R programming for statistics

Laboratory

Aleš Kozubík
University of Žilina

Project: Innovative Open Source Courses for Computer Science

31. 5. 2021

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

Innovative Open Source Courses for Computer Science

This teaching material was written as one of the outputs of the project
"Innovative Open Source Courses for Computer Science",
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.

Innovative Open Source Courses for Computer Science

Project was implemented under the Erasmus+.
Project name: "Innovative Open Source courses for Computer Science curriculum"
Project no.: 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: This project has been funded with support from the European Commission. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

R programming for statistics

I. Introduction into R environment

Installing R

It is freely available from the Comprehensive R Archive Network (shortly CRAN)

The internet location is `https://cran.r-project.org`

Here are at disposal pre-compiled binaries for all common platforms Linux, Mac OS, and Windows.

You can select the most suitable mirror for downloading the installation package.

Installing R packages

R comes with a huge set of packages extending its base core.

They increase the power of R.

To install package we use the function `install.packages()`

R first run

Once we have installed R, we can try if it works correctly.

We start the R environment interface simply from command prompt typing:

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

It displays a short introductory note that is followed by the sign

```
>
```

assigning the R prompt.

Leaving the R environment

The R environment is now ready for an interactive mode of work.

In order to finish work in R environment we simply type

```
> q()
```

R reacts by question:

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

If we select `y`, the whole working history is saved in the file `.Rhistory` saved in the working directory.

Workspace and navigation

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.

If we saved the history when leaving R environment, we can return to the commands from previous session.

Communication with the OS

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 find using the `getwd()` function.

The current working directory can be changed using the `setwd()` function.

To run the OS commands we apply the `system()` function.

To create a new directory

```
> system("mkdir new")
```

Getting help

The general function for getting help has a simple form `help()`, or shortly in the operator form `?`.

To get some information about the additional packages, we use

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

Some packages include also code demonstrations, we run using function `demo()`

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

Using R as calculator

Console prompt enables interactively compute the operations and functions

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

If we don't see new prompt, it may be because we entered an incomplete command

```
> 5-
+
```

We have to type the rest of the command and then press Enter or cancel the command pressing Esc key.

Objects

R is object-oriented language

Everything in R is an object and it represents some data that has been stored in memory

Objects can be given any name, 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)`),

Creating objects

We create a new object simply with the assignment operator

The assigning operator has two possible forms: <- or =.

It is recommended to use <- as = can sometimes lead to errors:

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

Listing and removing objects

The list of all created object we obtain as an answer of the `ls()` function.

The objects we will not use in the future can be removed from the memory using the `rm()` function.

R programming for statistics

II. Data structures in R

Data type `numeric`

The data type `numeric` represents the real decimal numbers.

It is the default type of each new object

If we assign to any variable the decimal

The type of any object we recognise using function `class()`

Data type numeric – an example

Let us see the example.

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

Note

Number is represented as vector with length 1. Sign [1] means the first position in the vector.

Data type numeric

Inserting an integer in the variable does not change its type, but it remains numeric

See the example

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

We can also ask using function `is.integer()`.

```
> is.integer(z)
[1] FALSE
```

Data type integer

To create the object of the integer type, we use the function `as.integer()`

Example

```
> a <- as.integer(12)
> a
[1] 12
> class(a)
[1] "integer"
> is.integer(a)
[1] TRUE
```

Data type integer

Alternatively, the variables of the integer type can be submitted as whole numbers ended by letter L

Example

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

Data type integer

What happens if we insert value that is not integer?

```
> as.integer(2.718)
[1] 2
> as.integer(TRUE)
[1] 1
```

The value is rounded or transformed to integer.

Data type integer

But excluding characters or strings are exception

They are not transformed

```
> as.integer("frcka")
[1] NA
Warning message:
NAs introduced by coercion
```

Change of the type

When making any computations, it is important to keep in mind that a variable may be retyped.

Example

```
> x<-as.integer(20)
> class(x)
[1] "integer"
> x<-x/3+1
> x
[1] 7.666667
> class(x)
[1] "numeric"
```

Data type complex

R environment provides also the possibility to work with complex numbersT

The complex value is in R defined via the imaginary unit `i`

Example

```
> z<-1+2i
> class(z)
[1] "complex"
```

Data type complex

Keep in mind, that value -1 is not of the complex type and therefore

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

We must enter

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

Data type `complex`

Do you know alternative solution?

Data type `complex`

Do you know alternative solution?

We use the function `as.complex()`

Data type complex

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

sqrt() and as.complex() functions has to be entered in the given order

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

Data type complex

When entering the complex number with unit imaginary part, it is necessary to write the coefficient

Otherwise the imaginary unit is understood as object

Let us see

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

Data type logical

It 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"
```

Data type `logical`

Here are defined all standard logical operations

`&`	Logical `AND`
`|`	Logical `OR`
`!`	Negation

Data type logical

Illustration

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

Data type character

It is used to store the string values, strings are entered using the quotation marks

```
> x<-"facina"
> class(x)
[1] "character"
#But as well
> x<-as.character(3.1415926)
> x
[1] "3.1415926"
> class(x)
[1] "character"
```

Data type character

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

```
> name<-"Donald"
> surname<-"Knuth"
> paste(name,surname)
[1] "Donald␣Knuth"
# To add any separator
> paste(name,surname,sep=",")
[1] "Donald,Knuth"
```

Data type character

How to arrange concatenation without spaces?

Data type `character`

How to arrange concatenation without spaces?

We define the separator in the function `paste()` to be empty, means we define `sep=""`

Data type character

How to arrange concatenation without spaces?

We define the separator in the function `paste()` to be empty, means we define `sep=""`

```
paste(name,surname,sep="")
```

Data type `character`

Sometimes it is useful to obtain formatted output using the `sprintf()` function

Its syntax is same as in C language

Formatting labels

`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.

Data type character – formatted output

```
> 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"
```

Data type character – function substr()

Extracting some substring is a typical operation.

Here is implemented the function substr()

Its arguments are the original string and the start and end positions of the substring that should be extracted

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

Data type character – function sub()

To replace some part of the string by another substring, we apply the function sub()

It is important to pay attention to unambiguity of the substring, because only the first occurrence is substituted

Let us see the example

```
> 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"
```

Data type character – function gsub()

This function differs from sub() in that gsub() substitutes all matches respectively

Let us see the change in the previous example

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

Vectors

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 referred as components

The number of components of the vector is referred as its length.

Vectors

Vector `v` is created by combine function `c()`

Its length we get using the `length()` function

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

Vectors

Vector of logical values

