

Introduction to Programming using R

Organizational Matters

Lecturers:

- Tim Mensinger (tim.mensinger@uni-bonn.de)
- Moritz Brinker (s3mobrin@uni-bonn.de)
- Florian Schoner (florian.schoner@uni-bonn.de)

Timetable:

- Monday - Thursday, 9am - 4pm; Friday, 9am - 2pm
- Morning: Lectures and presentation of solutions
- Lunchbreak: 12-1pm
- Afternoon: Supervised learning

Preliminaries

Before starting the class make sure to install the required software. We will be using the programming language R [R Core Team, 2019] and the development environment RStudio [RStudio Team, 2015]. We recommend first installing R and then RStudio (desktop).¹

¹ In case you have problems with the installation process press [here](#).

- R: <https://cran.rstudio.com>
- RStudio: <https://rstudio.com/products/rstudio/download>

1 Introduction

“[...] computers and mathematics are like beer and potato chips: two fine tastes that are best enjoyed together. Mathematics provides the foundations of our models and of the algorithms we use to solve them. Computers are the engines that run these algorithms.” – [Stachurski, 2009]

IN THIS SECTION we will present the very basics of R. We will go through some arithmetic, variables, special numerical objects, comments and data types.²

² All statements are typed into the R console and the results are displayed after [1].

1.1 Arithmetic

```
1 + 1
```

```
[1] 2
```

```
1 - 1
```

```
[1] 0
```

Decimals:³

```
2.5 * 4
```

```
[1] 10
```

```
1 / 3
```

```
[1] 0.3333333
```

```
2 ^ 2 ^ 3
```

```
[1] 256
```

```
2 ** 2 ** 3
```

```
[1] 256
```

Parentheses:⁴

```
(2 ** 2) ** 3
```

```
[1] 64
```

1.2 Variables

```
x <- 10 + 5
```

```
x
```

```
[1] 15
```

```
x <- x + x
```

```
x
```

```
[1] 30
```

```
x ** 2
```

```
[1] 900
```

```
y <- x
```

```
y
```

```
[1] 30
```

³ Note that the decimal mark is denoted by a dot (.) and not a comma (,).

⁴ If in doubt use parentheses to ensure that R will compute the correct expression.

1.3 Special Numerical Objects

Constants, infinity and NaNs (Not a Number):

```
pi
[1] 3.141593

1/0
[1] Inf

0/0
[1] NaN
```

1.4 Data Types

- `numeric`: `x <- 1.25`
- `integer`: `x <- 1L`
- `character`: `x <- "this_works"`⁵
- `logical`: `x <- TRUE`, `y <- FALSE`
- `complex`: `x <- 1 + 2i`

⁵ In other programming languages this data type is known as a *String*.

2 Data Structures

In this section we consider the most common data structures used in R. This includes

- `list`
- (atomic) vector
- `matrix`
- `data.frame`

2.1 Lists

Lists are objects which can contain different objects of different data types.⁶

```
x <- list(1, 1.25, "this works?")
print(x)

[[1]]
[1] 1

[[2]]
[1] 1.25

[[3]]
[1] "this works?"
```

⁶ `list` is a (built-in) function which takes as argument multiple objects and combines them to a list. See section `function.print` is a (built-in) function which prints its argument on the console.

```
x <- list(x, 1 + 2i)
print(x)
```

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

```
[[1]][[2]]
[1] 1.25
```

```
[[1]][[3]]
[1] "this works?"
```

```
[[2]]
[1] 1+2i
```

```
length(x)

[1] 2
```

2.2 Vectors

Vectors are similar to lists in that they can contain multiple objects, however, any vector can contain only objects of one data type.⁷

```
x <- c(1, 2, 3)
print(x)
```

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

```
x <- c("this", "actually", "works")
print(x)
```

```
[1] "this"      "actually" "works"
```

```
x <- c("wait?", 1)
print(x)
```

```
[1] "wait?" "1"
```

```
length(x)

[1] 2
```

⁷ `c` is a (built-in) function which takes as argument multiple objects of the same data type and combines them to a vector. `c` stands for **combine**.

2.3 Indexing of Lists and Vectors

We access elements of lists and vectors via their index. If `x` is a list (or vector) we get the `i`-th element as `x[i]`.⁸ Note that for a list (or vector) of length `n` we can of course only ask for elements 1 to `n`, otherwise R returns `NA` which stands for Not Available. If we supply a vector of indices we can access more than one element, i.e. `x[c(1, 3, 5)]` will return the first, third and fifth element of `x`.

