I. Introduction to R

Table of Contents

1. Introduction 2
1.1. The R Graphical user interface (GUI) and RStudio 2
1.2. Starting an R session 5
2. Assigning values in R 9
2.1. Creating objects with basic data types 9
2.2. Atomic vectors 13
2.3. Arrays 18
3. Generating sequences in R 23
4. Writing data into file 25
5. R packages 26
6. References 27

1. Introduction

1.1. The R Graphical user interface (GUI) and RStudio

The R programming language is a useful tool – among others – to analyze time series and fields, to conduct trend analysis, hypothesis testing, principal component analysis, and cluster analysis. In R data can be easily visualized.

R is an open-source programming language which was created in the 1990s by Ross Ihaka and Robert Gentleman for statistical analysis and for visualization of their results (Ihaka & Gentleman, 1996).

Installation of R (RGui, R graphical user interface):

The last stable version of R (version 4.1.2) was published on 1 November 2021 which is available for Windows, Linux and Mac OS X on the following link: Comprehensive R Archive Network (CRAN). (If Windows 10 is used, it is recommended to download the latest 64-bit version of R: `R-4.1.2-win.exe`. On the website mentioned above *Download R for Windows*, then *install R for the first time* shall be selected.)

The RGui can be easily installed by using the default settings. Desktop shortcut can be created by choosing *Select Additional Tasks*, then *Create desktop shortcut* during the installation. If we install it in the following folder: `C:\Program Files\R`, then the RGui can be launched by the file `RGui.exe` which is available in the folder: `C:\Program Files\R\R-4.1.2\bin\x64` (see in the left-hand side of Figure 1).

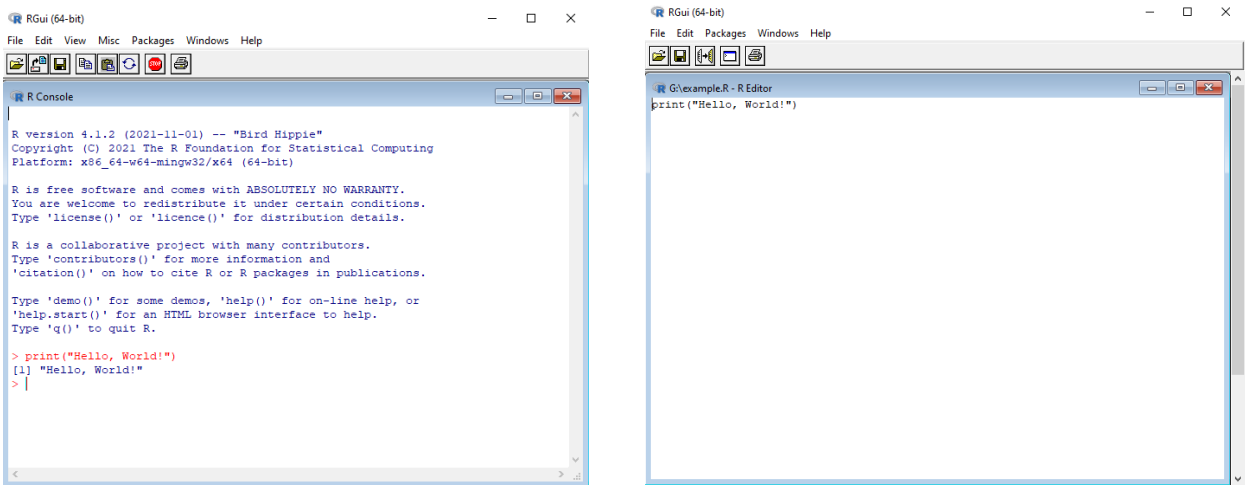


Figure 1. Console window and script editor of the RGui.

Installation of RStudio (integrated development environment, IDE):

In the course the RStudio will be used which facilitates our work. Free version of RStudio can be downloaded from this website. (It is recommended to download `RStudio-2021.09.2-382.exe` in Windows 10.) The RStudio can also be easily installed by using the default settings.

Please note that, RStudio cannot be installed without the installation of the R (RGui). If RStudio is installed in the following folder: `C:\Program Files\RStudio`, then it can be launched by the file `rstudio.exe` which is available in the folder: `C:\Program Files\RStudio\bin`.

In RStudio a new script file can be opened by choosing `File/New File/R Script`. After this, we got four panels in the RStudio (see in Figure 2).