```
> v<-c(TRUE,TRUE,FALSE,TRUE,FALSE)
>v
[1]  TRUE  TRUE FALSE  TRUE FALSE
```

Vector of characters

```
> a<-c("aa","bb","cc","dd","ee","ff")
> a
[1] "aa" "bb" "cc" "dd" "ee" "ff"
```

Vectors

Vectors can be combined using the combine function c()

```
> a<-c(1,2,3)
>b<-c(4,5,6)
>c(b,a)
[1] 4 5 6 1 2 3
# See the retyping of components
> a<-c("a","b","c")
> c(a,b)
[1] "a" "b" "c" "4" "5" "6"
```

Vectors – the arithmetics

The vector arithmetic is implemented component-wise.

The arithmetic operations are performed component-by-component.

- + 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.

Vectors – the arithmetics

```
> 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
```

Vectors – the arithmetics

Attention on the recycling rule

If the length of the vectors does not match, the shorter one is used repeatedly.

This rule is limited by condition that the length of the longer vector is a multiple of the shorter one.

Vectors – the arithmetics

```
> v<-c(10,20,30)
> u<-1:9
> u+v
[1] 11 22 33 14 25 36 17 28 39
# Number is vector with length 1
> b<-c(1,2,3,4)
> 5*b
[1]  5 10 15 20
```

Vectors – selecting the components

The components we want to select from the vector are submitted by indices in brackets []

```
> v<-1:10
> v[3:5]
[1] 3 4 5
```

Note

Operator : defines the scope of numbers from the first to the second.

Vectors – selecting the components

We can select the components also using the vector of logical values.

The length of both vectors should be the same, otherwise the remaining positions are assumed to be TRUE.

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

Vectors – selecting the components

The selected components must not be in the continuous sequence of indices.

We define them using the combine c() function

```
> a<-c("aa","bb","cc","dd","ee","ff")
> a[c(2,3,5)]
[1] "bb" "cc" "ee"
# The indices can repeat
> a[c(2,2,3,5)]
[1] "bb" "bb" "cc" "ee"
```

Vectors – assigning names to the components

We can assign some names to the components.

We define the names using the function `names()`.

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

Vectors – assigning names to the components

Once we have assign the names to the components, we can select them by these names.

In the previous demonstration we can use

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

Matrix

Matrix is a two dimensional collection of data of the same type arranged in rectangular layout.

We create it using the function `matrix()` with following 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.

Matrix – entering examples

```
> 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
```

Matrix – accessing the elements

The single elements of the matrix are accessed by pair of coma separated indices in brackets.

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

Matrix – accessing the elements

The single elements of the matrix are accessed by pair of coma separated indices in brackets.

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

Omitting one of the indices leads to extracting of the row or column

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

Matrix – extracting submatrices

We define the rows and columns using the `c()` function.

```
> 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
```

Matrix – assigning names

We assign names to rows and columns using the `dimnames()` and `list()` functions

```
> 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
```

Matrix – transposing

We can transpose the matrix using the function `t()`

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

Other function are defined in package `matlib`.

Matrix – the operations

Defined component-wise

It is important when multplying matrices. The common operation $*$ means multiplication of the elements on the same positions.

The standard multiplication is defined as operation %*%.

Matrix – the operations

Let us compare

```
> C<-B[c(1,2),c(1,2)]
> C*C
     [,1] [,2]
[1,]    9   36
[2,]   16   49
> C%*%C
     [,1] [,2]
[1,]   33   60
[2,]   40   73
```

Matrix – combining

To combine matrices it is necessary they have the same number of row or columns.

If they have the same number of rows, we can combine the columns with the `cbind()` function.

```
> 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
```

Note

Let us note the `diag()` function. It produces diagonal matrix with given vector on its diagonal.

Matrix – combining

If the matrices have the same number of columns, we can combine the columns with the `rbind()` function.

```
> C<-matrix(1:12,nrow=3)
> 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
```

Array

Arrays are generalizations of the matrix data structure.

They are more than two dimensional matrices

An array we can create using the `array()` function.

Its syntax is

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

Array – creation

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")
```

Array – creation, cuntinuation

Now we create the array z, that contains the integers from 1 to 36 ($3 \times 4 \times 3$ is 36)

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

To see the structure of the array, type now z in the R interface

The output is too long to be displayed here.

Array – accessing the elements

We access the elements of the array using the brackets in the same mode as in matrices.

```
> z[2,3,1]
[1] 8
> z[2:3,2:3,2]
    B2 B3
A2 17 20
A3 18 21
```

Data structure frame

Frame is the most common structure for storing the data.

It enables storing the column vectors of the different data types.

Data frames are created using the function `data.frame()`, general syntax

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

Data frame – creation

We create a short data frame inckuding data about scoring of the basketball 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
```

Data frame – accessing the cells

Here are several ways how to access the each cell of the data frame

We can use the index notation

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

Data frame – accessing the cells

Another option is to use the column names.

The column names are submitted as character vector.

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

Data frame – accessing the cells

The third possibility is to use the $ notation.

It consists from the data frame name on the first place and column name on the second place, that are separated by the $ sign.

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

Data frame – double brackets operator

To access the single column we apply the he double square bracket `[[]]`

Compare these two listings

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

```
> players[[4]]
[1]  7  4  6 12
```

Data frame – double brackets operator

The double brackets operator is equivalent to use of coma in the one bracket operator

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

Data frame – assigning names to rows

We use the function `row.names()` whose argument is a vector of characters

```
> 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
```

Data frame – assigning names to rows

Now we can extract rows by indices or by names

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

Data frame – extracting rows

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
```

Data frame – assigning names to rows

It can bring some discomfort if we have to write the name of the data frame very frequently.

The `attach()` function adds the data frame to the search path.

It enables to write only the column names.

To remove the frame from the search path, we simply use the `detach()` function.

Data frame – attach() demonstration

After attaching the data frame players, we easily compute the sscoring percentages of each player

```
> 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
```

Data frame – alternative to attach()

An alternative to the attaching the frame to the search path is using the `with()` function.

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

Data frame – merging the datasets

We frequently need to merge date from two or more datasets.

We use function `merge()`.

The arguments are the names of two data frames to be merged.

Third argument `by=''column_name''` defines the key variable for joining the data.