Examples:

```
x <- c(2, 4, 6, 8, 10)
print(x[1])

[1] 2

print(x[2])

[1] 4

print(x[6])

[1] NA

x[6] <- 0
print(x)

[1] 2 4 6 8 10 0

print(x[c(1, 3, 5)])

[1] 2 6 10

x[3] <- 100
print(x)

[1] 2 4 100 8 10 0

x[c(2, 4)] <- -100
print(x)

[1] 2 -100 100 -100 10 0

print(x[-1])

[1] -100 100 -100 10 0

print(x[-c(1, 2)])

[1] 100 -100 10 0
```

Useful commands:

⁸This is slightly different to many other programming languages which start indexing at 0 instead of 1.

```
x <- 1:10
print(x)

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

x <- seq(from=1, to=10, by=2)
print(x)

[1] 1 3 5 7 9

x <- seq(from=0, to=1, length.out=20)
print(x)

[1] 0.00000000 0.05263158 0.10526316 0.15789474 0.21052632 0.26315789
[7] 0.31578947 0.36842105 0.42105263 0.47368421 0.52631579 0.57894737
[13] 0.63157895 0.68421053 0.73684211 0.78947368 0.84210526 0.89473684
[19] 0.94736842 1.00000000
```

2.4 Calculating with Vectors

```
x <- 1:10
print(x)

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

y <- -4:5
print(y)

[1] -4 -3 -2 -1 0 1 2 3 4 5

print(x + y)

[1] -3 -1 1 3 5 7 9 11 13 15

print(x * y)

[1] -4 -6 -6 -4 0 6 14 24 36 50

print(x ** y)

[1] 1.000000e+00 1.250000e-01 1.111111e-01 2.500000e-01 1.000000e+00
[6] 6.000000e+00 4.900000e+01 5.120000e+02 6.561000e+03 1.000000e+05

print(2 * x)

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

print(10 + x)

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

```
print(x ** 2)
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

Recycling:

```
x <- 1:4
```

```
y <- c(1, 5, 10)
```

```
print(x + y)
```

Warning in x + y: longer object length is not a multiple of shorter object length

```
[1] 2 7 13 5
```

(Some) useful functions:⁹

```
x <- -5:5
```

```
print(x)
```

```
[1] -5 -4 -3 -2 -1 0 1 2 3 4 5
```

```
print(sum(x))
```

```
[1] 0
```

```
print(mean(x))
```

```
[1] 0
```

```
print(sd(x))
```

```
[1] 3.316625
```

```
print(var(x))
```

```
[1] 11
```

```
print(cumsum(x))
```

```
[1] -5 -9 -12 -14 -15 -15 -14 -12 -9 -5 0
```

⁹ `sum` is a (built-in) function which sums all elements of its argument (also works on matrices). `mean` computes the mean of its argument, `sd` the (unbiased) standard deviation, `var` the variance and `cumsum` the cumulative sum.

2.5 Matrices

Matrices represent two dimensional arrays which works similar to vectors in that matrices can only contain objects of a single data type.

To create a matrix we need to know how many rows and columns it should have and what data it should contain.¹⁰

¹⁰ Note that `1:x` is equivalent to `c(1,2,...,x)`. Also note `matrix(data, rows, cols)` is not equal to `matrix(data, cols, rows)`; if you do not know which argument comes when, simply ask R for help: `?matrix`.

```

data <- 1:9
rows <- 3
cols <- 3

x <- matrix(data, rows, cols)
print(x)

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

y <- matrix(data, rows, cols, byrow=TRUE)
print(y)

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

dim(x)

[1] 3 3

nrow(x)

[1] 3

ncol(x)

[1] 3

```

2.6 Combining Vectors and Matrices

```

x <- matrix(1:9, 3)
print(x)

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

y <- matrix(1:6, 2)
print(y)

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

```



```
z <- rbind(x, y)
print(z)
```

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

```
x <- cbind(1:3, 4:6)
print(x)
```

