most familiar measure of centre. In the R environment it is implemented as the `mean()` function and its use is very simple. On the website of the Slovak central bank `https://www.nbs.sk/en/monetary-policy/macroeconomic-database` we can find macroeconomic database. We can download for example the `.csv` file containing the numbers of new passenger cars registrations in selected time period. This file is implicitly saved as `macrostat.csv`. Due to the use of comma as the decimal separator, we read the data using the `read.csv2()` function. To compute the mean, we have to use `as.numeric()` because `cars[1,]` gives the values in the list format. So to obtain the average value of the monthly registered new passenger cars (in thousands) in years 2017-18 we can use the code:

```
cars<-read.csv2("macrostat.csv",header=FALSE,sep=";")
mean(as.numeric(cars[1,]))
[1] 8.090125
```

So we see, the monthly average of the newly registered cars is 8.090 thousands.

It is often the case that the values of the statistical trait of interest are ordered in a sequence of absolute frequencies. In this case, we modify the relation (6.1) for calculating the sample mean to the form:

$$\overline{x} = \frac{x_1 \cdot n_1 + x_2 \cdot n_2 + \cdots + x_k \cdot n_k}{n_1 + n_2 + \cdots + n_k} = \frac{\sum_{i=1}^{k} x_i \cdot n_i}{\sum_{i=1}^{k} n_i}. \qquad (6.2)$$

where $x_i$'s denote the values of the variable and $n_i$ denotes the absolute frequency of $x_i$.

In this case, we need to define a custom function to calculate the mean value. As input values, we will specify two vectors. The first vector contains the values that the random variable takes, and the second is a vector of their multiplicities. Before performing the calculation according to the relation (6.2), it is necessary to verify that both vectors have the same length. The corresponding function `mean2()` can then be defined as follows:

```
mean2<-function(arg1,arg2){
    if (length(arg1)==length(arg2)){
        s<-sum(arg1*arg2)/sum(arg2)
    }
    else{s<-c("Arguments␣are␣not␣of␣equal␣length")}
    return(s)
}
```

The use of the just defined function `mean2()` we can illustrate on the variable that takes the values from the set $\{1, 2, \ldots, 10\}$. We can generate the absolute frequencies of these values using the Poisson distribution. The feasible values are stored in the vector `a` and the absolute frequencies in the vector `b`. See the listing.

```
a<-c(1,2,3,4,5,6,7,8,9,10)
b<-rpois(10,20)
mean2(a,b)
5.38613861386139
```

Another alternative is to sort the measured values of the variable into class intervals. It means structuring the data into a table containing the class intervals and their absolute frequencies. Assuming a uniform distribution of values within each interval, we choose the midpoints of these intervals as their representatives.

## The median

We can characterise median as the middle value if the observed values are sorted from smallest to largest. Formally we can say, that variable value is larger then median equals to the probability, that its value is less then median and consequently this probability equals to 1/2. Having a sample of numbers $x_1, x_2, \dots, x_n$ the median $\tilde{x}$ is given as

$$\tilde{x} = \begin{cases} \left(\frac{n+1}{2}\right)\text{-th ordered value} & n \text{ is odd} \\ \frac{1}{2}\left(\left(\frac{n}{2}\right)\text{-th} + \left(\frac{n}{2}\right)\text{-th ordered value}\right) & n \text{ is even} \end{cases} \tag{6.3}$$

Let us note, this measure is more robust then mean. The robustness of the value means it is not strongly influenced by extremal values of the variable. In the R environment is implemented the function `median()`. So we can simply find the median of monthly newly registered passenger cars using the code

```
> median(as.numeric(cars[1,]))
[1] 8.2425
```

## Quantiles

The median, defined in the previous subsection, divides the sample into two equally likely subsets. To refine the measure of location, we can divide the sample into more then two equally likely parts. If we divide the data for example into four parts, we obtain *quartiles*. Similarly, dividing the sample into the one hundred parts, we obtain *percentiles*.

Generally, we can divide the sample into any number $q$ of equally likely parts. These values are called *$q$-quantiles*, and the $k$-th $q$-quantile for the random variable $X$ is defined by formula

$$\mathbb{P}\left(X < x\right) \leq \frac{k}{q}. \tag{6.4}$$

To find the quantiles, in R is implemented the `quantile()` function. Without any optional parameters it gets the minimum of the sample, the first quartile, median, the third quartile and the maximal value of the sample. We can illustrate it on the COVID-19 data, downloaded from the official website of the Slovak government `https://korona.gov.sk`. In the following source code we download the data from the distanced source at first, and then we compute the quartiles of the daily increase:

```
data<-read.csv("https://mapa.covid.chat/export/csv",header=T,sep=";")
> quantile(data[,4])
   0%    25%    50%    75%   100%
    0     30    232   1737  15278
```

We can also set some optional arguments of the `quantile()` function:

- `probs` numeric vector of probabilities with values in $\langle 0, 1\rangle$, that defines the probability levels for the required quantiles,
- `na.rm` logical value, if true, any NA and NaN's are removed from `data` before the quantiles are computed,
- `names` logical value, if true, the result has a names attribute. Set to `FALSE` for speed-up with many `probs`.

To find the deciles of the daily increases, we use the `quantile()` function in the form:

```
> quantile(data[,4],probs=seq(0,1,by=0.1))
   0%    10%    20%    30%    40%    50%    60%    70%    80%    90%   100%
    0      6     20     43     91    232    642   1293   2034   3041  15278
```

### 6.1.2 Measures of variability

The random samples are frequently characterised by considerable variability of the measured data. Therefore, the fact that we know the mean value, the most frequent value and the median value are not sufficient to describe the statistical population. There are a number of different measures that express the variability of the values in the sample.

#### The variation range

The variation range, or shortly only range, is s one of the most basic and the simplest measures of variation. It depends only on two values in the sample – the greatest and the smallest values. It is defined as the difference

$$R = \max\{x_1, \dots, x_n\} - \min\{x_1, \dots, x_n\} \tag{6.5}$$

The shortcoming of this measure is that it does not use all the measured values. However, it is well applicable for a quick orientation on the extent of variability of a given sample.

The `range()` function determines the range of variation in the R language environment. Its outputs are two values – the greatest and the smallest value in the sample. If we want to express the variation range as a single value by definition (6.5), we use the `max()` and `min()` functions. We can illustrate this in the following source code.

```
> x<-c(5,10,12,4,16,8,9)
> range(x)
[1]  4 16
> R<-max(x)-min(x)
> R
[1] 12
>
```

#### Interquartile range

The interquartile range is a measure of statistical dispersion. It is defined as the difference between the upper and lower quartiles of the data. Assigning these quartiles as $Q_3$ and $Q_1$, we can express the interquartile range $IQR$ as

$$IQR = Q_3 - Q_1. \tag{6.6}$$

In order to compute the interquartile range, here is the function `IQR()` is implemented in the R language.

```
> x<-c(5,10,12,4,16,8,9)
> IQR(x)
[1] 4.5
>
```

### Mean absolute deviation

The mean absolute deviation $MAD$ of a dataset is the average distance between each data point and the mean. For the sample $x_1, \dots, x_n$, it is defined by formula

$$MAD = \frac{1}{n} \sum_{i=1}^{n} |x_i - \overline{x}| \,. \tag{6.7}$$

In the R environment, the `mad()` function is implemented to calculate the mean absolute deviation. We illustrate its use in the source code:

```
> x<-c(5,10,12,4,16,8,9)
> mad(x)
[1] 4.4478
>
```

### Variance and standard deviation

Variance is the most popular and the most frequently used measure of the sample variation. If we have the sample $x_1, \dots, x_n$, we define the sample variation $s^2$ by formula

$$s^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2 \,. \tag{6.8}$$

In the case of the data ordered in the sequence absolute frequencies $n_1, \dots, n_k$, $\sum_{i=1}^{k} n_i = n$, of the values $x_1, \dots, x_k$, we modify the formula (6.8) into the form

$$s^2 = \frac{1}{n} \sum_{i=1}^{k} n_i \, (x_i - \overline{x})^2 \,. \tag{6.9}$$

The standard deviation is then defined as the square root from the variance, means:

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2}, \tag{6.10}$$

or in the case of the ordered data set

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n} \sum_{i=1}^{k} n_i \, (x_i - \overline{x})^2}. \tag{6.11}$$

There is a simple direct relationship between standard deviation and variance. However, the variance comes out in squares of the measurement units. The purpose of introducing a standard deviation to describe the variability of a sample is to achieve the same measurement units as in the original dataset.

We must be careful when using the `var()` and `sd()` functions implemented in R. This is because their results are an unbiased estimates of the variance and standard deviation of the whole population.[1] If we want to compute the sampling variance according to the relation (6.8), we have to define our own function, which we illustrate in the following source code.

[1]We mention this estimate property in the next section

```
> variance<-function(x) sum((x-mean(x))^2)/length(x)
> stdev<-function(x) sqrt(variance(x))
> variance(x)
[1] 14.40816
> stdev(x)
[1] 3.795809
> var(x) # compare results
[1] 16.80952
> sd(x)
[1] 4.099942
```

Besides these absolute variability measures, we can define the relative variability measures. These relative measures of variability remove the problems of comparing sets with different values of the trait of interest. They are also functional when comparing the variability of multiple traits measured in incompatible units. The essence of their construction is to compare the absolute measure of variability with the arithmetic average of the given sample.

### The Coefficient of Variation

The coefficient of variation is a statistical measure of the relative dispersion of data points in a data series around the mean. It shows the extent of variability in relation to the mean of the population. The coefficient of variation $CV$ is defined as the ratio of the standard deviation $s$ to the mean $\overline{x}$

$$CV = \frac{s}{\overline{x}}. \tag{6.12}$$

The coefficient of variation is frequently expressed as a percentage. Then the result obtained by definition (6.12) has to be multiplied by 100. The coefficient of variation has got no implementation among the function in R, but we can easily compute it using the existing functions or define new function. Let us see the source code.

```
> cv<-function(x) variance(x)/mean(x) * 100
> cv(x)
[1] 157.5893
```

## 6.1.3 Skewness and kurtosis

Skewness and kurtosis are the measures which tell about the shape of the data distribution. These measures give us an indication of how small or large values are concentrated in the statistical population. We are also interested in how strong the concentration of values near the mean is.

### Skewness

Skewness measures the asymmetry of the distribution or data set. We define the skewness $\gamma_1$ as

$$\gamma_1 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^3}{s^3}. \tag{6.13}$$

Depending on the value of $\gamma_1$ there exist 3 types of skewness. If $\gamma_1 > 0$ we speak about positive skewness. It means that majority of the data are less than the average value. On the other hand there are the negative skewed data, when $\gamma_1 < 0$ In this case is the majority of the values in the dataset greater than the mean. Finally zero skewness $\gamma_1 = 0$ represents the symmetric distribution of the data.

### Kurtosis

Kurtosis measures the sharpness of the peak in the data distribution.We define the kurtosis $\gamma_2$ as

$$\gamma_2 = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^4}{s^4}. \tag{6.14}$$

Similarly like in the case of skewness, here are also three types of kurtosis. If $\gamma_2 = 3$ we speak about mesokurtic distribution. The value $\gamma_2$ is compared with 3, because the kurtosis of the normal distribution equals 3. So we compare the kurtosis of the sample with the Gaussian curve. In the case $\gamma_2 < 3$ is the distribution more flat than normal distribution and we speak about platykurtic distribution. In the opposite case, when $\gamma_2 > 3$, the analysed distribution has a greater peak in the mean than normal distribution. This kind of distributions is said to be leptokurtic.

To compute the skewness and kurtosis in R we need the `moments` package. In this package are defined the functions `skewness()` and `kurtosis()`. let us see the source code.

```
> library(moments)
> skewness(x)
[1] 0.3598295
> kurtosis(x)
[1] 2.252963
>
```

## 6.2 Parameter estimates

One of the goals of the statistical analysis is to estimate the parameters of the original distribution from which the random selection comes. We distinguish two types of estimates:

- **the point estimates** that provide the estimate of the exact value of the parameter,
- **confidence intervals** that give the intervals, that contain the real value of the parameter wit given probability (reported as the confidence level).

### 6.2.1 Point estimates

To estimate the value of the parameter we aim to choose the sample characteristic that approximate the parameter $\Theta$ with the best quality. The quality of the estimator is guaranteed by its properties:

- **Unbiased estimate** of the parameter $\theta$ is such estimate $T$ that equality $\mathbb{E}(T) = \theta$ holds.
- **Consistent estimate** can be characterised by increasing accuracy with increasing sample size. Formally we can write $\lim_{n\to\infty} T_n = \Theta$, where indices represent the size of the sample used in estimation.
- **Efficient estimate** can be interpreted as the best possible estimate. The notion best possible relies on the loss measured by the mean squared error of $T$. It is the value $MSE(T) = \mathbb{E}\left((T - \Theta)^2\right)$. The efficient estimate minimizes this value.

These properties influence implementation of some sample characteristics in the R language. Let us look on the sample mean. Having the random sample $X_1, \dots, X_n$ from the distribution with the mean $\mu$, we can calculate:

$$\mathbb{E}(\overline{x}) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}(X_i) = \frac{1}{n}n\mu = \mu.$$

So we have shown that the average is the unbiased estimate of the mean. Now let us look on the sample variance. Using the identity

$$\sum_{i=1}^{n}(X_i - \overline{X})^2 = \sum_{i=1}^{n}(X_i - \mu + \mu - \overline{X})^2 = \sum_{i=1}^{n}(X_i - \mu)^2 - n(\overline{X} - \mu)^2,$$

we get

$$\begin{aligned}\mathbb{E}\left(\sum_{i=1}^{n}(X_i - \overline{X})^2\right) &= \mathbb{E}\left(\textstyle\sum_{i=1}^{n}(X_i - \mu)^2 - n(\overline{X} - \mu)^2\right)\\ &\textstyle\sum_{i=1}^{n}\mathbb{D}\left(\overline{X_i}\right) - n\mathbb{D}\left(\overline{X}\right)\\ &n\sigma^2 - \frac{n\sigma^2}{n} = (n-1)\sigma^2,\end{aligned}$$

and therefore

$$\mathbb{E}\left(s^2\right) = \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2\right) = \frac{n-1}{n}\sigma^2.$$

So we see the sample variance is not unbiased estimate of the random variable variance. In order to get the unbiased estimate we have to multiply the sample variance by $\frac{n}{n-1}$. So we obtain the unbiased estimate of the variance

$$s_{n-1}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2.$$

The index $n-1$ is used to emphasize using the coefficient $1/(n-1)$ instead of $1/n$. This result explains, why the function `var()` is implemented differently from sample variance. It represents the unbiased estimate of the original random variable.