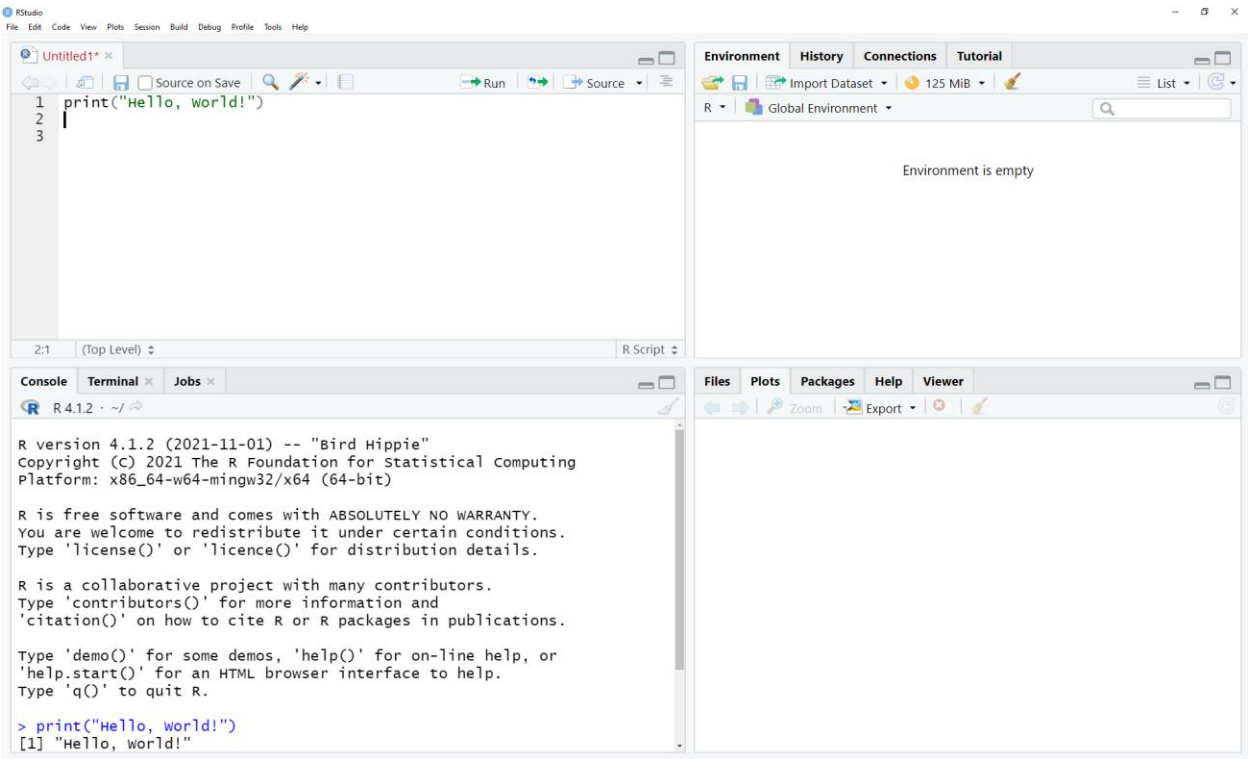


Figure 2. View of the four panels in RStudio.

Please note that in some cases (for example, in case of developing a new R package) it is necessary to run Rtools which can be downloaded from this website. After the installation, instructions on this website shall be followed.) In this course we will not use Rtools.

Remarks on programming paradigms:

There are different classes of programming paradigms, e.g., procedural, object-oriented, and declarative languages.

Procedural languages:	**Object-oriented languages:**	**Declarative languages:**
Fortran	Python	SQL
C	Java	HTML
Pascal	PHP	
	C++	
	Ruby	
	R	
	ABAP	

In **object-oriented languages**, data belong to a *class* and operations can be applied on them by using *functions and methods*. In the classes there are *instances* which have the same *attributes*. For example, there is the class of the `students`. Every student has an `age`. Let us create an instance of the class: `Peter(18)`. There are procedures – known as methods – which belong to the class. These methods can be applied on the instances of the given class. For example, every student can `learn`. If we want Peter to start to learn then the command is: `Peter.learn`. If we create a subclass in a class, then the methods of the original class can also be applied on the instances of the subclass. (Because of the previously mentioned, object-oriented programming is useful to create computer games.) Classes and methods are the elements of languages e.g., Python and Java. R can be programmed without classes and methods in the S3 object system, however, data still can belong to objects. For example, a statistical linear regression model created in R can be considered as an object. We will see that the regression parameters, the fitted values and the residuals of a linear model are belong to the same `lm`-type object[1].

In **procedural programming languages**, there are variables on which procedures can be applied. There are no classes, no methods and no objects. It is important to note that, we did not state that object-oriented programming languages are more advanced/better than procedural programming languages. For example, a group of the function in R are written in Fortran and C.

In **declarative programming languages** we declare the properties of the desired result. For example, in the Structured Query Language (SQL) – which is a relational database management system – we only declare the data structure of the result. We do not define the order of the procedures. Its syntax is SELECT database FROM * table WHERE condition. (In case of functional programming languages to which the above-mentioned programming language belong, the order of the procedure is important.)

In this course, the R S3 object system is suitable for us to read two- or multidimensional arrays (objects), visualizing their data and applying simple operations on those objects.

[1] It is worth mentioning that R can be programmed as object-oriented language in the same sense as Java when the S4 object system is used. In S4 we have to define a class and its attributes (representations).

1.2. Starting an R session and basic concepts of R

The **R session** starts when we open RStudio and it finishes when we close RStudio. During the R session objects (for example numeric vectors, arrays, lists) are created and operations are applied on them. The **R workspace** contains these objects which includes a **working directory**. It means that if any file (e.g., txt, png) is created in R, then it is saved in the working directory according to the default settings.

In RStudio **commands** can be entered in the *script window* (see topleft in Figure 2), and in the *console window* (R Console; see bottomleft in Figure 2). To run the code, in the script window we go to the line of the code and push the Run button, or push Ctrl+ENTER. In the console window, commands can be directly written, then it runs when we push ENTER.

All **objects** that are created during the R session are listed in the Global Environment (see topright in Figure 2 in the Environment window).