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

(Some) useful functions:¹¹

¹¹ Explain them here formally.

```
m <- matrix(1:9, 3)
print(m)
```

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

```
print(rowSums(m))
```

```
[1] 12 15 18
```

```
print(colSums(m))
```

```
[1] 6 15 24
```

```
print(rowMeans(m))
```

```
[1] 4 5 6
```

In more general settings we might wish to apply an arbitrary function to the rows or columns of a matrix. We can do this with the function `apply`.¹² Example:

```
m <- matrix(1:9, 3)

print(apply(m, 1, sd))

[1] 3 3 3

print(apply(m, 2, sd))

[1] 1 1 1
```

¹² `apply(X, MARGIN, FUN)`, `X` = matrix of interest, `MARGIN = 1` to apply the function over the rows and `2` to apply the function over the columns, `FUN` = the function of interest.

2.7 Matrix algebra

```
X <- matrix(1:9, 3)
y <- -1:1
```

```
X * y
```

```
      [,1] [,2] [,3]
[1,]   -1   -4   -7
[2,]    0    0    0
[3,]    3    6    9
```

```
X %*% y
```

```
      [,1]
[1,]     6
[2,]     6
[3,]     6
```

```
X * X
```

```
      [,1] [,2] [,3]
[1,]     1   16   49
[2,]     4   25   64
[3,]     9   36   81
```

```
X %*% X
```

```
      [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
[3,]   42   96  150
```

(Some) useful functions:¹³

¹³ Explain em.

```
t(X)
```

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

```
diag(X)
```

```
[1] 1 5 9
```

```
A <- matrix(c(1, 10, -2, 3), 2)
print(A)
```

```

      [,1] [,2]
[1,]    1  -2
[2,]   10   3

solve(A)

      [,1]      [,2]
[1,] 0.1304348 0.08695652
[2,] -0.4347826 0.04347826

b <- c(-1, 1)
solve(A, b)

[1] -0.04347826  0.47826087

```

2.8 Data Frames

Data frames represent data sets. The difference to matrices is that different columns can have different data types. Note that there are many different ways of creating a data frame.

```

x <- c("Micheal", "Sofia", "Jonah")
y <- c(1.0, 1.3, 3.0)
z <- c("a", "b", "c")

df <- data.frame(name=x, grades=y, type=z)
print(df)

  name grades type
1 Micheal  1.0   a
2  Sofia   1.3   b
3  Jonah   3.0   c

m <- matrix(1:9, 3)
df <- as.data.frame(m)
print(df)

  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9

```

The iris data set:

```

head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5         1.4         0.2  setosa

```

```

2          4.9          3.0          1.4          0.2 setosa
3          4.7          3.2          1.3          0.2 setosa
4          4.6          3.1          1.5          0.2 setosa
5          5.0          3.6          1.4          0.2 setosa
6          5.4          3.9          1.7          0.4 setosa

```

```
str(iris)
```

```

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

```

2.9 Indexing of Matrices and Data Frames

Matrices and data frames constitute two dimensional objects, this means we can ask for submatrices, columns, rows or individual elements.

```

m <- matrix(1:9, 3)
print(m)

```

```

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

```

```
print(m[1, 1])
```

```
[1] 1
```

```
print(m[1, ])
```

```
[1] 1 4 7
```

```
print(m[, 1])
```

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

```
print(m[c(1, 2), c(2, 3)])
```

```

      [,1] [,2]
[1,]    4    7
[2,]    5    8

```

When dealing with data frames we can on top access columns by their respective names.¹⁴

¹⁴ To access a column of a data frame by name use `df$column_name`. Note the different results of `df[["grade"]]` and `df["grade"]`.

```
df <- data.frame(name=c("Thomas", "Susan"), grade=c(1, 2))
print(df)
```

```
      name grade
1 Thomas     1
2 Susan      2
```

```
print(df$name)
```

```
[1] Thomas Susan
Levels: Susan Thomas
```

```
print(df[["grade"]])
```

```
[1] 1 2
```

```
print(df["grade"])
```

```
      grade
1         1
2         2
```

3 Logical Operators

IN THIS SECTION we consider logical operators which form the direct equivalent to logical operators in mathematics. We first note that we can induce boolean values by comparison via relations ($<$, $>$, $<=$, $>=$) or (in)equalities ($==$, $!=$). On boolean values we may use logical operators as *and* ($&$), *or* ($|$), but also quantifier as \exists (**any**) and \forall (**all**).

```
x <- 1:10
```

```
x < 5
```

