Benchmarking in R

how to check and compare speed of code execution?

Time efficiency

Benchmarking

- Comparing different ways of solving the same problem
- Same problem can be solved in different ways. Some are more time efficient some are slower.
- We need to identify which part of the program slows down our program/code
- We will discuss also byte coding
- In each body of the function there is information about vytes

Structure of the class

1. system.time() –measure the time needed for execution of the code. Returns three numbers.
2. Benchmark() – relative time of our codes, comparison of time of codes across different option
3. Microbenchmark() - repeating code and see the distribution of time
4. Byte compiler - 0101010101001
5. Profiling profvis()– identifying which element of our code are the slowest

Sys.time()

```
> system.time(runif(1e7))
   user  system elapsed
   0.19    0.03    0.21
> 
```

Shows time in seconds
User – how long it took to generate the random numbers for the user
System – time needed for memory allocation or disk access
Elapse - sum of user and system

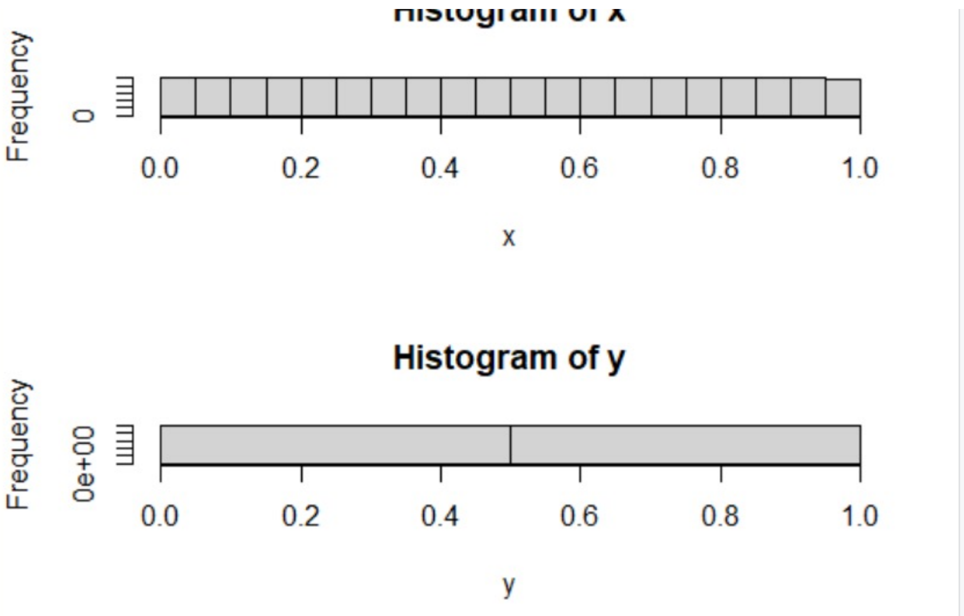
Due to rounding to decimal place might not be seen as the sum.

We see that it tool less than half of the second

It may happen that user>elapsed → parrarel on several course programming

Speed of execution of longer code with sys.time → use {}

```
Browse[1]>
> system.time(runif(1e7))
   user  system elapsed 
   0.19    0.03    0.21 
> system.time({
+   x <- runif(1e6)
+   y <- ifelse(x > 0.5, 1, 0)
+   layout(matrix(1:2, nrow = 2, ncol = 1)) # divides the device up into as
+   # many rows and columns as there are in matrix mat,
+   # with the column-widths and the row-heights specified in the respective argumen
ts.
+   hist(x)
+   hist(y, breaks = c(0, 0.5, 1))
+   layout(matrix(1))
+   print(summary(as.factor(y)))
+   rm(x, y)
+   })
     0      1 
500762 499238 
   user  system elapsed 
   0.73    0.04    0.86 
> 
```