In R lots of **functions** are available in R. Their syntax is: `function(parameter)`. For example, we can print the following text "Hello, World!" with the `print` function.

```
print(x="Hello, World!")
[1] "Hello, World!"
```

Note that, the `print` function has parameters. The text or object which content is printed on the screen is given after the parameter `x`. However, the name of the parameter can be omitted, therefore, the following code will also work:

```
print("Hello, World!")
```

Note that texts are always given between quotation marks or apostrophes.

There are functions which can be used without parameters, for example, `citation` and `quit`:

```
citation()
quit()
```

Information about a function can be retrieved by using function `help`. This is shown in the bottomright panel of the RStudio.

For example:

```
help(print)
```

The following command also works:

```
?print
```

We can open the HTML version of R's online documentation with the following command:

```
help.start()
```

Documentation can also be found in the directory of the installed R.
For example, *`C:\Program Files\R\R-4.1.2\bin\x64\doc\manual`*.

Note that, the previously listed functions (e.g., `print`, `quit`, `help`, `help.start`) are built-in functions which means that those functions are available just after the installation of the R. Those are in automatically insalled R libraries such as `base` and `utils`.

Lots of functions are available in R libraries – called as **R packages** – which can be installed after the installation of the R (see Chapter 5; e.g., later in the semester we will use the function `image.plot` from the R package `field` to create maps with cartesian projection).

Of course, we can define our own functions. (Example will be shown later in the semester.)

Files with specific extensions in R:

For the purpose to write reusable codes, commands written in the script window can be saved into .R script files in the RStudio by choosing the Saves as… option in the File menu. (The extension shall be given in the file title, e.g., test.R.)

The R files can open in RStudio by choosing the Open File… option in the File menu.

The codes of the .R script can be loaded and run with the source function:

```
source("test.R")
```

Note that, the code written previously works if the `test.R` file is in the working directory. We can define the path (i.e. other folder) of the .R file by adding it to the file title as follows:

```
source(file="C:/folder/test.R")
```

We can edit these files not only in RStudio/RGui but also in any word composer, for example in NotePad or NotePad++.

It is beneficial to add comments to our codes which help to understand our commands. Lines of comments begins with one or more hash marks (#).

Objects created during the R session will be deleted after closing the session without storing them. One or more objects can be stored in .RData binary files. We can create an RData file by using function `save.image` in which the title of the RData can be given:

```
save.image(file="filetitle.RData")
```

With the previously written code, the RData file will be created in the working directory. We can define the path of the RData file by adding it to the file title as follows:

```
save.image(file="C:/folder/filetitle.RData")
```

The RData file can be loaded by the load function:

```
load("filetitle.RData") # if the file is in the working directory
load("C:/folder/filetitle.RData") # if the file path is given
```

A single object can be stored in RDS binary files with the `saveRDS` function:
```
saveRDS(object=objectname, file="filetitle.RDS")
```

The RDS file can be loaded with the `readRDS` function:
```
readRDS(file=filetitle.RDS)
```

In the R Console previously executed commands can be retrieved by using the numeric pad of the keyboard with up and down arrows. The cursor can be positioned in the current line by using left and right arrows. Commands can also be stored in a specific R files, namely into .Rhistory files. We can create/load an RHistory file by using function `savehistory/loadhistory`:
```
savehistory("fileName.Rhistory")
loadhistory("fileName.Rhistory")
```

In the `saveRDS`, `readRDS`, `savehistory` and `loadhistory` functions, the file path can be given similarly to the functions `source`, `save.image` and `load`.

Changing the path of the working directory:

Path of the working directory can be retrieved by the function `getwd`:
```
getwd()
```

A custom working directory can be choosen by using function `setwd`. Its syntax is: `setwd(dir="path")`
where path is e.g., `C:/folder/`

Files stored in the working directory can be listed by using function `list.files`:
```
list.files()
```

If there is no file in the working directory, then `character(0)` is printed on the screen.

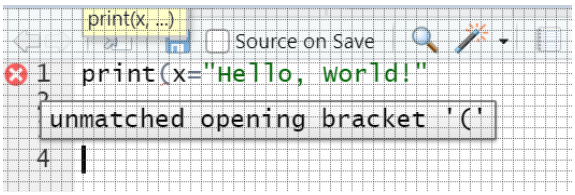
The list of objects can be retrieved by `ls` or `objects` functions:
```
ls()
character(0)
```

Objects can be removed by function `rm`. All objects can be removed by the following command:
```
rm(list=ls())
```

In this case we used the combination of two functions: `rm` and `ls`. With `ls()` we make a list of the names of the objects which were created during the R session. Then, we executed the function `rm` to remove all objects.

Remarks:

- In the R Console the lines starts with the prompt symbol > . If the expression is incomplete (for example, we use the print function, but we miss the closing parenthesis), then the prompt character changes to +. In this case, if we push the ESC button then we get back the > symbol.
- The R is case-sensitive, so we can store objects with the names `a` and `A` at the same time.
- The decimal separator is dot by default.
- R can handle accent marks which usually appear above a character (e.g., á, ä, é) but it is advised to avoid them.
- In the script window white x in a red circle appears before the lines of codes which contains error. An example can be seen below:



Keyboard shortcuts:

In the followings some keyboards shortcuts are listed which facilitate the work in RStudio.

- Selected lines in the script window can be commented out simultaneously by pressing Ctrl+Shift+c.
- All lines of codes in the script window can be selected by Ctrl+a. As it was mentioned before, the codes can be run by Ctrl+ENTER or by pushing the Run button.
- The R script can be saved by Ctrl+s.
- The R Console can be cleared by pressing Ctrl+l.
- In the script window and in the R Console copy and paste works with Ctrl+c and Ctrl+v.

