Case Study 1

AKSTA Statistical Computing 107.258

Maximilian Hagn

21. April 2022

Ratio of Fibonacci numbers

For-Loop

In the following code snippet the ratio of the nth Fibonacci number to the following number is calculated. For this purpose, the start values are determined first. Then a for-loop is executed n - 1 times. In each step the next number is calculated and stored. After the last and the next number has been calculated, the loop is terminated and the ratio is returned.

```
fibonacciRatioForLoop <- function(n) {

  F_x <- 0
  F_1 <- 0
  F_2 <- 1

  for(x in 0:(n-1)) {
    F_x = F_1 + F_2
    F_1 = F_2
    F_2 = F_x
  }

  return(F_2/F_1)

}
```

While-Loop

In the following code snippet, the same principle is applied as in the calculation with the for-loop. The only difference is that in addition it must be counted in which iteration the program is, so that the termination condition can take effect.

```
fibonacciRatioWhileLoop <- function(n) {

  F_x <- 0
  F_1 <- 0
  F_2 <- 1
  x = 0

  while(x <= (n-1)) {
    f_i = F_1 + F_2
    F_1 = F_2
    F_2 = f_i
    x = x + 1
  }

  return(F_2/F_1)

}
```

Benchmark

To accomplish this task, the package "microbenchmark" must be installed first, with the following code:

```
package.install('microbenchmark')
```

Afterwards the library can be included:

```
library(microbenchmark)
```

To perform the benchmark test, the two expressions whose runtime is to be checked must be passed to the function. In the first step, the runtime is calculated for n=100:

```
n <- 100
benchmark_100 = microbenchmark(fibonacciRatioForLoop(n), fibonacciRatioWhileLoop(n))
benchmark_100
```

```
## Unit: microseconds
##                         expr   min    lq      mean median     uq       max neval
##    fibonacciRatioForLoop(n) 5.471 5.886 130.85507  6.421  7.306 12417.908   100
##  fibonacciRatioWhileLoop(n) 9.401 9.646  39.34879 10.226 11.216  2892.501   100
```

It can be observed that the median of the execution times of the for-loop takes less time than the execution of the while-loop. Furthermore, the minimum measured runtime of the for-loop is lower as well, while the maximum measured time is also higher than that of the while-loop. It can thus be stated that the for-loop has a greater spread of execution times and the while-loop thus acts more consistently in terms of time consumption.

In the next step, the runtime is calculated for n=1000:

```
n <- 1000
benchmark_1000 = microbenchmark(fibonacciRatioForLoop(n), fibonacciRatioWhileLoop(n))
benchmark_1000
```

```
## Unit: microseconds
##                         expr   min      lq      mean  median      uq    max
##    fibonacciRatioForLoop(n) 49.40 54.9800  58.74739  56.145  63.615  73.09
##  fibonacciRatioWhileLoop(n) 91.61 97.3495 103.03858 104.300 106.215 118.65
##  neval
##    100
##    100
```

At higher n, however, it can now be noted that both loop variants behave rather consistently and fewer outliers can be detected. This may also be due to the fact that the absolute runtimes are now longer and thus overhead generated by the hardware or the operating system has less influence on the result. In general, however, it can be said that at high n it is clear that the for-loop takes less time and thus seems to be more optimized.

Plot

In the following example, it should be checked, from which n the sequence converges. This can be done by entering the first 100 values returned by the function "fibonacciRatioForLoop" into a vector.

```
ratios <- 1
for(x in 0:100) {
    ratios[x] <- fibonacciRatioForLoop(x);
}
```

To find the exact value, all values from the vector "ratio" can be displayed:

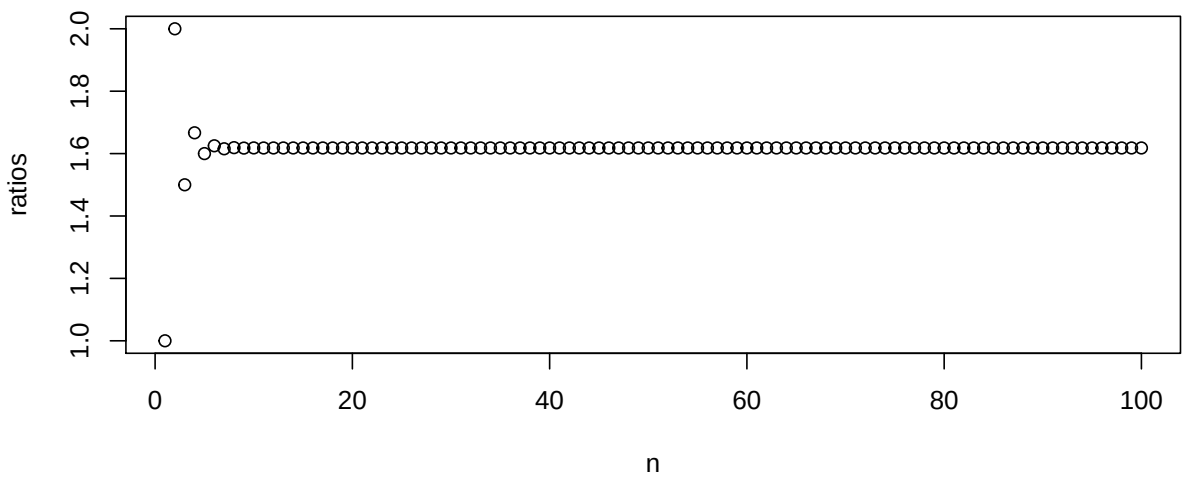
```
print(ratios)
```

```
##   [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##   [9] 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037 1.618033 1.618034
##  [17] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [25] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [33] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [41] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [49] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [57] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [65] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [73] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [81] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [89] 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034 1.618034
##  [97] 1.618034 1.618034 1.618034 1.618034
```

Using these values, it can be seen that from n=7 onwards, only slight differences are recognizable. From n=16 there are no differences at all and each calculated value is the same.

This can also be illustrated by visualization in a plot:

```
plot(ratios, type = "p", xlab="n", ylab="ratios")
```



It can be seen that the sequence is approaching towards the value 1.618034. Calculating the formula for the golden ratio given in the second task, we get the following value:

```
print((sqrt(5) + 1)/2)
```

```
## [1] 1.618034
```

Thus, it can be stated that the sequence of the Fibonacci ratio converges towards the golden ratio.

The golden ratio

To check whether the golden ratio satisfies the Fibonacci-like relationship first the value of the golden ratio is stored in a variable. In a second step the for-loop is executed for all test values from 1 to 1000. In each iteration, the ratio of the last Fibonacci number to the next is calculated and then compared with the == operator.

```
goldenRatio <- (sqrt(5) + 1)/2
testValues = c(1, 5, 10, 15, 16, 17, 18, 19, 20, 30, 40, 50, 60, 70, 80, 90, 100,
               200, 300, 400, 500, 600, 700, 800, 900, 1000)

CheckEquality_1 <- function(values) {
  for(i in values){
     cat(sprintf("%f == F_%d/F_%d -> %s\n",
                 goldenRatio,
                 i+1,
                 i,
                 fibonacciRatioForLoop(i) == goldenRatio))
  }
}

CheckEquality_1(testValues)
```

```
## 1.618034 == F_2/F_1 -> FALSE
## 1.618034 == F_6/F_5 -> FALSE
## 1.618034 == F_11/F_10 -> FALSE
## 1.618034 == F_16/F_15 -> FALSE
## 1.618034 == F_17/F_16 -> FALSE
## 1.618034 == F_18/F_17 -> FALSE
## 1.618034 == F_19/F_18 -> FALSE
## 1.618034 == F_20/F_19 -> FALSE
## 1.618034 == F_21/F_20 -> FALSE
## 1.618034 == F_31/F_30 -> FALSE
## 1.618034 == F_41/F_40 -> TRUE
## 1.618034 == F_51/F_50 -> TRUE
## 1.618034 == F_61/F_60 -> TRUE
## 1.618034 == F_71/F_70 -> TRUE
## 1.618034 == F_81/F_80 -> TRUE
## 1.618034 == F_91/F_90 -> TRUE
## 1.618034 == F_101/F_100 -> TRUE
## 1.618034 == F_201/F_200 -> FALSE
## 1.618034 == F_301/F_300 -> FALSE
## 1.618034 == F_401/F_400 -> FALSE
## 1.618034 == F_501/F_500 -> FALSE
## 1.618034 == F_601/F_600 -> TRUE
## 1.618034 == F_701/F_700 -> TRUE
## 1.618034 == F_801/F_800 -> TRUE
## 1.618034 == F_901/F_900 -> FALSE
## 1.618034 == F_1001/F_1000 -> FALSE
```