Data frame – merging the datasets

To demonstrate the merging, we create new frame `rebounds` at first.

```
> 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")
```

Data frame – merging the datasets

Now we are ready to merge data frames `players` and `rebounds`.

```
> 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
```

Data frame – merging the datasets

Alternative is adding the rows to the existing frame using the function `rbind()`.

Arguments are names of two data frames.

To illustrate it, we prepare new data frame `players2`

```
> 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")
```

Data frame – merging the datasets

Now we merge these data frames

```
> 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
```

Lists

Lists represent the most complex data structure.

Lists are ordered collections of objects.

To create a list, we use the function `list()`. Its syntax is simple:

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

Its arguments are names of existing objects

Lists

Optionally we can name the object in the created list:

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

Lists

We create the list named NBA from our existing data frames players and players2

```
> 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
```

Lists

Now we can add a next member of the list using the concatenate function `c()`

```
> NBA<-c(NBA,list(club="Celtics",city="Boston",
  Players=players2))
> NBA
```

The output is too long to be displayed here, see it directly in R.

Note

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.

Lists – accessing the elements

We have to distinguish among the single and double bracket operators.

Try the following commands (some outputs are too long to be displayed here)

```
> NBA[3]
> NBA[[3]]
> NBA[3][2]
$<NA>
NULL
> NBA[[3]][2,]
        playerID position attempted made
Player2        2    guard         6    4
```

Lists – modifying the elements

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
```

Entering data from the keyboard

The simplest method (but also the most time consuming for large samples)

We work in two steps

- Create the empty data frame with the variable names and types we want to store in the dataset.
- Invoke the simple data editor using the function `edit()`, whose argument is the name of the data frame we want to edit.

Entering data from the keyboard

We create empty data frame named `mydata` with four variables: `name` that has type character and three numeric variables `age`, `height` and `weight`.

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

Note

Let us note, that assigning like `numeric(0)` and `character(0)` create a variable of the given type but without any data.

Entering data from the .csv file

Comma separated values, one of the most used data format.

The first row should, but must not, contain the column names.

Example of the file structure

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

Entering data from the .csv file

We assume the data are saved in the file `mydata.csv`.

We import the data by `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
```

Entering data from the `.csv` file

Options of the `read.csv()` function.

- `header` logical value, indicates whether the input file contains the names of variables as the first line, default value `TRUE`.
- `sep` defines the field separator character, the default value is comma,
- `dec` defines the character used in the file for decimal points, default value is `.`, 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.
- `stringsAsFactors` which is a logical value that indicates whether the strings are converted to factors, to prevent converting we set it to `FALSE`.
- `row.names` a vector of row names

Writing data into the `.csv` file

R can create `csv` file form existing data frame.

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.

Common syntax

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

`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

Writing data into the `.csv` file

Selected options of the `write.csv()` function

- `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.

Entering data from the Excel files

Here 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.

Entering data from the Excel files

We install the package by standard command:

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

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

```
library("xlsx")
```

Entering data from the Excel files

This package provides two functions for reading the contents of an Excel worksheet into a R data.frame: `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.

Entering data from the Excel 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.

Writing data into the Excel files

Package `xlsx` provides two functions fro writing `write.xlsx()` and `write.xlsx2()`

General syntax

```
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)
```

Writing data into the Excel files

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.

Reading data from the JSON files

JSON (JavaScript Object Notation) is a lightweight data-interchange format

To get JSON files into R, we first need to install or load the `rjson` package.

We can use the `fromJSON()` function

Usage depends on the location of the `.json` file

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

a

Writing data into the JSONI files

It 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 create a JSON object we use `toJSON()` function:

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

Then we use the `write()` function

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

R programming for statistics

III. Probability distributions in R

Random samples

Standard function `sample()` with syntax

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

Arguments

- `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.

Random samples – examples

The simplest use with only the first argument

```
> 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
```

Random samples – examples

The second argument states the sample size

Randomly selected 5 integers from 1 to 40

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

Random samples – examples

We simulate rolling the dice 50 times

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

Error because the the sample size exceeds the length of the data to be sampled.

Random samples – examples

We simulate rolling the dice 50 times – we must set the `replace` argument

```
> 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
```

Random samples – examples

We can simulate tossing the unfair coin with higher frequencies of heads than tails.

Let us suppose, that heads fall twice more than tails, we set the argument `prob=c(2/3,1/3)`.

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

Random samples – ensuring same result

If we take samples, they will be random and they change each time we apply the `sample()` function

If we need to reconstruct the same sample, we can use the `set.seed()` function

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

Discrete distribution

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

$$p(x) = \mathbb{P}\left(X = x\right), \, x \in H. \tag{1}$$

Discrete distribution

We mention some of them:

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

Bernoulli distribution

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)`

Binomial distribution

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.
- `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.
- `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.

Binomial distribution – examples

Example

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.

Binomial distribution – solution 1

The probability of answering a question correctly by random is $\frac{1}{5} = 0.2$.

We can find the probability of having exactly 6 correct answers by random attempts using the `dbinom()`

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

Binomial distribution – solution 1

To find the probability of having six or less correct answers by random attempts, we apply the function `dbinom()` with $x = 0, \ldots, 6$ and sum the results.

So we get:

```
> 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
```

Binomial distribution – solution 2

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

So we get the same value

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

Binomial distribution – example continuation

Example

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?

Binomial distribution – solution

As we are searching for the probability $\mathbb{P}(X > 10)$ 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
```

Binomial distribution – example 2

Example

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.

Binomial distribution – 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.

So we have

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

Binomial distribution – example 3

Example

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.

Binomial distribution – solution

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
```

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
```

Hypergeometric distribution

Four functions for handling hypergeometric distribution in R:

- `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.

Hypergeometrlc distribution – example 1

Example

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?

Hypergeometric distribution – solution

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.

Therefore we have

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

Hypergeometric distribution – solution

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.

We can use summation of the `dhyper()` function 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, where $x = 2$ 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.

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

Hypergeometrlc distribution – example 1

Example

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.

Hypergeometric distribution – solution

The shipment contains $m = 10$ defective DVD players and $n = 90$ non-defecive DVD players and inspector randomly selects $k = 15$, the inspection is repeated $N = 10$ times.

To simulate their results we apply the function `rhyper()`.

So we get:

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

Negative binomial distribution

Four 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.

Negative binomial distribution – examples

Example

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?

Negative binomial distribution – solution

We need to find $\mathbb{P}(X = 2)$.

Note that is technically a geometric random variable, since we are only looking for one success.

Due to the implementation of the `dnbinom()` function, we set `x=2` failures prior `n=1` success and `p=0.2`.

So we have

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

To the second question we choose `x=4` failures prior `n=3` successes.

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

Poisson distribution

Four functions for handling Poisson distribution in R:

- `dpois(x,l)` calculates the probability mass function value $\mathbb{P}(X = x)$ of the Poisson distribution with the parameter λ 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, $\mathbb{P}(X \leq x)$ the argument `l` is the parameter of the distribution. Stating the additional argument `lower.tail=FALSE` we get the probability $\mathbb{P}(X > x)$.
- `rpois(k,l)` is used for generating random numbers from a given Poisson distribution, `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, `q` is a vector of the quantile levels required and `l` is the parameter of the distribution.

Poisson distribution – examples

Example

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

Poisson distribution – solution

Overflow occurs once in 100 years, we can consider it to be a rare event and the number of overflows follows the Poisson distribution.

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.

We have