Which function is more time efficienct with sys.time?

```
> system.time(m1 <- my_mean1(myData$x))
   user  system elapsed 
   0.02    0.00    0.02 
> system.time(m2 <- my_mean2(myData$x))
   user  system elapsed 
   3.12    0.05    3.20 
> system.time(m3 <- mean(myData$x))
   user  system elapsed 
   0.02    0.00    0.02 
> system.time(m4 <- mean(as.numeric(myData$x)))
   user  system elapsed 
   0.02    0.00    0.02 
```

We create several functions to calculate mean

1. my_mean1 function is vectorized. Sum of all vectors elements / length of vector
2. my_mean2 – we have loop over elements of vector. We loop over the element of vector . At each iteration we increase the value of sum. Looopover every single element of a vector

Dataframe complex object. Transformation of dataframe to vector is time consuming

Loop is the slowest. It takes almost 3 seconds

Are the result identical?

```
> identical(m1, m3)
[1] FALSE
> identical(m2, m3)
[1] FALSE
> identical(m4, m3)
[1] TRUE
> # lets see the results
> m1
[1] 0.0001645981
> m2
[1] 0.0001645981
> m3
[1] 0.0001645981
> m4
[1] 0.0001645981
```

Are the result identical?
Up to 10 decimal place mean is the same for all functions.
Due to rounding it may seem that there is no difference

Our functions build in mean() is written in C++ our user defined functions written in R → each programming language different rounding policy across programming languages.
Do we care that they are not indendtical?

Function compiled from different programming languages might give us different precision after the comma.
Do we care? Yes if we work for cern

```
> (m1 == m3)
[1] FALSE
> (abs(m1 - m3) < 1e-15)
[1] TRUE
>
```

Benchmarking – average time needed to execute the code

- Can we trust sys.time?

Benchmarking – average time needed to execute the code

- We run our code 100 times and then we verify the average time of execution.
- We compare codes that return the same outcome.

Benchmark() – how to read the output?

Says which code. It can be a label

Not useful always NA.
Derived processes

How much time we
run it?

We limit ourseves to 100000 elements in order not to wait too long

```
> benchmark(m1 <- my_mean1(myData$x[1:100000]),
+           m2 <- my_mean2(myData$x[1:100000]),
+           m3 <- mean(myData$x[1:100000]),
+           m4 <- mean(as.numeric(myData$x[1:100000]))
+           )
                                    test replications elapsed relative user.self sys.self user.child sys.child
1         m1 <- my_mean1(myData$x[1:1e+05])          100    0.08    1.000      0.04     0.04         NA        NA
2         m2 <- my_mean2(myData$x[1:1e+05])          100    4.03   50.375      3.97     0.03         NA        NA
3            m3 <- mean(myData$x[1:1e+05])          100    0.10    1.250      0.05     0.04         NA        NA
4 m4 <- mean(as.numeric(myData$x[1:1e+05]))          100    0.08    1.000      0.07     0.02         NA        NA
>
```

Time in seconds

Total time executing 100 repetitions of a particular code

Loop took 4 seconds to run 100 times

Vectorized

1 – fastest code
4.03/0.08 = 50.375 – the slowest code loop .
Using loop was 50 times slower than the fastest approach.

0.1/0.08= 1.25
Writing own function faster than base mean() it is strange.
Base mean() is more complex than our function.

Divide the number needed for 100 replications by the fastest time

Time for memory allocation

Time by user

```
> (compare_mean <- benchmark("my_mean1" = {m1 <- my_mean1(myData$x[1:100000])},
+                           "my_mean2" = {m2 <- my_mean2(myData$x[1:100000])},
+                           "mean" = {m3 <- mean(myData$x[1:100000])},
+                           "mean_on_num" = {m4 <- mean(as.numeric(myData$x[1:100000]))}
+                           )
+   )
         test replications elapsed relative user.self sys.self user.child sys.child
3        mean          100    0.06    1.000      0.05     0.01         NA        NA
4 mean_on_num          100    0.08    1.333      0.05     0.04         NA        NA
1    my_mean1          100    0.06    1.000      0.07     0.00         NA        NA
2    my_mean2          100    2.96   49.333      2.95     0.00         NA        NA
```

