Advanced Programming with **R** (session 2)

M2 Statistics and Econometrics

Thibault Laurent

Toulouse School of Economics, CNRS

Last update: 2023-09-27

Table of contents


Before starting

Packages and software versions

Install the following packages:

```
install.packages(c("ggplot2", "kableExtra", "Matrix", "microbenchma
```

And load them:

```
require("Matrix") # for presenting S4 class of object
require("microbenchmark") # comparing computational time
require("pryr") # access memory, internal R codes on github
require("Rcpp") # integrate C++ code
require("reticulate") # interface with python
```

This document has been compiled under this version of **R**

```
R.Version()$version.string
```

```
## [1] "R version 4.3.1 (2023-06-16)"
```


1. Style guide

Style guide for **R** code (1)

- There is no PEP 8 guide style as for **Python** (https://www.python.org/dev/peps/pep-0008/)
- However, one can follow Hadley Wickham recommendations in his **R** advanced book.
- An alternative is the Google **R** Style Guide
- Among these commitment, we can cite:
  - Use an appropriate file naming like

```
source("lagrangian_computation.R")
load("final_data_basis.RData")
```

  - Use an appropriate **R** objects naming like

```
reg_tree
rf_reg
```

Style guide for R code (2)

- Other recommendations among many others
  - Use spaces between operators, argument function, etc. :

```
1 + 2 + 3 + 4 == 4 * (4 + 1) / 2
a <- c(5, NA, 4, 3)
mean(a, na.rm = TRUE)
```

  - With exceptions for operators : and ::

```
1:10
stats::lm
```

  - Try to limit the number of characters per line to 80 (it can actually depend on the type of document: report, presentation, etc) and do not hesitate to break line
  - Qualify namespaces for all external functions

```
ozone.rf <- randomForest::randomForest(Ozone ~ ., data = airqual
  mtry = 3, importance = TRUE, na.action = na.omit)
```

Style guide for R code (3)

- Other recommendations among many others
  - Affectation operator: use <- instead of =

```
a <- 1
```

  - Use spaces before/after {, }, (,) and indent inside the **for**, **while**

```
x <- runif(10)
mean_sq <- 0
for (i in seq_along(x)) {
  mean_sq <- mean_sq + x[i] ^ 2
}
```

  - Same thing inside condition **if**/**else**:

```
if (a == 0) {
  log(x)
} else {
  (x ^ a - 1) / a
}
```

👩‍🎓 Training: Exercise 2.1

- Re-write the following code properly by using the recommendations seen in the section

```
my_mean=function(x)
{
# verification
if(!is.numeric(x))
  stop("x must be a numeric vector")
# initialization
n= length(x)
res =0
for(k in 1:n)
{
  res= res+x[k]
}
# return res
  return(res/ n)
}
```


2. Reminders

if/else syntax

The principle of **if/else** is the following:

```
if (<condition(s)>) {
  <instruction 1>
} else {
  <instruction 2>
}
```

For example, if a variable x is **numeric** we want to compute the mean but if it is a **character**, we want to compute the mode. The algorithm is

- **if** x is **numeric do**

  $\bar{x}$

  else if x is **character do**

  $Mode(x)$

  end for

Application

```
x <- c("F", "F", "M", "F")
if (is.numeric(x)) {
  cat("mean =", mean(x))
} else {
  if (is.character(x)) {
    cat("mode =", max(table(x)))
  }
}
```

```
## mode = 3
```

```
x <- c(10, 11, 12, 15, 13, 12)
if (is.numeric(x)) {
  cat("mean =", mean(x))
} else {
  if (is.character(x)) {
    cat("mode =", max(table(x)))
  }
}
```

```
## mean = 12.16667
```

Instruction **if** must be followed by **TRUE** ou **FALSE**

Even if we can test several conditions inside the instruction **if**, the result must be **TRUE** or **FALSE**.

Example: in the previous example, in the second situation, we allow **x** to be a **factor**. We check two conditions, but the result is still a scalar **TRUE** or **FALSE**

```
x <- factor(c("m", "m", "f", "f"))
if (is.character(x) | is.factor(x)) {
  table(x)
}
```

```
## x
## f m
## 2 2
```

Recall: | is the OR operator and **&** is the AND operator

Changing all the elements of a vector

It is possible to use a "vectorized" version of **if/else**. For example, we change the sign of the negative values of a vector of **numeric**.

```
x <- c(1, 3.4, 2, -3, -2)
```

Option 1:

```
x[x < 0] <- - x[x < 0]
```

Option 2: use *ifelse()* function

```
x <- ifelse(x < 0, -x, x)
```

Nested loop

We want to compute the income tax with respect to the value of salary:

- €0 — €15,000 23%
- €15,000 — €28,000 27%
- if > €28,000 38%

Application:

```
x <- 44000
if (x < 15000) {
  x * 0.23
} else {
  if (x >= 15000 & x < 28000) {
    15000 * 0.23 + (x - 15000) * 0.27
  } else {
    if (x >= 28000 & x < 55000) {
      15000 * 0.23 + (28000 - 15000) * 0.27 + (x - 28000) * 0.38
    }
  }
}
```

```
## [1] 13040
```

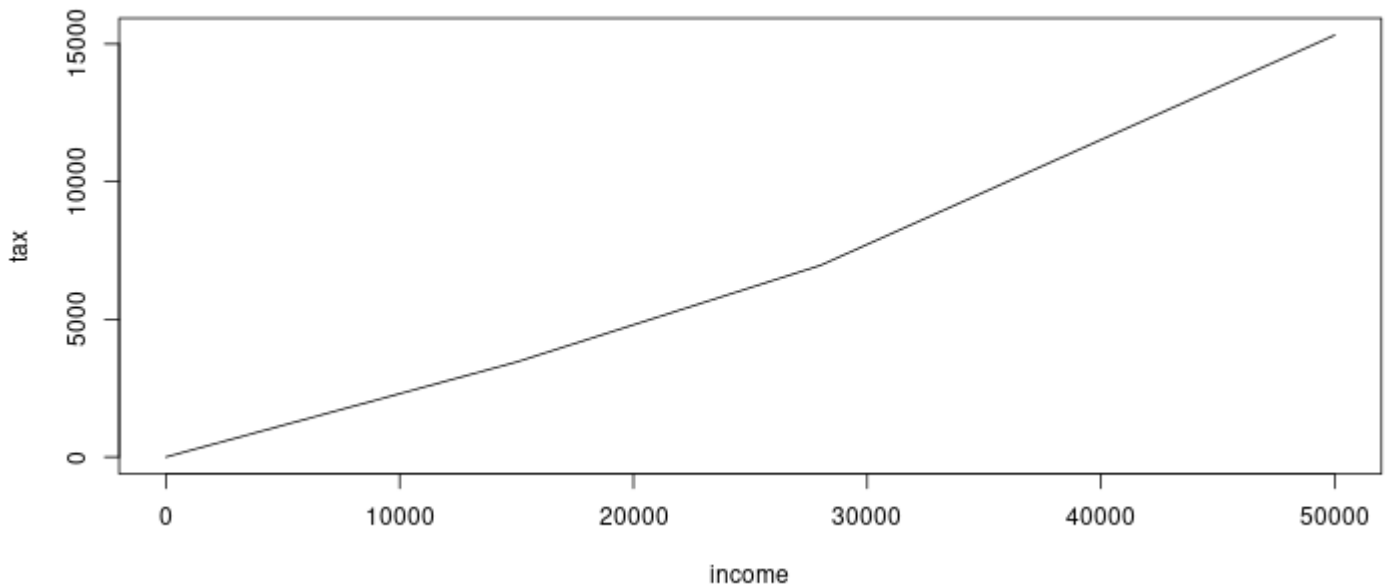
Why creating its own function?