In this text we introduce two methods of the point estimates constructing:

- the method of moments,
- the method of maximal likelihood.

We will suppose, we have the sample $X_1, \dots, X_n$ from the distribution that depends on the vector of parameters $\theta = (\theta_1, \dots, \theta_m)$.

## The method of moments

Let us further suppose that there exist all moments $\nu_k = \mathbb{E}\left(X_i^k\right)$, $k = 1, \dots, m$. The sample moments $\nu_k$ are defined as $\nu_k = \frac{1}{n}\sum_{i=1}^{n} X_i^k$ for $k = 1, \dots, m$. The principle of the method of moments is the equality of the theoretical and sample moments. It means the method of moments estimator for $\theta_1, \theta_2, \dots, \theta_k$ denoted by $\hat{\theta}_1, \hat{\theta}_2, \dots, \hat{\theta}_k$ is defined as the solution (if there is one) of the equations:

$$\nu_k = \nu_k, \quad k = 1, \dots, m.$$

Alternatively we can use the $r$-th central moment defined as $\mu_r = \mathbb{E}\left((X - \mathbb{E}(X))^r\right)$ and $r$-th sample central moment $m_r = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^r$.

We illustrate the method on the case of the uniform distribution wit parameters $a$ and $b$. It is known, that the moments of the uniformly distributed random variable $X$ are

$$\mathbb{E}(X) = \frac{a+b}{2} \quad \text{and} \quad \mathbb{D}(X) = \frac{(b-a)^2}{12}.$$

To find the estimates of the parameters $a$ and $b$ we have to solve the equations:

$$\begin{aligned} \overline{x} &= \tfrac{a+b}{2}, \\ s^2 &= \tfrac{(b-a)^2}{12}. \end{aligned}$$

Solving these equations we obtain:

$$\begin{aligned} a &= = \overline{x} - \sqrt{3}s, \\ b &= = \overline{x} + \sqrt{3}s. \end{aligned}$$

Now we are ready to implement this estimates in R language. We take advance the previously defined functions `variation()`resp. `stdev()`. And here is the source code.

```
> x<-runif(1000,1,3) #generating the random sample
> a<-mean(x)-sqrt(3)*stdev(x)
> b<-mean(x)+sqrt(3)*stdev(x)
> a
[1] 1.018323 #can differ for other samples
> b
[1] 3.033395 #can differ for other samples
```

## The maximum likelihood method

This method is based on maximizing a likelihood function so that, under the assumed statistical model, the observed data is the most likely. The point in the parameter space that maximizes the likelihood function is said to be the maximum likelihood estimate. Formally, let us assume that $X_1, \dots, X_n$ are independent identically distributed random variables from the distribution with density $f(x, \theta)$. The joined density is the product of these univariate density functions:

$$L(x_1, \dots, x_n; \theta) = \prod_{i=1}^{n} f(x, \theta).$$

Just introduced function $L(x_1, \dots, x_n; \theta)$ is then called the likelihood function. To get the estimates of the parameters, we maximize this function by standard process known from the mathematical analysis. To make the work easier, we maximize the function $\ln L(x_1, \dots, x_n; \theta)$ instead of direct maximization of $L(x_1, \dots, x_n; \theta)$. The natural logarithm is increasing function, therefore it save the extremes and moreover convert the product to the sum of functions.

We illustrate the method on the problem of estimating the probability $p$ of some random event $A$. We can interpret this situation as an results of the alternative random variable that is indicator of the random event $A$. Therefore we have $\mathbb{P}(X = 1) = p$ and $\mathbb{P}(X = 0) = 1 - p$. We perform $n$ random trials and observe the occurrence of event $A$. So we get a sample $X_1, \dots, X_n$ of random variables with the densities $f(x_i, p) = p^{x_i}(1-p)^{1-x_i}$, where $x_i \in \{0, 1\}$. The corresponding likelihood function has the form

$$L(x, p) = \prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i} = p^{\sum_{i=1}^{n} x_i}(1-p)^{n-\sum_{i=1}^{n} x_i}.$$

In order to state the estimate of $p$ we will maximize the function $L(x, p)$ with respect to the parameter $p$. To do so, we take natural logarithm of the likelihood function

$$\ln(L(x, p)) = \sum_{i=1}^{n} x_i \ln p + \left(n - \sum_{i=1}^{n} x_i\right) \ln(1-p),$$

and we set its derivative with respect to $p$ equal to 0:

$$\frac{\mathrm{d} \ln L(x, p)}{\mathrm{d}x} = \frac{\sum_{i=1}^{n} x_i}{p} - \frac{n - \sum_{i=1}^{n} x_i}{1-p} = 0.$$

Multiplying the equation by $\frac{1}{n}$ we have

$$\overline{x} \cdot \frac{1}{p} - (1 - \overline{x}) \cdot \frac{1}{1-p} = 0,$$

and its solution is

$$p = \overline{x}.$$

To confirm that $p = \overline{x}$ really maximizes the likelihood function, we have to verify the second order condition. For the second derivative of the likelihood function logarithm we have

$$\frac{\mathrm{d}^2 \ \ln L(x, p)}{\mathrm{d}x^2} = -\frac{\overline{x}}{p} + (1 - \overline{x}) \cdot \frac{1}{(1-p)^2}.$$

Substituting $p = \overline{x}$ we get

$$\left(\frac{\mathrm{d}^2 \ \ln L(x, p)}{\mathrm{d}x^2}\right)_{p=\overline{x}} = -\frac{1}{1 - \overline{x}} < 0.$$

So we have the most likely estimate $p = \overline{x}$. We can use this approach to determine just how biased an unfair coin is. At first we generate the sample of 0 and 1 which indicates tossing head or tail. To get sample of unfair coin, we declare the vector of probabilities `prob`, how we can see in the source code:

```
> x<-sample(c(0,1),1000,replace=TRUE,prob=c(2/3,1/3))
> mean(x)
[1] 0.304
```

### 6.2.2 Confidence intervals

The confidence interval can be defined as the range of estimates for an unknown parameter, that contains the real value of the parameter with given probability. This probability that the parameter is within the given interval is reported as the confidence level. The most common confidence level used in practice is 95%, but other levels (such as 90% or 99%) are also frequently used.

Formally, let $\mathbf{X} = (X_1, \dots, X_n)$ is a random sample from distribution that depends on the unknown parameter $\theta$. A confidence interval for the parameter $\theta$, with confidence level $\alpha$, is an interval with random endpoints $(u(\mathbf{X}); v(\mathbf{X}))$, determined by the pair of random variables $u(\mathbf{X})$ and $v(\mathbf{X})$, with the property:

$$\mathbb{P}\left(u(\mathbf{X}) < \theta < v(\mathbf{X})\right) = \alpha.$$

The derivation of the confidence interval we can illustrate on the normal distribution. Let as assume, at first, that $X_1, \dots, X_n$ is a random sample from the normal distribution $N(\mu, \sigma^2)$, whose standard deviation $\sigma$ is known. We want to find the confidence interval for the mean $\mu$. Because $\overline{X}$ has the normal distribution $N\left(\mu, \frac{\sigma^2}{n}\right)$, we have

$$\mathbb{P}\left(\left|\frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}}\right| < c\right) = 2\Phi(c) - 1, \text{ for all } c > 0,$$

what is equivalent to

$$\mathbb{P}\left(\overline{X} - c\frac{\sigma}{\sqrt{n}} < \mu < \overline{X} + c\frac{\sigma}{\sqrt{n}}\right) = 2\Phi(c) - 1, \text{ for all } c > 0.$$

From the last relation, it follows that the confidence interval for the mean with the confidence level $\alpha = 2\Phi(c) - 1$ has the form

$$\left(\overline{X} - c\frac{\sigma}{\sqrt{n}}; \overline{X} + c\frac{\sigma}{\sqrt{n}}\right).$$

If we take in account, that $c = \Phi^{-1}\left(\frac{\alpha+1}{2}\right)$, we can write the confidence interval in the form

$$\left(\overline{X} - \Phi^{-1}\left(\frac{\alpha+1}{2}\right)\frac{\sigma}{\sqrt{n}}; \overline{X} + \Phi^{-1}\left(\frac{\alpha+1}{2}\right)\frac{\sigma}{\sqrt{n}}\right).$$

Clearly, we use the quantile function `qnorm()` to find the bounds of the confidence interval. Here follows the complete process of stating the confidence level as the R source code:

```
x<-rnorm(100,1,2) # we generate some sample
> n<-length(x)
> sample.mean<-mean(x)
> sample.sd<-2 #  standard deviation is known
> alpha<-0.95 # seting the confidence level 95\%
> c<-qnorm((alpha+1)/2,0,1)
> margin<-c*sample.sd/sqrt(n)
> lower.bound<-sample.mean-margin
> upper.bound<-sample.mean+margin
> print(c(lower.bound,upper.bound))
[1] 0.8149884 1.5989740
```

Now let us see, how is the confidence interval changed, if the standard deviation is unknown. Then we have to use the unbiased estimate of the standard deviation $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\overline{X} - X_i)^2, s = \sqrt{s^2}$. Then the random variable

$$T = \frac{\overline{X} - \mu}{s}\sqrt{n},$$

follows the Student's $t$-distribution with $n-1$ degrees of freedom. The confidence interval is then given

$$\left(\overline{X} - c\frac{s}{\sqrt{n}}; \overline{X} + c\frac{s}{\sqrt{n}}\right).$$

Here $c$ is the corresponding quantile of the Student distribution, so we apply the `qt()` function in R. We modify the source code to the following form.

```
> n<-length(x) # we use previously generated sample
> sample.mean<-mean(x)
> sample.sd<-sd(x) # estimate instead of unknown standard deviation
> alpha<-0.95
> c<-qt((alpha+1)/2,df=n-1)
> margin<-c*sample.sd/sqrt(n)
> lower.bound<-sample.mean-margin
> upper.bound<-sample.mean+margin
> print(c(lower.bound,upper.bound))
[1] 0.8118763 1.6020861
```

Let us now look at the confidence interval for the variance of the normal distribution. If $s^2$ is the unbiased estimate of the variance, then the random variable

$$Y = \frac{(n-1)s^2}{\sigma^2} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{\sigma^2} = \sum_{i=1}^{n}\left(\frac{x_i - \overline{x}}{\sigma}\right)^2$$

follows the $\chi^2(n-1)$ distribution. For the confidence interval we get

$$\mathbb{P}(c_1 < \chi^2 < c_2) = \alpha$$
$$\mathbb{P}\left(c_1 < \frac{(n-1)s^2}{\sigma^2} < c_2\right) = \alpha$$
$$\mathbb{P}\left(\sigma \in \left(\frac{(n-1)s^2}{c_2}; \frac{(n-1)s^2}{c_1}\right)\right) = \alpha,$$

where $c_1$ and $c_2$ are the critical values of the $\chi^2(n-1)$ distribution. When computing, we have to respect that $\chi^2$ distribution is not symmetric, what is important for stating the critical values $c_1$ and $c_2$. It is easily visible from the source code.

```
> n<-length(x) # the sample already generated sooner
> sample.var<-var(x)
> c1<-qchisq(1-(alpha+1)/2,df=n-1) # consequent of asymmetry
> c2<-qchisq((alpha+1)/2,df=n-1) # consequent of asymmetry
> lower.bound<-sample.var*(n-1)/c2
> upper.bound<-sample.var*(n-1)/c1
> print(c(lower.bound,upper.bound))
[1] 3.056626 5.350767
```

As the last example, we will show the confidence interval for the probability of a random event. The unbiased estimate of the probability $p$ of a random event occurring is $\hat{p} = \frac{m}{n}$, where $m$ is the number of occurrences of the observed event in a series of $n$ random trials. We can also interpret this value as the proportion $m$ of elements with the observed property in a random sample of size $n$. We then speak about the so-called population proportion. We know, that $\mathbb{E}(\hat{p}) = p$ and $\mathbb{D}(\hat{p}) = \frac{pq}{n}$, where $q = 1-p$. The approximative equality $\frac{pq}{n} \approx \frac{\hat{p}\hat{q}}{n}$ holds for the large $n$. Therefore the random variable

$$Z = \frac{\frac{m}{n} - p}{\sqrt{\frac{\hat{p}\hat{q}}{n}}}$$

follows the standardized normal distribution $N(0,1)$. Then we can write the confidence interval for the probability $p$ with confidence level $\alpha$ as

$$\left(\hat{p} - \Phi^{-1}\left(\frac{\alpha+1}{2}\right)\cdot\sqrt{\frac{\hat{p}\hat{q}}{n}}; \hat{p} + \Phi^{-1}\left(\frac{\alpha+1}{2}\right)\cdot\sqrt{\frac{\hat{p}\hat{q}}{n}}\right).$$

**Example 6.2.1** *Suppose 250 randomly selected people are surveyed to determine if they own a tablet. Of the 250 surveyed, 98 reported owning a tablet. Using a 95% confidence level, compute a confidence interval estimate for the true proportion of people who own tablets.*

**Solution**: As the first step we calculate the unbiased point estimate of the probability $p$ as $\hat{p} = 98/250$ and further we define $\hat{q} = 1 - \hat{p}$. Now we can calculate the bounds of the confidence interval with use of the `qnorm()` function. Let us see the source code:

```
> n<-250
> p<-98/n
> q<-1-p
> c<-qnorm((1+alpha)/2,0,1)
> lower.bound<-p-c*sqrt(p*q/n)
> upper.bound<-p+c*sqrt(p*q/n)
> print(c(lower.bound,upper.bound))
[1] 0.3314836 0.4525164
```

So we obtained the 95% confidence interval (0.3315; 0.4525) for the proportion of people owning the tablet.

# Hypotheses testing

By hypothesis we understand an educated guess about something in the world around us. It should be testable, either by experiment or observation. A statistical hypothesis is then any numerical assumption about the parameters of one or more basic ensembles or about the type of probability distribution in the basic population. However, this assumption is based on other sources of information, not on the basis of random selection. Hypothesis testing is then understood as testing the validity of a statistical hypothesis using the knowledge obtained by statistical investigation in a sample. Hence, we specify a decision rule by which we decide the validity or invalidity of the hypothesis.

We categorize two basic types of tests:

- **parametric tests** involving unknown parameters, they are based on known distributions.
- **non-parametric tests** concerning in general properties of the underlying population, no knowledge of the distribution in the underlying population is required.