In the followings we create various types of objects on which arithmetic operations are performed.

2. Assigning values in R

2.1. Creating objects with basic data types

In R, the basic data types are **integer, numeric, complex, logical** and **character (string)**. Based on these elements, numerical- or character-valued vectors, and arrays can be created. Objects with more complexity can also be defined such as **data frame** which is a table containing columns with different types of data. However, each column shall have the same length. **List** is another complex data structure in R, where list slices can contain different types of data (e.g., a string, a logical vector, and a multidimensional numeric array). The **lm**-type data or the **ncdf4**-type data are considered as more complex objects. In the latter the metadata of a multidimensional NetCDF-array is stored. This file format is widely used in meteorology and will be discussed later in the semester.

First, we create objects with a single value which is numeric or logical or character (in Chapter 2.1), then we create objects with multiple elements, i.e., vectors (in Chapter 2.2) and arrays (in Chapter 2.3). Basic operations on the objects are exemplified by exercises.

Exercise 1. Create an empty folder (`test`) in the Windows File Explorer and set it as the working directory in the current R session! Then, check whether the folder is empty. Check it if there are any objects in the global environment. If it is not empty, then remove the objects. Please use the following R functions: `setwd`, `getwd`, `list.files`, `ls`, `rm`.

Arithmetic operations:
addition: **+**
subtraction: **-**
multiplication: ***
division: **/**
exponentiation: ****** or **^**
If the base is `x` and the exponent (power) is `y` then `x**y` or `x^y`.

Order of the operations can be changed by using parentheses `()`.

Exercise 2. Form the sum of 5 and 8!

```
5+8
```

The result is:

```
[1] 13
```

The result is showed after the line of the executed command. `[1]` means that the result is a row vector which only has one component.

The `sum` function can also be used for addition:

```
sum(5,8)
[1] 13
```

Without assigning the values to a name, the R will show the result of an operator, but it is not stored to the global environment. Because of this, the result is not reusable.

Values can be assigned to a name as follows:

```
objectName (identifier) <- value
value -> objectName (identifier)
objectName (identifier) = value
```

Using spaces before and after the operators **<-** **->** **=** are not obligatory.

Remember that the R is case-sensitive therefore, for example the names `object` and `OBJECT` can be assigned to values at the same time.

Exercise 3. Store 10 and 20 as objects then print them on the screen!

```
object <- 10
OBJECT <- 20
object
[1] 10
OBJECT
[1] 20
```

In this case, the function `print` was not required to print the values in the screen.

Names of the objects can contain dots and underscores. Writing in *camelCase* style is also used in R. Special characters, for example percent signs or spaces shall be avoided.

Reserved words in R:

Every programming language has reserved words that cannot be used as identifiers such as the name of functions.

Exercise 4. Get the list of the reserved words of R! Explain the meaning of the following reserved words!

List of reserved words can be seen by using the command: `?reserved`

TRUE and FALSE are logical constants.

Inf means „Infinity”. For example, when a number or a logical constant is divided by 0.

NaN stands for „Not a Number”. For example, when 0 is divided by 0.

NA stands for „Not Available”, which means missing values.

NULL means an empty object.

Built-in constants:

In R there are only a few built-in constants, which are the followings:

`pi`: The value of pi: 3,141593 which is the ratio of the circumference of a circle and its dimeter.

`letters`: The 26 lower-case letters of the Roman alphabet.

`LETTERS`: The 26 upper-case letters of the Roman alphabet.

`month.name`: Name of each month in English.

`month.abb`: Abbreviatied name of each month in English.

Other constants can be defined by the user, for example with functions (see Exercise 5).

Exercise 5. Store as object `e` the Euler's number by using the `exp` function! Euler's number is the value of the exponential function where the argument is 1.

```
e <- exp(1)
e
[1] 2.718282
```

Exercise 6. Store the results of the following operations: 1+2 and 2*5. The former shall be stored as `x` while the latter as `y`. Then, divide `x` by `y` and store the ratio as object `z`. Print object `z` on the screen!

```
x <- 1+2
y <- 2*5
z <- x/y
z
[1] 0.3
```

Here four commands were given in four lines. Each command was evaluated after pressing ENTER. If we insert commands into braces/curly brackets `{...}` – which is called a **code block** – then commands are evaluated after the closing brace. In cases of loops (e.g., `for`, `while`) or `if` statements shall be given in blocks. Note that after the opening brace, the prompt sign changes from > to +.

```
{x <- 1+2
y <- 2*5
z <- x/y
}
z
[1] 0.3
```

Exercise 7. Calculate the 9th power of 3 in two ways!

```
3**9
[1] 19683

3^9
[1] 19683
```

Modulo operation and integer division:

If we divide `x` by `y`, then modulo operation gives the remainder (in other words the fractional part) of the division. Integer division means that if we divide x by y, then the remainder is discarded.

Exercise 8. Calculate the modulo of 9/2 then make integer division!

```
9%%2
[1] 1

9%/%2
[1] 4
)
```

Type and dimension of objects:

Type and dimension of an object can be checked by using functions `class` and `dim`, respectively.

```
class(x=objectName)
dim(x=objectName)
```

Exercise 9. Check the type of the objects `x`, `y`, and `z` by using the function `class`!

```
class(x)
[1] "numeric"

class(y)
[1] "numeric"

class(z)
[1] "numeric"
```

Objects `x`, `y` are integer, while z is real number. Their type is in R `numeric`.