In the previous example, instead of repeating the same code for each variable, the idea is to create a function and use it every time we want to apply it. The syntax is :

```
my_tax <- function(x) {
# verification
  stopifnot(is.numeric(x))
  # instructions
  if (x < 15000) {
    tax <- x * 0.23
    } else {
      if (x >= 15000 & x < 28000) {
        tax <- 15000 * 0.23 + (x - 15000) * 0.27
        } else {
          if (x >= 28000 & x < 55000) {
            tax <- 15000 * 0.23 + (28000 - 15000) * 0.27 + (x - 280
          }
        }
    }
  return(tax) # results
}
```

Application

```
income <- seq(0, 50000, 1000)
plot(income, sapply(income, my_tax), type = "l", ylab = "tax")
```



Remark: our function `my_tax()` is not vectorized which explains why we use `sapply()`

Global v.s. local variable

Global variables are declared outside any function, and they can be accessed (used) on any function in the program. Local variables are declared inside a function, and can be used only inside that function. It is possible to have local variables with the same name in different functions.

Example: here **beta_0** and **beta_1** are used inside *f()* but have not been defined inside, so there are choosen as global. **x** has been defined as global variable, but as it is also defined inside the function, the local is used.

```
f <- function(x) {
  return(beta_0 + beta_1 * x)
}
x <- 1
beta_0 <- 1
beta_1 <- 2
f(c(0, 1, 2))
```

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

for, while loop

We consider the vector $(x_1, \ldots, x_n)$. To compute the average mean of the vector, the formula is

$$\bar{x} = S_n/n$$

with $S_n = \sum_{i=1}^{n} x_i = x_1 + \ldots + x_n$. The computation must be done step by step:

- Step 0: Initialize $S = 0$
- Step 1: $S = S + x_1$
- Step 2: $S = S + x_2 (= x_1 + x_2)$
- Step 3: $S = S + x_3 (= x_1 + x_2 + x_3)$
- $\vdots$
- Step n: $S = S + x_n (= x_0 + \ldots + x_n)$

Algorithm to compute the mean of a vector

This can be written like this:

- **initialisation**: $S = 0$
- **for** i from 1 to n, **do**

  S = S + x[i]

  end for
- **finalization**: At the end, we divide S/n

On **R**, it can be done like this:

```
S <- 0
for (i in 1:n) {
  S <- S + x[i]
}
S <- S / n
```

Applications

- Example 1

```
x <- c(5, 10, 3, 3, 6, 9, 1, 2, 3, 1, 11, 12)
for (i in 1:length(x)) {
  cat("Value", i, "equals", x[i], "; ")
}
```

```
## Value 1 equals 5 ; Value 2 equals 10 ; Value 3 equals 3 ; Value 4 equals
```

- Example 2

```
my_mean <- 0
for (i in 1:length(x)) {
  my_mean <- my_mean + x[i]
}
my_mean / length(x)
```

```
## [1] 5.5
```

Remark: many functions are already vectorized (like function *mean()*) which allows to avoid to re-program these functions.

break and **next** reserved words

break and **next** are two reserved words. It allow user to make verification at each step i of the loop with an **if** instruction. If **break** is called during a **for** loop, it will stop the loop at step i. **next** will skip step i. Using **break** is equivalent to use **while** instruction. For example, if a vector has missing values, we skip the missing values.

```
x <- c(5, 10, 3, NA, 6, 9, 1, 2, 3, NA, 11, 12)
S <- 0
for (i in 1:length(x)) {
  if (is.na(x[i])) {
    cat("step", i, "not executed ~ ")
    next
  }
  S <- S + x[i]
  cat("step", i, "executed ~ ")
}
```

```
## step 1 executed ~ step 2 executed ~ step 3 executed ~ step 4 not executed
```

```
S
```

```
## [1] 62
```

Double Loop **for**

To browse the elements of a matrix, we need to nest two loops: one loop for the row, one loop for the column.

```
for (i in 1:3) {
  for (j in 1:4) {
    cat("i =", i, "and j =", j, ifelse(j == 4, "\n", "; "))
  }
}
```

```
## i = 1 and j = 1 ; i = 1 and j = 2 ; i = 1 and j = 3 ; i = 1 and j = 4
## i = 2 and j = 1 ; i = 2 and j = 2 ; i = 2 and j = 3 ; i = 2 and j = 4
## i = 3 and j = 1 ; i = 3 and j = 2 ; i = 3 and j = 3 ; i = 3 and j = 4
```

Application: compute the sum of the elements of the following matrix:

```
x_mat <- matrix(x, nrow = 4, ncol = 3)
S <- 0
for (i in 1:nrow(x_mat)) {
  for (j in 1:ncol(x_mat)) {
    S <- S + x_mat[i, j]
  }
}
```

while loop

Example: we want to compute the sum all the elements of a vector of size n and stop the computation if there is a missing value NA. For doing this, we need to browse the elements of the vector x. At each iteration, we check that x[i] is not a missing value and if not, we update the sum and increase i by 1, etc.

This can be written like this:

- **initialization:** $i = 1$ and $S = 0$
- **while** $i \leq n$ and x[i] $\neq$ NA **do**

  S = S + x[i]

  i = i + 1

  end while
- **finalization**: return i

while loop

Application: we consider the following vector

```
x <- c(5, 10, 3, 3, NA, 9, 1, 2, 3, 1, 11, 12)
```

```
i <- 1
S <- 0
while (i <= length(x) && !is.na(x[i])) {
  S <- S + x[i]
  i <- i + 1
}
```

👩‍🎓 Training: Exercise 2.2.a

We consider the following simulated vector of size 10000:

```
set.seed(1)
x <- rnorm(10000)
```

The algorithm to find the maximum is:

- **initialisation**: $m = x[1]$
- **for** i from 2 to n, **do**

    if (x[i] > m) **do**

      m = x[i]

    end if

  end for

Program in **R** this algorithm

👩‍🎓 Training: Exercise 2.2.b

Create a function that allows to compute the maximum for any vector of **numeric**. It must take into account the possibility that there exists some missing values.