Arguments of benchmark

- 1. we can decide which column to print
- 2. Which column to sort result
- 3. change the repetitions – more replication more precision
- Loop

```
> (compare_mean1a <- benchmark("my_mean1" = {m1 <- my_mean1(myData$x[1:10000])},
+                              "my_mean2" = {m2 <- my_mean2(myData$x[1:10000])},
+                              "mean" = {m3 <- mean(myData$x[1:10000])},
+                              "mean_on_num" = {m4 <- mean(as.numeric(myData$x[1:10000]))},
+                              columns = c("test", "replications", "elapsed", "relative"),
+                              order = "relative",
+                              replications = 500
+                              )
+  )
         test replications elapsed relative
3        mean          500    0.01        1
4 mean_on_num          500    0.03        3
1    my_mean1          500    0.05        5
2    my_mean2          500    1.60      160
> 
```

Loop 160 slower than other options

Microbenchmark() – not in seconds

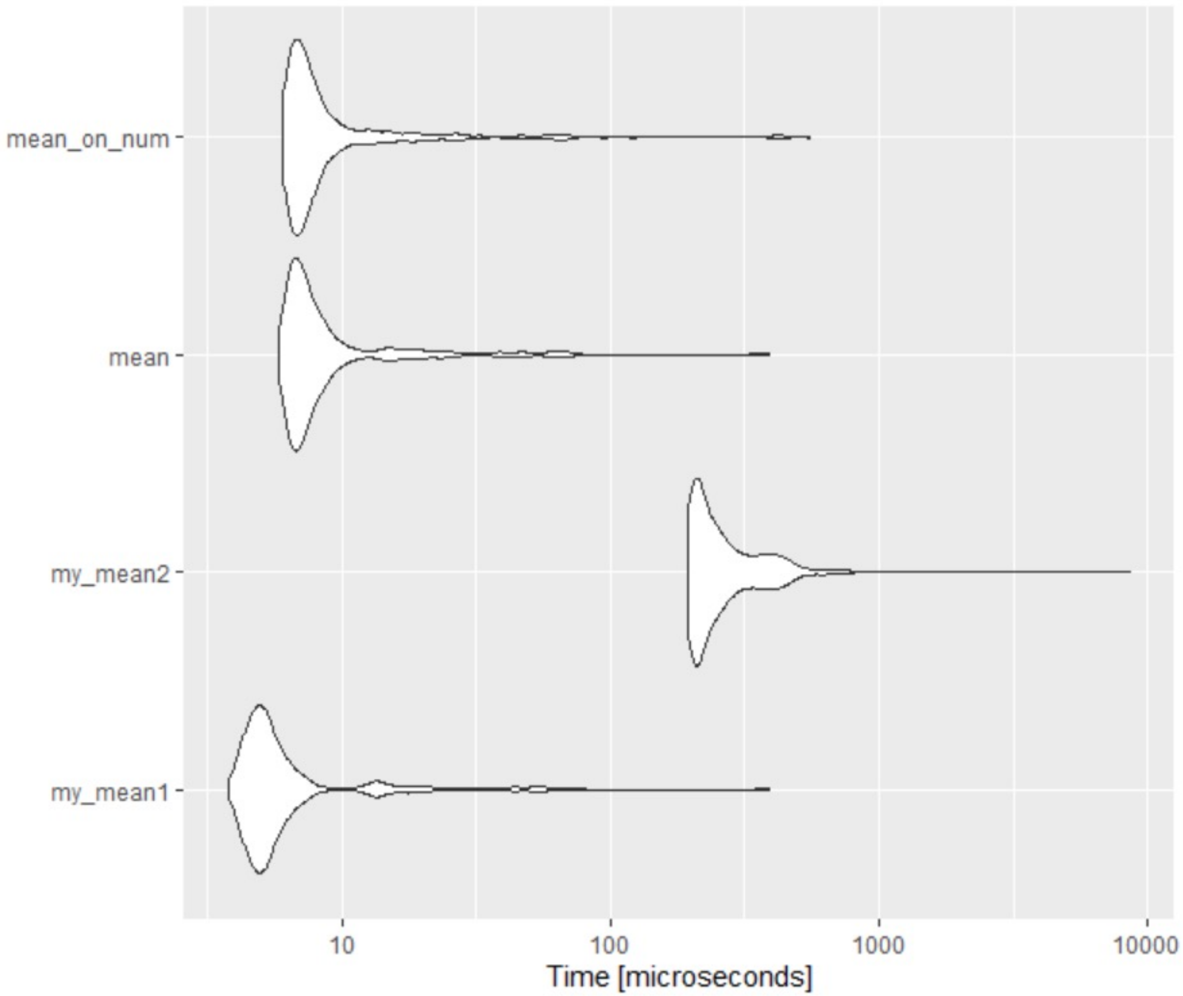
```
> (compare_mean2 <- microbenchmark("my_mean1" = {m1 <- my_mean1(myData$x[1:10000])},
+                                  "my_mean2" = {m2 <- my_mean2(myData$x[1:10000])},
+                                  "mean" = {m3 <- mean(myData$x[1:10000])},
+                                  "mean_on_num" = {m4 <- mean(as.numeric(myData$x[1:10000]))}
+                                  )
+ )
Unit: microseconds
        expr      min        lq       mean    median        uq       max neval cld
    my_mean1   26.001   28.9010   43.27996   32.6515   47.1500   178.701   100  a
    my_mean2 2005.600 2614.9010 3308.41698 2925.1010 3374.3500 12472.701   100   b
        mean   34.001   36.6505   59.31698   46.8515   66.3020   197.201   100  a
 mean_on_num   34.501   39.1010   64.63202   53.3015   81.7515   155.902   100  a
>
```

