Network Modeling - HS 2022

Introduction to `R`

C. Stadtfeld, A. Espinosa-Rada, A. Uzaheta
Early versions by K. Mepham, V. Amati

This is not a course to learn `R`, it just is an introduction that is supposed to create the basic understanding of `R`. There are a lot of excellent online courses available. Recommended are, for example, the following:

https://r4ds.had.co.nz/
http://www.statmethods.net/
http://www.burns-stat.com/pages/Tutor/hints_R_begin.html
http://data.princeton.edu/R/gettingStarted.html
https://adv-r.hadley.nz/

You can go to any of these sites to learn the basics of R or refresh your knowledge.

What is R?

`R` is a language and environment for statistical computing and graphics. It is a GNU project providing a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, …) and graphical techniques, and it is highly extensible (source http://www.r-project.org/).

RStudio is a convenient interface consisting of four windows (locations according the default setup):

- editor window (upper left): collections and commands are edited and saved
- console (lower left): where R computes and execute the commands
- environment (upper right): shows which data and values R has in its memory
- files/plots/packages/help window(lower right): you can open files, view plots (also previous plots), install and load packages or use the help function.

Rscripts

`R` scripts are files containing the sequence of commands that should be executed.

The Rscript `RIntro.R` contains the commands that are illustrated in this introduction. You can load the file in RStudio (File -> open file) and run the commands. To do this you select the commands and click on the icon Run in the upper right corner of the editor window. You can also use the keyboard shortcut Ctrl+Enter for Windows and Cmd+Return for Mac.

R as a calculator

The program R can be used as a calculator. Standard arithmetic operators are `+`, `-`, `*`, and `/` for add, subtract, multiply and divide, and `^` for exponentiation. Mathematical functions, such as `sqrt`, `exp`, and `log` are also defined and compute the square root, exponential and logarithm of a number.

```
# R as a calculator
5 + 8
```

```
## [1] 13
```

```
15^2
```

```
## [1] 225
```

```
exp(3)
```

```
## [1] 20.08554
```

```
log(1)
```

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

```
sqrt(4)
```

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

The relational operators `<=`, `<`, `==`, `>`, `>=` and `!=` for less than or equal, less than, equal, greater than, greater than or equal, and not equal can be used to create logical expressions that take values `TRUE` and `FALSE`. `TRUE` and `FALSE` are binary statements and are internally evaluated as 1 and 0, respectively.

```
5 > 3
```

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

```
a <- 4
b <- 6
a == b
```

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

```
(a == b) * 1 # FALSE is evaluated as 0
```

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

```
a <= b
```

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

```
(a <= b) * 1 # TRUE is evaluated as 1
```

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

Data structures

Everything in R is an object. An object is a data structure having some characteristics (attributes) and methods which act on its attributes. The most frequently used objects are vectors, matrices, array, and lists.

To assign a value to an object we use the code `name <- value`. The symbol `<-` is the assignment operator, read as "gets". R now accepts the equal sign as well as an assignment operator. A valid name consists of letters, numbers, and the dot or underscore characters. R is case sensitive which means that capital and small letters make differences.

Vectors

Vectors are the simplest objects in R. A `vector` is defined as an ordered collection of numbers or characters. Different functions allow creating a `vector`.

- The function `c()`, short for concatenate, is used to create vectors from scalars or other vectors. Since `c()` is the function that concatenates values, please do not use `c` as a name of an R object.
- The colon operator `:` is used to generate a sequence of integers.
- The `seq()` function creates a sequence given the starting and stopping points (arguments `from =` and `to =`) and an increment (argument `by =`).
- The rep function is used for repeat or replicate elements a certain number of times
  - argument `times =` giving the number of times to repeat the whole `vector`.
  - argument `length.out =` with the desire length of the output `vector`.
  - argument `each =` repeating `each` times each element of the `vector`.

