Group 21 Lab 1 Report

Stefano Toffol (steto820), Mim Kemal Tekin (mimte666)

28 January, 2019

Note: all the inline results of the R outputs are computed dynamically using Rmarkdown, exploiting the syntax 'r code'. The computation used are all reported at the end of each question (and in the appendix)

Question 1

We were asked to evaluate the following snippets:

```r
# Computation 1
x1 <- 1/3
x2 <- 1/4
if((x1-x2)==(1/12)) print("Subtraction is correct") else print("Subtraction is wrong")

# Computation 2
x1 <- 1
x2 <- 1/2
if((x1-x2)==(1/2)) print("Subtraction is correct") else print("Subtraction is wrong")
```

Which return respectively `Subtraction is wrong` and `Subtraction is correct`.

The first operation returns a different result due to approximation errors of `x1`: $\frac{1}{3} = 0.\overline{3}$ is a recurring decimal number, as well as the theoretical result of the sum ($\frac{1}{12} = 0.08\overline{3}$). Printing the result of the two quantities, specifying `options(digits = 22)` makes the situation clear: $\frac{1}{3}$ is equal to 0.3333333333333333148296, which leads to 0.0833333333333333148296 when we subtract $\frac{1}{4}$ to it; on the other hand the theoretical result, that is $\frac{1}{12}$, is equal to 0.0833333333333333287074. The difference then is of a magnitude of 10^{-17}, and is entirely caused by the presence of recurring decimal number, rounded to the closest decimal number that can be stored in a computer.
For the second computations there are no problems because all the quantities involved, 1, $\frac{1}{2}$ and consequentially the result $\frac{1}{4}$ are all exact powers of 2, which are stored without errors in any machine.

A possible solution for the first operation is not to consider the actual equality of two quantities, but their *approximated* equality. This is carried out by, for instance, the function `all.equal()`. In the `if()` condition, testing `all.equal(1/3 - 1/4, 1/12)` will, in fact, return TRUE.

```r
# Print the various quantities
sprintf("%.22f", 1/3)
sprintf("%.22f", 1/3 - 1/4)
sprintf("%.22f", 1/12)
```

Question 2

We are asked to write a `R` function that computes the derivative, according to the following formula:

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

The function is implemented as following:

```r
my_derivative <- function(f, x, epsilon = 1e-15) {
    return( (f(x+epsilon)-f(x)) / epsilon )

}
```

The derivative has to be tested on the identity function:

```
my_lm <- function(x) {

  return( x )

}
```

The derivative of the function should, of course, be constant and equal to 1. The R results were instead pretty different.
When $x = 1$ the function returns 1.110223, a finite number relatively close to our target, even though substantially different. On the other hand when $x = 100000$ the output of the function becomes 0.

The first case is a result of approximation: storing exactly 10^{-15} is impossible and `R` round it to 1.000000000000000077705 4e-15. The rounding difference is then amplified when computing $f(x + \epsilon)$ and successively when dividing the numerator of the derivative by ϵ.
The second is a case of underflow and is caused by the presence of a large number ($x = 100000$) and its linear combination with a very small number $\epsilon = 10^{-15}$. In fact the linear regression of $f(x + \epsilon)$ returns exactly the same result of $f(x)$, consequentially leading to a derivative of zero.

```
epsilon = 1e-15

# Print the various quantities (1st case)
sprintf("%1.22e", epsilon)
sprintf("%.22f", my_lm(1+epsilon))
print(my_derivative(my_lm, 1))

# Print the various quantities (2nd case)
my_lm(100000)
my_lm(100000 + epsilon)
my_lm(100000 + epsilon) == my_lm(100000)  # TRUE
my_derivative(my_lm, 100000)
```

Question 3

In this task we are asked to implement a function to compute the variance using the following formula:

$$\text{Var}(\overrightarrow{x}) = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)$$

The data to experiment on, a 10000 unit vector, is generated from a Gaussian random variable with $\mu = 10^8$ and $\sigma^2 = 1$. We therefore have a high number of huge values that differ of a very tiny amount. The aim of this assignment is to investigate the difference between this estimation of the variance and the default one from `R` considering an increasing amount of observations from the numeric vector. The resulting code is the following:

```r
# Implement variance function
myvar <- function(x) {

  n <- length(x)
  return( (sum(x^2) - sum(x)^2/n)/(n-1) )

}

set.seed(12345)
data <- rnorm(1e+04, 1e+08)
my_diff <- rep(NA, 1e+04)
indexes <- c()
i <- 1

while(i<=1e+04) {

  indexes <- c(indexes, i)
  my_diff[i] <- myvar(data[indexes]) - var(data[indexes])
  i <- i+1

}
my_cum_diff <- cumsum(my_diff[-1])
```

The outcome of the whole process is plotted in Figure 1. As we can see there are no cases in which the differences between the two methods were exactly equal to zero. The range of them is asymmetric and goes from -3.95016 to 3.48086. Our implementation does not grant a sufficient level of precision also because on average the two methods will differ of approximately -1.