Adjust the time units automatically

Distribution of time needed to run the particular code

Min time of execution
Max time of execution
Lq – lower quintile in 25 % of executing time of execution was not higher than …
Up – upper quartile only 25% of execution took longer than …
Neval - time of execution
Cld - statistical differences across different codes. Multicomparable tests. Same letters – no statisticall difference across codes. Different letters – statistical differences in terms of time execution.

Microbenchmark() – graphical analysis



Violin plot (require ggplot)

Horizontal xaxis – time

Yaxis – our functions

Frequence of our data

My_mean1 – most frequent time the bulb in violin plot

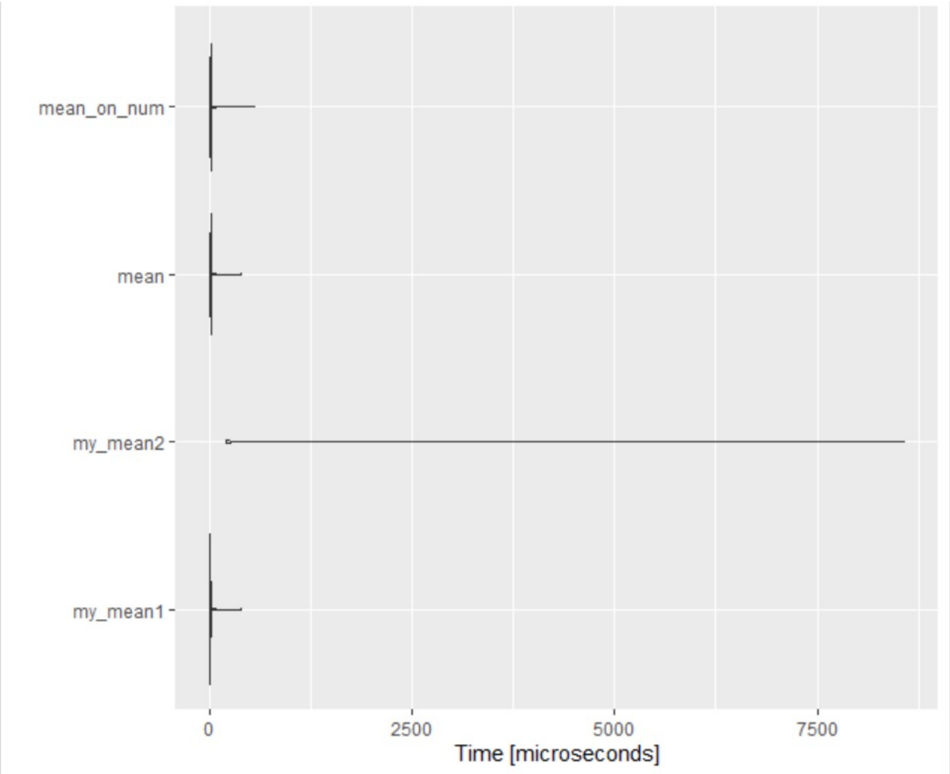
We see the tail with max time

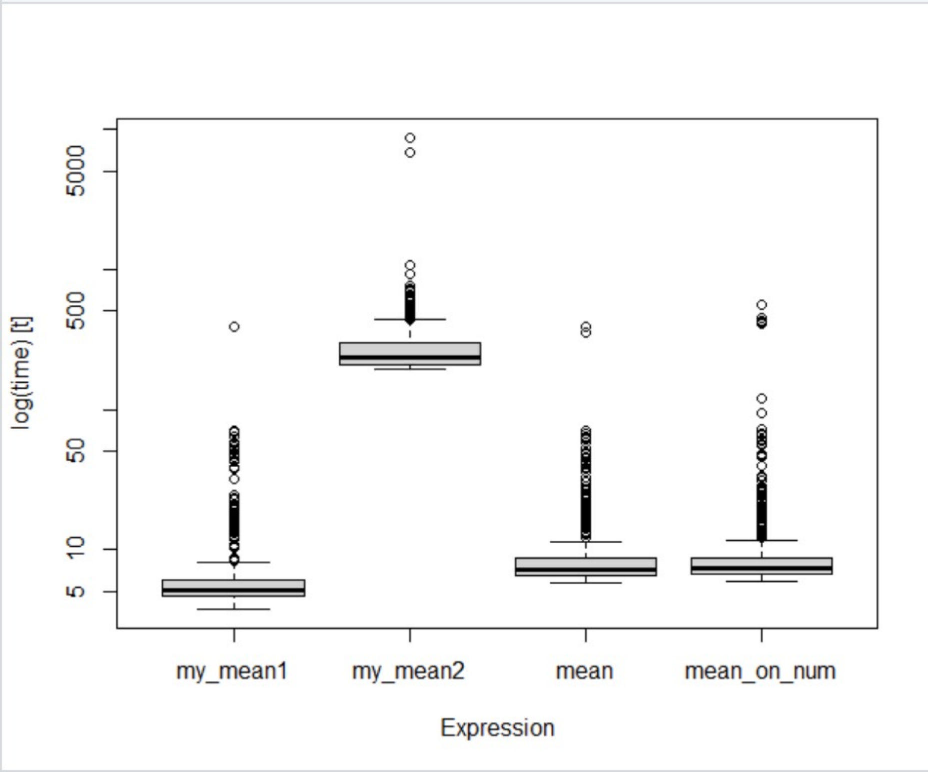
Main part of distribution is the bulb of violin plot

The scale is transform logarimically

Violin plot using linear scale not visible distribution of violin plot

- Boxplot for microbenchmark similar as violin
- Suggest my_mean1 is the fastest





Byte compiler 42:00

Memory address

- Compiling the code to byte – improve the time efficiency
- Different levels of compilations