Exercise 10. Create the objects `plotName` and `True` which types are character and logical, respectively. Henceforth, the type of the former is called as a string. The assigning values are: "My_first_plot.png" and TRUE.

```
plotName <- "My_first_plot.png"
True <- TRUE

class(plotName)
[1] "character"

class(True)
[1] "logical"
```

We repeat it again, a text – which is a string – is always given between quotation marks ("**. . .**") or apostrophes ('**. . .**').

2.2. Atomic vectors

Values or objects which have the same type can be concatenated with the function `c`. It can be used to create vectors of numeric (and integer), logical or string-type values. Its syntax is the following:

```
vector1 <- c(value1, value2, value3, ...)
vector2 <- c(objectName1, objectName2, ...)
```

For example, the function `c` can be used to create numeric vectors.

Note that, a vector can only contain the same type of values. For example, we cannot define vector with both logical and numeric values.

A single component of a vector can be selected (in other words: retrieved) by using squared brackets `[…]` just after the name of the vector. In the brackets the index of the selected value shall be given. For example, the 1st component of `vector1` can be selected as follows:

```
vector1[1]
```

Exercise 11. Which type has the vector which contains a string (e.g., apple) as the 1st component and an integer (e.g., 9) as the 2nd component?

```
vector3 <- c("apple",9)
class(vector3)
[1] "character"
```

Check the type of the 1st and 2nd components, respectively:

```
class(vector3[1])
class(vector3[2])
```

This means that 9 is not considered as a number in R. Therefore, arithmetic operation cannot be applied on the 2nd component of the vector. For example, the following code will produce the error: „non-numeric argument to binary operator".

```
vector3[2] * 2
```

Numeric vectors:

Exercise 12. Create two numeric vectors with the numbers 1,2,3,4,5 and 5,4,3,2,1, respectively! Store them in objects `a` and `b` then check their types with the function `class`!

```
a <- c(1,2,3,4,5)
b <- c(5,4,3,2,1)
a
[1]  1  2  3  4  5
b
[1] 5  4  3  2  1
class(a)
[1] "numeric"
class(b)
[1] "numeric"
```

The previously stored objects are arithmetic sequences where the common difference is 1 and -1, respectively. In R, these sequences also can be created by using colon.

```
a2 <- 1:5
b2 <- 5:1
a2
[1] 1 2 3 4 5
b2
[1] 5 4 3 2 1
```

As we mentioned the function `c` can be used to concatenate objects also.

Exercise 13. Concatenate objects `a2` and `b2` but do not store the results as an object!

```
c(a2,b2)
[1] 1 2 3 4 5 5 4 3 2 1
```

The magnitude of a numeric vector can be changed by adding a number to it or multiplying it with a number. Consequently, **arithmetic operations can also be applied on numeric vectors.**

Exercise 14. Create the arithmetic sequence 1,2,3,4,5,6,7,8,9,10. Then transform it into the arithmetic sequence 0,1,2,3,4,5,6,7,8,9 and store it as object `c`! Please note that `c` is not a reserved constant therefore, it can be used as an object name.

```
c <- 1:10-1
c
[1] 0 1 2 3 4 5 6 7 8 9
```

Scalar multiplication:

Exercise 15. Multiply (divide) the numeric vector `a` (`b`) with 2!

```
a*2
[1]  2  4  6  8 10
b/2
[1] 2.5 2.0 1.5 1.0 0.5
```

Other operations with numeric vectors:

Exercise 16.

Add each component of `a` to each component of `a2`!

```
a+a2
[1]  2  4  6  8 10
```

Subtract each component of `b` from each component of `a`!

```
a-b
[1] -4 -2  0  2  4
```

Multiply each component of `a` with each component of `b`!

```
a*b
[1] 5 8 9 8 5
```

Check the length of vectors `a` and `c` with function `length`! What happens when the components of vectors `a` and `c` with different lengths are added?

```
length(a)
[1] 5

length(c)
[1] 10

a+c
[1]  1  3  5  7  9  6  8 10 12 14
```

The components of the vector with smaller length will be used again.

Dot and cross products:

Dot product can be formed by using the operator %*%.

Exercise 17. Make the dot product of numeric vectors `a` and `b`!

```
dotProd <- a%*%b
dotProd
      [,1]
[1,]   35
```

Check the type of the dot product with function `class`!

```
class(dotProd)
[1] "matrix"
```

The result is a scalar value however, it is stored as a matrix. It can be transformed to a scalar (numeric) by using functions `drop` or `as.numeric`.

Transform object `dotProd` with function `as.numeric`!

```
as.numeric(dotProd)
[1] 35

class(as.numeric(dotProd))
[1] "numeric"
```

Component(s) of a vector can be retrieved by using brackets: `[]`.