The first step in hypothesis testing is to formulate a statistical hypothesis, which is the research question within the experiment. We formulate it in the form of a **zero hypothesis** $H_0$, which expresses a statement about the zero difference between the tested data sets, and the **alternative hypothesis** $H_1$, which denies the validity of $H_0$.The alternative hypothesis can be either **two sided** e.g.

$$H_0 : \mu_1 = \mu_2 \text{ versus } H_1 : \mu_1 \neq \mu_2,$$

or **one sided** e.g.

$$H_0 : \mu_1 = \mu_2 \text{ versus } H_1 : \mu_1 > \mu_2 \text{ resp. } H_1 : \mu_1 < \mu_2.$$

The error $\alpha$ chosen by the experimentalist that determines the probability of rejecting the null hypothesis despite it being true is called the **significance level of the test**. The most frequented values are $\alpha = 0.05$ and $\alpha = 0.01$.

The decision to reject the null hypothesis may be burdened with error due to the random nature of the selection. We distinguish between two types of error.

1. A **type I error** is the mistaken rejection of an actually true null hypothesis (also known as a “false positive” finding or conclusion). The decision to reject the null hypothesis depends on the size of the significance level $\alpha$. At a higher value, the probability of a type 1 error increases. The significance level is also the probability of a type I error.
2. A **type II error** is the mistaken non-rejection of an actually false null hypothesis (also known as a “false negative” finding or conclusion). The rate of the type II error is denoted by$\beta$ and related to the power of a test, which equals $1 - \beta$.

Table 7.1: Test error classification

| Decision<br>Hypothesis | REJECT<br>$H_0$ | DO NOT REJECT<br>$H_0$ |
|---|---|---|
| $H_0$ holds | Type I error<br>$\alpha$ | True<br>$1-\alpha$ |
| $H_0$ does not hold | True<br>$1-\beta$ | Type II error<br>$\beta$ |

The types of errors are surveyed in table 7.1.

The next important step in hypothesis testing is the selection of an appropriate test criterion as the random variable $Q = f(X, \Theta)$. Using the data in the sample and the assumed value of $\Theta_0$, we calculate the specific value of the test criterion $q = f(x, \Theta_0$. Now we determine the critical values $c_1$ and $qc2$, that correspond to the quantiles

$$F(c_1) = \frac{\alpha}{2}, \quad F(c_2) = 1 - \frac{\alpha}{2},$$

where $F()$ is the distribution function of the random variable $Q$.These values determine the so-called critical area of the test, which is the interval $(c_1; c_2)$ (by other words, the critical area is the confidence interval for the values of the testing variable $Q$). We reject the null hypothesis, if the value $q$ lies outside the critical area.

The $p$-value is also an important characteristic of statistical tests. The $p$-value is used as an alternative to rejection points to provide the smallest level of significance at which the null hypothesis would be rejected. A very small $p$-value means that such an extreme observed outcome would be very unlikely under the null hypothesis. Reporting $p$-values of statistical tests is common practice in academic publications of many quantitative fields. Formally, let us consider an observed test-statistic $q$ from unknown distribution $Q$. Then the $p$-value is:

- $p = \mathbb{P}(Q \geq q|H_0)$ for a one-sided right-tail test,
- $p = \mathbb{P}(Q \leq q|H_0)$ for a one-sided left-tail test,
- $p = 2\min\{\mathbb{P}(Q \geq q|H_0), \mathbb{P}(Q \leq q|H_0)\}$ for a two-sided test. If distribution $Q$ is symmetric about zero, then $p = \mathbb{P}\left(|Q| \geq |q| \middle| H_0\right)$.

## 7.1 Parametric tests

Parametric tests are those that make assumptions about the parameters of the population distribution from which the sample is drawn. This is often the assumption that the population data are normally distributed. here are more types of the parametric tests:

- The one-sample test is a statistical hypothesis test used to determine whether an unknown population parameter is different from a specific value.
- The two-sample test is a test performed on the data of two random samples, each independently obtained from a different given population. The purpose of the test is to determine whether the difference between these two populations is statistically significant.

- The paired test compares the parameters of two measurements taken from the same individual, object, or related units. While two-sample test is used when the data of two samples are statistically independent, the paired test is used when data is in the form of matched pairs.

### 7.1.1 One-sample tests

A one sample test compares the value of any parameter of distribution stated for sample to a pre-specified value and tests for a deviation from that value. For example we might know that the average birth weight for babies in some country is 3,410 grams and wish to compare the average birth weight of a sample of babies in another country to this value.

#### One sample $t$-test

Let $X_1, \dots, X_n$ is a random sample from the normal distribution $N(\mu, \sigma^2)$, where $n \geq 2$. We will deal with the test of the hypothesis $H_0 : \mu = \mu_0$ with alternative $H_1 : \mu \neq \mu_0$. In such situation we speak about two side alternative. Alternatively we can state the alternative hypothesis as $H_1 : \mu < \mu_0$ or $H_1 : \mu > \mu_0$ and speak about the one side alternative. The corresponding testing statistics has the form:

$$T = \frac{\overline{X} - \mu}{s}\sqrt{n}$$

and it follows the $t$-distribution with $n - 1$ degrees of freedom.

**Example 7.1.1** *The machine fills boxes with detergent. Each box should contain 5 kg of powder. We selected 10 boxes randomly from the production and their contents were accurately weighed. We found the following deviations from the required weight (in dag):*

$$-5,\ 4,\ -1,\ -8,\ 7,\ -6,\ 4,\ -3,\ 2,\ -2$$

*We have to verify that there is no systematic deviation of the machine settings.*

**Solution**: The data will be treated as a selection from the normal distribution $N(\mu, \sigma^2)$. If there is no system error, the hypothesis $H_0 : \mu = 0$ holds, while the deviation is evidenced by the fact that $H_1 : \mu \neq 0$. If we compute manually, we have

```
> T<-(mean(x)-0)/sd(x)*sqrt(length(x))
> T
[1] -0.7436885
> qt(0.975,length(x)-1)
[1] 2.262157
```

We compare the value of the statistics $T$ with the critical value of the $t$-distribution. Because $|T| < t_{0.975}(9)$, we ca not reject the hypothesis, that the automatic machine is correctly set. We can also use the implemented function `t.test()` that provides complete information about the test and confidence intervals.

```
> t.test(x,mu=0,alternative="two.sided",conf.level=0.95)

        One Sample t-test

```

```
data:  x
t = -0.74369, df = 9, p-value = 0.476
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -4.445988  2.245988
sample estimates:
mean of x
     -1.1
>
```

As the output of the function `t.test()` we get the value of the testing statistics, number of degree of freedom, the $p$-value, the 95% confidence interval $(-4.446; 2.246)$ for the $\mu$ parameter and the mean of the sample.

The size of our sample is small, $n = 10$, so we should test the normality at first. We apply Shapiro-Wilk test which examines if a variable is normally distributed in some population. In the R environment there is implemented the function `shapiro.test()`. Using it, we have

```
> shapiro.test(x)

        Shapiro-Wilk normality test

data:  x
W = 0.98509, p-value = 0.9866
```

From the output, the p-value is greater than the significance level 0.05 implying that the distribution of the data are not significantly different from normal distribution. In other words, we can assume the normality.

### Wilcoxon test

Let us suppose, that $X_1, \dots, X_n$ is a random sample from the continuous distribution with the density $f$ that is symmetric according to the point $a$. This kind of symmetry means that equality $f(x+a) = f(x-a)$ holds. Therefore $a$ equals to the median $\tilde{x}$. If there exists finite mean value of the distribution, it is also $\mathbb{E}(X_i) = a$ for each $i = 1, \dots, n$. However, the finiteness of the mean value is not generally assumed. The Wilcoxon test is designed to test the hypothesis $H_0 : \tilde{x} = x_0$ with alternative $H_1 : \tilde{x} \neq x_0$.

We assume that none of the values in the sample equals to $x_0$. If some of the values $X_i$ equals to $x_0$, it is usually omitted from the sample. The test is then based on ordering of the variables $Y_i = X_i - x_0$ into the sequence, non-decreasing according to their absolute values

$$|Y_{(1)}| \leq |Y_{(2)}| \leq \cdots \leq |Y_{(n)}|.$$

Let us assign as $R_i^+$ the order of the variable $|Y_i|$ in the sequence above. Further we define

$$S^+ = \sum_{Y_i \geq 0} R_i^+, \quad S^- = \sum_{Y_i < 0} R_i^+.$$

If the value of $\min\{S^+, S^-\}$ less than the critical value $w(\alpha)$[1] we reject the hypothesis.

The complete procedure is quite laborious, therefore we rather use the implemented function `wilcox.test()`. let us see an example.

[1]The values are tabled in the statistical tables.

**Example** 7.1.2 *Fifteen subjects were asked to estimate independently, without prior practice, when one minute would elapse from a given signal.The following results were obtained (in seconds)*

$$54, 49, 46, 56, 64, 52, 67, 57, 51, 59, 58, 62, 63, 57, 52.$$

*We want to test the hypothesis that in the human population, half of the people underestimate the length of one minute while the second half overestimate it.*

**Solution**: In the R environment we use the function `wilcox.test()`. According to the conditions, we use the parameters `mu=60` and `conf.level=0.95`. So we obtain

```
> x<-c(54, 49, 46, 56, 64,52,67,57,51,59,58,62,63,57,52)
> wilcox.test(x,mu=60,conf.level=0.95)

        Wilcoxon signed rank test with continuity correction

data:  x
V = 25, p-value = 0.04973
alternative hypothesis: true location is not equal to 60
```

Due to the $p$-value less than 0.05, we can reject the hypothesis with the significance level 95%. As a part of the answer of the function we also receive the warning message `cannot compute exact p-value with ties` what is caused by small sample size.

## Test for the variance of the normal distribution

Let $X_1, \ldots, X_n$ is a random sample from the normal distribution $N(\mu, \sigma^2)$ and we suppose that both parameters of the distribution are unknown. We will use the unbiased estimate of $\sigma^2$

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left(X_i - \overline{X}\right)^2 = \frac{1}{n-1} \left( \sum_{i=1}^{n} X_i^2 + n\overline{X} \right).$$

Our aim is to test the hypothesis $H_0 : \sigma^2 = \sigma_0^2$ with two-side alternative $H_1 : \sigma^2 \neq \sigma_0^2$ or one-side alternative $H_1 : \sigma^2 > \sigma_0^2$ resp. $H_1 : \sigma^2 < \sigma_0^2$. The testing statistics has the form

$$W = \frac{(n-1)s^2}{\sigma_0^2},$$

and it follows the $\chi^2$ distribution with $n-1$ degrees of freedom if the mean is known and $n-2$ if the mean has to be estimated from the sample. As the $\chi^2$ distribution is not symmetric, we must compare the value of the statistics $W$ with two critical values $\chi_{n-1}\left(\frac{\alpha}{2}\right)$ and $\chi_{n-1}\left(1-\frac{\alpha}{2}\right)$. In the case of the one-side alternatives the corresponding confidence intervals the critical values are $\chi^2_{n-1}(\alpha)$ for the alternative $\sigma > \sigma_0^2$ and $\chi^2_{n-1}(1-\alpha)$ for the alternative $\sigma < \sigma_0^2$.

The `varTest()` function to perform a one-sample test for variance is part of the `EnvStats` package. It is not to be confused with the `var.test()` function, which is used for tests comparing the variances of two random samples, or the `chi.test()` function, which is used for goodness-of-fit tests.

**Example** 7.1.3 *In the cement factory, they fill cement into bags weighing 50 kg. The filling equipment is set up so that the amount of cement in the bags follows the distribution*

$N(50, 0.15^2)$. *In order to verify the assumption* $\sigma = 0.15$, *the control weighed the contents of 10 bags and found the following values:*

$$50.12, 49.81, 50.00, 50.18, 49.95, 50.03, 49.98, 50.10, 50.14$$

*We have to test the hypothesis* $H_0 : \sigma^2 = 0.15^2$ *on the significance level 95%.*

**Solution**: As the first step we have to load the `EnvStats` package. Then we can apply the `varTest()` function with argument `sigma.squared=0.0225`. So we have

```
> library("EnvStats")
> x<-c(50.12, 49.81,50.00,50.18,49.95,50.03,49.98,50.10,50.14)
> varTest(x,alternative="two.sided",conf.level=0.95,
  sigma.squared=0.15^2)

Results of Hypothesis Test
--------------------------
Null Hypothesis:                 variance = 0.0225
Alternative Hypothesis:          True variance is not equal to 0.0225
Test Name:                       Chi-Squared Test on Variance
Estimated Parameter(s):          variance = 0.01320278
Data:                            x
Test Statistic:                  Chi-Squared = 4.694321
Test Statistic Parameter:        df = 8
P-value:                         0.4206164
95\% Confidence Interval:        LCL = 0.006023664
                                 UCL = 0.048456546
```

So we can not reject the hypothesis, that the filling machine works correctly. Let us note, the number of degrees of freedom `df=8` that corresponds to the fact, that mean was estimated from the sample. Last two values give us the information about the 95% confidence interval for the variance $(0.006; 0.0485)$. From the sense of the assignment, we would obviously be interested in the one-sided alternative, which variance does not exceed the allowed limit. In this case, we are testing a one-sided alternative and thus must specify the parameter `alternative="greater"`.

### Significance test for a population proportion

Now, let us suppose we need to verify if probability of some event is equal to $p_0$. This probability can be also interpreted, as the proportion of the elements in some population, which have some property. To estimate $\hat{p}$ the probability $p_0$, we can proceed by conducting $n$ independent random trials and observing the occurrence of the event. We can also describe this procedure by selecting a sample of size $n$ from the population under study and determining the proportion $\hat{p}$ of elements with the observed probability in that sample. Thus, in both cases, it is in the final a hypothesis test for the $p$ parameter of the binomial distribution. This test is used we have a simple random sample where each observation can result in just two possible outcomes, a success and a failure.

Formally, let $Y \sim Bi(n, p)$ and we test the hypothesis $H_0 : p = p_0$ against the alternative $H_1 : p \neq p_0$. For sufficiently large values of $n$, we use an approximation of the binomial distribution by the normal distribution, and the test statistic then takes the form

$$U = \frac{Y - np_0}{\sqrt{np_0(1 - p_0)}}, \tag{7.1}$$

and it asymptotically follows the standardized normal distribution. We reject the hypothesis $H_0$ if its value exceeds the critical value of the standardized normal distribution. i.e if $|U| \geq u_{\frac{\alpha}{2}}$.

To perform this test in R environment, we use the function `prop.test(sample)`. Its full syntax takes the form
`prop.test (x, n, p = 0.5, alternative = "two.sided", conf.level = alpha).`
Here `x` is number of positive results (successes), `n` is the number of trials, `p` the hypothesized probability of success, parameter `alternative` we can set also to `less` or `greater` if we perform the one-sided test. The last parameter `conf.level` we set to be equal to the requested confidence level.