It can be seen that the equality is reached for the first time at the number 40, but from then on it is not always continuously true. For example, it is noticeable that between the values 200 and 500 the relationship is never reached, but afterwards it is satisfied again. Similarly with n=900 and n=1000 there is no equality. The "==" operator checks if the two values are exactly the same. Thus not all values are true here and even with small deviations the value "FALSE" is returned.

In the second task, the same principle is used to calculate and compare Fibonacci ratios. The only difference is that the "all.equal" function is used instead of the "==" operator.

```
goldenRatio <- (sqrt(5) + 1)/2
testValues = c(1, 5, 10, 15, 16, 17, 18, 19, 20, 30, 40, 50, 60, 70, 80, 90, 100,
               200, 300, 400, 500, 600, 700, 800, 900, 1000)

CheckEquality_2 <- function(values) {
   for(i in values){
     cat(sprintf("all.equal(%f, F_%d/F_%d -> %s\n",
                 goldenRatio,
                 i+1,
                 i,
                 all.equal(goldenRatio, fibonacciRatioForLoop(i))))
   }

}
CheckEquality_2(testValues)
```

```
## all.equal(1.618034, F_2/F_1 -> Mean relative difference: 0.381966
## all.equal(1.618034, F_6/F_5 -> Mean relative difference: 0.01114562
## all.equal(1.618034, F_11/F_10 -> Mean relative difference: 9.136361e-05
## all.equal(1.618034, F_16/F_15 -> Mean relative difference: 7.427932e-07
## all.equal(1.618034, F_17/F_16 -> Mean relative difference: 2.83722e-07
## all.equal(1.618034, F_18/F_17 -> Mean relative difference: 1.083721e-07
## all.equal(1.618034, F_19/F_18 -> Mean relative difference: 4.139447e-08
## all.equal(1.618034, F_20/F_19 -> Mean relative difference: 1.581128e-08
## all.equal(1.618034, F_21/F_20 -> TRUE
## all.equal(1.618034, F_31/F_30 -> TRUE
## all.equal(1.618034, F_41/F_40 -> TRUE
## all.equal(1.618034, F_51/F_50 -> TRUE
## all.equal(1.618034, F_61/F_60 -> TRUE
## all.equal(1.618034, F_71/F_70 -> TRUE
## all.equal(1.618034, F_81/F_80 -> TRUE
## all.equal(1.618034, F_91/F_90 -> TRUE
## all.equal(1.618034, F_101/F_100 -> TRUE
## all.equal(1.618034, F_201/F_200 -> TRUE
## all.equal(1.618034, F_301/F_300 -> TRUE
## all.equal(1.618034, F_401/F_400 -> TRUE
## all.equal(1.618034, F_501/F_500 -> TRUE
## all.equal(1.618034, F_601/F_600 -> TRUE
## all.equal(1.618034, F_701/F_700 -> TRUE
## all.equal(1.618034, F_801/F_800 -> TRUE
## all.equal(1.618034, F_901/F_900 -> TRUE
## all.equal(1.618034, F_1001/F_1000 -> TRUE
```

In this output it is clearly noticeable that the equality is already given from the value 20 and in contrast to the previous example the golden ratio satisfies from the value 20 all Fibonacci-like relationships. This can be explained by the fact that the "all.equal" function only checks for near equality and thus returns "TRUE" even for values that are almost equal. Furthermore, in the previous task it was already found out that the series converges against the golden ratio and this assertion can thus be confirmed.