Examples of the syntax:

```
objectName[indexOfComponent]
objectName[c(indexOfComponent, indexOfComponent)]
objectName[indexOfComponent:indexOfComponent]
```

Component(s) can also be omitted from a vector.
Examples of the syntax:

```
objectName[-indexOfComponent]
objectName[c(indexOfComponent, indexOfComponent)]
objectName[-(indexOfComponent:indexOfComponent)]
```

Exercise 18. Retrieve the 5th component of vector `a` then the 1st and 4th components of vector `b`, finally the components between the 2nd and 8th components of vector `c`!

```
a[5]
[1] 5

b[c(1,4)]
[1] 5 2

c[2:8]
[1] 1 2 3 4 5 6 7
```

Empty and null vectors:
An empty vector can be created by the function `c`.

```
null <- c()
null
NULL
```

A vector which has the value 0 can also be created by the function `c`.

```
null <- c(0)
```

Assume that the result of a for loop is a numeric vector. In this case, an empty / null vector shall be defined before the loop to store the results. For example: add 1,2,3,4,5 to the vector `a`!

```
result <- c(0)
for (i in 1:5) {
     result[i] <- a[i]+i
}
result
[1]  2  4  6  8 10
```

String vectors:

Exercise 19. Create a string vector with the function `c` with the following elements: apple, tangerine, peach, and orange.

```
fruits <- c("apple","tangerine","peach","orange")
fruits
[1] "apple" "tangerine" "peach" "orange"
```

Check the type of the object `fruits`.

```
class(fruits)
[1] "character"
```

Logical vectors: These vectors contains the reserved words TRUE or FALSE.

Exercise 20. Create a logical vector with the following elements: FALSE, TRUE, TRUE.

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

What type of object can we get if we concatenate FALSE, TRUE, TRUE inserting them into quotation marks?

2.3. Arrays

Arrays can be filled with numeric, string or logical values but each array shall contain elements with the same type. Rows/columns of two-dimensional arrays can be considered as vectors. Vectors in each row (column) shall have the same length.

Two-dimensional arrays can be created with function `matrix`:

```
matrix(objectName or values, nrow=rowNumber, ncol=columnNumber,
byrow=FALSE or TRUE)
```

Two- or higher dimensional arrays can be created with function `array`:

```
array(objectName or values, dim=c(dimension_1, dimension_2, ...,
dimension_n))
```

Two-dimensional arrays:

Exercise 21. Create two two-dimensional arrays with functions `array` and `matrix`, respectively! Array `A` shall contain 1, 2, …, 100 and array `B` shall contain 100, 99, …, 1.

```
A <- array(1:100, dim=c(10,10))
A
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1   11   21   31   41   51   61   71   81    91
 [2,]    2   12   22   32   42   52   62   72   82    92
 [3,]    3   13   23   33   43   53   63   73   83    93
 [4,]    4   14   24   34   44   54   64   74   84    94
 [5,]    5   15   25   35   45   55   65   75   85    95
 [6,]    6   16   26   36   46   56   66   76   86    96
 [7,]    7   17   27   37   47   57   67   77   87    97
 [8,]    8   18   28   38   48   58   68   78   88    98
 [9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100
```

```
B <- matrix(100:1, nrow=10, ncol=10)
B
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]  100   90   80   70   60   50   40   30   20    10
 [2,]   99   89   79   69   59   49   39   29   19     9
 [3,]   98   88   78   68   58   48   38   28   18     8
 [4,]   97   87   77   67   57   47   37   27   17     7
 [5,]   96   86   76   66   56   46   36   26   16     6
 [6,]   95   85   75   65   55   45   35   25   15     5
 [7,]   94   84   74   64   54   44   34   24   14     4
 [8,]   93   83   73   63   53   43   33   23   13     3
 [9,]   92   82   72   62   52   42   32   22   12     2
[10,]   91   81   71   61   51   41   31   21   11     1
```

Check the arrays! The first column of the arrays are filled with data at first, then the second column etc. If we use the function `matrix` then parameter `byrow=TRUE` makes it possible to fill the first row with data at first, then the second row etc.

Check the type of the arrays, respectively!

```
class(A)
[1] "matrix"
class(B)
[1] "matrix"
```

If the product of the numbers of rows and columns are smaller than the numbers of the data then the smaller number of data will be stored. For example:

```
C <- matrix(1:100, nrow=2, ncol=5)
C
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrix operations:

It is possible to add/multiply elements of different arrays.

Exercise 22. Multiply the elements of matrix `A` with 2 (a scalar)!

```
D <- matrix(2*1:100, nrow=10, ncol=10) # or
D <- 2*A
D
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    2   22   42   62   82  102  122  142  162   182
 [2,]    4   24   44   64   84  104  124  144  164   184
 [3,]    6   26   46   66   86  106  126  146  166   186
 [4,]    8   28   48   68   88  108  128  148  168   188
 [5,]   10   30   50   70   90  110  130  150  170   190
 [6,]   12   32   52   72   92  112  132  152  172   192
 [7,]   14   34   54   74   94  114  134  154  174   194
 [8,]   16   36   56   76   96  116  136  156  176   196
 [9,]   18   38   58   78   98  118  138  158  178   198
[10,]   20   40   60   80  100  120  140  160  180   200
```

Exercise 23. Transpose matrix `A`!

```
t(A)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1    2    3    4    5    6    7    8    9    10
 [2,]   11   12   13   14   15   16   17   18   19    20
 [3,]   21   22   23   24   25   26   27   28   29    30
 [4,]   31   32   33   34   35   36   37   38   39    40
 [5,]   41   42   43   44   45   46   47   48   49    50
 [6,]   51   52   53   54   55   56   57   58   59    60
 [7,]   61   62   63   64   65   66   67   68   69    70
 [8,]   71   72   73   74   75   76   77   78   79    80
 [9,]   81   82   83   84   85   86   87   88   89    90
[10,]   91   92   93   94   95   96   97   98   99   100
```

Exercise 24. Form the matrix product of `A` and `B`!