**Example 7.1.4** *Suppose we toss a coin 100 times and get 52 heads. verify if the coin is fair, means if probability of tossing the head* $p = 0.5$.

**Solution**: We can use `prop.test` to assess whether or not the coin is fair. To do so we enter the following values: `x=52`, `n=100`, `p=0.5`, `alternative=two.sided` and `conf.level=0.95`. here is the source code and answer

```
> prop.test(x=52,n=100,p=0.5,alternative="two.sided",
  conf.level=0.95)

        1-sample proportions test with continuity correction

data:  52 out of 100, null probability 0.5
X-squared = 0.09, df = 1, p-value = 0.7642
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.4183183 0.6201278
sample estimates:
   p
0.52
```

Since the p-value is greater than 0.05, we cannot reject the null hypothesis that the population proportion is 0.5. Therefore we can consider the coin to be fair.

### 7.1.2 Two-sample tests

A two-sample test is a test performed on the data of two random samples, each independently obtained from a different given population. The purpose of the test is to determine whether the difference between these two populations is statistically significant.

#### Two sample *t*-test

Let $X_1, \ldots, X_m$ is a random sample from the normal distribution $N(\mu_1, \sigma^2)$ and let $Y_1, \ldots, Y_n$ is a random sample from the normal distribution $N(\mu_2, \sigma^2)$, and let $m \geq 2$, $n \geq 2$ and $\sigma^2 > 0$. We further suppose, that both samples are independent. Let us assign

$$\begin{aligned} \overline{X} &= \tfrac{1}{m} \textstyle\sum_{i=1}^{m} X_i & \qquad \overline{Y} &= \tfrac{1}{n} \textstyle\sum_{i=1}^{n} Y_i \\ S_X^2 &= \tfrac{1}{m-1} \textstyle\sum_{i=1}^{m} (X_i - \overline{X})^2 & \qquad S_Y^2 &= \tfrac{1}{n-1} \textstyle\sum_{i=1}^{n} (Y_i - \overline{Y})^2. \end{aligned}$$

A two-sample test refers to the hypothesis $H_0 : \mu_1 - \mu_2 = \delta$, where $\delta$ is a given number (most frequently we set $\delta = 0$) against the alternative .$H_1 : \mu_1 - \mu_2 \neq \delta$. The testing statistics has the form

$$T = \frac{\overline{X} - \overline{Y} - \delta}{\sqrt{(m-1)S_X^2 + (n-1)S_Y^2}} \sqrt{\frac{mn(m+n-2)}{m+n}}, \tag{7.2}$$

that follows the Student's distribution $t(m+n-2)$ with $m+n-2$ degrees of freedom. The hypothesis is rejected if $|T| \geq t_{m+n-2}(\alpha)$.

To perform the test in the R environment, we use the function `t.test()` . Its full syntax is

```
t.test(x,y,mu,alternative,var.equal = FALSE, conf.level = alpha),
```

where xand `y` are samples, `mu` is the difference of the means, `alternative` determines, as usually, if we perform one=sided or two=sided test, `conf.level` is set to the required confidence level $\alpha$. The parameter `var.equal` is logical value indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise the Welch (or Satterthwaite) approximation to the degrees of freedom is used.

**Example** 7.1.5 *Suppose we have measured the height of 10 men and 10 women, with results summarized in the table:*

| | **Height (in cm)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Men** | *187* | *185* | *180* | *192* | *178* | *185* | *183* | *176* | *181* | *190* |
| **Women** | *175* | *168* | *170* | *174* | *180* | *176* | *168* | *163* | *182* | *171* |

*Let us test, if the average height of men is greater than average height of women.*

**Solution**: At first, we must prepare the data sets `men_height` and `women_height`. Then we can apply the `t.test()` function, how we can see in the next source code.

```
> women_height<-c(175,168,170,174,180,176,168,163,182,171)
> men_height <- c(187,185,180,192,178,185,183,176,181,190)
> t.test(women_height, men_height, conf.level=0.95)

        Welch Two Sample t-test

data:  women_height and men_height
t = -4.4816, df = 17.705, p-value = 0.0002994
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -16.162827  -5.837173
sample estimates:
mean of x mean of y
    172.7     183.7
```

We see, that the p-value is less than 0.05, so we can reject the hypothesis, that the average height is the same for men and women. Moreover, we see from the answer, that the function `t.test()` performed automatically the Welch modification of the test. It is because we have not declared equal variances of both samples. We can compare it with the next source code, where we set `var.equal=TRUE`. In such situation the function performs directly the $t$-test.

```
>  t.test(women_height, men_height, conf.level=0.95,var.equal=TRUE)

        Two Sample t-test

data:  women_height and men_height
t = -4.4816, df = 18, p-value = 0.0002885
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -16.156659  -5.843341
sample estimates:
mean of x mean of y
    172.7     183.7
```

Finally, we can see from the answer, that average heights are 172.2 cm for women and 183.7 cm for men. So we can illustrate also test on the difference of the means. To do so, we have to set the value of the mu variable of the `t.test()` function.

```
>  t.test(women_height, men_height, mu=-10,conf.level=0.95,
  var.equal=TRUE)

        Two Sample t-test

data:  women_height and men_height
t = -0.40742, df = 18, p-value = 0.6885
alternative hypothesis:true difference in means is not equal to -10
95 percent confidence interval:
 -16.156659  -5.843341
sample estimates:
mean of x mean of y
    172.7     183.7
```

Since the p-value is greater than 0.05, we cannot reject the hypothesis that the difference of the average heights of men and women is 10 cm. Let us note, that we had to set `mu=-10` since we entered as the first variable `women_height` and the average height of women is less than average height of men.

### F-test of equality of variances

This test is used to test the null hypothesis that two independent samples have the same variance. we will assume that $X_1, \dots, X_m$ and $Y_1, \dots, Y_n$ are two independent samples from the normal distributions $N(\mu_1, \sigma_1^2)$ and $N(\mu_2, \sigma_2^2)$ respectively. Further we assume $m \geq 2, n \geq 2$, $\sigma_1^2 > 0$ and $\sigma_2^2 > 0$. If the hypothesis $H_0 : \sigma_1^2 = \sigma_2^2$ holds, the random variable $Z = \frac{S_X^2}{S_Y^2}$ follows the Fisher's F-distribution $F(m-1, n-1)$ with $m-1$ and $n-1$ degrees of freedom. As usual, we reject the hypothesis $H_0 : \sigma_1^2 = \sigma_2^2$ if the value of the random variable $F$ fell outside the interval with the endpoints $F_{m-1,n-1}\left(1-\frac{\alpha}{2}\right)$ and $F_{m-1,n-1}\left(\frac{\alpha}{2}\right)$. In the R environment we perform this test using the function `var.text()`. Its full syntax is
`var.test(x, y, ratio = 1, alternative = "two.sided",conf.level = 0.95)`, where x and y are the samples, `ratio` is the tested ratio between the samples ( if we test equality of variances, we set `ratio = 1`, which is also implicit value), `alternative` and `conf.level` have the usual meaning.

**Example** 7.1.6 *Let us return to the example 7.1.5. Before we perform the t-test, we should verify, if the variances of both samples are same. Therefore, we test the assumption that the variances of the measured heights of women and men are equal.*

**Solution**: We have already insert the values of the samples. Su we can just call the `var.test()` function, how we can see in the source code.

```
> var.test(women_height,men_height,ratio=1,conf.level=0.95)

        F test to compare two variances

data:  women_height and men_height
F = 1.2965, num df = 9, denom df = 9, p-value = 0.7052
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.3220284 5.2196392
sample estimates:
ratio of variances
          1.296485
```

Due to the p-value greater than 0.05, we can not reject the hypothesis, that both samples have the same variance. So, we could consequently set in the $t$-test the variable `var.equal` to be TRUE.

## 7.1.3 Homogeneity test of two binomial distributions

Now, we turn our attention towards testing whether one population proportion $p_1$ equals a second population proportion $p_2$. It means, we will test the hypothesis $H_0 : p_1 = p_2$ against the alternative $H_1 : p_1 \neq p_2$.

These proportions can also be interpreted as the occurrence of a random event $A$ in two observations. Suppose event $A$ occurred in $m$ independent observations $X$ times and in a second series of $n$ independent observations $Y$ times. The values $p_1$ and $p_2$ are the probabilities of occurrence of event $A$ in these two series of trials. For the distribution of the random variables $X$ and $Y$ clearly holds $X \sim \text{Bin}\left((,m), p_1\right)$ and $Y \sim \text{Bin}\left((,n), p_2\right)$. Therefore, the hypothesis $H_0$ is sometimes called *the hypothesis of homogeneity of two binomial distributions.*

Let us assign $x = \frac{X}{m}$ and $y = \frac{Y}{n}$, the estimates of the proportions (or probabilities) $p_1$ and $p_2$. As the testing statistics we define the random variable

$$U = \frac{x - y}{\sqrt{\frac{x(1-x)}{m} + \frac{y(1-y)}{n}}}. \tag{7.3}$$

If the hypothesis $H_0$ holds, then random variable $U$ follows the standardized normal distribution $N(0, 1)$. Therefore, we reject the hypothesis if $|U| \geq u\left(\frac{\alpha}{2}\right)$.

In the R environment we perform this test using the function `prop.test()`, whose arguments x and n are entered as vectors. Other arguments are same as in the one-sample case.

**Example** 7.1.7 *Let us say we have two groups of student A and B. Group A with an early morning class of 200 students with 142 female students. Group B with a late class of 200 students with 90 female students. Use a 95% confidence level. We want to know, whether the proportions of females are the same in the two groups of the student?*

**Solution**: We use the `prop.test()` function and we set `x=c(142,90)` and `n=c(200,200)`. The result we obtain after running the following source code:

```
> prop.test(x = c(142, 90), n = c(200, 200))

2-sample test for equality of proportions with continuity correction

data:  c(142, 90) out of c(200, 200)
X-squared = 26.693, df = 1, p-value = 2.384e-07
alternative hypothesis: two.sided
95 percent confidence interval:
 0.1616802 0.3583198
sample estimates:
prop 1 prop 2
  0.71   0.45
```

We see the p-value is less then 0.05, so we can reject the hypothesis and say that there is significant difference between these two proportions.

### 7.1.4 Paired tests

Paired samples typically consist of a sample of matched pairs of similar units, or one group of units that has been tested twice. In practice, we frequently meet such situations where the values of two variables that are related to each other (e.g. visual acuity of the left and right eye, blood clotting before and after administration of a certain drug) are measured for each object. It means, we have sample of pairs random variables $(X_1, Y_1), \dots, (X_n, Y_n)$. These $n$ pairs can be considered mutually independent because they refer to different objects. In contrast, the quantities $X_i$ and $Y_i$ with the same index can no longer be considered independent because they are values measured at the same object.

Let us assume, that $(X_1, Y_1), \dots, (X_n, Y_n)$ is a sample from some two-dimensional distribution. If this distribution has finite first moments, we can ask if their means are equal. So, we want to test the hypothesis $H_0 : \mathbb{E}(X_i) - \mathbb{E}(Y_i) = \mu_0$ against the alternative $H_1 : \mathbb{E}(X_i) - \mathbb{E}(Y_i) \neq \mu_0$, where $\mu_0$ is given constant. We construct new random variables

$$Z_1 = X_1 - Y_1, \dots, Z_n = X_n - Y_n.$$

Due to the assumptions, the random variables $Z_1, \dots, Z-n$ are independent and identically distributed with the mean $\mathbb{E}(Z_i) = \mathbb{E}(X_i) - \mathbb{E}(Y_i)$. If it can be further assumed that the $Z_i$ variables follow a normal distribution, we can use one-sample $t$-test, introduced in the subsection 7.1.1.

To perform the paired test in the R environment we use the function `t.test()`, where we set the optional parameter `paired=TRUE`. We illustrate its use in the example.

**Example 7.1.8** *When testing a new treatment, the weight of 10 mice was measured before and after applying the tested therapeutic preparation. The results are summarized in the table:*

| | **Weight (in g)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Before** | *210.3* | *205.1* | *208.6* | *200.1* | *241.3* | *185.8* | *243.2* | *221.5* | *196.3* | *230.5* |
| **After** | *315.2* | *308.4* | *310.5* | *304.5* | *350.8* | *295.2* | *350.4* | *341.7* | *308.7* | *343.3* |

*Let us verify, if there is a significant difference in the weight of mice before and after treatment.*

**Solution**: Because our measurements record the weights of the same individual, we use a paired test. When performing the computation in the R environment, we first initialize the `before` and `after` vectors. We then use the Shapiro-Wilk test to make sure that the differences of the two masses are not significantly different from a normal distribution. See the source code:

```
> before<-c(210.3,205.1,208.6,200.1,241.3,185.8,243.2,221.5,
  196.3,230.5)
> after<-c(315.2,308.4,310.5,304.5,350.8,295.2,350.4,341.7,
  308.7,344.3)
> d<-before-after
> shapiro.test(d)

        Shapiro-Wilk normality test

data:  d
W = 0.93906, p-value = 0.5426
```

From the output, the p-value is greater than 0.05 implying that the distribution of the differences (d) are not significantly different from normal distribution. In other words, we can assume the normality. Now we are ready to perform the paired *t*-test:

```
> t.test(before, after, paired = TRUE)

        Paired t-test

data:  before and after
t = -61.228, df = 9, p-value = 4.169e-13
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -112.7161 -104.6839
sample estimates:
mean of the differences
                 -108.7
```

## 7.2 Non-parametric tests

In statistics, non-parametric tests are methods of statistical analysis that do not require a distribution to meet the required assumptions to be analysed (especially if the data is not normally distributed). Due to this reason, they are sometimes referred to as distribution-free tests. Non-parametric tests serve as an alternative to parametric tests such as *t*-test. Let us note, that non-parametric tests are only alternative but no substitute of the parametric tests. If the data meets the required assumptions for performing the parametric tests, the relevant parametric test must be applied.

In order to obtain reliable results from statistical analysis, it is necessary to be able to correctly identify whether to use a parametric or non-parametric test. Here are more reasons, why to use non-parametric tests:

1. Data do not meet the assumptions about the population sample. Application of the parametric tests requires various assumptions to be satisfied. For example the data follows the normal distribution.

2. The sample size is too small. In this case, it is possible that we may not be able to verify the distribution of the data. It means, the use of non-parametric tests is the only suitable option.
3. The analysed data is ordinal or nominal. Unlike parametric tests that can work only with continuous data, non-parametric tests are the only solution to other data types such as ordinal or nominal data.