The differences may again be explained by both the rounding and over-floating phenomena: the computer, being unable to perfectly store every float number, especially when their integer part is already big like in this case, approximates it to the nearest one and ignores the effect of small decimal parts. In our implementation there are several different parts in which this problem may occur: squaring each number of our data vector, taking thier sum (and of their squares), and, finally, taking the difference between the two quantities; dividing the result by $n-1$ instead should not give any trouble since the numerator should be of the same magnitude of the denominator. Probably for bigger samples the effect of over-floating will increase even more: considering thousands of values around 10^8, squaring them and summing them up will quite likely lead to huge numbers, difficult to be handled by any machine.

In particular several trends going towards zero are present among the points: their presence is probably caused by considering a new number in the sample which is rounded up/down to the nearest storable number. This rounding effect is then amplified in the various step described before, but is soon cancelled by the successive sample(s), which includes the presence of a number rounded in the opposite direction.

Is interesting to observe also the plot displaying the trend of the cumulative sum of the differences in Figure 2. Until approximately the first 6500 observations the value of the cumulative difference fluctuate around -40. After that it drops all of a sudden, up to almost -400. This behaviour show an interesting aspect of the phenomena: the over-floating effect seems to be the dominant one for big samples, having a bigger impact on the bias between the two functions.

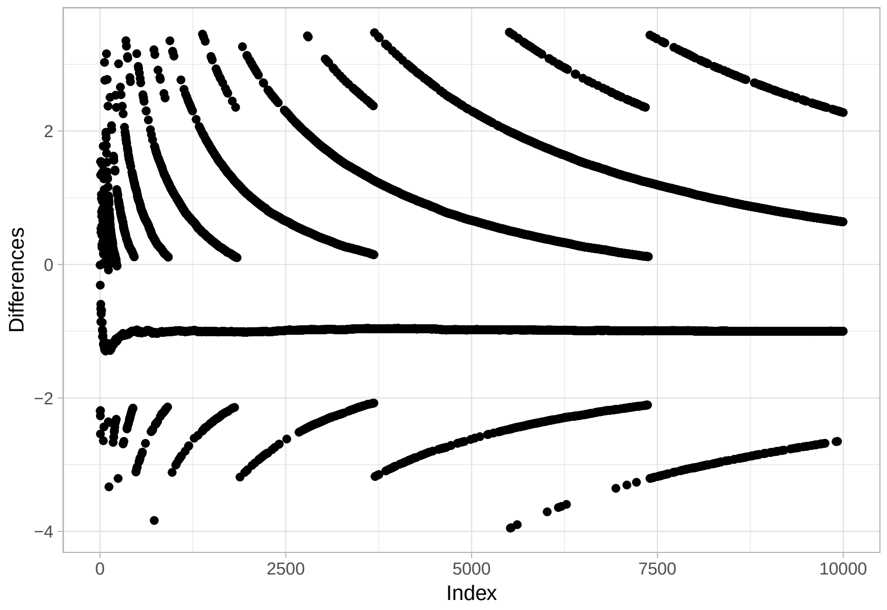

Figure 1: Behaviour of the differences between the variance function implemented and the default one.