```
A %*% B
```

```
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8] [,9] [,10]
 [1,] 43105 38505 33905 29305 24705 20105 15505 10905 6305  1705
 [2,] 44060 39360 34660 29960 25260 20560 15860 11160 6460  1760
 [3,] 45015 40215 35415 30615 25815 21015 16215 11415 6615  1815
 [4,] 45970 41070 36170 31270 26370 21470 16570 11670 6770  1870
 [5,] 46925 41925 36925 31925 26925 21925 16925 11925 6925  1925
 [6,] 47880 42780 37680 32580 27480 22380 17280 12180 7080  1980
 [7,] 48835 43635 38435 33235 28035 22835 17635 12435 7235  2035
 [8,] 49790 44490 39190 33890 28590 23290 17990 12690 7390  2090
 [9,] 50745 45345 39945 34545 29145 23745 18345 12945 7545  2145
[10,] 51700 46200 40700 35200 29700 24200 18700 13200 7700  2200
```

Exercise 25. Invert matrix `E` with the function `solve`! Matrix `E` shall be defined as follows.

```
E <- matrix(c(4,3,2,1), nrow=2, ncol=2)
E
```

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

```
solve(E)
```

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

Rows/columns/elements of a matrix can be retrieved by using brackets: `[]`.

```
objectName[indexOfRow,]
objectName[,indexOfColumn]
objectName[indexOfRow,indexOfColumn]
```

Rows/columns of a matrix can also be omitted by using brackets: `[]`.

```
objectName[-indexOfRow,]
objectName[,-indexOfColumn]
```

Retrieving a part of an array will be useful in meteorological data processing.

Exercise 26. Retrieve the first column/row/element of matrix `D`! After that, retrieve the common elements of the 2nd and 3rd rows as well as the 2nd, 3rd and 4th columns!

```
D[,1]
[1]  2  4  6  8 10 12 14 16 18 20

D[1,]
[1]   2  22  42  62  82 102 122 142 162 182

D[1,1]
[1] 2

D[2:3,2:4]
     [,1] [,2] [,3]
[1,]   24   44   64
[2,]   26   46   66
```

Creating matrices by concatenating vectors:

Vectors can be concatenated to matrices and a matrix can be concatenated with another matrix by using functions `cbind` and `rbind`.

Let us to see some example! The syntax when a matrix and a column of another matrix is concatenated to create a new matrix is the following:

```
newObjectName <- cbind(objectName1, objectName2[,number of column])
```

The syntax when a matrix and a row of another matrix is concatenated to create a new matrix is the following:

```
newObjectName <- rbind(objectName1, objectName2[number of row,])
```

Exercise 27. Create a new matrix (`F`) by concatenating the first row of matrix `A` and the second row of matrix `B`!

```
F <- rbind(A[1,], B[2,])
F
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   11   21   31   41   51   61   71   81    91
[2,]   99   89   79   69   59   49   39   29   19     9
```

The rows of matrices `A` and `B` shall have the same lengths.

Multidimensional arrays:

Function `array` can be used to create three- or higher dimensional – in other words: multidimensional – matrices.

Exercise 28. Create a three-dimensional array (`array1`) which has 10x10x10 elements. Fill the array with natural numbers from 1 to 1000!

```
array1 <- array(1:1000, dim=(c(10,10,10)))
```

Check the type and dimension of the array!

```
class(tomb)
[1] "array"

dim(tomb)
[1] 10 10 10
```

Meteorological data are mainly stored in multidimensional arrays. Usually, the dimensions are geographical longitude, geographical latitude, pressure level and time.

The previously created array can be considered as an object that contains ten data tables which are consisted of 10 rows and 10 columns. Print the content of 1st and 10th data tables!

```
array1[,,1]
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    1   11   21   31   41   51   61   71   81    91
 [2,]    2   12   22   32   42   52   62   72   82    92
 [3,]    3   13   23   33   43   53   63   73   83    93
 [4,]    4   14   24   34   44   54   64   74   84    94
 [5,]    5   15   25   35   45   55   65   75   85    95
 [6,]    6   16   26   36   46   56   66   76   86    96
 [7,]    7   17   27   37   47   57   67   77   87    97
 [8,]    8   18   28   38   48   58   68   78   88    98
 [9,]    9   19   29   39   49   59   69   79   89    99
[10,]   10   20   30   40   50   60   70   80   90   100

array1[,,10]
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]  901  911  921  931  941  951  961  971  981   991
 [2,]  902  912  922  932  942  952  962  972  982   992
 [3,]  903  913  923  933  943  953  963  973  983   993
 [4,]  904  914  924  934  944  954  964  974  984   994
 [5,]  905  915  925  935  945  955  965  975  985   995
 [6,]  906  916  926  936  946  956  966  976  986   996
 [7,]  907  917  927  937  947  957  967  977  987   997
 [8,]  908  918  928  938  948  958  968  978  988   998
 [9,]  909  919  929  939  949  959  969  979  989   999
[10,]  910  920  930  940  950  960  970  980  990  1000
```

For example, an empty three-dimensional array which has 3x4x5 elements can be created as follows.

```
array(dim=(c(3,4,5)))
```

3. Generating sequences in R

Arithmetic sequences can be created by using function `seq`:

The syntax of the function is the following.

```
seq(from=start value, to=end value, by=increment of the sequence)
```

Exercise 29. Create an arithmetic sequence in the interval [1,20] with the increment of 2 and store it as `sequence1`!