Game of craps

The game starts by throwing two dice. These are stored in the variables “diceResult_1” and “diceResult_2” in my program and generated by “runif()”. The first parameter describes that one number should be generated, whereas the second and third parameters describe the limits. The calculated number is then rounded down. This process could also be done by generating only a random number between two and twelve, but here the approach to generate two numbers was chosen to represent the abstract concept of the game. After the sum of the two dice has been calculated, a check is made to see if the numbers rolled together add up to 7 or 11. If this condition is met, a string containing the diced value and “Won!” is returned. If not satisfied, the number of dice pips are stored and written to the output that the player must roll again. The while-loop is now executed until either the condition to win or the condition to lose is met. The player wins if the sum of the current rolls equals the sum of the first rolls. If this condition is met, a string is returned that contains the current value of the dice and “Won!”. If the player rolls the value seven or eleven after the first round, the current value and “Lost!” are returned. To better track the progress of the game, after the first round, as well as after all rolls, the number that was rolled is also returned. The first round and the remaining rounds are intentionally treated separately in this implementation, since checking each round if it is the first round would have a negative effect on performance.

The code is shown in the following snippet:

```
game_of_craps <- function() {

  diceResult_1 <- floor(runif(1, min=1, max=6))
  diceResult_2 <- floor(runif(1, min=1, max=6))
  diceSum = diceResult_1 + diceResult_2

  if(diceSum == 7 || diceSum == 11){
     return(paste("x = ", diceSum, "- Won!"))
  }

  firstRoll <- diceSum
  print(paste("x = ", diceSum, "- Roll Again!"))

  while(TRUE) {

    diceResult_1 <- floor(runif(1, min=1, max=7))
    diceResult_2 <- floor(runif(1, min=1, max=7))
    diceSum = diceResult_1 + diceResult_2

      if(diceSum == firstRoll){
        return(paste("x = ", diceSum, "- Won!"))
      }

      if(diceSum == 7 || diceSum == 11){
        return(paste("x = ", diceSum, "- Lost!"))
      }

      print(paste("x = ", diceSum, "- Roll Again!"))
  }
}
```

Five runs of the game are given below:

```
game_of_craps()
```

```
## [1] "x =  9 - Roll Again!"
## [1] "x =  5 - Roll Again!"
## [1] "x =  2 - Roll Again!"
## [1] "x =  6 - Roll Again!"
## [1] "x =  6 - Roll Again!"
```

```
## [1] "x =  7 - Lost!"
```

```
game_of_craps()
```

```
## [1] "x =  3 - Roll Again!"
## [1] "x =  8 - Roll Again!"
## [1] "x =  4 - Roll Again!"
```

```
## [1] "x =  11 - Lost!"
```

```
game_of_craps()
```

```
## [1] "x =  6 - Roll Again!"
```

```
## [1] "x =  7 - Lost!"
```

```
game_of_craps()
```

```
## [1] "x =  6 - Roll Again!"
## [1] "x =  12 - Roll Again!"
```

```
## [1] "x =  6 - Won!"
```

```
game_of_craps()
```

```
## [1] "x =  4 - Roll Again!"
```

```
## [1] "x =  7 - Lost!"
```

Readable and efficient code

Code Wrapped in Function “foobar0”

In this implementation, the code was inserted into the “foobar0” function. For this purpose, it was only additionally changed that the value of x is returned.

```
foobar0 <- function(x, z) {
  set.seed(1)

  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)
}
```

Rewrite

For the implementation, the repetitive pieces of code were moved to their own functions. The input was outsourced to the function “checkInput” and the calculation of the values to the function “computeValues”. These functions are executed one after the other in the main function “foobar”.

```
checkInput <- function(x, step) {

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

}

computeValues <- function(x, z, step) {

  fit <- lm(x ~ z)
  r <- fit$residuals
  x <- step * sin(r) + (.01 * step)
  return(x)

}

foobar <- function(x, z) {

  set.seed(1)
  checkInput(x, 1)
  x <- computeValues(x, z, 1)
  checkInput(x, 2)
  x <- computeValues(x, z, 2)
  checkInput(x, 3)
  x <- computeValues(x, z, 3)
  checkInput(x, 4)
  x <- computeValues(x, z, 4)

  return(x)

}
```

Equality

If the two implementations are compared with the function “all.equal”, the result returns true. Thus, it can be stated that the semantics of the program remains the same.

```
normValue <- rnorm(100)
all.equal(foobar0(normValue, normValue), foobar(normValue, normValue))
```

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