### 7.2.1 Sign test

In particular, we use the sign test when the distribution of the variables $X_i$ is highly skewed. Because the probability of an error of the second kind is relatively high compared to other tests, it is desirable to have a larger number of observations available.

Let $X_1, \dots, X_n$ is a sample from the continuous distribution with median $\tilde{x}$. we test the null hypothesis $H_0 : \tilde{x} = x_0$, where $x_0$ is a given number. At first, we have to compute the differences $X_1 - x_0, \dots, X_n - x_0$. Let us denote as $Y$ the number of the positive differences. If the null hypothesis $H_0$ holds, the random variable $Y$ follows the binomial distribution Bin $\left(n, \frac{1}{2}\right)$. We reject the hypothesis $H_0$ if the value of $Y$ is close to zero or $n$.

The sign test is a special case of the binomial test where the probability of success under the null hypothesis is $p = 0.5$. Therefore, in the R environment we use the `binom.test()` function to perform the sign test. We illustrate its use in example.

**Example 7.2.1** *20 subjects were asked to estimate, without prior practice, the time when 1 minute had elapsed since a given signal. The following values (in seconds) were obtained:*

*53,48,45,55,63,51,66,56,50,58,62,49,52,65,47,57,54,67,42,55*

*We want to decide whether people are overestimating or underestimating their time interval estimates.*

**Solution**: If the median were 60, we would get the difference values $Y_i = X_i - 60$. We first determine how many of these values are negative (we only verify if $X > 60$) and then test whether this random variable follows a binomial distribution. The full procedure is shown in the source code.

```
> x<-c(53,48,45,55,63,51,66,56,50,58,62,49,52,65,47,57,54,67,42,55)
> d<-sum(x>60)
> binom.test(d,20)

        Exact binomial test

data:  d and 20
number of successes = 5, number of trials = 20, p-value = 0.04139
alternative hypothesis:true probability of success
  is not equal to 0.5
95 percent confidence interval:
 0.08657147 0.49104587
sample estimates:
probability of success
                  0.25
```

The p-value is less than 0.05, so we can reject the hypothesis that median of the estimates is 60. We see, that most of the people has tendency underestimate the length of the time interval.

### 7.2.2 Wilcoxon signed rank test

The one-sample Wilcoxon signed rank test is a non-parametric alternative to one-sample t-test when the data cannot be assumed to be normally distributed. It is used to determine whether the median of the sample is equal to a known standard value (i.e. theoretical value).

Let $X_1, \dots, X_n$ is a sample from the continuous distribution, whose density is symetric around the point $a$, i.e. $f(a + x) = f(a - x)$. In this case the value $a$ must be equal to the median $\tilde{x}$. The null hypothesis is again $H_0 : \tilde{x} = x_0$ against the alternative $H_1 : \tilde{x} \neq x_0$ Similarly like in the sign test, we define the random variables $Y_i = X_i - x_0$ . The variables $Y_i$ are then sorted in non-decreasing sequence according their absolute values

$$|Y|_{(1)} \leq |Y|_{(2)} \leq \cdots \leq |Y|_{(n)}.$$

Let us further denote as $R_i^+$ the order of the value $|Y_i|$ in the sequence and we define

$$S^+ = \sum_{Y_i \geq 0} R_i^+, \quad S^- = \sum_{Y_i < 0} R_i^+.$$

We reject the hypothesis, if the value of $\min(S^+, S^-)$ is less or equal tu the critical value $w(\alpha)$ of the test.

In the R environment is implemented the `wilcox.test()` function. It takes the form:

```
wilcox.test(x,mu=0,alternative="two sided")
```

where `x` is the sample, `mu` the hypothesised value (default 0), and `alternative` can be ”two sided”,”grater” or ”less”.

let us return to the previous example concerned in the time interval estimation. We can use the Wilcoxon test, and we get the following results:

```
> x<-c(53,48,45,55,63,51,66,56,50,58,62,49,52,65,47,57,54,67,42,55)
> wilcox.test(x,mu=60)

        Wilcoxon signed rank test with continuity correction

data:  x
V = 33, p-value = 0.007559
alternative hypothesis: true location is not equal to 60
```

Due to the very small p-value this Wilcoxon test also rejects the hypothesis, that median equals 60.

The Wilcoxon test can be applied also in its paired form to test if the difference between two means is significantly different from zero. Using the Wilcoxon Signed-Rank Test, we can decide whether the corresponding data population distributions are identical without assuming them to follow the normal distribution. In the R environment we use the `wilcox.test()` function with option `paired=TRUE`.

We will illustrate it on the data form example 7.1.8. Without verifying the normality of the weight distribution, we can use the Wilcoxon paired test, how shows the source code:

```
> before <-c(210.3,205.1,208.6,200.1,241.3,185.8,243.2,221.5,
196.3,230.5)
> after <-c(315.2,308.4,310.5,304.5,350.8,295.2,350.4,341.7,
308.7,344.3)
> wilcox.test(before,after,paired=TRUE)

        Wilcoxon signed rank test

data:  before and after
V = 0, p-value = 0.001953
alternative hypothesis: true location shift is not equal to 0
```

Similarly like in the example 7.1.8, the Wilcoxon test also rejects the hypothesis, that the treatment does not influence the weight of the mice.

### 7.2.3 Mann-Whitney-Wilcoxon test

The Mann-Whitney-Wilcoxon test is used to test whether two samples are likely to derive from the same population (i.e., that the two populations have the same shape). So, it is a non-parametric test of the null hypothesis that, for randomly selected values X and Y from two populations, the probability of X being greater than Y is equal to the probability of Y being greater than X. The general assumptions of the test are as follows:

1. All observations from both groups are independent of each other.
2. The responses are at least ordinal (i.e., one can at least say, of any two observations, which is the greater).
3. Under the null hypothesis $H_0$, the distributions of both populations are equal.
4. The alternative hypothesis H1 is that the distributions are not equal.

Let us assume, that $X_1, \dots, X_n$ and $Y_1, \dots, Y_m$ are independent identically distributed samples from $X$ and $Y$ respectively. The corresponding Mann-Whitney-Wilcoxon U statistic is defined as:

$$U = \sum_{i=1}^{n} \sum_{j=1}^{m} S(X_i, Y_j),$$

where

$$S(X_i, Y_j) = \begin{cases} 1 & \text{if } X_i > Y_j, \\ 1/2 & \text{if } X_i = Y_j, \\ 0 & \text{if } X_i < Y_j. \end{cases}$$

In order to perform the Mann-Whitney-Wilcoxon test in the R environment, we use the `wilcox.test()` function. However, we need to sort the data into a data.frame structure, in which there will be one component that will act as an identifier of membership in one of the two groups being compared. We illustrate it in example.

**Example** 7.2.2 *Let us consider the clinical trial designed to investigate the effectiveness of a new drug to reduce symptoms of asthma in children. A total of $n = 10$ participants are randomized to receive either the new drug or a placebo. Participants record the number of episodes of shortness of breath over a 1 week period following receipt of the assigned treatment. The data are summarized in the table.*

| **Placebo** | 7 | 5 | 6 | 4 | 12 |
|---|---|---|---|---|---|
| **New drug** | 3 | 5 | 4 | 2 | 1 |

*Is there a statistically significant difference in number of the episodes over 1 week?*

**Solution**. Here is a small sample size, that means the non-parametric test is a suitable method. At first we create the vector x that contains all episode numbers and vector y that contains only values 0 or 1, that indicates placebo or new drug application. We join these vectors to build up the data frame. Then we are ready to perform the test, how it is shown in the source code.

```
>x<-c(7,5,6,4,12,3,5,4,2,1)
> y<-c(0,0,0,0,0,1,1,1,1,1)
> treat<-data.frame(x,y)
> wilcox.test(x~y,data=treat)

        Wilcoxon rank sum test with continuity correction

data:  x by y
W = 23, p-value = 0.03558
alternative hypothesis: true location shift is not equal to 0
```

The p-value demonstrates, that we can confirm the decrease of the episodes after application of the new drug on the confidence level exceeding 95%.

### 7.2.4 Kruskal-Wallis test

The Kruskal-Wallis test is used in a situation where we have three or more independent samples on and we want to test for differences among the sample means. Sample observations in each group are assumed to come from populations with the same shape of distribution. It is mainly used for selections that are considerably different from normal.

Suppose we have $I$ random selections, each of size $n_i$. The null hypothesis has the form

$$H_0 : \mu_1 = \cdots = \mu_I.$$

The method of the Kruskal-Wallis test we can describe in a few steps:

1. Let $Y_{i1}, \dots, Y_{in_i}$, $i = 1, \dots, I$ is a sample from continuous distribution and all samples are independent. All values $Y_{ij}$ form a joint random sample with size $n = n_1 + \cdots + n_I$. Rank all data from all groups together; i.e., rank the data from 1 to n ignoring group membership. We denote as $R_{ij}$ the rank of the value $Y_{ij}$ in the joint sample.
2. Let us denote the sum of all ranks in group s $T_i$, $i = 1, \dots, I$. Clearly $T_1 + \cdots + T_I = \frac{n(n+1)}{2}$. The testing statistics is

$$Q = \frac{12}{n(n+1)} \sum_{i=1}^{I} \frac{T_i^2}{n_i} - 3(n+1).$$

   The random variable $Q$ is approximately $\chi^2$-distributed with $I - 1$ degrees of freedom.

3. Finally, the decision to reject or not the null hypothesis is made by comparing $Q$ to a critical value $Q_\alpha$ obtained from a table or a software for a given significance or $\alpha$ level. If $Q$ is greater than $Q_\alpha$, the null hypothesis is rejected.

The test can be performed in the R environment using the function `kruskal.test()`. It can be used in two versions:

`kruskal.test(x,g)`

where x is a numeric vector of data values, or a list of numeric data vectors, and g a vector or factor object giving the group for the corresponding elements of `x`. The other version is:

`kruskal.test(formula, data, subset,na.action)`

where `formula` is a formula of the form `response~group` where response gives the data values and group a vector or factor of the corresponding groups, `data` an optional matrix or data frame containing the variables in the `formula`, `subset` is an optional vector specifying a subset of observations to be used and `na.action` is a function which indicates what should happen when the data contain `NA`s.

**Example** 7.2.3 *Let us suppose, we have the following data on the scoring percentage of basketball players by position*

| **Position** | **Percentage** | | | | | |
|---|---|---|---|---|---|---|
| **Center** | *69.7* | *64.2* | *65.3* | *67.5* | *62.1* | |
| **Forward** | *58.3* | *60.2* | *57.2* | *54.8* | *61.3* | *60.5* |
| **Guard** | *52.1* | *53.2* | *58.6* | *51.2* | *52.8* | |

*We want to know if there is any significant difference between the average average scoring percentages on 3 different positions.*

**Solution:** We will illustrate both versions of using the `kruskal.test()` function. For the first version we must enter the numeric vectors of the percentages before we call the function. This procedure is illustrated in the source code.

```
> x<-c(69.7,64.2,65.3,67.5,62.1)
> y<-c(58.3,60.2,57.2,54.8,61.3,60.5)
> z<-c(52.1,53.2,58.6,51.2,52.8)
> kruskal.test(list(x, y, z))

        Kruskal-Wallis rank sum test

data:  list(x, y, z)
Kruskal-Wallis chi-squared = 12.035, df = 2, p-value = 0.002435
```

Equivalently we can use the group variable, and proceed with the following source code:

```
> g <- factor(rep(1:3, c(5, 6, 5)),
+               labels = c("Center",
+                          "Forward",
+                          "Guard"))
> x <- c(x, y, z)
> kruskal.test(x, g)

        Kruskal-Wallis rank sum test

data:  x and g
Kruskal-Wallis chi-squared = 12.035, df = 2, p-value = 0.002435
```

Let us note, we had to join all three vectors in one, to have the same length as the vector of the factors g.

To illustrate the second version, with use of `formula`, we must prepare the data frame, that is composed from the vector x and groups g. We can see it in the source code

```
> data<-data.frame(g,x)
> kruskal.test(x~g,data=data)

        Kruskal-Wallis rank sum test

data:  x by g
Kruskal-Wallis chi-squared = 12.035, df = 2, p-value = 0.002435
```

How we can see, all alternatives give the same p-value that is less than 0.05. So we can reject the hypothesis, that players on all positions have the same scoring percentage.

## 7.3 Goodness of fit tests

In the data science, occasionally, we receive a dataset and we would like to know what is the generative distribution for that dataset. Basically, the process of finding the right distribution for a set of data can be broken down into four steps:

1. visualization, plot the histogram of data,
2. guess what distribution would fit to the data the best,
3. use some statistical test for goodness of fit,
4. repeat 2 and 3 if measure of goodness is not satisfactory.

The first task is relatively simple. In R, we can use `hist()` to plot the histogram of a vector of data. This function was introduced in the section 5.3. The second task is a little bit tricky. It is mainly based on our experience and your knowledge of statistical distribution. Fortunately, how we can see on figure 5.40, we can plot the histogram together with the supposed density. This make guessing the real distribution easier.

There are three well-known and widely use goodness of fit tests:

1. Chi Square test
2. Kolmogorov–Smirnov test
3. Cramér–von Mises criterion

All of the above tests are for statistical null hypothesis testing. For goodness of fit we have the following hypothesis:

$H_0$: The data is consistent with a specified reference distribution.

$H_1$: The data is NOT consistent with a specified reference distribution

### 7.3.1 Kolmogorov-Smirnov test

The Kolmogorov-Smirnov test is popular test, probably due to its extreme simplicity. It is used for asking two possible questions:

1. Are two sample distributions the same, or are they significantly different?
2. Does a particular sample distribution arise from a particular hypothesized distribution?

The testing statistic is based on the difference between the empirical and theoretical distribution functions or in the two sample case, on the difference between the corresponding empirical distribution functions. So, in the one sample case, the Kolmogorov-Smirnov statistic has form

$$D_n = \sup_x |F_n(x) - F(x)|,$$

where $F_n(x)$ is the empirical distribution function. In the case of two samples with sizes $n$ resp. $m$, the Kolmogorov-Smirnov statistic takes the form

$$D_{n,m} = \sup_x |F_n(x) - F_m(x)|,$$

where $F_n(x)$ and $F_m(x)$ are empirical distribution functions.