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

```
sum(x < 5)
```

```
[1] 4
```

```
x[x < 5]
```

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

```
x[!(x < 5)]
```

```
[1] 5 6 7 8 9 10
```

```
x[(x < 3) | (x > 7)]
```

```
[1] 1 2 8 9 10
```

```
x[(x < 8) & (x > 3)]
```

```
[1] 4 5 6 7
```

```
any(x < 8)
```

```
[1] TRUE
```

```
all(x < 8)
```

```
[1] FALSE
```

Often we are interested in the (indices of the) elements of a vector (matrix) that fulfill a certain condition.

```
x <- c(3, 2, -100, 400)
```

```
which(x > 100)
```

```
[1] 4
```

4 Conditional Expressions

In many scenarios our decisions depend on the specific state of the situation. For example, *if* it rains we will take the umbrella with us. Or a little more complex. *If* it rains we will take the umbrella, otherwise, *if* we fixed the flat bike tires already we will go by bike. (We illustrate the a fictional conditional decision tree on the blackboard.) This brings us to conditional expressions.

4.1 *if*

```
x <- 10
```

```
if (x < 10) {  
  print("x is smaller than 10.")  
}
```

4.2 *else*

```
x <- 10
```

```
if (x < 10) {  
  print("x is smaller than 10.")  
} else{  
  print("x is *not* smaller than 10.")  
}
```

```
[1] "x is *not* smaller than 10."
```

4.3 *else if*

```
x <- 10
if (x < 10) {
  print("x is smaller than 10.")
} else if (x > 0) {
  print("x is between 0 and 10.")
} else {
  print("x is either smaller than 0 or bigger than 10.")
}

[1] "x is between 0 and 10."
```

4.4 *Short digression to User Input*

```
cat("Please choose which type of regression should be run:\n")
x <- readline(prompt="Linear regression (1); Polynomial regression (2): ")
x <- as.integer(x)

if (x == 1) {
  print("Okay lets do linear regression!")
} else if (x == 2) {
  print("Oh no I hate polynomial regression :)")
} else {
  print("There were only two options what did you do?")
}
```

5 *Control Flow Statements*

When working on nearly any project we often find ourselves repeating simple tasks over and over again. If this happens with tasks that cannot be managed on a computer we hire research assistants; however, if it can be done on a computer there are cheaper ways.

5.1 *For Loops*

Let's say we want to create a list with 10 entries and the i th entry is a matrix of dimension $i \times i$ filled with numbers 1 to i^2 . This can be achieved very easily with a for loop.

```
matrices <- list()
for (i in 1:10) {
  imatrix <- matrix(1:(i ** 2), nrow=i)

  matrices[[i]] <- imatrix
}
matrices[[5]]
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	6	11	16	21
[2,]	2	7	12	17	22
[3,]	3	8	13	18	23
[4,]	4	9	14	19	24
[5,]	5	10	15	20	25

5.2 Short digression to Monte Carlo simulation

Say we have two uniform random variables on $[0, 1]$, i.e. $X, Y \sim \mathcal{U}[0, 1]$. And say we want to estimate $\mathbb{P}(X + Y \in [0.75, 1.25])$ without doing any analytical mathematics. One solution to problems of this kind are so called *Monte Carlo estimates*, in which we *simulate* (in this case) two uniform random variables for many many times and each time we simply check if the sum of the realizations fulfills the statement. The frequency of times when the statement was fulfilled then approximates the probability.

```
count <- 0
nsim <- 10000
for (i in 1:nsim) {
  x <- runif(1, min=0, max=1)
  y <- runif(1, min=0, max=1)

  z <- x + y
  if (z >= 0.75 && z <= 1.25) {
    count <- count + 1
  }
}
count / nsim

[1] 0.4466

# analytical solution = 7/6 = 0.4375
```

5.3 While Loops

For loops are very useful if we know exactly how many times we need to execute some statement. If we do not know the number of repetitions before starting the loop we can use **while** loops.

Cherry picking results: Once we introduced linear models and ordinary least squares regression we will show a simple example on how to cherry pick your data such that you can claim statistical significance even if there is none.

Example:


```

userinput <- NULL
while(is.null(userinput)) {
  input <- readline("Type in a number between 0 and 10. \n")
  input <- as.integer(input)

  if (is.numeric(input)) {
    if (input >= 0 && input <= 10) {
      userinput <- input
    }
  }
}
userinput

```

6 Functions

Functions are arguably the most important building block when writing large programs. We have already seen the use of many (built-in) functions. Functions, in general, allow us to use a piece of code multiple times in a program without repeating all of the code at every instance.

6.1 A normal example

```

normaldensity <- function(x, mu, sigma) {
  constant <- 1 / sqrt(2 * pi * sigma ** 2)
  exponential <- exp(- (x - mu) ** 2 / (2 * sigma ** 2))

  return(constant * exponential)
}

normaldensity(x=0, mu=0, sigma=1)

[1] 0.3989423

normaldensity(x=1, mu=0, sigma=1)

[1] 0.2419707

standardnormaldensity <- function(x) {
  normaldensity(x, mu=0, sigma=1)
}

curve(standardnormaldensity, from=-4, to=4, xlab="x", ylab="f(x)", ylim=c(0, 0.5))

```

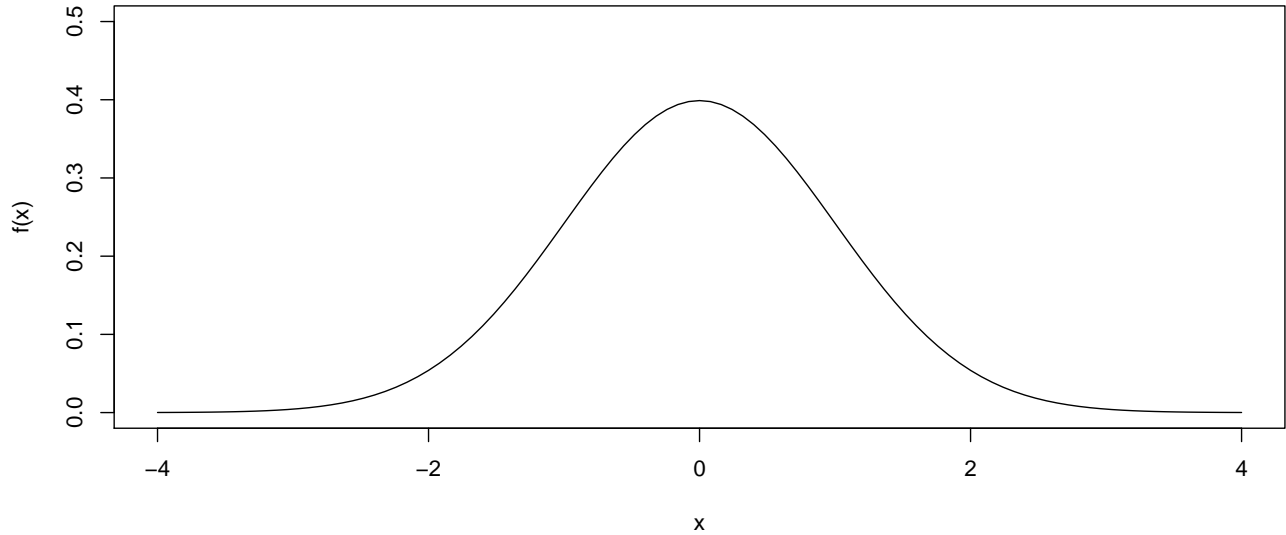


Figure 1: The standard normal density.

6.2 Recursive functions

The fibonacci sequence is defined by the following (recursive) function

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n > 1. \end{cases}$$

We can implement this function easily using an R function.

```
fibonacci <- function(n) {
  if (n == 0) {
    return(0)
  } else if (n == 1) {
    return(1)
  } else {
    return(fibonacci(n - 1) + fibonacci(n - 2))
  }
}

n <- 1:10
fib <- sapply(n, fibonacci)
rbind(n, fib)
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]
n	1	2	3	4	5	6	7	8	9	10
fib	1	1	2	3	5	8	13	21	34	55

References

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019. URL <https://www.R-project.org/>.

RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA, 2015. URL <http://www.rstudio.com/>.

John Stachurski. *Economic Dynamics: Theory and Computation*, volume 1 of *MIT Press Books*. The MIT Press, 2009. ISBN ARRAY(0x3ef60d08). URL <https://ideas.repec.org/b/mtp/titles/0262012774.html>.