To test your function, execute the following codes:

```
x1 <- c(1000, 10, 6, NA)
x2 <- c(NA, 1000, 10, 6)
x3 <- c(NA, NA, 1000, 10, 6)
my_max(x1)
```

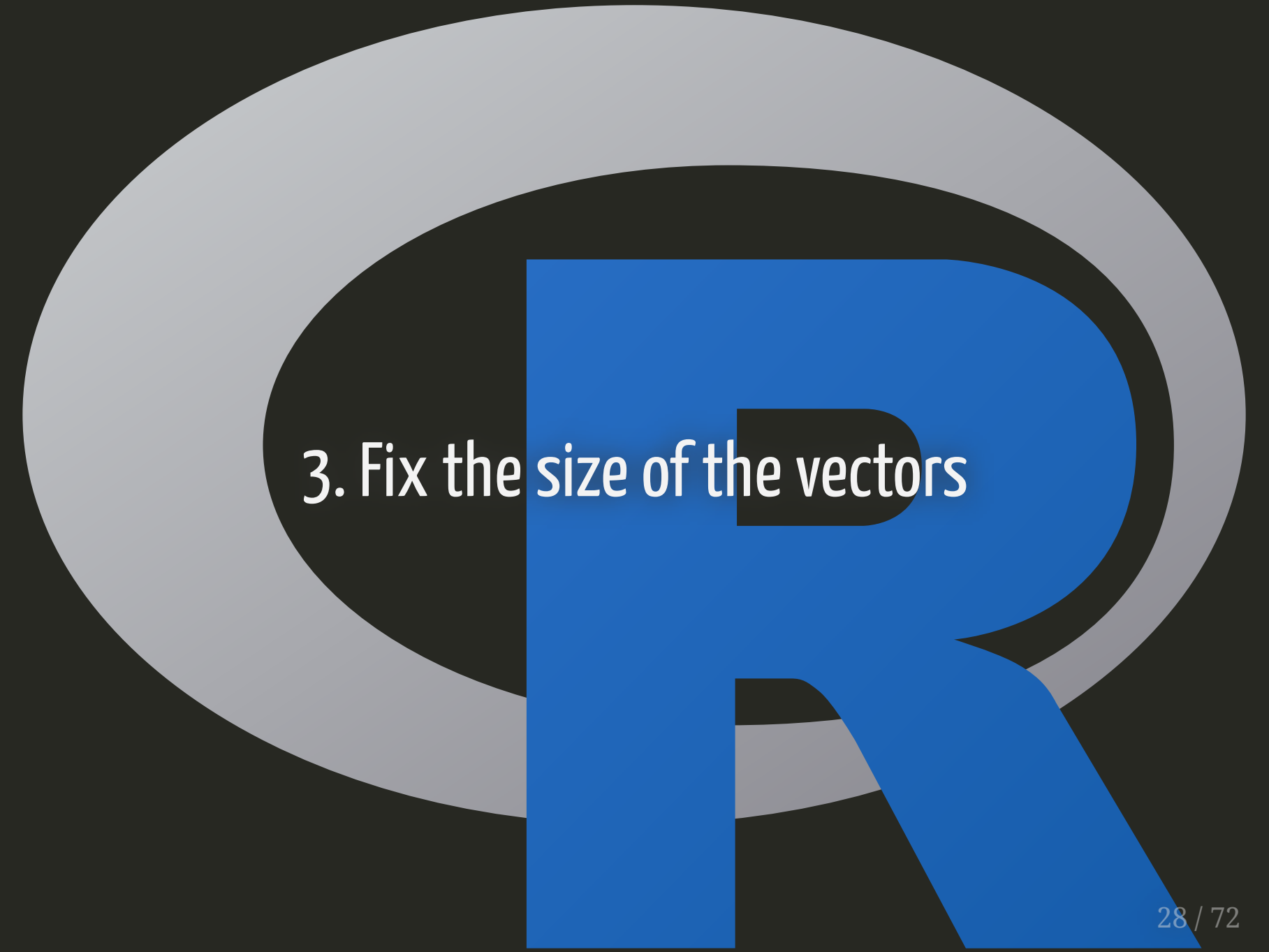
```
## [1] 1000
```

```
my_max(x2)
```

```
## [1] 1000
```

```
my_max(x3)
```

```
## [1] 1000
```


3. Fix the size of the vectors

Fix the size of the vectors

Objective: create a function *trunc_rnorm()* which takes as input argument an integer **n** and a positive scalar **a**. It returns a vector of size **n** where each elements follows a gaussian $\mathcal{N}(0, 1)$ and is between $[-a, a]$;

```
trunc_rnorm.1 <- function(n, a)
 x <- numeric(0)
 i <- 1
 while (i <= n) {
  r <- rnorm(1)
  if (r > - abs(a) & r < abs(a)
    x <- c(x, r)
    i <- i + 1
   }
  }
 x
}
```

```
trunc_rnorm.2 <- function(n, a)
 x <- numeric(n)
 i <- 1
 while (i <= n) {
  r <- rnorm(1)
  if (r > - abs(a) & r < abs(a)
   x[i] <- r
   i <- i + 1
   }
  }
 x
}
```

Comparing results (1)

By using *system.time()*, we should repeat several times the function because the computation time presents some variance:

```
B <- 100
my_time <- data.frame(method_1 = numeric(B), method_2 = numeric(B))
for (k in 1:100) {
  my_time[k, "method_1"] <- system.time(trunc_rnorm.1(n = 10000, a
  my_time[k, "method_2"] <- system.time(trunc_rnorm.2(n = 10000, a
}
```

Then we use *pivot_longer()* (see previous chapter) to transform the data into long format, which allows to plot a boxplot

```
my_time <- tidyr::pivot_longer(my_time, cols = 1:2, names_to = "met
library(ggplot2)
p <- ggplot(my_time, aes(x = method, y = value)) +
  geom_boxplot(outlier.colour = "black", outlier.shape = 16,
               outlier.size = 2, notch = FALSE)
```

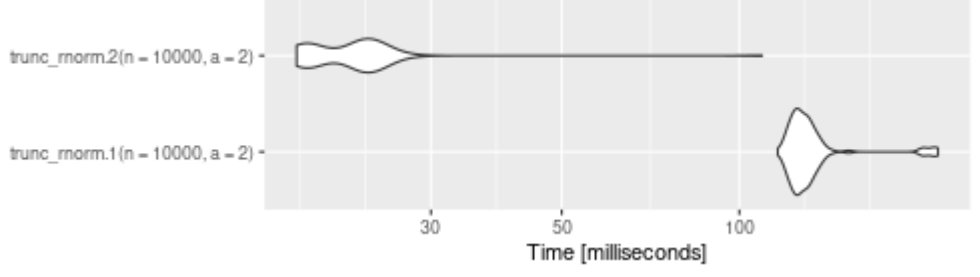
Comparing results (2)

To measure computational time, function *microbenchmark()* from package **microbenchmark** repeats several time the same code and returns summary statistics:

```
mbm <- microbenchmark::microbenchmark(
  trunc_rnorm.1(n = 10000, a = 2),
  trunc_rnorm.2(n = 10000, a = 2), times = 100L)
```

Function *autoplot()* from package **ggplot2** allows to plot the results of **microbenchmark**

```
ggplot2::autoplot(mbm)
```



More information about how measuring computational time here

Why is it longer?

R stores the object somewhere in memory. If the size of the vector is fixed, it is possible to modify this object without changing its memory location. Function *address()* from package **pryr** allows to give the memory location of the object

```
x <- numeric(10)
for (i in 1:10) {
  x[i] <- ifelse(rnorm(1) > 0, 1, 0)
  print(pryr::address(x))
}
```