The following lines demonstrate the use of these functions.

```
# Function c
vec0 <- c(2, 3, 4)
vec0
```

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

```
# WARNING:
# do not use c as a name of an R object!!!

Vec0
```

```
## Error in eval(expr, envir, enclos): object 'Vec0' not found
```

```
# WARNING:
# R is case sensitive!!! Therefore, Vec0 is different from vec0

# Colon operator
vec1 <- 1:3
vec1
```

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

```r
# The seq function
vec2 <- seq(from = 2, to = 5, by = 0.5)
vec2
```

```
## [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
# The rep function
vec3 <- rep(1:3, times = 2)
vec3
```

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

```r
# Character vector
vec4 <- c("a", "b", "c")
vec4
```

```
## [1] "a" "b" "c"
```

`R` operations are vectorized. Arithmetic and relational operators work element by element. When computing an operation (e.g., the sum) between two vectors of different length, the shorter one will be repeated until it has the same length as the longer one. This rule is referred to as the recycling rule.

```r
vec0 + vec1
```

```
## [1] 3 5 7
```

```r
log(vec1)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```r
length(vec1)
```

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

```r
sum(vec1)
```

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

```r
max(vec1)
```

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

```r
min(vec1)
```

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

```r
# The recycling rule
vec0 + vec3
```

```
## [1] 3 5 7 3 5 7
```

To extract specific elements from a `vector` we use one of the subsetting operators. The square brackets are used for this, the subscript of the elements wanted is placed inside them.

```r
vec2[1]
```

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

```r
vec2[c(1,3)]
```

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

Matrix

A `matrix` is a generalization of a `vector` and defined as an ordered collection of elements having all the same nature (usually numbers). We can create matrices using:

- the function `matrix()`, with arguments the values of the cells (`data =`), number of rows (`nrow =`), number of columns (`ncol =`), and whether the matrix should be filled by row or by column (`byrow =`, default value `FALSE`)
- the functions `rbind()` and `cbind()` to bind vectors by row or by column. *Note:* `rbind()` and `cbind()` have different methods depending on the class of the object. They work in `vector`, `matrix` or `data.frame` objects, returning an objects of the same class and applying coercion when needed it.

The following lines demonstrate the use of these methods.

```r
# The function matrix
mat1 <- matrix(1:10, nrow = 2, ncol = 5, byrow = FALSE)
mat1
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```r
mat2 <- matrix(1:10, nrow = 2, ncol = 5, byrow = TRUE)
mat2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

```r
# Functions rbind and cbind
mat3 <- rbind(vec0, vec1)
mat3
```

```
##      [,1] [,2] [,3]
## vec0    2    3    4
## vec1    1    2    3
```

```
# or as column vectors
mat4 <- cbind(vec0, vec1)
mat4
```

```
##      vec0 vec1
## [1,]    2    1
## [2,]    3    2
## [3,]    4    3
```

```
# Number of rows and columns
dim(mat1)
```

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

The following lines illustrate how to compute some basic operations with matrices and how to extract elements from it.

```
mat1 * mat2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   15   28   45
## [2,]   12   28   48   72  100
```

```
# WARNING: the operator * computes the element-wise (Hadamard) product!!!
#          The operator for the matrix product is %*%
mat5 <- mat1 %*% t(mat2)
mat5
```

```
##      [,1] [,2]
## [1,]   95  220
## [2,]  110  260
```

```
solve(mat5) # Calculates the inverse
```

```
##       [,1]  [,2]
## [1,]  0.52 -0.44
## [2,] -0.22  0.19
```

```
det(mat5) # Calculates the determinant
```

```
## [1] 500
```

```
t(mat1) # Calculates the transpose
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    7    8
## [5,]    9   10
```

```
# WARNING:
# t() is the function that computes the transpose of a matrix.
# Do not use t as the name of an R object!!!
```

The elements of a matrix may be exatracted using the row and column subscripts in square brackets, separated by a comma.

```
mat1
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```
mat1[1, 2]      # single cell
```

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

```
mat1[1,]       # first raw
```

```
## [1] 1 3 5 7 9
```

```
mat1[, 1]       # first column
```

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

```
mat1[-1, -1]    # A negative pair results in a minor matrix
```

```
## [1]  4  6  8 10
```

```
              # where a column and a row are omitted.
```

Array

An `array` is a collection of matrices having the same number of rows and columns. An `array` can be created using the function `array()` with arguments the values of the cells (`data =`), and a vector specifying the number of rows, columns and matrices (`dim =`).

```
arr0 <- array(1:48, dim = c(4, 4, 3))
arr0
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   21   25   29
## [2,]   18   22   26   30
## [3,]   19   23   27   31
## [4,]   20   24   28   32
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   33   37   41   45
## [2,]   34   38   42   46
## [3,]   35   39   43   47
## [4,]   36   40   44   48
```

```
arr1 <- array(c(mat1, mat2), dim = c(2, 5, 2))
arr2 <- array(c(mat1, mat2, mat4), dim = c(2, 5, 3))
```

The elements of an **array** are extracted using the row, column and matrix subscripts in square brackets, separated by a comma. Arrays can stored multidimensional objects beyond collections of matrices (3-dim), therefore the dimensions don't have an specified name in general. Here, the names row, column and matrix are specific for the arrays that we are using.