To perform a one-sample or two-sample Kolmogorov-Smirnov test in R environment we can use the `ks.test()` function. We illustrate its use in the following source code. As the first step, we generate the numeric vector x as an sample from the gamma distribution with the sample size $n = 1000$. In the second step we add some normal distributed noise to the data. Then we conduct the Kolmogorov-Smirnov test to verify, if the random variable follow normal distribution.

```
> num_of_samples = 1000
> x <- rgamma(num_of_samples, shape = 10, scale = 3)
> x <- x + rnorm(length(x), mean=0, sd = .1)
> ks.test(x,"pnorm")

        One-sample Kolmogorov-Smirnov test

data:  x
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

The resulting p-value is much less than 0.05, so we reject the hypothesis of a normal distribution.

To illustrate the two sample version, we generate second sample y from the gamma distribution. Then we again apply the `ks.test()` function.

```
> y<-rgamma(num_of_samples,shape=5,scale=1)
> ks.test(x,y)
        Two-sample Kolmogorov-Smirnov test

data:  x and y
D = 0.992, p-value < 2.2e-16
alternative hypothesis: two-sided
```

The resulting p-value is again extremely small, so we can reject the hypothesis, that samples x and y come the same distribution from.

### 7.3.2 Chi-square goodness of fit test

The chi square test for goodness of fit is a non-parametric test to test whether the observed values that falls into two or more categories follows a particular distribution of not. We can say that it compares the observed proportions with the expected chances.

For the chi-square goodness-of-fit computation, the data are divided into $n$ bins and the test statistic is defined as

$$\chi^2 = \sum_{i=1}^{n} \frac{(X_i - E_i)^2}{E_i},$$

where $X_i$ denotes number of observations for bin $i$ and $E_i$ an expected count for bin $i$, asserted by the null hypothesis. The expected frequency $E_i$ is calculated by:

$$E_i = (F(Y_u) - F(Y_l))N$$

where $F$ is the cumulative distribution function for the probability distribution being tested, $Y_u$ and $Y_l$ are the upper limit and the lower limit for class $i$, and $N$ is the sample size.

If the null hypothesis holds, then the resulting statistic follows the chi-square distribution $\chi^2(n - c)$ with $n - c$ degrees of freedom, where $n$ is the number of bins and $c$ is the number of estimated parameters (including location and scale parameters and shape parameters) for the distribution plus one.

In the R environment, The Chi-Square Goodness of Fit Test can then be performed using the `chisq.test()` function, which has the following syntax.

```
chisq.test(x, p)
```

where x are the observed frequencies, represented numerically as a vector and p a numerical vector of proportions to be expected.

**Example** 7.3.1 *We expect, that every day, an equal number of clients enter a business, according to a vendor. To test this theory, we record the number of customers who visit the shop in a given week and discovers the following.*

*Monday: 260 customers, Tuesday: 245 customers, Wednesday: 270 customers, Thursday: 250 customers, and Friday: 230 customers.*
*Let us verify the assumption, that customers come into the shop uniformly during the week.*

**Solution**: First, we will prepare two arrays to store our observed frequencies and expected customer proportions for each day. Because we expect uniformly distributed data, the vector p will contain five equal values that must add up to 1. Then we can apply the `chisq.test()` function. Let us see the source code:

```
> visitors<-c(260,245,270,250,230)
> expect<-c(0.2,0.2,0.2,0.2,0.2)
> chisq.test(x=visitors,p=expect)

        Chi-squared test for given probabilities

data:  visitors
X-squared = 3.6653, df = 4, p-value = 0.4532
```

The p-value is large, so we can not reject the null hypothesis.

In the next example, we illustrate the use of the chi-square test to verify the Poisson distribution.

**Example** 7.3.2 *The insurance portfolio includes 1,000 car accident insurance policies. Insured drivers reported from 0 up to 4 claims during the year as shown in the table below:*

| *Number of claims* | *0* | *1* | *2* | *3* | *4* |
|---|---|---|---|---|---|
| *Number of reports* | *384* | *380* | *170* | *54* | *23* |

*Let us verify, if the number of reported claims follow Poisson distribution.*

**Solution**: At first we must enter the numeric vector of claims. To find the vector of poissonian probabilities, we must estimate the value of the $\lambda$ parameter. We find it as a mean of the sample. As the next step, we have to generate the vector of probabilities of the Poisson distribution. The sum of all values in the probability vector must be equal to 1, so we determine the last entry as the complementary value of the distribution function. It means, that we interpret the last group as more than 3 reported claims. Then we can use the `chisq.test()` function, how illustrates the source code.

```
> x<-c(384,380,170,54,23)
> lambda<-(0*x[1]+x[2]+2*x[3]+3*x[4]+4*x[5])/sum(x)
> p<-c(dpois(0,lambda),dpois(1,lambda),dpois(2,lambda),
dpois(3,lambda),1-ppois(3,lambda))
> chisq.test(x=x,p=p)
        Chi-squared test for given probabilities

data:  x
X-squared = 2.9718, df = 4, p-value = 0.5626
```

The p-value is much greater than 0.05, so we cannot reject the null-hypothesis, that number of reported claims follows the Poisson distribution.

Chi-square test of goodness-of-fit is not very suitable for verifying continuous distributions. Nevertheless, we illustrate its use for a test of the normality of a distribution. Let us note that the presented approach is applicable as well for other kinds of continuous distributions.

Let us suppose that $X_1, \ldots, X_n$ is a random sample. We want to test the hypothesis $H_0$, this sample comes from the normal distribution $N(\mu, \sigma^2)$ with unknown parameters $\mu$ and $\sigma^2$. First we create class intervals

$$J_1 = (-\infty; b_1)\,, J_2 = \langle b_1; b_2), J_3 = \langle b_2; b_3), \ldots, J_{k-1} = \langle b_{k-2}; b_{k-1}), J_k = \langle b_{k-1}; \infty),$$

where $k \geq 4$. The probability $p - i$ that the random variable $X_i$ falls in the interval $J_i$ is then determined by the values of the distribution function

$$p_i = F(b_i) - F(b_{i-i}) \text{ for } i = 2, \ldots, k-1$$

and $p_1 = F(b_1)$ resp. $p_k = 1 - F(b_{k-1})$. The test statistic then takes the form

$$\chi^2 = \sum_{i=1}^{k} \frac{(X_i - np_i)^2}{np_i}.$$

Since we have assumed that the parameters of the normal distribution are unknown, it is necessary to determine their estimates. Therefore, the test statistic has a distribution $\chi^2(n-3)$.

To perform this test in the R environment, we apply the `gofTest()` function from the `EnvStats` package. This function has several arguments that allow specification of several alternatives to the goodness-of-fit test and also provides a wider range of distributions that can be verified. Among the arguments of the `gofTest()` function, here are at least a few of the most common ones:

- `test` is a character string defining which goodness-of-fit test to perform. If no value is specified, the default is `"sw"` Shapiro-Wilk test. Other possibilities are `"chisq"` for Chi-squared test, `"ks"` for Kolmogorov-Smirnov test, `cmv` for Cramer-von Mises test, and some others (complete list we get using the `help(gofTest)`.
- `distribution` is a character string denoting the distribution abbreviation. The default value is `"norm"` representing the normal distribution. From the other distributions let us mention at least `"exp"` for exponential distribution, `"gamma"` for the Gamma distribution, `logis` for the logistic distribution, `"unif"` for the uniform distribution, and `"lnorm"` for the logarithmic-normal distribution. The complete list we can find in the help file of the `Distribution.df`.
- `n.classes` for the case when `test="chisq"`, the number of class intervals into which the observations are to be allocated.
- `cut.points` for the case when `test="chisq"`, a vector of cutpoints that defines the class intervals.
- `est.arg.list` is a list of arguments to be passed to the function estimating the distribution parameters. For example, to use the most likely estimates we have to set the value of the argument `est.arg.list=list(method="mle")`.
- complete specification of all arguments one can find using the `help(gofTest)`.

To illustrate how does the function `gofTest()` work, we generate the random sample from the normal distribution $N(1, 10)$. The sample obtained is entered into the vector `x`. Now we are ready tu run the `gofTest()` function. The whole process we see in the following source code.

```
> library("EnvStats")
> x<-rnorm(1000,1,10)
> gofTest(x,distribution="norm",test="chisq",
  est.arg.list=list(method="mle"))
Results of Goodness-of-Fit Test
-------------------------------
Test Method:                     Chi-square GOF
Hypothesized Distribution:       Normal
Estimated Parameter(s):          mean = 1.026369
                                 sd   = 9.965374
Estimation Method:               mle/mme
Data:                            x
Sample Size:                     1000
Test Statistic:                  Chi-square = 29.44
Test Statistic Parameter:        df = 29
P-value:                         0.4423341
Alternative Hypothesis:          True cdf does not equal the
                                 Normal Distribution.
```

### 7.3.3 Cramér – von Mises criterion

The Cramér – von Mises criterion is a criterion used to verify the goodness of fit of a cumulative distribution function $F$ compared to a given empirical distribution function $F_n$, or for comparing two empirical distributions.

Let $X_1, \dots, X_n$ is a random sample from the distribution with cumulative distribution function $F$. The Cramér – von Mises criterion works with sample $X_{(1)} \le X_{(2)} \le \dots \le X_{(n)}$ which is oredered increasingly. The test statistic is defined as

$$T = \frac{1}{12n} + \sum_{i=1}^{n} \left[ \frac{2i-1}{2n} - F(X_{(i)}) \right]^2 .$$

If this value is larger than the tabulated value, then the hypothesis that the data came from the assumed distribution can be rejected. Because we are working in the R environment, we can rely on the implementation of critical values in this environment and we do not need to work with tables.

In the R environment, we perform the Cramér – von Mises test again using the `gofTest()` function from the `Envstats` package. In this case, we need to specify the argument `test="cvm "`. The following source code illustrates the usage. First, we generate a vector `x`, which represents a random selection from a normal distribution. We then perform a test. The high p-value does not allow us to reject the null hypothesis that the sample comes from a normal distribution.

```
> library("EnvStats")
> x<-rnorm(1000,1,10)
> gofTest(x,distribution="norm",test="cvm",
  est.arg.list=list(method="mle"))

Results of Goodness-of-Fit Test
-------------------------------
Test Method:                     Cramer-von Mises GOF
Hypothesized Distribution:       Normal
Estimated Parameter(s):          mean = 1.026369
                                 sd   = 9.960390
Estimation Method:               mle/mme
Data:                            x
Sample Size:                     1000
Test Statistic:                  W = 0.03370085
Test Statistic Parameter:        n = 1000
P-value:                         0.7923351
Alternative Hypothesis:          True cdf does not equal the
                                 Normal Distribution.
```

## 7.4 Chi-square test of independence

The Chi-square test of independence is a statistical hypothesis test used to determine whether two categorical or nominal variables are likely to be related or not. As with all prior statistical tests we need to define null and alternative hypotheses. To verify independence, we set them as follows:

$H_0$ In the population, the two categorical variables are independent.

$H_1$ In the population, the two categorical variables are dependent.

We assume that we have the data arranged into a contingency table that has $r \times c$ cells. Let us further denote by $O_{ij}$ the number of empirical observations corresponding to the cell in the $i$-th row and $j$-th column. Next, let us denote by $n_{i.} = \sum_{k=1}^{c} O_{ik}$ the sum of the observations in the $i$-th row and $n_{.j} = \sum_{k=1}^{r} O_{ik}$ the sum of the observations in the $j$-th column. Obviously

$$n = \sum_{i=1}^{r} n_{i.} = \sum_{j=1}^{c} n_{.j} = \sum_{i=1}^{r} \sum_{j=1}^{c} O_{ij},$$

where $n$ is the sample size. We use the observed data to estimate the theoretical probabilities of occurrence of individual values of categorical variables. Using the previous label, we can estimate the probabilities $p_{i.} = \frac{n_{i.}}{n}$ and $p_{.j} = \frac{n_{.j}}{n}$. Assuming independence, we determine the probability of simultaneous occurrence of the values of each categorical variable as the product $p_{ij} = p_{i.}p_{.j}$. For the theoretical frequencies of occurrence of these characters, we then get

$$n_{ij} = np_{ij} = \frac{n_{i.}n.j}{n}.$$

The test statistic for the independence test is then based on the differences between the theoretical and empirical frequencies and takes the form

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - n_{ij})^2}{n_{ij}} = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{\left(O_{ij} - \frac{n_{i.}n_{.j}}{n}\right)^2}{\frac{n_{i.}n_{.j}}{n}}.$$

If the zero hypothesis holds, this statistic follows the chi-squared distribution $\chi^2((r-1)(c-1)$.

When working in the R environment, we can simply use the function `chisq.test()` to perform the independence test, where we specify the contingency table as a variable.We illustrate the procedure with the following example.

**Example** 7.4.1 *We have observed students' results on mathematics and statistics examinations. The frequencies of the resulting grades are summarized in the table:*

| *Statistics* | *Mathematics* | | | | | |
|---|---|---|---|---|---|---|
| | *A* | *B* | *C* | *D* | *E* | *FX* |
| *A* | *5* | *8* | *10* | *6* | *3* | *2* |
| *B* | *8* | *7* | *12* | *8* | *3* | *1* |
| *C* | *9* | *4* | *15* | *10* | *7* | *4* |
| *D* | *2* | *3* | *8* | *9* | *15* | *10* |
| *E* | *2* | *2* | *7* | *12* | *15* | *18* |
| *FX* | *1* | *3* | *8* | *9* | *17* | *20* |

*We want to test whether the resulting marks in mathematics and statistics are independent.*

**Solution**: At first we must enter the values in the contingency table structure. Therefore we enter the data as a matrix. Then we are ready to apply the `chisq.test()` function. here is the simple source code:

```
> data<-matrix(c(5,8,10,6,3,2,8,7,12,8,3,1,9,4,15,
   10,7,4,2,3,8,9,15,10,2,2,7,12,15,18,1,3,8,9,17,20),ncol=6)
> chisq.test(data)