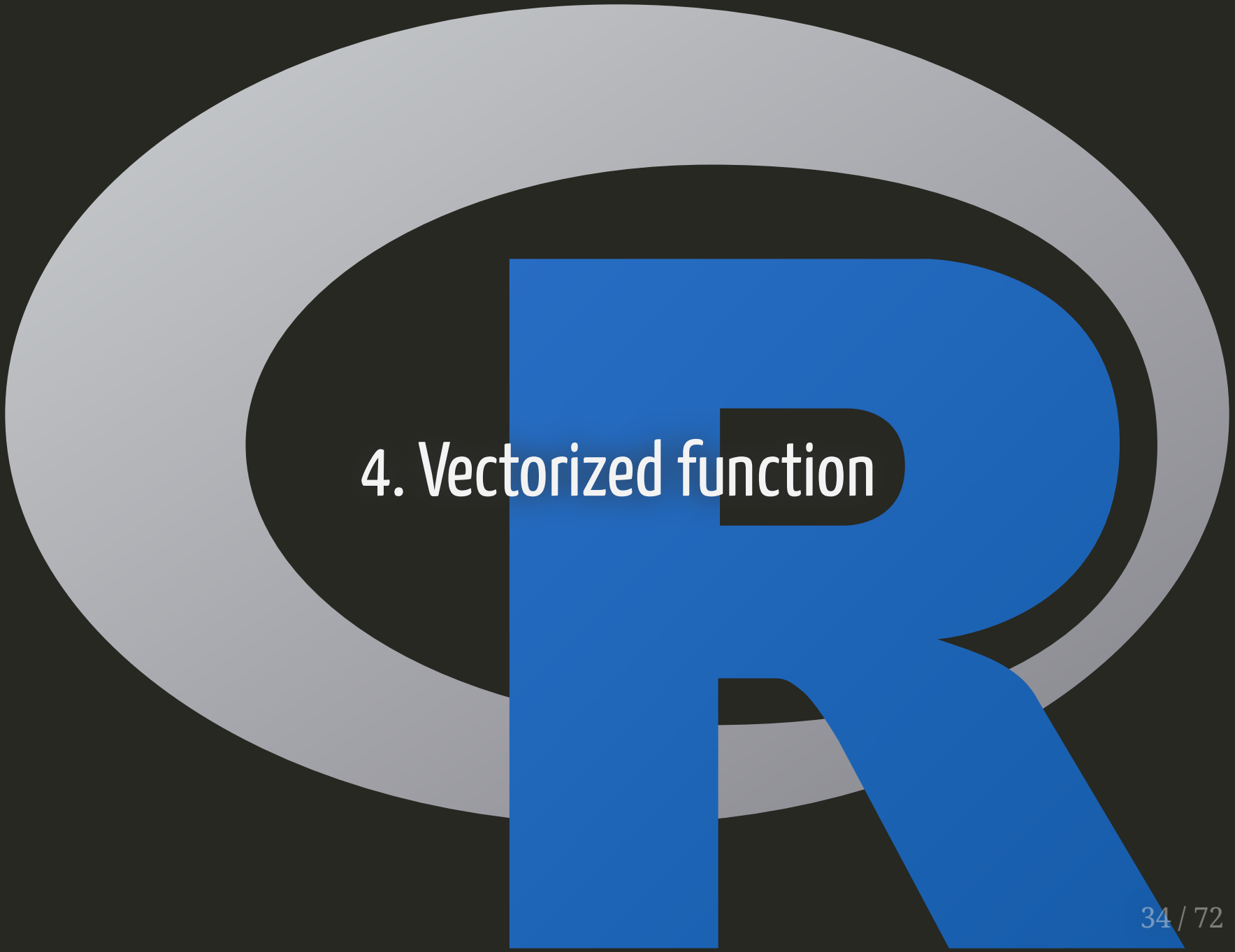
However, if the size of the vector changes, it will modify the location. It is like if it was creating a new object at each step.

```
for (i in 11:20) {
  x[i] <- ifelse(rnorm(1) > 0, 1, 0)
  print(pryr::address(x))
}
```

👩‍🎓 Training: exercise 2.3

Compare the computational time between the three expressions and represent the result in a plot

```
n <- 10 ^
# expression 1
x <- numeric(n)
for (k in 1:n)
  x[k] <- (5 == sample(1:10, 1))
mean(x)
# expression 2
x <- NULL
for (k in 1:n)
  x <- c(x, (5 == sample(1:10, 1)))
mean(x)
# expression 3
x <- 0
for (k in 1:n)
  x <- x + (5 == sample(1:10, 1))
x/n
```


4. Vectorized function

Use preprogrammed vectorized function

Objective: compute the sum of the elements of a simulated vector **vec**

```
vec <- rnorm(10000000)
```

- Solution 1: we program the function

```
my_sum <- function(x) {
  res <- 0
  for (k in seq_along(x))
    res <- res + x[k]
  return(res)
}
```

- Solution 2: we use function *sum()*

```
ggplot2::autoplot(
  microbenchmark::microbenchmark(
    my_sum(vec),
    sum(vec), times = 10L))
```

Why such a difference?

Most of the **R** base functions which are vectorized are calling **C**, **C++**, or **FORTRAN** program to carry out operations.

In a compiled language (which is the case with **C**, **C++**, or **FORTRAN**), the target machine directly translates the program.

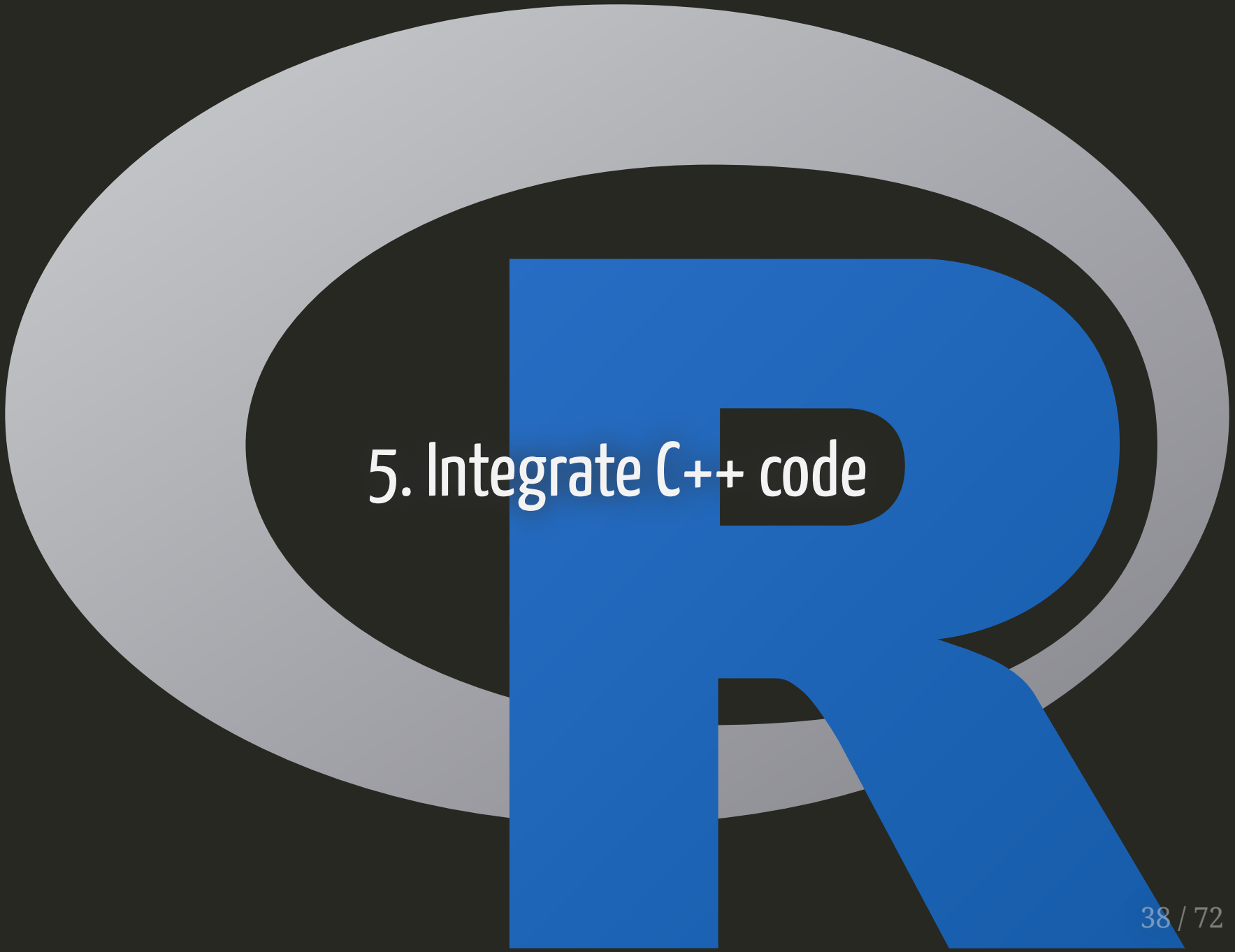
In an interpreted language (which is the case of **R**), the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code.

