Case Study 1

Nicolas Bschor 12132344

2022-03-29

First we have to define the libraries we will use in the project.

```
library(microbenchmark)
```

Ratio of Fibonacci numbers

1

In this exercise we write the Ratio of Fibonacci numbers given as $r_n = F_{n+1}/F_n$ with F_n as the nth Fibonacci number. First with a `for` loop:

```
ratio_fibonacci_for <- function(n) {
  f0 <- 0
  f1 <- 1
  r_f = 1:n

  if(n == 0) {
    return(0)
  }

  for(i in 1:n) {
    f_new <- f0 + f1
    f0 <- f1
    f1 <- f_new

    r_f[i]<- f1/ f0
  }

  return(r_f)
}
```

and afterwards with a `while` loop:

```
ratio_fibonacci_while <- function(n) {
  i <- 0
  f0 <- 0
  f1 <- 1
  r_f = 1:n
```

```r
  if(n == 0) {
    return(0)
  }

  while(i < n) {
    i <- i+1
    f_new <- f0 + f1
    f0 <- f1
    f1 <- f_new

    r_f[i]<- f1/ f0
  }

  return(r_f)
}
```

We can check if both functions create the same output.

```r
set.seed(420)
functions_equal <- TRUE

for(rand_int in sample.int(n = 100, size = 10)) {
  n_rf_for <- ratio_fibonacci_for(rand_int)
  n_rf_while <- ratio_fibonacci_while(rand_int)

  if(!all.equal(n_rf_for, n_rf_while)) {
    functions_equal <- FALSE
    break
  }
}

if(functions_equal) {
  print("Functions produce the same output")
} else {
  print("Functions don't produce the same output")
}
```

```
## [1] "Functions produce the same output"
```

2

With the `microbenchmark` command we can compare the two functions with 100 and 1000 as input values.

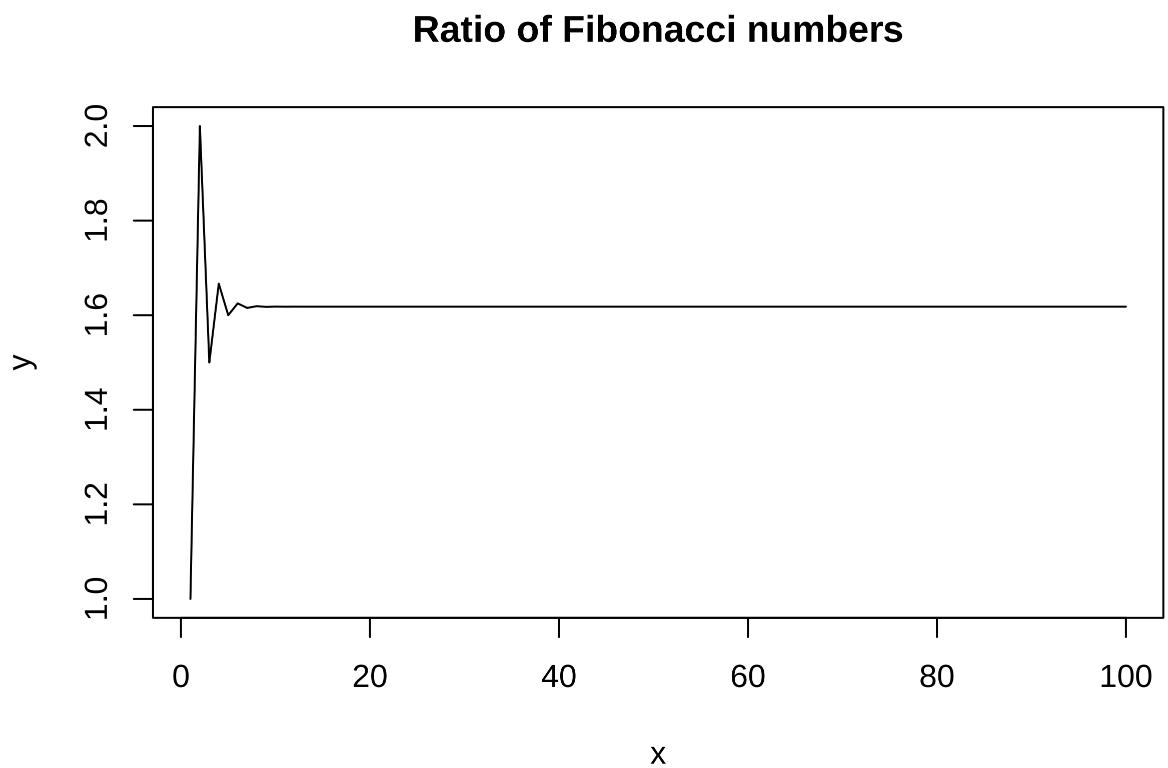
```r
microbenchmark(ratio_fibonacci_for(100), ratio_fibonacci_for(1000),
               ratio_fibonacci_while(100), ratio_fibonacci_while(1000))
```

```
## Unit: microseconds
##                         expr   min     lq    mean median     uq   max neval
##     ratio_fibonacci_for(100)  11.2  11.90  15.261   12.3  13.80  76.4   100
##    ratio_fibonacci_for(1000)  92.2  95.55 107.425   97.2 102.95 279.5   100
##   ratio_fibonacci_while(100)  15.2  16.10  19.899   16.5  19.75  52.1   100
##  ratio_fibonacci_while(1000) 131.9 139.30 153.461  140.5 145.60 363.0   100
```