```
# How could you address the first matrix of arr1?
arr1[, , 1]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

```
# How could you address the first row of each matrix in arr1?
arr1[1, , ]
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    3    2
## [3,]    5    3
## [4,]    7    4
## [5,]    9    5
```

```
# How could you address the first column of each matrix in arr1?
arr1[, 1 , ]
```

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

Data frame

A data frame extends the matrix object. It can have columns of different data type and can be thought of the usual case by variable data matrix. Data frames can be created using the function `data.frame` with arguments the variables that constitute the column of the object.

```
# Creating data frames
peopleid <- 1:10
gender <- rep(c("Male", "Female", NA), times = c(4, 5, 1))
# data frame can also handle missing values that are usually coded as NA (not available)
set.seed(1908)
age <- sample(20:30, 10)
age
```

```
##  [1] 25 26 21 20 22 24 27 29 30 28
```

```
height <- rnorm(n = 10, mean = 165, sd = 5) |> ceiling()
dat1 <- data.frame(id = peopleid, gender = gender, age = age, height = height)

# Inspecting a data frame
head(dat1) # shows the first six lines
```

```
##   id gender age height
## 1  1   Male  25    167
## 2  2   Male  26    162
## 3  3   Male  21    162
## 4  4   Male  20    155
## 5  5 Female  22    154
## 6  6 Female  24    159
```

```
dim(dat1) # returns the number of rows and columns
```

```
## [1] 10  4
```

```
names(dat1) # shows the names of the columns
```

```
## [1] "id"     "gender" "age"    "height"
```

```
summary(dat1) # summary statistics for each column
```

```
##        id           gender               age            height     
##  Min.   : 1.00   Length:10          Min.   :20.00   Min.   :154.0  
##  1st Qu.: 3.25   Class :character   1st Qu.:22.50   1st Qu.:159.8  
##  Median : 5.50   Mode  :character   Median :25.50   Median :164.0  
##  Mean   : 5.50                      Mean   :25.20   Mean   :163.7  
##  3rd Qu.: 7.75                      3rd Qu.:27.75   3rd Qu.:167.0  
##  Max.   :10.00                      Max.   :30.00   Max.   :175.0
```

```r
# The elements of a data.frame can be addressed as in a matrix
dat1[1, 2] # single cell
```

```
## [1] "Male"
```

```r
dat1[1, ] # first raw
```

```
##   id gender age height
## 1  1   Male  25    167
```

```r
dat1[, 1] # first column
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
# OBSERVATION: name of the columns can be used to address columns
dat1$id
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
dat1[, "id"]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
# How could you extract id and age variables?
dat1[, c("id", "age")]
```

```
##    id age
## 1   1  25
## 2   2  26
## 3   3  21
## 4   4  20
## 5   5  22
## 6   6  24
## 7   7  27
## 8   8  29
## 9   9  30
## 10 10  28
```

Lists

A `list` is an object consisting of an ordered collection of objects which might have different nature, i.e. they might be vectors, matrices, array or lists as well. Many `R` functions return the results in a list and therefore it is important to understand how lists work. To extract elements of a list is possible to use another subsetting operator, the double square bracket. The difference between the simple square bracket and the double square bracket is preserving vs. simplifying.

> "Simplifying subsets returns the simplest possible data structure that can represent the output, and is useful interactively because it usually gives you what you want. Preserving subsetting keeps the structure of the output the same as the input, and is generally better for programming because the result will always be the same type." from Advance R by Hadley Wickham.

```
list0 <- list(vec0,mat1,dat1)
str(list0)
```

```
## List of 3
##  $ : num [1:3] 2 3 4
##  $ : int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
##  $ :'data.frame':    10 obs. of  4 variables:
##   ..$ id    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   ..$ gender: chr [1:10] "Male" "Male" "Male" "Male" ...
##   ..$ age   : int [1:10] 25 26 21 20 22 24 27 29 30 28
##   ..$ height: num [1:10] 167 162 162 155 154 159 167 166 175 170
```

```
list0[[1]]
```

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

```
list0[[3]][1:3,]
```

```
##   id gender age height
## 1  1   Male  25    167
## 2  2   Male  26    162
## 3  3   Male  21    162
```