```
> 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
```

Poisson distribution – example 2

Example

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.

Poisson distribution – solution

"Some policies" means "1 or more policies"

We have to calculate the probability $\mathbb{P}(X > 0) = 1 - \mathbb{P}(X \leq 0)$.

The parameter of the distribution is `l=3`

We apply the function `ppois()` with the additional argument `lower.tail` set to `FALSE`

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

Alternatively we can use `dpois()`:

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

Poisson distribution – example 3

Example

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.

Poisson distribution – 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
```

Poisson distribution – example 4

Example

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.

Poisson distribution – solution

To find a maximum arrivals with at least 90% certainty level means to find the 90% quantile.

We use the `qpois()` function with arguments `q=0.9` and `l=2` average arrivals per minute.

So we get

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

Continuous distribution

We mention some of them:

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

Many other are implemented in R.

Continuous distribution

We mention some of them:

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

Many other are implemented in R.

We will concern in three "blue" distributions.

Uniform distribution

Four functions for handling uniform distribution in R:

- `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.

Uniform distribution – example

Example

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.

Uniform distribution – solution a)

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 $\mathbb{P}(X > 3) = 1 - F(3)$.

We get

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

Uniform distribution – solution b)

The question b) is about the 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
```

Uniform distribution – simulation

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.

To see the graph, let us run the code

```
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))
```

Exponential distribution

Four functions for handling exponential distribution in R:

- `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.

Exponential distribution – example

Example

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.

Exponential distribution – solution

The mean completing time, the checkout processing rate equals to its inverted value,

Hence the processing rate is $\frac{1}{3}$ checkouts per minute. So the question a) is answered as the probability $\mathbb{P}(X < 2)$

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

The question b) is answered as the probability $\mathbb{P}(X > 5)$

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

Exponential distribution – example 2

Example

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).

Exponential distribution – solution a)

We denote X the random variable that represents the time to the malfunction of the device.

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 $\lambda = 1/30$.

We get the result by the command:

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

Exponential distribution – solution b)

In order to answer question b), we have to find the probability $\mathbb{P}(X \geq 72)$

To get the answer using the `pexp()` function, we have to set the argument `lower.tail=FALSE`

We get

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

Exponential distribution – quantiles

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.

Normal distribution

Four functions for handling normal distribution in R:

- `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.

Normal distribution – example

Example

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.

Normal distribution – solution a)

We apply the function `pnorm()` of the normal distribution with mean 70 and standard deviation 10.
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`.

We have

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

Normal distribution – solution b)

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
```

Normal distribution – example 2

Example

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.

Normal distribution – solution

Let us denote the random variable that describes the height as X,

To answer the question a) we have to compute the probability $\mathbb{P}(X \geq 200)$.

In b) we have to find $\mathbb{P}(X < 160)$.

Using the function `pnorm()` we get

```
> pnorm(200,179,6.68,lower.tail=FALSE)
[1] 0.000834096
> pnorm(160,179,6.68)
[1] 0.002225376
```

R programming for statistics

IV. Programming in R

Functions

Almost all actions in R run through functions.

Here is a rich collection of built-in function

Other functions can be defined by user

The built-in functions we can divide into

- math functions,
- string functions,
- specialized statistical and probability functions,
- other useful functions.

Math functions

Some of them we have already mention in lesson 1.

Here we introduce some additional details

The logarithmic function `log()` computes as the default value the natural logarithm.

To get logarithm with any base, we must declare the `base` argument of the function `log()`.

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

Math functions

Trigonometric functions work with argument given in radians.

When using the grades, we have to reshape the value as $r = \frac{\pi \cdot \alpha}{180}$, where r is the new measure in radians and α is the old value given in grades or we can also use the function `deg2rad()` from the `REdaS` package.

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

```
> tan(45)
[1] 1.619775
> tan(deg2rad(45))
[1] 1
```

Math functions – complex numbers

Functions for computing with the complex numbers

- `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 $\overline{z}$.

String functions

Function `nchar()` determines the size of each elements of an character vector

```
> z<-c("yellow","black","white")
> nchar(z)
[1] 6 5 5
> str<-"This is a long string"
> nchar(str)
[1] 21
```

String functions

Argument keepNA is logical, it states if NA should be returned where ever x is NA

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

String functions

List of the string functions

- `nchar()` Number of characters in string.
- `substr()` Extract or replace substrings.
- `grep()` Search for a pattern in string.
- `strsplit()`S plits 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.

String functions

To find a subscribed pattern in the string we use the grep() function

```
> 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)
```

String functions

To substitute the found pattern by another string we use the function `sub()`.

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

Here is optional argument `ignore.case`, logical value.

String functions

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.

```
> 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"
```

String functions

Function `strsplit()` splits the elements of the character vector `x` at positions defined by the second `split` argument.

```
> 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"
```

String functions

Function `strsplit()` splits the elements of the character vector `x` at positions defined by the second `split` argument.

We have alrady mention function `paste()` to contcatenate the strings.

Arguments are the strings to be concatenated and `sep` that defines how to separate them

```
> 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"
```

String functions

Two related functions `toupper()` and `tolower()` transform given string into the upper case and lower case letters

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

Elementary statistical functions

- `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.

Elementary statistical functions – `mean()` optional arguments

`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.

```
> 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
```

Elementary statistical functions – quantiles()

Default outcome are the quartiles

To specify the probability levels for the quantiles, we must set the optional argument prob in the form of the numeric vector.

```
> delay<-c(0,9,0,42,14,0,11)
> quantile(delay)
  0%  25%  50%  75% 100%
 0.0  0.0  9.0 12.5 42.0
 > quantile(delay,prob=c(0,0.33,0.67,1))
   0%   33%   67%  100%
 0.00  0.00 11.06 42.00
```

Elementary statistical functions – `mad()`

Median absolute deviation is a robust measure of the variability of a univariate sample of quantitative data.