Given the results, we can see that the function with a `for` loop is faster in general with 100 or 1000 values.

3

```
rf <- ratio_fibonacci_for(100)
plot(rf, type="l", main="Ratio of Fibonacci numbers", xlab="x", ylab="y")
```



In the plot we can see, that the ratio fibonacci series converge to 1.618034.

The golden ratio

```
phi <- (sqrt(5) + 1) / 2
x <- 1:1000
phi_power = phi^(x+1)
phi_sum = phi^(x) + phi^(x-1)

res_equal_op <- phi_sum == phi_power
length(res_equal_op[res_equal_op==TRUE]) == length(res_equal_op)
```

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

```
res_all_equal <- all.equal(phi_power, phi_sum)
res_all_equal
```

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

With the `==` operator we can see that the two arrays doesn't are the same. So there are some differences, but these are very small ($\Delta \approx 10^-16$). This is probably do to the numeric handling of the numbers. The `all.equal` function tested the two arrays if they are 'near equality', so the small rounding differences are not taken into account. Using this function, we can see that $\Phi^{n+1} = \Phi^n + \Phi^{n-1}$.

Game of craps

```
roll_dice <- function() {return( sum(sample(1:6, 2)) )}

game_of_craps <- function() {

  res_1 <- roll_dice()

  if(res_1 == 7 || res_1 == 11) {
    return(TRUE)
  }

  while(TRUE) {
    res_n <- roll_dice()

    if(res_n == res_1)
      return(TRUE)
    else if(res_n == 7 || res_n == 11)
      return(FALSE)
  }
}
```

We created the function `roll_dice` which returns the sum of two random numbers ranging from 1 to 6. In the `game_of_craps` function the main game is simulated. It returns `TRUE` if the game is won, `FALSE` when the player lost the game.

First the function `roll_dice` is called and the result is stored in the `res_1` variable. If the result is 7 or 11 the player wins the game, if not the second part of the games starts in a `while` loop. It starts by roll the dice again and the result is stored in the variable `res_n`. Now the result of the first roll_dice and the one in the while loop are compared. If they are equal the program stops with `TRUE`. If the new result is 7 or 11 the program terminates with `FALSE`. In every other case the program starts again.

Readable and efficient code

Function foobar0

First we can define the 'bad' function foobar0.

```
foobar0 <- function(x, z) {
  if (sum(x >= .001) < 1) {
    stop("step 1 requires 1 observation(s) with value >= .001")
  }
  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- sin(r) + .01

  if (sum(x >= .002) < 2) {
    stop("step 2 requires 2 observation(s) with value >= .002")
  }
  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 2 * sin(r) + .02

  if (sum(x >= .003) < 3) {
    stop("step 3 requires 3 observation(s) with value >= .003")
  }
  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 3 * sin(r) + .03

  if (sum(x >= .004) < 4) {
    stop("step 4 requires 4 observation(s) with value >= .004")
  }
  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- 4 * sin(r) + .04

  return(x)
}
```

Function improve

Now we can improve the function by reducing repetitive and combine different parts in functions.

```
transform_input <- function(x, z, n) {
  fit <- lm(x ~ z)
  r <- fit$residuals
  return(n * sin(r) + (.01 * n))
}

check_input <- function(x, n) {
  if (sum(x >= (n*.001)) < n) {
    stop(paste("step", n, "requires", n,
               "observation(s) with value >=", (n * .001), sep = " "))
  }
}

foobar <- function(x, z) {
  for(i in 1:4) {
    check_input(x, i)
```

```r
    x <- transform_input(x, z, i)
  }

  return(x)
}
```

Validation

To check the new function we can use the following code.

```r
for(i in 1:00) {
  set.seed(1)
  x <- rnorm(100)
  z <- rnorm(100)

  if(!all.equal(foobar0(x,z), foobar(x, z))) {
    stop("Functions produce different output")
  }
}

print("Functions produce equal output")
```

```
## [1] "Functions produce equal output"
```