Object structure and coercion

If we have an object `x` and we would like to know what it is, we can use the following functions:

```
x <- mat2
is.vector(x)
```

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

```
is.matrix(x)
```

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

```
is.data.frame(x)
```

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

```
str(x)
```

```
##  int [1:2, 1:5] 1 6 2 7 3 8 4 9 5 10
```

```
class(x)
```

```
## [1] "matrix" "array"
```

We can also coerce objects into particular structures

```
y <- as.vector(mat2)
z <- as.matrix(y) # Vector are coerced into column matrix

u <- as.data.frame(mat2)
u
```

```
##   V1 V2 V3 V4 V5
## 1  1  2  3  4  5
## 2  6  7  8  9 10
```

```
# OBSERVATION: R automatically creates column names
# when using the function as.data.frame
# To change these column names we can use the function names

names(u) <- c("a","b","c","d","e")
u
```

```
##   a b c d  e
## 1 1 2 3 4  5
## 2 6 7 8 9 10
```

Control flow statements

From Advance R.

> There are two primary tools of control flow: choices and loops. Choices like `if` statements and `switch()` calls, allows you to run different code depending on the input. Loops, like `for` and `while`, allow you to repeatedly run code, typically with changing options.

Conditional statements

The `if` statement allows to run a block of code between curly brackets if the test expression that gives the condition is evaluated to the logical value `TRUE`. It's worth to mention that the if statement expects a test expression of length one. That's why the longer form of the logical operators (`&&` and `||`) is preferred. See ``?`&&` `` for details.

```
# if (condition: logical value) {expression to do}

if (a  > b && b > 1) {
  print("a is bigger than b")
  c <- a + b
} else if (b > a) {
  print("a is bigger than b")
  c <- b - a
} else
  print("a is equal to b")
```

```
## [1] "a is bigger than b"
```

```
# the longer form of the logical operators (&&, ||) is preferred for
# programming control-flow and typically use in `if` clauses

# vectorised if
# ifelse(condition, when TRUE, when FALSE)
vec2Modified <- ifelse(vec2 > 3, vec2 + 2, vec2 * 2)
vec2Modified
```

```
## [1] 4.0 5.0 6.0 5.5 6.0 6.5 7.0
```

The `ifelse()` function is the vectorized version of the conditional statement. It's useful to replace values in a vector given a condition.

Loops

The for and while construction perform loops in R.

```
# for (i in set of values) {expression to do}

# For example, we can use a for loop to compute the sum of the values
# in each row of the matrix mat2
for (i in 1:nrow(mat2)) {
  x <- sum(mat2[i,])
  print(x)}
```

```
## [1] 15
## [1] 40
```

```
# Other loop construction:
# while (condition) {expression}
# repeat {expression}
```

Function definition

In the previous lines, we have encountered a few functions. We can also create our own functions. To specify a function we use the following code:

```
name <- function(arguments) {
  body of the function
}
```

```
# We can create a function sumrow that computes the sum of each row
# in a matrix x and return them in a vector called rowsum

sumrow <- function(x){
  rowsum <- NULL
  for (i in 1:nrow(x)) {
    rowsum[i] <- sum(x[i,])
  }
  return(rowsum)}

sumrow(mat2)
```

```
## [1] 15 40
```

It's worth to mention that this is just an illustration on how to create a function from scratch. The sum of the row value problem had different solutions in `R` that doesn't require to create a new function or using `for` loops. For this particular problem the `rowSums()` function does the trick. In general, it's better to use vectorized versions of the operations and avoid the use of `for` loops when heavy computation are required.

The `apply` family

Base `R` is equipped with a huge variety of function to perform different tasks. The `apply` family of functions is a useful toolbox to solve specific problems without the need to use `for` loops. The `apply()` function apply a function to an array for an specific dimension(s). For example, it may be used to compute the sum by row in a matrix. The `lapply()` function apply a function to a list. `sapply()`, `vapply()` and `replicate()` are wrappers versions of the `lapply()` for common tasks. Finally, the `tapply()` function is useful to compute summary statistics by levels of a `factor` or a `character` vector.

```
# apply a function to an array in a given dimension
apply(X = mat2, MARGIN = 1, FUN = sum)
```

```
## [1] 15 40
```

```
# apply a function to a vector by each level of a factor variable
tapply(X = dat1$age, INDEX = dat1$gender, FUN = mean)
```

```
## Female   Male
##   26.4   23.0
```

```
# sum the rows of a numeric matrix-like for each level of a grouping variable
rowsum(x = dat1[, c("age", "height")], group = dat1$gender)
```

```
## Warning in rowsum.data.frame(x = dat1[, c("age", "height")], group =
## dat1$gender): missing values for 'group'
```

```
##        age height
## Female 132    821
## Male    92    646
## <NA>    28    170
```

```
# apply a function to each component of a list
lapply(X = list0, FUN = typeof)
```

```
## [[1]]
## [1] "double"
##
```

```
## [[2]]
## [1] "integer"
##
## [[3]]
## [1] "list"
```

```
  # same with a simplify output lapply(x, f, simplify = "array")
sapply(X = list0, FUN = typeof)
```

```
## [1] "double"  "integer" "list"
```

```
# other functions of the apply family
# vapply(), replicate(), mapply()
```