For the sample $X_1, \ldots, X_n$ it is defined by formula:

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

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

Useful functions – seq()

The function seq() generates the sequence of numbers starting by from value and ends in to value. The last argument by defines the step of the sequence.

```
> 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
```

Useful functions – rep()

Function rep() has two arguments, the vector x to be repeated and number n of the repeating cycles

```
> 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"
```

Useful functions – sort() a order()

The functions `sort()` and `order()` are joined with sorting the vectorx

`sort()` gives sorted values while `order()` gives the indices of ordered the components in the original vector.

```
> 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
```

Useful functions – `rev()`

Gives vector `x` in reverse order

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

Conditional statements – `if` statement

`if()` statement performs operations based on a simple condition

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

More than one commands need to be braced

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

Conditional statements – `if ... else` statement

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

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

```
> 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"
```

Conditional statements – `if ... else` statement

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
}
```

Conditional statements – `if ... else` statement an example

Example

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.

Conditional statements – if ... else solution

```
> 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
```

Conditional statements – `ifelse` statement

The `if` and `if ... else` statements should not be applied when the condition being evaluated is a vector.

`if` statement evaluates the condition only for the first element of the vector.

```
> 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
```

Conditional statements – `ifelse` statement

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

```
ifelse(condition, expression1, expression2)
```

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

Conditional statements – switch

switch() tests an expression against elements of a list. Each value in the list is called case

The syntax of the switch() function:

```
switch (expression, list)
```

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

Conditional statements – switch

If the expression is a character string, switch() will return the value based on the name of the element.

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

Conditional statements – switch

In case of multiple matches, the value of first matching element is returned

We can define the default value which is returned if no match is present

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

Loops – `for`

`for` loop allows us to repeat a command or a block of commands a fixed number of times

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

Loops – for

```
> 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
```

Loops – for

We can stop the loop before it has looped through all the items applying the `break` statement

```
> 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
```

Loops – for

With the `next` statement, we can skip an iteration without terminating the loop.

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

Loops – while

Suitable when we want to repeat a command or block of commands until a given condition is satisfied

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

Loops – while

Use of the `while` loop in simulating the die rolls, 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
```

Loops – `repeat`

Similar to `while` loop, but the block of commands is executed at least once, no matter what the result of the condition

```
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

Loops – repeat

Use of the `while` loop in simulating the die rolls, until the first roll of six

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

User defined function

The general structure of a function is given below.

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

User defined function

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.

User defined function

Define function cubes() that prints the third powers of numbers in sequence

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

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

User defined function

We can define this function as well without arguments. In such circumstances it produces the sequence of the cube powers of the constant length.

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

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

User defined function

The arguments to a function call can be supplied in the same sequence as defined in the function

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

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

User defined function

Alternatively we can call the function by names of the arguments

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

User defined function

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)
   }
}
```

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

User defined function

What happens if we want to put the value `cubes(2,2)` in some variable?

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

Variable `z` does not include any value

User defined function

We must redefine the function using `return()`

```
cubes<-function(start=1,end=10){
   for(i in start:end) {
      b <- i^3
      return(b)
   }
}
> z<-cubes(2,2)
> z
[1] 8
```

User defined function

The function cubes() actually returns only one value

To extend the result to whole scope, we must define the output variable as vector

```
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
   }
   return(b)
}
```

User defined function

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

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

User defined function

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.

```
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)
}
```

User defined function

Now we can use it to get output as a list

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

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

Running R scripts

An R script is simply a text file containing (almost) the same commands that you would enter

We can create it in any simple text editor and save with extension `.R`

There are basically two Linux commands to run the script

```
Rscript filename.R
```

and is preferred. The older command is

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

R programming for statistics

V. Elementary graphics

Scatter plots

We create it simply using `plot()` function.

In the simplest use, the function has two arguments `x` and `y`

These variables are vectors that contain the values we want to plot.

The length of the vectors must be the same.

Scatter plots

Example

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:

Temp.	28	30.2	32	31	29.5	26	31.5	30	29	34
Sales (€)	540	560	530	570	525	490	530	530	500	580

Scatter plots

At first we define two numeric vectors:

`x` that contains the temperatures

`y` that will represent the daily sales

Then we draw the scatter plot

```
> 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)
```

How to save plot

We can use `dev.copy()` command, to copy the contents of the graph window to a file without having to re-enter the commands.

To create a `png` file `newplot.png` with our graph, we enter:

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

How to save plot

Alternatively, we can redirect the output from screen to the file.

We can use the functions

`pdf()`	Vector `pdf` format, best choice when used with `pdflatex`
`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.

Options for saving the graphs

`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, default 72 dpi.
`units`	Units of measure.
`bg`	Background colour.
`fg`	Foreground colour.
`family`	The fonts used (default Helvetica).

Modifying the plot – dot characters

The dot character is given by value of `pch` argument of the `plot()` function

Possible values

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

Modifying the plot – dot characters

Let us try to modify our plot

```
> plot(x,y,pch=17)
> plot(x,y,pch=1)
```

Modifying the plot – dot characters

Let us try to modify our plot

```
> plot(x,y,pch=17)
> plot(x,y,pch=1)
```

What to modify next?

Modifying the plot – dot characters

Let us try to modify our plot

```
> plot(x,y,pch=17)
> plot(x,y,pch=1)
```

What to modify next? The line type

Modifying the plot – line type

The line type is set by the `type` argument of the `plot()` function

Possible values

`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.

Modifying the plot – line type

Let us try to modify our plot

```
> plot(x,y,type="l")
> dev.off()
> plot(x,y,type="s")
> dev.off()
> plot(x,y,pch=17,type="b")
> dev.off()
```

Modifying the plot – line type

Let us try to modify our plot

```
> plot(x,y,type="l")
> dev.off()
> plot(x,y,type="s")
> dev.off()
> plot(x,y,pch=17,type="b")
> dev.off()
```

What to modify next?

Modifying the plot – line type

Let us try to modify our plot

```
> plot(x,y,type="l")
> dev.off()
> plot(x,y,type="s")
> dev.off()
> plot(x,y,pch=17,type="b")
> dev.off()
```

What to modify next? The line style

Modifying the plot – line style

The line style is set by the `lty` argument of the `plot()` function

Possible values

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.