It explains why the computational time is better when using internal functions which are calling **C**, **C++**, or **FORTRAN**.

👩‍🎓 Training

Exercise 2.4

Program a function *my_sd()* which computes the standard deviation of a vector of numeric without calling function *sum()* neither *mean()*. Moreover, you have to use only one loop. Compare the computational time with function *sd()*


5. Integrate C++ code

How can I integrate a C++ code

- If you program your **C++** function in a separated file, you can lauch your file in **R** with *sourceCpp()* from **Rcpp** package

```
download.file(url = "http://www.thibault.laurent.free.fr/cours/R_av
Rcpp::sourceCpp("sumcplusplus.cpp")
```

- If you program your **C++** in a dedicated chunk in Markdown, the building shared library is automatically done
- Call the **C++** function from **R**

```
sum_rcpp(vec)
```

```
## [1] 4164.232
```

Overview of a C++ file

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double sum_rcpp(NumericVector x) {
  double res = 0;
  int n = x.size();

  for(int i = 0; i < n; i++) {
    res = res + x(i);
  }

  return res;
}
```

Main differences between **R** and **C++** languages

- the type of the objects must be defined (input and output arguments, internal objects, even argument i in a loop !) and it can not be changed. For example, if you define object **res** as an integer, the result will be necessarily an integer even if the **x** vector is a numeric
- **Rcpp** also attempts to provide many of the base **R** functions within the **C++** scope
- a line of code ends by a ;
- use = operaror to create a new object
- syntax in **for** is a little bit different
- in vector use (,) instead of [,]; index in vector starts from 0

More informations: Hadley Wickham's book, package author doc, slides from Duke university

Calling **Python** code from **RStudio**

- Create a chunk by specifying the language used:

````
```{python}
import pandas
flights = pandas.read_csv("http://www.thibault.laurent.free.fr/cours
flights = flights[flights['Dest'] == "TPA"]
flights = flights[['UniqueCarrier', 'DepDelay', 'ArrDelay']]
flights = flights.dropna()
```
````

- Possibility to interact between **R** and the **Python** object created:

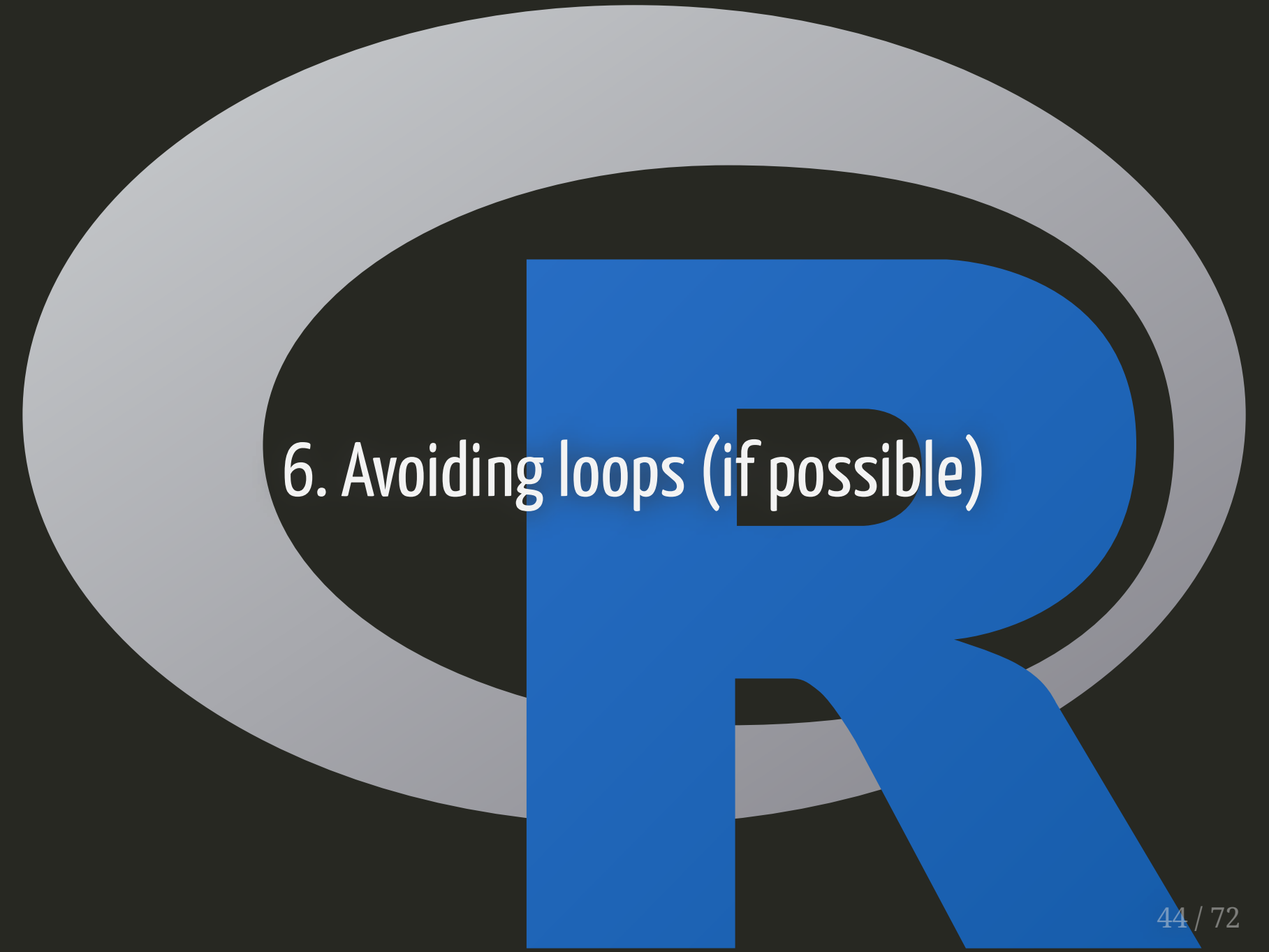
````
```{r}
library(ggplot2)
ggplot(py$flights, aes(UniqueCarrier, ArrDelay)) + geom_point() + g

```
````

👩‍🎓 Training

Exercise 2.5

Program a function *my_sd_cpp()* in **C++** which computes the standard deviation of a vector of numeric. Compare the computational time with function *sd()*.

Remark: in **C++**, one can use *pow(a, b)* to compute a^b.


6. Avoiding loops (if possible)

Function *l-s-t-apply()*

- for **array**, use *apply()*

```
apply(iris[, 1:4], 2, mean)
```

- Function *lapply()* is used on a **list** object. It applies the function **FUN** to each element of the list :

```
my_list <- list(a = 1:3, b = "a string")
lapply(my_list, nchar)
```

- Function *sapply()* uses *lapply()* but returns an array/matrix when it is possible.