R code

High level language programming

Understood by human

Translated

Byte code

Understood by computer

Compilation to byte code. Package compiler all functions are by default are compliled when they are used for the first time

Four different levels of compilation:
0 – no translation
1 – some translation
2
3 – by default in R all functions are precompiled

```
> median
function (x, na.rm = FALSE, ...)
UseMethod("median")
<bytecode: 0x000001862e306f38>
<environment: namespace:stats>
```

Showing how time efficiency is improved due to pre compilation

- cmpfun() allows to pre-compile functions
- the cmpfile() function is used to precompile the code saved in an external file

```
> my_mean2
function(x) {
  result <- NA
  n <- length(x)
  for(i in 1:n)
    result <- sum(result, x[i], na.rm = T)
  result <- result/n
  return(result)
}
>
```

Not compiled no byte code!!!!

Newley defined

Not used yet

Pre compilation of function in R

```
> my_mean2_cmp <- cmpfun(my_mean2)
> my_mean2_cmp
function(x) {
  result <- NA
  n <- length(x)
  for(i in 1:n)
    result <- sum(result, x[i], na.rm = T)
  result <- result/n
  return(result)
}
<bytecode: 0x000001865055e1d8>
>
```

Compare the time efficiency before and after compilation

my_mean2_cmp <- cmpfun(my_mean2)

turn off compilation

enableJIT(0)

and compare the efficieny once again

benchmark("my_mean2" = {m1 <- my_mean2(myData$x[1:10000])},

"my_mean2_cmp" = {m2 <- my_mean2_cmp(myData$x[1:10000])}

)[, 1:6]

```
> benchmark("my_mean2" = {m1 <- my_mean2(myData$x[1:10000])},
+           "my_mean2_cmp" = {m2 <- my_mean2_cmp(myData$x[1:10000])}
+           )[, 1:6]
          test replications elapsed relative user.self sys.self
1     my_mean2          100    0.71    1.972      0.69        0
2 my_mean2_cmp          100    0.36    1.000      0.36        0
```

Function after compilation was two times faster than not compiled version of a function!!!!!

Both compiled – time is the same

```
> enableJIT(3)
[1] 0
>
> # and compare the efficieny once again
>
> benchmark("my_mean2" = {m1 <- my_mean2(myData$x[1:10000])},
+           "my_mean2_cmp" = {m2 <- my_mean2_cmp(myData$x[1:10000])}
+           )[, 1:6]
          test replications elapsed relative user.self sys.self
1     my_mean2          100    0.34    1.000      0.33        0
2 my_mean2_cmp          100    0.36    1.059      0.36        0
> |
```

My_mean2 – 1 compilation 99 already compiled !!!!

External files with function definitions and run compile them
External files with function definitions and run compile them
Manual compilation before we run it – we can share with our collegue the compiled files

```
> cmpfile(infile = "my_mean2.R",  # source file
+         outfile = "my_mean2_cmp.R") # destination file
saving to file "my_mean2_cmp.R" ... done
>
> # lets look into "my_mean2_cmp.R"
>
> # lets delete my_mean2() function from our workspace
>
> rm(my_mean2)
>
> # and read it from the file
>
> source("my_mean2.R")
>
> my_mean2
function(x) {
  result <- NA
  n <- length(x)
  for(i in 1:n)
    result <- sum(result, x[i], na.rm = TRUE)
  result <- result/n
  return(result)
}
```