The line width is set by the `lwd` argument of the `plot()` function

Modifying the plot – line style

Let us try to modify our plot

```
> plot(x,y,type="l",lty=5)
> dev.off()
> plot(x,y,type="l",lty=1,lwd=2)
> dev.off()
```

Modifying the plot – line style

Let us try to modify our plot

```
> plot(x,y,type="l",lty=5)
> dev.off()
> plot(x,y,type="l",lty=1,lwd=2)
> dev.off()
```

What to modify next?

Modifying the plot – line style

Let us try to modify our plot

```
> plot(x,y,type="l",lty=5)
> dev.off()
> plot(x,y,type="l",lty=1,lwd=2)
> dev.off()
```

What to modify next? Colouring

Modifying the plot – colouring

Before we start, problem:

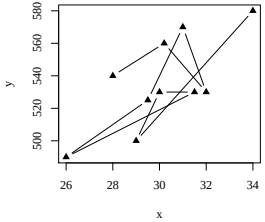
```
> plot(x,y,pch=17,type="l")
```

Modifying the plot – colouring

Before we start, problem:

```
> plot(x,y,pch=17,type="l")
```

gives

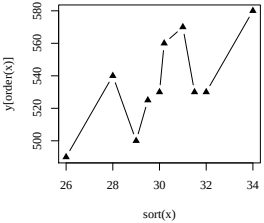


Modifying the plot – colouring

Before we start, problem:

```
> plot(x,y,pch=17,type="l")
```

but we want

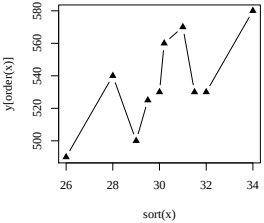


Modifying the plot – colouring

Before we start, problem:

```
> plot(x,y,pch=17,type="l")
```

but we want



How to arrange it?

Modifying the plot – line style

Answer

Use the `sort()` and `order()` functions

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

Modifying the plot – colouring

Setting the colours we can do by

- name of the colour, for example `col=red`
- by number of the colour, for example `col=636`
- by hexadecimal code, fo example `col="#FFCC00"`

The list of available colours we get as an answer of the function `colors()`.

Modifying the plot – colouring

Let us try

```
> plot(sort(x),y[order(x)],pch=17,type="b",
  col="red")
> dev.off()
> plot(sort(x),y[order(x)],pch=17,type="b",
  col=636)
> dev.off()
> plot(sort(x),y[order(x)],pch=17,type="b",
  col="#FFCC00")
> dev.off()
```

The list of available colours we get as an answer of the function `colors()`.

Modifying the plot – colouring

Other colouring options are

`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.

Modifying the plot – colouring

Let us try

```
>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()
>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()
```

Modifying the plot – colouring

Colours as vectors

We can set the value of the `col` argument as vector.

The colours from the vector are regularly varying

We can also use the `rainbow()` function, with predefined colour sequence.

Modifying the plot – colouring

Let us try

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

Modifying the plot – 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.

Alternative way, how to add the title and subtitle to the graph is using the function `title()`.

Modifying the plot – titles and subtitles

Let us try

```
> 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()
```

Modifying the plot – adding text into the plot

We can add some texts into the plotted graph using the functions `text()` and `mtext()`.

The difference `text()` places the given text on any position in the plotting area , `mtext()` function places the text into the margins.

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.

Modifying the plot – adding text into the plot

Let us try

```
> 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")
> dev.off()
```

Modifying the plot – adding text into the plot

Function `mtext()` 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. The lines are numbered from 0.

Let us try

```

> 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)
```

Modifying the plot – axes customization

To remove the plot box we set the option `axes=FALSE` inside the plotting function.

New axes we 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.

Let us try

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

Modifying the plot – axes customization

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.

Let us try

```
> 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")
```

Further axes customization

We are able to:

- set the number of the tick marks with specified 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` .

Further axes customization–ticks regions

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)`, `start` and `end` define the start and end value on each axis, and the value of `regions` defines the number of regions to divide the axis.

```
> 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(26,34,8))
> axis(2,col="blue",col.ticks="red",col.axis=555,
  yaxp=c(490,580,9))
```

Further axes customization–ticks marks length and orientation

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`.

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.

Further axes customization–ticks marks length and orientation

Try

```
> 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)
```

Note

We can remove the tick mark by setting the arguments `xaxt="n"` for the x-axis or `yaxt="n"` for the y-axis.

Further axes customization–ticks marks labels

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

```
> 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)
```

Limits and scaling

The limits for the axis we can define using the optional arguments `xlim` and `ylim` of the `plot()` function

The limits are 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.

`log="x"` sets the logarithmic scale to the `x`-axis,
`log="y"` sets the logarithmic scale to the `y`-axis and
`log="xy"` transforms both axis into the logarithmic scale.

Limits and scaling

Try

```
> plot(sort(x),y[order(x)],pch=17,type="b",
  col=30,axes=FALSE,ylim=c(400,600))
> 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)
```

Two dual vertical axes

Example

We want to plot in the same graph two characteristics of the patients' health condition, the temperature and the blood pressure.

Two dual vertical axes

Example

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.

Two dual vertical axes

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 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.

Two dual vertical axes

```
x<-1:100 # generating the values
y <- runif(100, min = 35, max = 40)
z <- y+10*runif(100,min=7,max=12)
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)
```

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.

For example

```
curve(x^2)
curve(x^2,xlim=c(-2,2),col="red",lwd=2)
```

Plotting two or more curves in one plot

We use the function `curve()` with argument `add=TRUE`.

For example

```
curve(x^2)
curve(sqrt(x),col="red",lwd=2,add=TRUE)
```

Plotting two or more curves in one plot

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.

```
set.seed(1)
x <- rnorm(100)
y <- x^2 + rnorm(100)
plot(y ~ x)
curve(x^2,add=TRUE)
```

Adding a legend

The function `legend()` enables adding a legend to the plots in R.

Some of the arguments:

- `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.

Adding a legend

Example

Let us create the user defined function `gonplot()`, that draws graphs of the $\sin x$ and $\cos x$ in the range $(-10; 10)$ in two colours and different line types. Then add the legend to the plot.

Adding a legend

The user defined function

```
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)
}
```

Adding a legend

Plot and adding the legend

```
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)
```

Adding a legend – note

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.

Bar graphs

Bar graph presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent.

To produce the bar graphs, R uses the function

```
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.

Bar graphs

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.