```
sapply(mtcars, mean)
```

- Function *tapply()* executes a function on a numeric variable with respect to the levels of a qualitative variable.

```
tapply(iris$Sepal.Length, iris$Species, sum)
```

Function *mapply()*

mapply() is a multivariate version of *sapply()*. For example, we have a **list** of two elements: the first element contains a vector of prices in dollar, the second element a vector of prices in pounds. We also have the rate change dollar/euros and pounds/euros in a vector of 2 elements.

```
price <- list(achat_1 = c(10, 11, 12, 90), achat_2 = c(10, 11, 12, 
taux <- c(taux_1 = 0.85, taux_2 = 1.12)
```

We would like to compute the sum of each element in **euros** knowing the change. For doing that, we first program a function which allows to do this computation for one element **x** of the first list and for one element **y** of the second list.

```
sum_convert <- function(x, y) sum(x) * y
```

Then, we use *mapply()* where the first argument is the multivariate function, the following arguments are **price** and **taux**

```
mapply(FUN = sum_convert, price, taux)
```

Create your own function in argument **FUN**

Functions *l-s-t-apply()* are particularly useful when argument **FUN** is an own created function. For example, if we need to compute summary statistics for several variables of a **data.frame**, we create first the function that we need to apply to each variable and then *sapply()* on it.

```
sum_stat <- function(x) c(min = min(x), max = max(x), mean = mean(x
                          med = median(x), sd = sd(x))
kableExtra::kbl(t(round(sapply(mtcars[, 1:4], sum_stat), 3)))
```

	min	max	mean	med	sd
mpg	10.4	33.9	20.091	19.2	6.027
cyl	4.0	8.0	6.188	6.0	1.786
disp	71.1	472.0	230.722	196.3	123.939
hp	52.0	335.0	146.688	123.0	68.563

Remark: to include properly a table in a Markdown document, we use function *kbl()* from package **kableExtra**

colSums(), rowSums(), colMeans(), rowMeans()

When applying function *apply()* with **FUN** = **sum** or **FUN** = **mean**, it is recommended to use instead one of the function *colSums()*, *rowSums()*, *colMeans()*, *rowMeans()* which are calling internal codes. A consequence is that the computational time is better.

```
x <- matrix(runif(10e6), nc = 5)
ggplot2::autoplot(
  microbenchmark::microbenchmark(
     apply(x, 2, mean),
     my_apply_2(x, mean),
     colMeans(x),
  times = 10L))
```

Function *replicate()*

Objective: we want to simulate 5 samples each of size 10, distributed under a $U_{[0,1]}$ and store it a **list**.

- Solution 1: use a **for** loop instruction

```
res <- vector("list", 5)
for (k in 1:5)
  res[[k]] <- runif(10)
```

- Solution 2: use *sapply()* and include any vector of size 5 instead of a **list** as first argument (a vector can be considered as a **list**, in that case each element is a scalar)

```
res <- sapply(integer(5), function(x) runif(10))
```

- Solution 3: use function *replicate()*

```
res <- replicate(5, runif(10))
```

Remark: solution 3 is equivalent to solution 2 because *replicate()* actually calls function *sapply()* by creating a vector of size n

👩‍🎓 Training

Exercise 2.6

- Simulate a list **xs** of 5 samples each of size 10 distributed under a $U_{[0,1]}$ (use if possible function *replicate()*).
- Simulate a vector **ws** of size 5 distributed under a binomial $\mathcal{B}(10, 0.5)$ (use function *rbinom()*).
- compute the sum of each element of **xs** and multiply it by the element of **ws** (use **for** loop and *mapply()* and compare computational time).


7. Recommendations inside a function

Create several functions (1)

- Do not hesitate to create small functions in your codes and call them inside your main function
- These functions should be local if it is only used once or global if there are called several times.
- Use a dot for private function

Example: compute a non parametric kernel K where K is one of the three options.

- **biweight** $K(x) = \frac{15}{16}(1 - (\frac{x}{h})^2)^2 1_{(\frac{x}{h})^2 \leq 1}$
- **triweight** $K(x) = \frac{35}{32}(1 - (\frac{x}{h})^2)^3 1_{(\frac{x}{h})^2 \leq 1}$
- **gaussian** $K(x) = \frac{1}{\sqrt{2\pi}} \exp(-0.5(\frac{x}{h})^2))$

Create several functions (2)

We first create small global functions:

```
.indicator <- function(x, h) ifelse((x/h) ^ 2 <= 1, 1, 0)
biweight <- function(x, h) 15/16 * (1 - (x/h) ^ 2) ^ 2 * .indicator
triweight <- function(x, h) 35/32 * (1 - (x/h) ^ 2) ^ 3 * .indicato
gaussian <- function(x, h) 1 / sqrt(2 * pi) * exp(-0.5 * (x/h) ^ 2)
```

Then we create the main function which calls others

```
f_noyau <- function(x, h, type = "bi") {
  if (type == "bi") {
    biweight(x, h)
  } else {
    if (type == "tri") {
      triweight(x, h)
    } else {
      gaussian(x, h)
    }
  }
}
```

Use *switch()* to avoid too much if/else

When there are too many nested conditions **if/else** with respect to an input parameter, you can use *switch()* function:

```
f_noyau.2 <- function(x, h, type = "bi") {
 switch(type,
  bi = biweight(x, h),
  tri = triweight(x, h),
  gauss = gaussian(x, h),
  "type should be among bi/tri/gauss")
}
```

Application:

```
x <- seq(-1, 1, 0.01)
plot(x, f_noyau.2(x, 0.3, type = "bi"), type = "l", ylab = "", ylim
lines(x, f_noyau.2(x, 0.3, type = "tri"), lty = 2)
lines(x, f_noyau.2(x, 0.3, type = "gauss"), lty = 3)
```

Function *stopifnot()*

When a function checks for validity of user-input arguments, function *stopifnot()* can be useful. It can contain several verifications:

```
stopifnot(1 < 2, length(1:2) == 2, pi < 2, cos(pi) > 3)
```

When it is used inside a function it stops it as soon as a condition is not verified. It is usually used at the top of the function:

```
try_stopifnot <- function (x, y, n) {
  stopifnot(length(x) == length(y),
            is.integer(n))
  (x + y) ^ n
}
try_stopifnot(2, 5, 2L)
```

Call options from another function

When a function *f()* calls another function *f1()* which has a lot of input argument, it is not necessary to declare all the input arguments in *f()*. Use instead ... among the input arguments of *f()* and *f1()*. It allows to use in *f()* all parameters known by *f1()*.

Example: *plot_reg()* calls function *plot()* which has many input arguments...