        Pearson's␣Chi-squared␣test

data:␣␣data
X-squared␣=␣75.839,␣df␣=␣25,␣p-value␣=␣5.047e-07
```

We see very small p-value of the test, so we can reject the hypothesis, that resulting marks in mathematics and statistics are independent.

# Regression and linear models

## 8.1 Measures of the statistical dependence

The fundamental tool for measuring statistical dependence is the covariance or the correlation coefficient. Let us assume, that $(x_1, y_1), ..., (x_n, y_n)$ is a random sample from two-dimensional distribution. Let us denote

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

We define the covariance $\operatorname{cov}(X, Y)$ by formula

$$\operatorname{cov}(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}).$$

The calculated value of the covariance is hard to interpret because it is highly influenced by the values of the random variables themselves, and is not standardised in any way. Therefore, we move from covariance to standardised values – the correlation coefficients. Given a series of $n$ measurements of the pair $(X_i, Y_i)$ indexed by $i = 1, \ldots, n$, the sample correlation coefficient is defined as

$$r_{X,Y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{(n-1)s_X s_Y},$$

where $\overline{x}$ and $\overline{y}$ are the sample means of $X$ and $Y$, and $s_x$ and $s_y$ are the unbiased estimates of the standard deviations of $X$ and $Y$.

A non-parametric alternative to the correlation coefficient is Spearman's rank correlation coefficient. It is a measure of rank correlation (statistical dependence between the rankings of two variables). It assesses how well the relationship between two variables can be described using a monotonic function. The Spearman correlation coefficient is defined as the correlation coefficient between the rank variables.

For a sample of size $n$, the $n$ raw scores $x_i, y_i$ are converted to ranks $R(x_i), R(y_i)$, and $r_S$ is computed as

$$r_S = \frac{\operatorname{cov}(R(X), R(Y))}{\sigma_{R(X)}\sigma_{R(Y)}},$$

The Kendall rank correlation coefficient, commonly referred to as Kendall's $\tau$ coefficient, is a statistic used to measure the ordinal association between two measured quantities. Let $(x_1, y_1), \ldots, (x_n, y_n)$ be a set of observations of the joint random variables X and Y, such that all the values of $x_i$ and $y_i$ are unique (ties are neglected for simplicity).

Any pair of observations $(x_i, y_i)$ and $(x_j, y_j)$, where $i < j$, are said to be **concordant** if the sort order of $(x_i, x_j)$ and $(y_i, y_j)$ agrees: that is, if either both $x_i > x_j$ and $y_i > y_j$ holds or both $x_i < x_j$ and $y_i < y_j$; otherwise they are said to be **discordant**. The Kendall $\tau$ coefficient is defined as:

$$\tau = \frac{\text{number of concordant pairs} - \text{number of discordant pairs}}{\binom{n}{2}}.$$

An explicit expression for Kendall's rank coefficient is

$$\tau = \frac{2}{n(n-1)} \sum_{i<j} \operatorname{sgn}(x_i - x_j)\operatorname{sgn}(y_i - y_j).$$

Once we are working in the R environment, we can compute the covariance or correlation coefficient using the `cov()` resp. `cor()` function. We illustrate their use in the example with the build up data set `faithful`. It consists of a collection of observations of the Old Faithful geyser in the Yellowstone National Park. There are two observation variables in the data set. The first one, called eruptions, is the duration of the geyser eruptions. The second one, called waiting, is the length of waiting period until the next eruption. Here we can see the corresponding source code:

```
> cov(faithful$eruptions,faithful$waiting)
[1] 13.97781
> cor(faithful$eruptions,faithful$waiting)
[1] 0.9008112
```

At this point it is important to note that for the `cov()` and `cor()` functions to work correctly, it is necessary that both vectors have the same length. In addition, if either vector contains unknown values, the `use=pairwise` argument must be used. Otherwise, the result would also contain an unknown value. Let us see simple example:

```
> x <-c(1,2,3,4,5)
> y <-c(1,2 3)
> y[5] <-5 # to assure same length
> cov(x,y) # y[4] is unknown value NA
[1] NA
> cov(x,y,use="pairwise")
[1] 2.916667
```

Alternatively, we can use as the function argument only the data set name, and we obtain the result in the form of covariance resp. correlation matrix, how illustrate the following source code:

```
> cov(faithful)
           eruptions   waiting
eruptions   1.302728  13.97781
waiting    13.977808 184.82331
> cor(faithful)
           eruptions   waiting
eruptions 1.0000000 0.9008112
waiting   0.9008112 1.0000000
```

In order to get the Spearman's rank correlation coefficient or the Kendall $\tau$, we have to specify the argument `method` in the function `cov()`, resp. `cor`. We can see it in the next listing:

```
> cor(faithful,method="spearman")
          eruptions   waiting
eruptions 1.0000000 0.7779721
waiting   0.7779721 1.0000000
> cor(faithful,method="kendall")
          eruptions   waiting
eruptions 1.0000000 0.5747674
waiting   0.5747674 1.0000000
```

Let us also note something about the covariance matrix and correlation matrix. Covariance matrix is a square matrix that displays the variance exhibited by elements of data sets and the covariance between a pair of data sets. The diagonal elements represent the variance and the off-diagonal elements represent the covariance. If we examine dependencies between multiple random variables $X_1, \dots, X_n$, the covariance matrix has the following structure:

$$\begin{pmatrix} s^2_{X_1} & \mathrm{cov}(X_1, X_2) & \dots & \mathrm{cov}(X_1, X_n) \\ \mathrm{cov}(X_2, X_1) & s^2_{X_1} & \dots & \mathrm{cov}(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{cov}(X_n, X_1) & \mathrm{cov}(X_n, X_2) & \dots & s^2_{X_n} \end{pmatrix}$$

Similarly, the correlation matrix is a square matrix, whose elements on position $(i, j)$ are the correlations coefficients between the $i$-th and $j$-th random variables.Thus the diagonal entries are all identically unity. The structure of the correlation matrix looks like this:

$$\begin{pmatrix} 1 & r_{X_1 X_2} & \dots & r_{X_1, X_n} \\ r_{X_2, X_1} & 1 & \dots & r_{X_2, X_n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{X_n, X_1} & r_{X_n, X_2} & \dots & 1 \end{pmatrix}$$

We illustrate the calculation of the covariance and correlation matrices using the built-in data set `trees`. This data set provides measurements of the diameter, height and volume of timber in 31 felled black cherry trees. Note that the diameter (in inches) is erroneously labelled Girth in the data.

```
> cov(trees)
           Girth   Height    Volume
Girth   9.847914 10.38333  49.88812
Height 10.383333 40.60000  62.66000
Volume 49.888118 62.66000 270.20280
> cor(trees)
           Girth    Height    Volume
Girth  1.0000000 0.5192801 0.9671194
Height 0.5192801 1.0000000 0.5982497
Volume 0.9671194 0.5982497 1.0000000
```

Let us conclude this section by stating that all values that enter into the calculation of the covariance matrix must be numerical. If they were not, an error would occur. We can illustrate this with the built-in data set `iris`. This data set contains measurements on 4 different attributes (in centimetres) for 50 flowers from 3 different species. There is one categorical variable `Species`, so the `cov()` function produces an error, how we can see from the source code:

```
 > cov(iris)
Error: is.numeric(x) || is.logical(x) is not TRUE
```

To get the covariance matrix, we have to exclude the last variable from the computation. The source code then looks like this:

```
> cov(iris[-c(5)])
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    0.6856935  -0.0424340    1.2743154   0.5162707
Sepal.Width    -0.0424340   0.1899794   -0.3296564  -0.1216394
Petal.Length    1.2743154  -0.3296564    3.1162779   1.2956094
Petal.Width     0.5162707  -0.1216394    1.2956094   0.5810063
```

## 8.2 Correlation test

Using a random sample, we can determine the sample correlation, but not the correlation coefficient of the whole population. The sample correlation coefficient $r$ is our estimate of the unknown population correlation coefficient. The hypothesis test lets us decide whether the value of the population correlation coefficient $\rho$ is ”close to zero” or ”significantly different from zero”. The null hypothesis and alternative hypothesis then have the form:

$H_0$: $\rho = 0$.

$H_1$: $\rho \neq 0$.

The null hypothesis means in words that the correlation coefficient does not significantly differ from zero and there is no significant linear relationship between $X$ and $Y$ in the population. In contrast, the alternative hypothesis states the population correlation coefficient differs significantly from zero. There is a significant linear relationship between $X$ and $Y$ in the population.

Let $(x_1, y_1), \dots, (x_n, y_n)$ to be a random sample from the two dimensional normal distribution with positive variances and correlation coefficient $\rho \in (-1; 1)$. If the hypothesis $\rho = 0$ holds, then the random variable

$$T = \frac{r}{\sqrt{1 - r^2}}\sqrt{n - 2}$$

follows the Student’s distribution $t(n - 2)$.

In the R environment, the `cor.test()` function is implemented to test the significance of the correlation coefficient. It returns both the correlation coefficient and the significance level (or $p$-value) of the correlation. This function can be used for all three types of correlation coefficient determining the `method` argument. We illustrate its use in the following source codes. We use again the build up dataset `trees`.

```
> cor.test(trees$Girth,trees$Volume)


        Pearson's␣product-moment␣correlation

data:␣trees$Girth␣and␣trees$Volume
t␣=␣20.478,␣df␣=␣29,␣p-value␣<␣2.2e-16
alternative␣hypothesis:␣true␣correlation␣is␣not␣equal␣to␣0
95␣percent␣confidence␣interval:
```

```
␣0.9322519␣0.9841887
sample␣estimates:
␣␣␣␣␣␣␣cor
0.9671194
```

To perform the test for the Spearman's rank correlation we submit also the method:

```
> cor.test(trees$Girth,trees$Volume,method="spearman")

        Spearman's␣rank␣correlation␣rho

data:␣␣trees$Girth␣and␣trees$Volume
S␣=␣224.61,␣p-value␣<␣2.2e-16
alternative␣hypothesis:␣true␣rho␣is␣not␣equal␣to␣0
sample␣estimates:
␣␣␣␣␣␣␣rho
0.9547151
```

Similarly we can perform the test for the Kendall's $\tau$:

```
> cor.test(trees$Girth,trees$Volume,method="kendall")

        Kendall's␣rank␣correlation␣tau

data:␣␣trees$Girth␣and␣trees$Volume
z␣=␣6.5313,␣p-value␣=␣6.519e-11
alternative␣hypothesis:␣true␣tau␣is␣not␣equal␣to␣0
sample␣estimates:
␣␣␣␣␣␣␣tau
0.8302746
```

## 8.3 Linear regression

We can characterise the linear regression as a linear approach for modelling the relationship between a scalar response and one or more explanatory variables (also known as dependent and independent variables). The case of one explanatory variable is called **simple linear regression**; for more than one, the process is called **multiple linear regression**.

In the case of simple linear regression we work with regression model with a single explanatory variable. That is, it concerns two-dimensional sample points with one independent variable and one dependent variable. The model with general regression line then has the form

$$Y_i = \alpha + \beta x_i + \varepsilon_i, \quad i = 1, \dots, n.$$

Let as assign $\overline{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$ and $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. Using the least square method we obtain the estimates of the regression coefficients

$$b = \frac{\sum_{i=1}^{n} X - iY_i - n\overline{x}\overline{Y}}{\sum_{i=1}^{n} x_i^2 - n\overline{x}^2}, \quad a = \overline{Y} - b\overline{x}.$$

To state the estimates of the regression coefficients in the R environment we use the function `lm()` whose syntax is

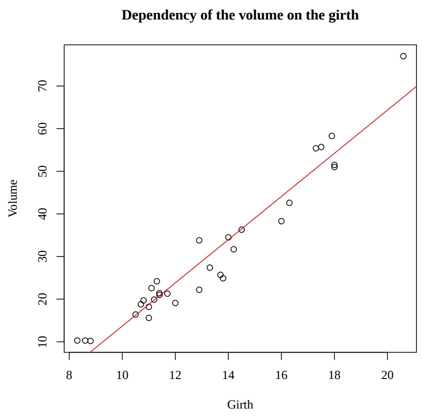


Figure 8.1: The scatter plot and linear regression line of the Girth and Volume variables from the dataset trees.

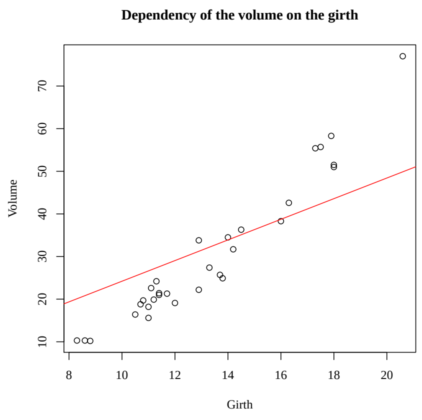


Figure 8.2: The scatter plot and "pure" linear regression line of the Girth and Volume variables from the dataset trees.

```
lm(resp~var1,dataset)
```

where

- `resp` is the response variable,
- `var1` is the explanatory variable,
- `dataset` is the name of the input data frame.

We can illustrate its use on the case of the dataset `trees` and find the linear model of the volume as variable dependent on the girth.

```
> lm(Volume~Girth,trees)

Call:
lm(formula = Volume ~ Girth, data = trees)

Coefficients:
(Intercept)        Girth
    -36.943        5.066
```

So, we see that the linear relation between the volume $V$ and girth $G$ has the form

$$V = -36.943 + 5.066G.$$

We can also illustrate the linear regression graphically. At first we prepare the scatter plot including the observed data. The regression line we then add to the graph using the function `abline()`. It has in our case the form.

```
abline(lm(Volume~Girth,trees),col="red")
```

The resulting graph we see in the figure 8.1. Let us note, that in case, when we want to model the "pure" linear dependency (means straight line passing through origin) $Y_i = \beta x_i$, we have to apply the `lm()` function with the syntax:

`lm(resp~-1+var1,dataset)`

Applying this approach on the `trees` dataset, we get:

```
> lm(Volume~-1+Girth,trees)

Call:
lm(formula = Volume ~ -1 + Girth, data = trees)

Coefficients:
Girth
2.421
```

That mean, the equation of the “pure” linear dependency among the `Girth` and volume is

$$V = 2.421G.$$

However, how we can see on the figure 8.2, this approximation is much worse then general linear relation. This example also shows that we need some indicators of the quality of the fitting by the prediction model. The commonly used indicators, implemented also in R, are:

- Coefficient of determination $R^2$. This coefficient states the proportion of the variation in the dependent variable that is explained by the model.
- Confidence test for the model coefficients. The null hypothesis states, that he coefficient equals zero. If we cannot reject this hypothesis, it is appropriate to exclude the variable from the model.
- F-test, that identifies if the model is significantly better prediction than the simple average.

Let us assume, a data set has $n$ values $x_1, \dots, x_n$, each associated with a fitted (or modelled, or predicted) value $f_1, \dots, f_n$. We define the residuals as $e_i = x_i - f_i$. The most general definition of the coefficient of determination is

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where

$$SS_{res} = \sum_{i=1}^{n} (x_i - f_i)^2 = \sum_{i=1}^{n} e_i^2,$$

is the so called residual sum of squares, and

$$SS_{tot} = \sum_{i=1}^{n} (x_i - \overline{x})^2,$$

is the total sum of squares.

In order to obtain information about the quality of fitting by the regression model in the R environment, it is necessary to store the output of the function `\lm` in some variable. Then the `summary()` function can be applied to this output. Thus, we obtain significant quantiles of the error distribution as well as the desired indicators of the fitting quality. We illustrate it on the previous dependency of `Volume` on the `Girth` of trees.