```
x<-c(2000,2400,1400,2600)
barplot(x)
```

Bar graphs – horizontal bars

We set the argument `horiz=T` to the true.

```
barplot(x,horiz=T)
```

Bar graphs – colouring and labels

We use these 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

It is similar like in the case of `plot()` function.

Bar graphs – colouring and labels

Let our vector x represent daily sales of some fruits.

We set their names as vector goods and use it to assign the names to bars.

```
goods<-c("orange","banana","apple","plum")
barplot(x,names.arg=goods,xlab="Fruit",
ylab="Sales",col="cyan",main="Monthly␣sale",
border="black")
```

Bar graphs – colouring and labels

We can modify the graph by different colours of the bars

We set the required colours as vector `colours` and use it as value of `col` argument

Argument `border` defines colour of the bars border

```
colours<-c("orange","yellow","red","blue")
barplot(x,names.arg=goods,xlab="Fruit",
ylab="Sales",col=colours,main="Monthly sale",
border="black")
```

Bar graphs – stacks

Using a matrix as input values we can colouring and labels.

We extent the sale volumes in vector `x` on more months.

Then we will present the information graphically.

We set at first

```
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)
```

Bar graphs – stacks

Now we are ready to draw the graph with stacks

We also add the legend

```
barplot(x,main="Sale volumes",
names.arg=months,xlab="Month",ylab="Sales",
col=colours,ylim=c(0,11000))
legend("topright",goods,fill=colours,ncol=2)
```

Bar graphs – stacks

The same information we can present alternatively by the grouped bar chart

We set argument beside=T

```
barplot(x, beside=T,main ="Sale volumes",
names.arg=months,xlab="Month",ylab="Sales",
col=colours,ylim=c(0,3000))
legend("topright",goods,fill=colours,ncol=2)
```

Bar graphs – filling by textures

Instead of colours, we can fill the bars by textures

The simplest are the parallel lines

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.

Bar graphs – filling by crossing lines

```
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")
```

Bar graphs – filling by crossing lines

```
angle1<-c(0,30,60,90)
angle2<-c(90,120,150,0)
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))
```

Histograms

A histogram is an approximate representation of the distribution of numerical data.

They represent the frequencies of values of a variable bucketed into ranges.

Histogram is similar to bar graph but the difference is it groups the values into continuous ranges.

Histograms give a rough sense of the density of the underlying distribution of the data

Histograms

Histogram can be created using the `hist()` function in R

```
hist(H,xlab,ylab,title, names.arg,col)
```

The parameters used in the function are 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.

Histograms–example

Example

Let us 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.

Histograms–example

At first, we simulate rolling the dice:

```
dice1<-sample(1:6,replace=T,10000)
dice2<-sample(1:6,replace=T,10000)
c<-dice1+dice2
```

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")
```

Histograms–example

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 plot as the histogram.

```
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)
```

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 graphs

he basic syntax for creating a pie-chart using the R is:

```
pie(data, labels, radius, main, col, clockwise)
```

The meaning of the arguments:

- `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.

Pie graph–example

Example

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 values we use as parameters of the pie chart:

```
data<-c(200,300,100,80,150)
labels<-c("housing","food","clothing",
"entertainment","other")
pie(data,labels,main="Monthly expenses")
```

Pie graph–colours modifying

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 (≥ 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).

Pie graph–use of rainbow()

```
description<-paste(labels,"\n",data,sep="")
pie(data,description,main="Monthly expenses",
col=rainbow(length(data)))
```

Note

We have also changed the labels. We add also the numerical values to their names.

Pie graph–further improvements

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.

```
description<-paste(labels,"\n",
trunc(100*data/sum(data)),"%", sep="")
pie(data,description,main="Monthly expenses",
col=heat.colors(length(data)))
```

Pie graph–further improvements

In order to get the 3D-effect in the chart, we must use the package `plotrix`.

We use `pie3D()` charts with 3D-effect.

```
library("plotrix")
pie3D(data,labels=description,
 main="Monthly expenses",
 col=rainbow(length(data)))
```

Pie graph–exploding the parts

We can further customize the 3D-chart appearance using the parameters

- `height` that states the height of the 3D pie (the default value is 0.1)
- `theta` that changes the viewing angle (the default angle is $\frac{\pi}{6}$).
- `explode` that defines exploding the part of the pie

```
pie3D(data,labels=description,
 main="Monthly expenses",
 col=terrain.colors(length(data)),
 height=0.2,theta=1.5,
 explode=0.1)
```

Note use of the `terrain.colors` palette

Fan plot

Useful alternative to the pie charts is `fun.plot()` defined in the `plotrix` package.

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 2π.
- `ticks` the number of ticks that would appear if the sectors were on a pie chart. Default is no ticks.

Fan plot

Illustration of the fun plot

```
fan.plot(data,labels=description,
 main="Monthly expenses",
 col=rainbow(length(data)),
 max.span=pi,ticks=max(data))
```

Fan plot

Illustration of the fun plot

```
fan.plot(data,labels=description,
 main="Monthly expenses",
 col=rainbow(length(data)),
 max.span=pi,ticks=max(data))
```

The disadvantage of the fun plot is a large white space above the chart.

We can remove this space by setting the new graphical device with user defined height and width.

A new graphical window we open by function `new.dev()`. The window size we define by arguments `height` and `width`.

Fan plot

```
dev.new(width=10,height=5,unit="cm")
fan.plot(data,labels=description,
 main="Monthly expenses",
 col=rainbow(length(data)),max.span=pi,
 ticks=max(data))
```

Box plot

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.
- `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.

Box plot–example

Example

Let us suppose, we have in data file `players.csv` the statistical data from the basketball game. This data file contains the players identification, his position and number of attempted and made shoots. By boxplot we can compare the points gained by position.

```
players<-read.csv("players.csv")
boxplot(made~position,data=players,
xlab="Position",ylab="Points gained",
main="Scoring by position")
```

Box plot

Similarly like the other types of the plots, we can modify the outlook of the plot.