Help

There are different ways to ask for help in R.

- When you do not know if there is already a function doing what you would like to do, you can search the answer using the internet.
- When you know the name of the function and you would like to know more, you can type `?name of the function` or `??name of the function` to get a description of the function, its arguments and examples.
- When you do not know the function but know some keywords that are in the documentation, you can type `help.search("keyword")`. It would return the name of functions and their respective packages where the keyword is presented. It only search in installed packages.

```
?matrix
?rowSums

help.search("regression")
```

Working with data

Setting the working directory

Setting the working directory corresponds to specifying the path of the folder in which data are and results will be saved. This avoids repeating the path every time we are loading a new file. By default `R` will set the working directory in a folder whose path can be retrieved using the command `getwd()` (getting working directory). To change the working directory into the desired folder, we use the function `setwd()` (set the working directory) and we specify the path of the selected folder as an argument of the function.

Since the backslash character has a special meaning to `R`, we have to either double-up the backslash (escape the backslash as special character) or use the forward slashes instead. If you are using RStudio as the `R` IDE, it sets the working directory for you to the project folder by default. This behavior is trigger by projects created by RStudio.

We can look at the files within our working directory with the command `list.files()` or use the Files pane in RStudio.

```r
getwd()
setwd("C:/Users/spadmin/Desktop/workshopSAOM")

list.files() # Listing the files in the working directory
```

Reading and writing data

Data contained in table format files can be read into `R` using the function `read.table()`. The outcome of this function is a data frame. We can also write data into table format files using the function `write.table()`. Other functions are available for files with particular extensions (e.g. from SAS or SPSS. Packages `foreign` and `readr`). The functions `read.csv()` and `write.csv()`, and `read.csv2()` and `write.csv2()` are used for comma and semicolon separated value files.

```r
# We export the data frame dat1 in a txt file comma separated
# ?write.table
write.table(dat1, "dat1.txt", sep = ",", row.names = FALSE, col.names = TRUE)

# # For tab spacing you should use  sep = "\t"
#
# We import data using the function read.table.
# ? read.table
dataImport <- read.table("dat1.txt", sep = ",", header = TRUE)
str(dataImport)
```

```
## 'data.frame':    10 obs. of  4 variables:
##  $ id    : int  1 2 3 4 5 6 7 8 9 10
##  $ gender: chr  "Male" "Male" "Male" "Male" ...
##  $ age   : int  25 26 21 20 22 24 27 29 30 28
##  $ height: int  167 162 162 155 154 159 167 166 175 170
```

Saving the workspace and objects to a file

`R` objects exist during your session but are deleted when you exit. However, you will be asked if you want to save an image of your workspace before you leave. You can save the workspace throughout the session using the command `save.image()` and load it with the command `load()`.

```r
save.image("Intro.RData")
load("Intro.RData")
```

`R` also offer a way to save single objects with the function `saveRDS()`, and read them back with the function `readRDS()`.

```r
saveRDS(dataImport, "dataImport.RDS")
dataImport <- readRDS("dataImport.RDS")
```

R projects

RStudio offers the possibility to create `R` projects. When you create a project the working directory is by default set to the project directory and the workspace is automatically saved into that folder. When you open a Rproject:

- a new R session (process) is started
- The scripts, `.RData` and `.Rhistory` files in the project's main directory are loaded (if project options indicate that it should be loaded)

- the working directory is set to the project directory.

To create a Rproject in Rstudio click on File -> New project

R packages

`R` is free software. When you install it, a series of functions and data sets are installed. Those functions and data sets are collected into libraries or packages. An R package is a collection of functions and data sets improving existing base R functions or adding new ones.

To install and load packages the commands `install.packages("name of the package")` and `library("name of the package")` are used.