```
plot_reg <- function(x, y, np = TRUE, ...) {
  plot(y ~ x, ...)
  abline(lm(y ~ x), col = "blue")
  if (np) {
    np.reg <- loess(y ~ x)
    x.seq <- seq(min(x), max(x), length.out = 25)
    lines(x.seq, predict(np.reg, x.seq), col = "red")
  }
}
```

Applications: we can use any arguments known by *plot()*

```
plot_reg(cars$speed, cars$dist, pch = 16, col = "pink",
         xlab = "variable explicative", ylab = "variable à explique
```

Other recommandations

- Do not keep un-used arguments (it costs time to evaluate)

```
f <- function(a = 5, b = 4, d = 3, e = 1)
  (a + b)^2
```

- It is possible to use function as input argument (like *apply()*):

```
randomise <- function(FUN) FUN(runif(1e3))
randomise(FUN = mean)
randomise(FUN = sum)
```

- It is possible to use function as output argument:

```
f_power <- function(exponent)
    function(x) x^exponent
f_power(2)(1:5)
f_power(3)(1:5)
```

👩‍🎓 Training

Exercise 2.7

Write a function *hist_extrm()* which has three input arguments:

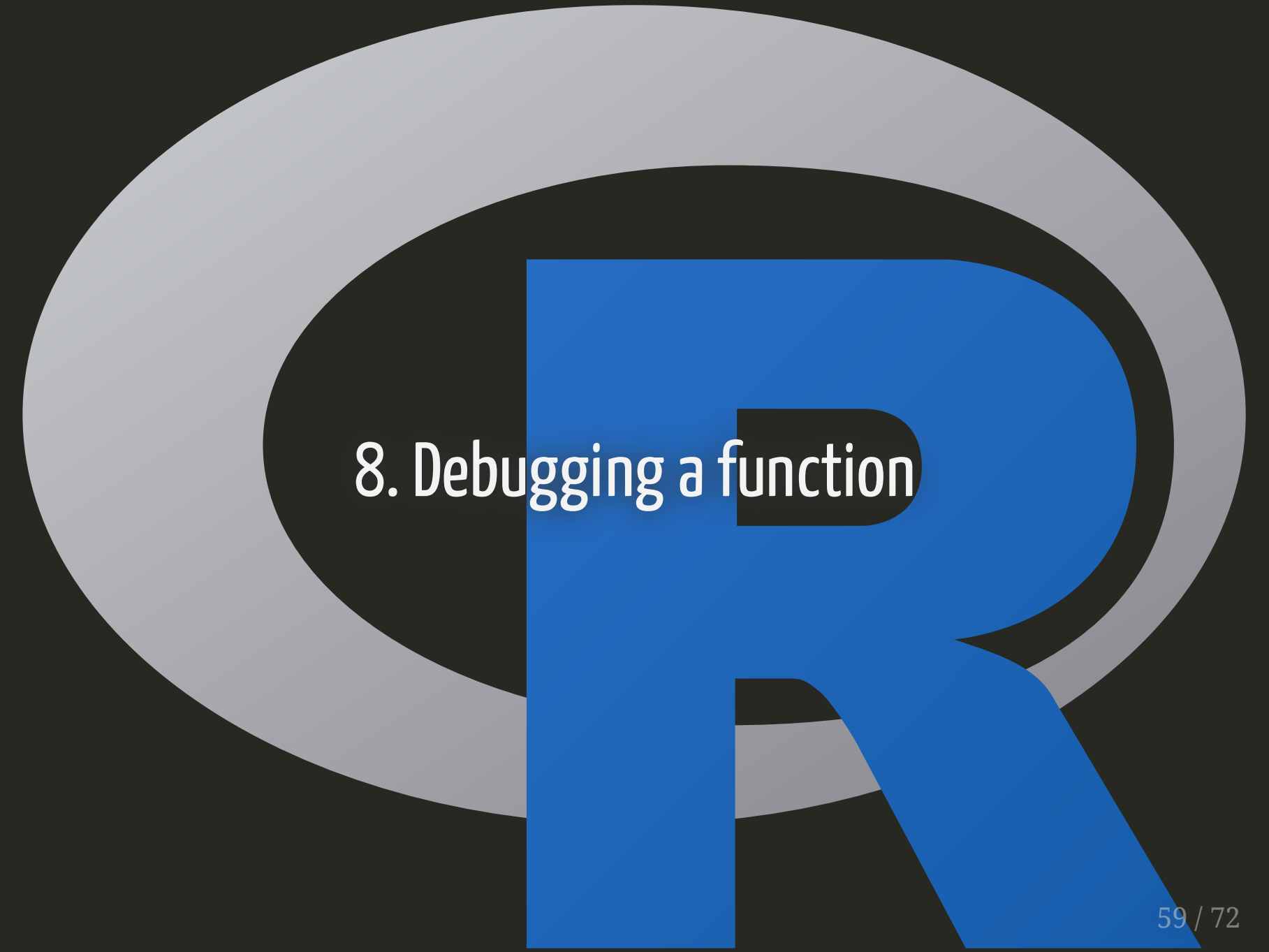
- an integer **n**,
- an integer **B**,
- ... which corresponds to the optional arguments of *hist()*.

This function will make the following job:

Repeat **B** times :

- simulate a random vector $\mathbf{x}$ $\mathcal{N}(0, 1)$ of size n
- checks if yes or no any value is upper to 1.96

The function plots the histogram of the extreme values and return the percentage of simulation where at least one extreme value appears.


8. Debugging a function

Debugg your function

- Distinguish *"error message"* and *"warning message"*
- Most of the time, the error message helps to understand what is wrong

```
sum(c("a", "b"))
```

- Use *traceback()* after an error message due to a call of a function

```
f_noyau.2(seq(-2, 2, by = 0.1), h = "n")
traceback()
```

Remark: it is easier to traceback when a function calls small functions

- Use *debugonce()* for executing a function step by step

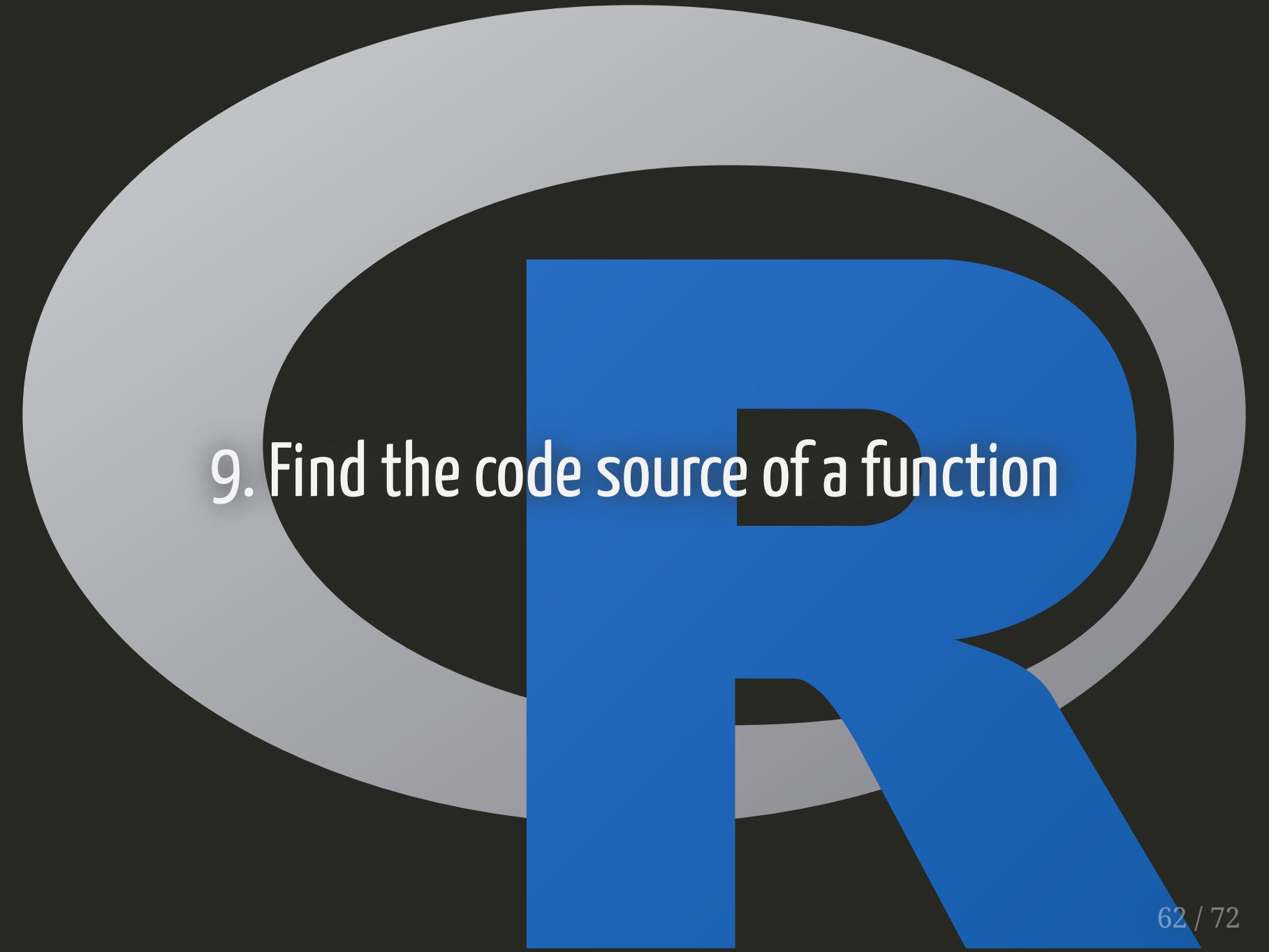
```
ex_bug.2 <- function(x) {
  x <- log(x)
  f_noyau.2(x, h = "n")
}
debugonce(ex_bug.2)
ex_bug.2(-5)
```

Function *try()*

If you are conscious that your code contains error and you do not want the function stops, use function *try()*:

```
f_error.1 <- function(x) {
  try(x <- log(x))
  x
}
f_error.1("10")
f_error.1(-1)
```

Remark: this is what *require()* is doing when it calls *library()*


9. Find the code source of a function

How can I get the code source of a R function? (1)

- Solution 1 : try to print the name of the function in your console

```
sapply
```

```
## function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
## {
##     FUN <- match.fun(FUN)
##     answer <- lapply(X = X, FUN = FUN, ...)
##     if (USE.NAMES && is.character(X) && is.null(names(answer)))
##         names(answer) <- X
##     if (!isFALSE(simplify))
##         simplify2array(answer, higher = (simplify == "array"))
##     else answer
## }
## <bytecode: 0x55ef26aec1b8>
## <environment: namespace:base>
```

How can I get the code source of a R function? (2)

Solution 2: the function belongs to the class S3 (there is a call to **UseMethod**).

```
summary
```

It means that the function can be applied to different class of objects. To print them, use *methods()* function:

```
methods("summary")
```

To get the codes, there are two options:

- if there is no asterisk, print the full name of the function:

```
summary.lm
```

- if there is a asterisk, use function *getAnywhere()*

```
getAnywhere("summary.ecdf")
```

How can I get the code source of a R function? (3)

Solution 3: the function calls internal program (there is a call to *.Primitive()* or *.Internal()*)

```
sum
```

Use the function **show_c_source()** from package **pryr** to get the source code from GitHub (need an account):

```
pryr::show_c_source(.Internal(mean(x)))
```

Solution 4: the function calls **C** code (there is a call to *.Call()*)

```
qnorm
```

In your explorer, print: site:https://svn.r-project.org/R/trunk/src qnorm

How can I get the code source of a R function? (4)

Solution 5: the function applies to an object of class **S4**. An object belongs to S4 if it calls @ to access to its aruguments.

```
require("Matrix")
m <- Matrix(rbinom(100, 1, 0.1), 10, 10)
str(m)
```

To obtain all the functions which can be applied on a S4 object, use *showMethods()*:

```
showMethods(class = "Matrix")
```

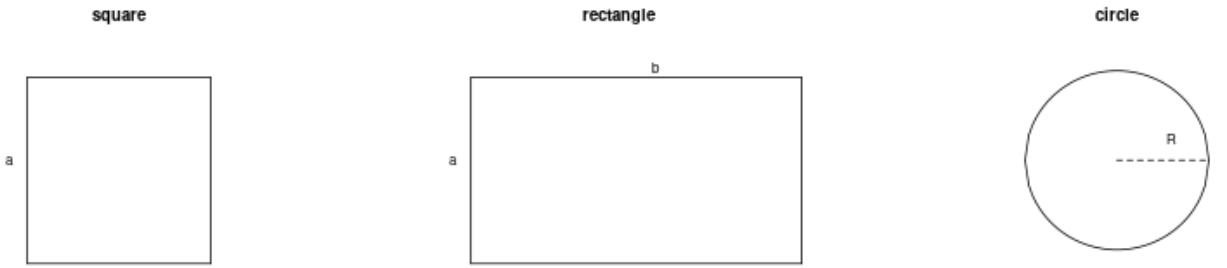
To get the code of one particular function, use *getMethod()*

```
getMethod("dim", "Matrix")
```


9. Short introduction to S3 method

S3 method (1)

Objective: we want to create a function which computes the area of square, a rectangle or a circle.



We need to define a class of object for each geometry.

- a square is defined by a
- a rectangle is defined by a and b
- a circle is defined by R

S3 method (2)

We create for each geometry an object which contains a value which allows to caracterize them:

```
squ <- 3
rec <- c(5, 6)
cir <- sqrt(10)
```

At this step, **squ**, **rec** and **circ** are simple vectors and we can only apply on them functions for vectors. Now, we define them as new classes of object with function *class()*.

```
class(squ) <- "carre"
class(rec) <- "rectangle"
class(cir) <- "cercle"
```

S3 method (3)

We would like to create a function *area()* which computes the good formula. We could do :

```
area <- function(x) {
   switch(class(x),
      carre = x ^ 2,
      rectangle = x[1] * x[2],
      cercle = pi * x ^ 2,
      "class should be among carre/rec/cercle")
}
```

The problem is that if we want to add a new class of object (like triangle), we should modify *area()*. That is why in S3 method, we create a method (here compute the area of a geometry that we call **getArea**) and then associate functions which refers to this method.

S3 method (4)

To create a method, we create a function **getArea** which calls *UseMethod()* like this:

```
getArea <- function(obj)
  UseMethod("getArea", obj)
```

Usually, we associate a first function which allows to treat the case where an object is unknown.

```
getArea.default <- function(obj) {
  stop("Méthode getArea non définie pour ce type d'objet")
}
```

At this step, as we did not associate **getArea** to any classes of object, *getArea()* will produce an error message:

```
getArea(cir)
```

S3 method (5)

We now associate one function for each class. The function must start with the name of the method (**getArea**), then a dot, then the name of the class.

```
getArea.cercle <- function(obj) {
  pi * obj[1] ^ 2
}
getArea.rectangle <- function(obj) {
  obj[1] * obj[2]
}
getArea.carre <- function(obj) {
  obj[1]^2
}
```

We can now use *getArea()* on each object and it will use the corresponding method with respect to the class of object:

```
getArea(cir)
getArea(rec)
getArea(squ)
```