```
sequence1 <- (seq(1,20,2))
sequence1
[1]  1  3  5  7  9 11 13 15 17 19
```

Exercise 30. Create an arithmetic sequence in the interval [20,1] with the increment of -2 and store it as `sequence2`!

```
sequence2 <- (seq(20,1,-2))
sequence2
 [1] 20 18 16 14 12 10  8  6  4  2
```

The order of the sequence can be reversed with function `rev`.

Exercise 31. Create an arithmetic sequence in the interval [100,0] with an increment of 10 by using the `rev` function and store as `sequence3`!

```
sequence3 <- rev(seq(0,100,10))
sequence3
[1]  100  90  80  70  60  50  40  30  20  10   0
```

Function `rep` can be used to repeat a sequence:

The syntax of the function is the following.

```
rep(sequence, times=number of times to repeat the sequence or each=number
of times to repeat each element of the sequence)
```

Exercise 32. Create a sequence in the interval [1,5] with an increment of 1 and store it as `sequence4`! After that, repeat it two times and print it on the screen!

```
sequence4 <- 1:5
sequence4
[1] 1 2 3 4 5
```

```
rep(sequence4, times=2)
[1] 1 2 3 4 5 1 2 3 4 5
```

The previously written commends can be combined in one command as follows.

```
sequence4 <- rep(1:5, times=2)
sequence4
[1] 1 2 3 4 5 1 2 3 4 5
```

Exercise 33. Repeat each element of `sequence4`!

```
rep(sequence4, each=2)
[1] 1 1 2 2 3 3 4 4 5 5 1 1 2 2 3 3 4 4 5 5
```

4. Writing data into file

Data can be written into file by usng function `cat`.

Exercise 34. Write the following text into a txt file: "Hello, world"! The name of the txt file shall be test.txt.

```
cat("Hello", file="test.txt")
cat(" world", file="test.txt", append=TRUE)
```

Line break can also be defined in the `cat` function by using the following parameter: `sep="\n"`.

Parameter `append=TRUE` serves to write data into existing files. (That was the case in our example.)

Data can also be written into file by using function `write.table`.

```
write.table(objectName, "fileName.txt", sep="", na="NA",
dec=",", col.names=TRUE or FALSE, row.names=TRUE or FALSE)
```

`objectName`: The name (identifier) of the object which content shall be written into file.
`"fileName.txt"`: The name of the file.

If the object is a data table then its columns can be separated by using parameter `sep`. Some examples are presented:

`sep=" "` – separation with spaces (default setting)
`sep="\t"` – separation with tabulators
`sep=","` – separation with commas
`sep=";"` – separation with semicolons

Decimal separator can be changed by using parameter `dec`.

`col.names`: TRUE means that column names will be added to the file.
`row.names`: TRUE means that row names will be added to the file.

(The default settings are TRUE in both cases.)

Objects can also be saved in prn or csv formats. We prefer the latter with commas or semicolons as separator.

Exercise 35. Print the content of matrix `A` with column names into the file matrix.csv. The separator shall be semicolon.

```
A[5,10] <- -10.2
write.table(A, "matrix.csv ", sep=";", row.names=FALSE)
```

5. R packages

Besides built-in R functions, additional functions are also available in R libraries (packages). The list of packages that can be installed are available in RStudio in `Tools/Install Packages...` where we can choose the package which we want to install. It is worth mentioning that there are packages, which are installed automatically with the installation of R. (For example, package sp which provides statistical methods to analyse spatial data, or base, or utils.)

Packages can be installed in custom folder with the following command:

```
install.packages("package", lib="path")
```

The packages can also be downloaded on the following homepage: https://cran.r-project.org/web/packages/. Older versions of the packages are available on the following homepage: https://cran.r-project.org/src/contrib/Archive/.

If a new R session is opened, the required packages must be loaded with the command below:

```
library(package) # or
require(package)
```

We can close the package with the following command:

```
detach_package(package)
```

Packages that will be used in the course are the followings:

To handle files in NetCDF format (functions `nc_open, ncvar_get, nc_close`):
ncdf4: https://cran.r-project.org/web/packages/ncdf4/index.html

To create maps with country borders (function `map`):
maps: https://cran.r-project.org/web/packages/maps/index.html

To create x-y plots with colorbar (function `image.plot`):
fields: https://cran.r-project.org/web/packages/fields/index.html

To add colorscale to x-y plots (functions `colorRampPalette, brewer.pal`):
RColorBrewer: https://cran.r-project.org/web/packages/RColorBrewer/index.html

Functions of an installed packages can be used without load the whole library. For example, function `image.plot` can be used as follows:

```
fields::image.plot(...)
```

6. References

Abari, K. (2013): Introduction to the R (Bevezetés az R-be in Hungarian) http://psycho.unideb.hu/munkatarsak/abari_kalman/szamitastechnika_II/bevezetes_az_R_be_2008_04.pdf

Ihaka, R., Gentleman, R. (1996): R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics, 5(3): 299–314. https://doi.org/10.2307/1390807

R Core Team (2021). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

Solymosi, N. (2005): Introduction to the usage of the R language and environment (Bevezetés az R-nyelv és környezet használatába in Hungarian) https://cran.r-project.org/doc/contrib/Solymosi-Rjegyzet.pdf

Documentation of R functions on the following homepage: https://www.rdocumentation.org/.