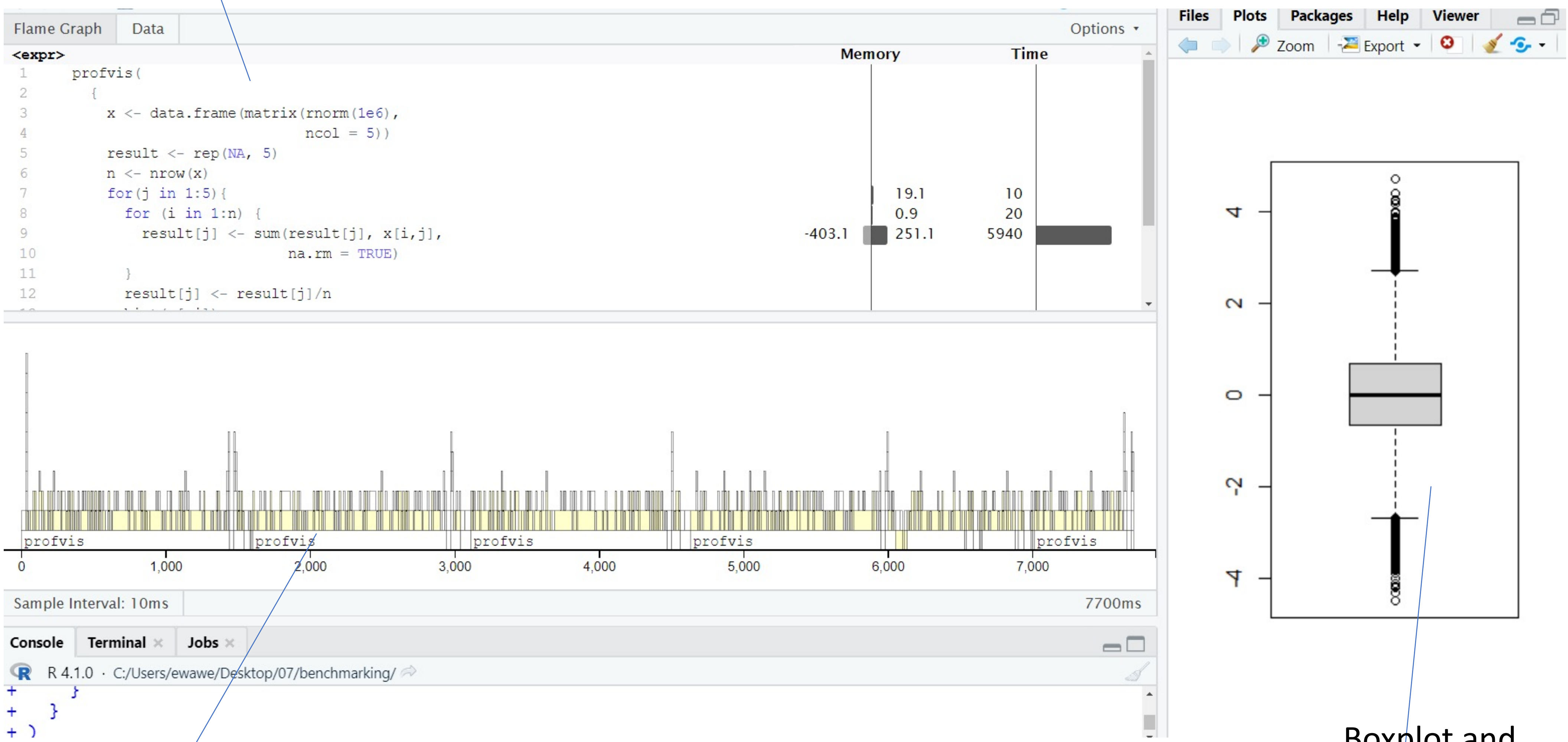
```
> # it is a NON-compiled version
>
> rm(my_mean2)
>
> # lets load the compiled version with loadcmp()
>
> loadcmp("my_mean2_cmp.R")
>
> my_mean2
function(x) {
  result <- NA
  n <- length(x)
  for(i in 1:n)
    result <- sum(result, x[i], na.rm = TRUE)
  result <- result/n
  return(result)
}
<bytecode: 0x0000018646cf9770>
```

Code profiling

- Identify pieces of the code that slow down the code
- Profvis() – graphically represents the time and memory
- Argument – code that we want to profile

Upper panel show for each line the memory use and time efficiency. Loop took the longest

Most time spend inside the loop.



Boxplot and histogram