We illustrate colouring 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)
```

Box plot

Setting the logical variable `horizontal` to `TRUE` we can rotate the boxes in the boxplot.

Moreover, the colours can vary from box to box

```
boxplot(made~position,data=players,
xlab="Position",ylab="Points gained",
main="Scoring by position",
col="col=c("blue","cyan","green"),
varwidth=TRUE,horizontal=TRUE)
```

Q-Q plot

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.

Q-Q plot

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.

Q-Q plot

In R, we have two functions to create Q-Q plots:

`qqnorm()` creates normal Q-Q plot (means the suggested theoretical distribution to be normal),

`qqplot()` function allows us to create a Q-Q plot to compare two datasets.

With `qqnorm()` is joined `qqline()` that produces line to a "theoretical", by default normal, quantile-quantile plot which passes through the 'probs' quantiles, by default the first and third quartiles.

Q-Q plot illustration

At first, we generate some sample from normal distribution

In the next step we compare it with theoretical distribution

```
x<-rnorm(100,mean=10,sd=1)
qqnorm(x)
qqline(x,col="steelblue",lwd=2)
```

Q-Q plot illustration

To illustrate the situation, when sample does not come from the supposed distribution, we generate the sample from exponential distribution.

```
x<-rexp(100,rate=1/10)
qqnorm(x)
qqline(x,col="steelblue",lwd=2)
```

Q-Q plot illustration

In order to compare, if two random samples come from the same distribution type, we will generate two vectors `x` and `y`

Then we apply the function `qqplot()` on these samples.

```
x<-rnorm(100,mean=10,sd=1)
y<-rnorm(100,mean=5,sd=3)
qqplot(x,y,main="Q-Q plot for two samples")
```

Q-Q plot illustration

`qqplot()` function does not cooperate with the `qqline()`

To add the auxiliary straight line, we use `abline()` function together with the `sort()` function.

```
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 function `lm()` produces the linear dependence model and provides the coefficients necessary to plot the straight line.

Q-Q plot

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 the same length as the given sample and then we use this vector as the second dataset entering into the `qqplot()` function.

```
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)
```

More graphs in one plot

In base R we can combine the graph 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.

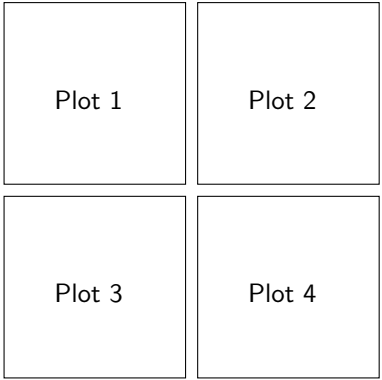
The decision of which graphical parameter we should use depends on how do we want our plots to be arranged:

- `mfrow` the plots will be arranged by rows,
- `mfcol` the plots will be arranged by columns.

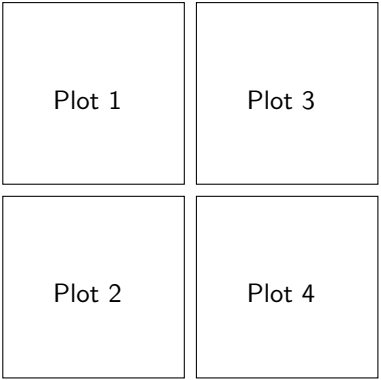
This setting is used as argument of the `par()` function, that defines parameters of the graphical device

More graphs in one plot

`mfrow=c(2,2)`



`mcol=c(2,2)`



More graphs in one plot–illustration

```
set.seed(5)
x <- rexp(80)
# Two rows, two columns
par(mfrow = c(2, 2))
# Plots
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))
```

More graphs in one plot–more complex structure

Frequently 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 centimetres with `lcm()` function.
- `heights` a vector for the height of the columns. You can also specify them in centimetres 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.

More graphs in one plot–more complex structure

We can preview a layout making use of the `layout.show()` function before adding the plots.

```
l <- layout(matrix(c(1,2,2,# First, second,
                     3,3,4), # third, fourth plot
            nrow=2,
            ncol=3,
            byrow=TRUE))
layout.show(l)
```

More graphs in one plot–more complex structure

We illustrate this method on the scatter plot accomplished with the marginals in the form of histogram and box plot.

```
 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")
```

R programming for statistics

VI. Descriptive sample characteristics

The mean

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
```

The mean

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 (??) for calculating the sample mean to the form:

$$\overline{x} = \frac{x_1 \cdot n_1 + x_2 \cdot n_2 + \cdots + x_k \cdot n_k}{n_1 + n_2 + \cdots + n_k} = \frac{\sum_{i=1}^{k} x_i \cdot n_i}{\sum_{i=1}^{k} n_i}. \tag{2}$$

where x_i's denote the values of the variable and n_i denotes the absolute frequency of x_i.

The mean

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 (2), it is necessary to verify that both vectors have the same length.

The mean

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 mean

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 their 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
```

The median

The function `median()` is implemented to state the median in the R environment.

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.

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

$$\mathbb{P}(X < x) \leq \frac{k}{q}. \tag{3}$$

Quantiles

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.

```
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
>
```

Quantiles

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`.

Quantiles

We illustrate it on finding the deciles of the daily increases

```
> 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
>
```

The variation range

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.

In order to express the variation range as a single value by definition (**??**), we use the `max()` and `min()` functions.

The variation range

Here is illustration of the 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

Here is implemented the function `IQR()` in the R language.

```
> x<-c(5,10,12,4,16,8,9)
> IQR(x)
[1] 4.5
>
```

Mean absolute deviation

Here is implemented the function `mad()` in the R language.

```
> x<-c(5,10,12,4,16,8,9)
> mad(x)
[1] 4.4478
>
```

Variance and standard deviation

We must use be the `var()` and `sd()` functions carefully.

Their results are an unbiased estimates of the variance and standard deviation of the whole population.

If we want to compute the sampling variance according to the relation (**??**), we have to define our own function, which we illustrate in the following source code.

Variance and standard deviation

```
> 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
```

Coefficient of variation

The coefficient of variation has got no implementation among the function in R.

We can compute it using the existing functions or we can define new function.

```
> cv<-function(x) variance(x)/mean(x) * 100
> cv(x)
[1] 157.5893
```

Skewness and kurtosis

To compute the skewness and kurtosis in R we need the `moments` package.

In this package are defined the functions `skewness()` and `kurtosis()`.

```
> library(moments)
> skewness(x)
[1] 0.3598295
> kurtosis(x)
[1] 2.252963
>
```

R programming for statistics

VII. Parameter estimates

Point estimates

Methods

In this course 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 $X_1, \ldots, X_n$ from the distribution that depends on the vector of parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$.

Confidence intervals

Example

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.

Confidence intervals

Solution: As the first step we calculate the unbiased point estimate of the probability p as $\hat{p} = \frac{98}{250}$ 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.

Confidence intervals

```
> 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.

Thanks for your attention.

R programming for statistics

Aleš Kozubík