```
> model<-lm(Volume~Girth,trees)
> summary(model)

Call:
lm(formula = Volume ~ Girth, data = trees)

Residuals:
   Min     1Q Median     3Q    Max
-8.065 -3.107  0.152  3.495  9.587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
Girth         5.0659     0.2474   20.48  < 2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 4.252 on 29 degrees of freedom
Multiple R-squared:  0.9353,    Adjusted R-squared:  0.9331
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

Descriptive statistics of residual errors are given as the first result. The second part of the output gives the estimates of the coefficient of the regression line and the resulting values of the null hypothesis test. Here we observe very small $p$-values. This means that we can reject the null hypothesis at a high confidence level. Last are the fitting quality indicators. Here we see that the value of the residual error is 4.252 and the coefficient of determination is 0.9353. This result means that the regression model explains 93.53 % of the volatility. Finally, on the last line we see the results of the F-test. Again, this shows a very small $p$-value, which confirms that the model has significantly better predictive ability than the simple mean estimation.

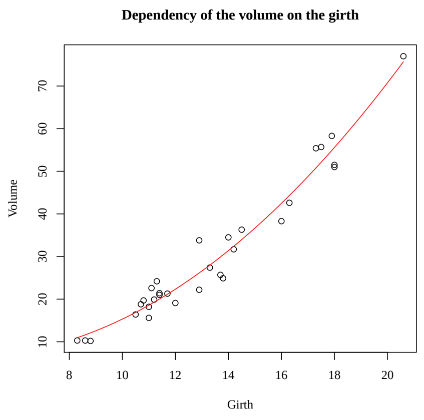
If we have the result stored in a variable, we can use the function `coef()` to get the coefficients of the regression line. The `confint()` function then provides confidence intervals for these coefficients. The default confidence level is 0.95, but this can be changed by specifying the argument `level=value`. The use of both functions is illustrated in the source code.

```
> coef(model)
(Intercept)       Girth
 -36.943459    5.065856
> confint(model,level=0.99)
                0.5 %     99.5 %
(Intercept) -46.21910 -27.667821
Girth         4.38399   5.747723
```

The function `lm()` can also be used for the case of polynomial regression. The regression function thus goes into the form

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n + \varepsilon.$$

The higher powers of the explanatory variable $x$ in the function `lm()` are then encapsulated in the function `I()` due to the special meaning of the symbols `^` and `*`. That means the arguments of the `lm()` function are `resp`, `var`, `I(var^2)`, `I(var^3)`, etc., and `dataset`. The following source code illustrates the quadratic regression function for the example with trees:



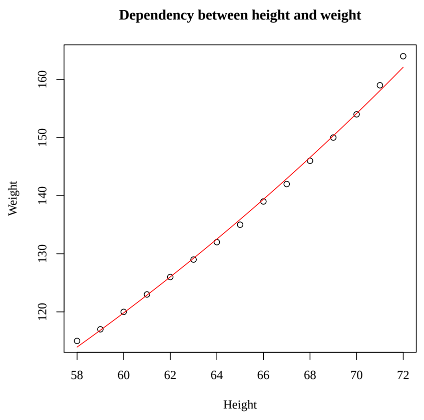


Figure 8.3: The scatter plot and quadratic regression curve of the `Girth` and `Volume` variables from the dataset `trees`.

Figure 8.4: The scatter plot and exponential regression curve of the women height and weight.

```
> lm(Volume~Girth+I( Girth^2),trees)

Call:
lm(formula = Volume ~ Girth + I(Girth^2), data = trees)

Coefficients:
(Intercept)        Girth    I(Girth^2)
    10.7863      -2.0921        0.2545
```

Therefore, the regression function has form

$$V = 10.7863 - 2.0921G + 0.2545G^2.$$

The result of the quadratic regression is graphically presented on figure 8.3. Let us note, that the graph of the regression curve is added to the plot using the `curve()` function. As her obligatory arguments are entered the regression function and `add=T` for adding the curve into the actual graph. The general form of the multi linear regression function with multiple explanatory variables is

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots \beta_n x_n + \varepsilon.$$

Also in this situation, we use the functions `lm()` to estimate the regression coefficients in the R environment. In this case, however, we specify the necessary number of explanatory variables `var1`, `var2` etc. In a multi linear regression, we can also take into account the interactions between the explanatory variables. These interactions are then characterized by different separators of the variables:

- `+` separates explanatory variables,
- `:` denotes interaction between variables,
- `*` denotes all possible interactions, for example `x*y*z` expands to `x+y+z+x:y+x:z+y:z+x:y:z`,

- ^ interactions up to the given grade, for example `(x+y+z)^2` expands to `x+y+z+x:y+x:z+y:z`.

To illustrate, we use the built-in dataset `mtcars`, which contains data on several car brands. We are looking for the dependence of consumption on the number of of cylinders, weight and power of the vehicle. These data are in the dataset assigned as `mpg` denoting the number of miles per gallon, `cyl` number of cylinders, `wt` for the weighth of the vehicle and `hp` the number of the horse power. We obtain estimates of the coefficients of the regression function using the following source code.

```
> lm(mpg~cyl+wt+hp,data=mtcars)

Call:
lm(formula = mpg ~ cyl + wt + hp, data = mtcars)

Coefficients:
(Intercept)          cyl           wt           hp
   38.75179     -0.94162     -3.16697     -0.01804
```

So, the corresponding model is

$$mpg = 38.75 - 0.94cyl - 3.17wt - 0.02hp.$$

Alternatively, we can look for the dependence of consumption on mass and power, taking into account their interaction with each other. In this case, we modify the source code to the following form:

```
> lm(mpg~wt+hp+wt*hp,data = mtcars)

Call:
lm(formula = mpg ~ wt + hp + wt * hp, data = mtcars)

Coefficients:
(Intercept)           wt           hp        wt:hp
   49.80842     -8.21662     -0.12010      0.02785
```

The corresponding model is

$$mpg = 49.81 - 8.22wt - 0.12hp + 0.03wt \cdot hp.$$

In the R environment, we can apply transformation functions directly in the linear model that is the output of the `lm()` function. Thus we can obtain the coefficients for the nonlinear form of the regression model. Using the dataset `women` we can determine the exponential dependence of the weight on the height. This data set gives the average heights and weights for American women aged 30–39. Its format is data frame with 15 observations on 2 variables: height in inches and weight in pounds. To determine the exponential dependence, we need to transform the variable `weight` using a logarithmic function. The source code for the calculation will then look like this:

```
> lm(log(weight)~height,data=women)

Call:
lm(formula = log(weight) ~ height, data = women)

Coefficients:
(Intercept)       height
    3.27508      0.02518
```

The resulting exponential regression function then takes the form

$$w = e^{3.27508+0.02518\cdot h}.$$

The graphical presentation of the result we can see on figure 8.4.

The implementation of `lm()` in the R environment allows factor variables to be included in the model. We illustrate this by modelling a person's height as a function of arm span. We use eye colour as the factor variable. For this purpose, we first create a short data file `people.csv` with the following contents:

```
Subject,Eye Colour,Height,Hand Span,Sex,Handedness,Height Cat
1,Brown,186,210,Male,R,Tall
2,Green,182,220,Male,R,Tall
3,Brown,147,167,Female,NA,NA
4,Green,157,180,Female,L,Short
5,Brown,170,193,Male,R,Medium
6,Blue,169,190,Female,L,Medium
7,Brown,174,217,Male,R,Medium
8,Blue,173,211,Male,R,Medium
9,Blue,166,193,Female,R,Medium
10,Blue,166,178,Female,R,Medium
11,Brown,163,223,Male,R,Medium
12,Blue,184,225,Male,R,Tall
13,Blue,176,214,Male,NA,Medium
14,Blue,183,218,Male,R,Tall
15,Green,160,190,Female,NA,Short
16,Brown,173,196,Male,R,Medium
```

After loading this file into the `people` variable, we use the `class()` function to make sure that the variable is of type `factor`. We then build the desired model using the following source code:

```
people<-read.csv("people.csv")
> class(people$Eye.Colour)
[1] "factor"
> lm(Height~Hand.Span+Eye.Colour,people)

Call:
lm(formula = Height ~ Hand.Span + Eye.Colour, data = people)

Coefficients:
   (Intercept)       Hand.Span   Eye.ColourBrown   Eye.ColourGreen
       82.8902          0.4456           -3.6233           -4.1924
```

Depending on the eye colour, we obtained the following height prediction models of the height $h$ in dependence on the hand span $hs$ :

$$\begin{aligned} h &= 82.89 + 0.45\cdot hs && \text{for persons with blue eyes}\\ h &= 82.89 + 0.45\cdot hs - 3.6233 && \text{for persons with brown eyes}\\ h &= 82.89 + 0.45\cdot hs - 4.1924 && \text{for persons with green eyes}\end{aligned}$$

The blue eye colour was used as the referral value in the previous computation. This can be changed using the `relevel()` function. Its syntax is

```
dataset$variable<-relevel(dataset$variable,"reflevel"),
```

where `reflevel` is the new referral value. In the previous model we can set as the referral value of the eye colour as green. Then we use the following source code:

```
> people$Eye.Colour<-relevel(people$Eye.Colour,"Green")
> lm(Height~Hand.Span+Eye.Colour,people)

Call:
lm(formula = Height ~ Hand.Span + Eye.Colour, data = people)

Coefficients:
   (Intercept)       Hand.Span    Eye.ColourBlue  Eye.ColourBrown
       78.6978          0.4456            4.1924           0.5690
```

The corresponding prediction models are then changed as follows:

$$\begin{aligned} h &= 78.6978 + 0.4456 \cdot hs && \text{for persons with green eyes} \\ h &= 78.6978 + 0.4456hs + 0.5690 && \text{for persons with brown eyes} \\ h &= 78.6978 + 0.4456hs + 4.1924 && \text{for persons with blue eyes.} \end{aligned}$$

# Bibliography

[1] Abdein, J. & Das Kumar, K.:*Data Manipuilation with R*, 2nd edition, Birmingham, Packt Publishing Ltd., 2014.

[2] Braun, W.,J. & Murdoch, D.,J.:*A First Course in Statistical Programming with R*, 2nd edition, New York, Cambridge University Press, 2016.

[3] Chang, W.:*R Graphics Cookbook.*, Sebastopol, United States, O'Reilly Media, Inc, 2013.

[4] Chiu, Y.:*R for Data Science Cookbook*, Birmingham, Packt Publishing Ltd., 2016.

[5] Crawley, M.,J.:*The R Book.*, 3rd edition, Chichester, England, John Wiley&Sons, Ltd., 2022.

[6] Crawley, M.J.:*Statistics: An Introduction Using R.* Boston, Addison-Wesley Publishing company, 2015.

[7] Dalgaard, P.:*Introductory Statistics with R*, 2nd edition, New York, Springer, 2008.

[8] Daróczi, G.:*Mastering Data Analysis with R*, Birmingham, Packt Publishing Ltd., 2015.

[9] DasGupta, A.:*Fundamentals of Probability: A First Course*, New York, Springer-Verlag, 2010.

[10] DeGroot, M.,H. & Schervish, M.,J.:*Probability and Statistics*, 4th edition, Boston, Addison-Wesley Publishing company, 2012.

[11] Devore, J.,L. & Berk, K.,N.*Modern Mathematical Statistics with Applications*, 2nd edition, New York, Springer, 2012.

[12] Everit, B.,S., & Hothorn, T.: *A Handbook of Statistical Analysis Using R*, Boca Raton, Chapman&Hall, CRC Press, Taylor & Francis Group, 2010.

[13] Field, A., Miles, J. & Field, Z.: *Discovering Statistics Using R*, Thousand Oaks, California, SAGE Publications, Ltd., 2012.

[14] Fox, J.: *An R Companion to Applied Regression*, 3rd edition, Thousand Oaks, California, SAGE Publications, Ltd., 2019.

[15] Holický, M.:*Aplikace teorie pravděpodobnosti a matematické statistiky*, Praha, ČVUT, 2015.

[16] Jakubowski, J., Sztencel, R.:*Wstęp do teorii prawdopodobieństwa*, Warszawa, Script, 2010.

[17] Kabacoff, R.:*R in Action*, New York, Manning Publications, 2015.

[18] Madsen, B.,S.:*Statistics for Non-Statisticians*, 2nd edition, Berlin Heidelberg, Springer-Verlag, 2016.

[19] Makhabel, B.:*Learning Data Mining with R*, Birmingham, Packt Publishing Ltd., 2015.

[20] Mittal, H.,V.:*R Graphs Cookbook*, Birmingham, Packt Publishing Ltd., 2011.

[21] Nánásiová, O., Kohnová, S.:*Štatistika a pravdepodobnosť. Základy matematickej štatistiky a teórie pravdepodobnosti*, Bratislava, STU 2016.

[22] Quick, J.,M.:*Statistical Analysis with R*, Birmingham, Packt Publishing Ltd., 2010.

[23] Randall, P.:*Foundastions and Applicatoins of Statistics, An Introduction Using R*, Providence, Rhode Island, American Mathematical Society, 2011.

[24] Ross, S.,M.:*A first course in probability*, 10-th edition, Boston, Pearson, 2018.

[25] Schmuller, J.:*Statistical Analysis with R For Dummies.*, Chichester, Hoboken, New Jersey, John Wiley&Sons, Inc., 2017.

[26] Schumacker, R. & Tomek, S.:*Understanding Statistics Using R*, New York, Springer, 2013.

[27] Spector, P.:*Data Manipulation with R*, New York, Springer, 2008.

[28] Suhov, Y. & Kelbert, M.:*Probability and Statistics by Example. Volume I. basic Probability and Statistics*, New Yourk, Cambbridge University Press, 2005.

[29] Verzani, J.:*Using R for Introductory Statistics.* Second edition, Boca Raton,CRC Press, Taylor & Francis Group, 2014.

[30] Wickham, H., & Grolemund, G.: *R for Data Science*, Sebastopol, United States, O'Reilly Media, Inc., 2017.

# List of Figures

# List of Tables

**Probability and Statistics with Programming in R**
**Study text**

Author: RNDr. Aleš Kozubík, CSc.
Žilinská univerzita v Žiline

Publisher: Mendel University in Brno, Zemědělská 1, 613 00 Brno, Czech Republic
Graphic editing and typesetting: Pavel Haluza
Year of publishing: 2022
First edition
Number of pages: 174

ISBN 978-80-7509-896-2 (online ; pdf)
DOI https://doi.org/10.11118/978-80-7509-896-2