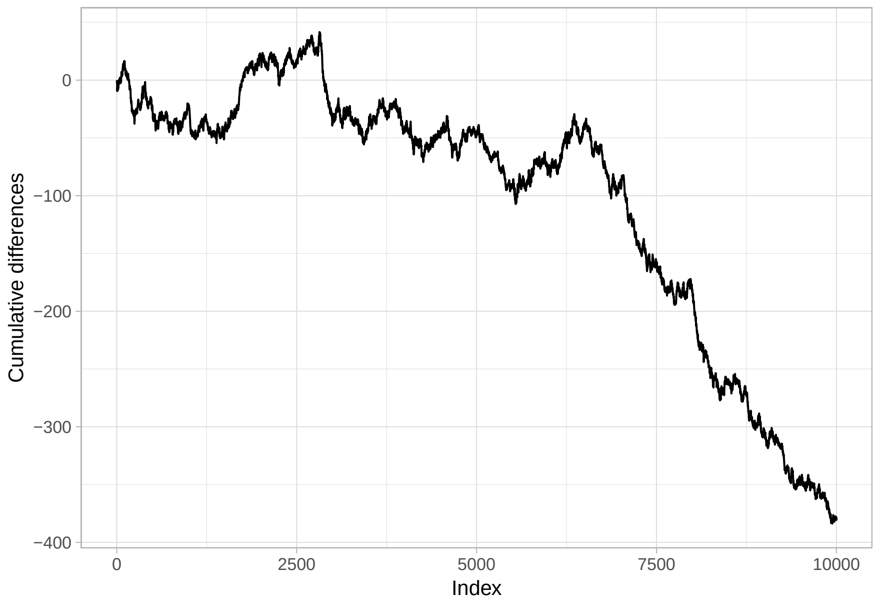

Figure 2: Behaviour of the cumulative differences between the variance function implemented and the default one.

`R` in-built `var` function is instead computed using the following formula:

$$\mathrm{Var}(\overrightarrow{x}) = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \frac{\sum_{j=1}^{n} x_j}{n}\right)^2$$

We implemented it in the following way and replicated the steps from above:

```r
# Implement variance function
R_var <- function(x) {

  n <- length(x)
  mu <- mean(x)
  return( sum((x-mean(x))^2)/(n-1) )

}

R_diff <- rep(NA, 1e+04)
indexes <- c()
i <- 1

while(i<=1e+04) {

  indexes <- c(indexes, i)
  R_diff[i] <- R_var(data[indexes]) - var(data[indexes])
  i <- i+1

}
```

As we can observe from the plot in Figure 3, the differences are still not all equal to zero. However the range has been shrunk dramatically (between -2.22045e-16 and 2.22045e-16), the range is symmetric, no trends appear any more, the unique values of the differences are just 6 and there are many cases in which the difference is exactly zero (in 6811 of the total 9999). The in-built method from `R` probably present just subtle differences from the one just tested, but the logic behind its computation should be the same.

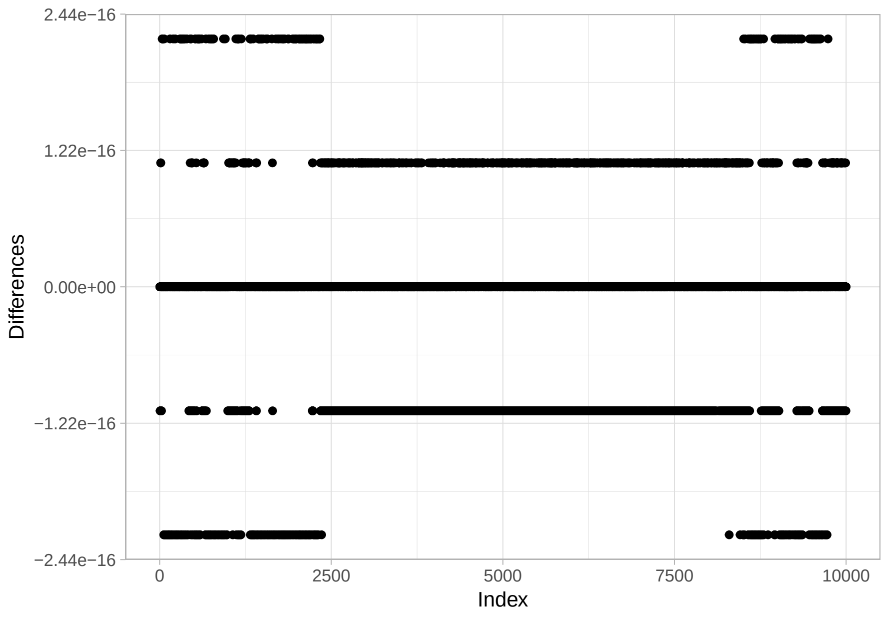

Figure 3: Behaviour of the differences between the variance function implemented according to the R formula and the actual default function from R.

Figure 4 instead show the cumulative sum of the newly computed differences: despite the increased precision of the new function (the maximum cumulative error is equal to 2.22045e-16), a decreasing trend, this time starting from the very beginning and being almost linear, is still present on the computed quantities. The way we computed the variance is probably still affected by some minor over-floating problems that are just slightly influencing our estimates.

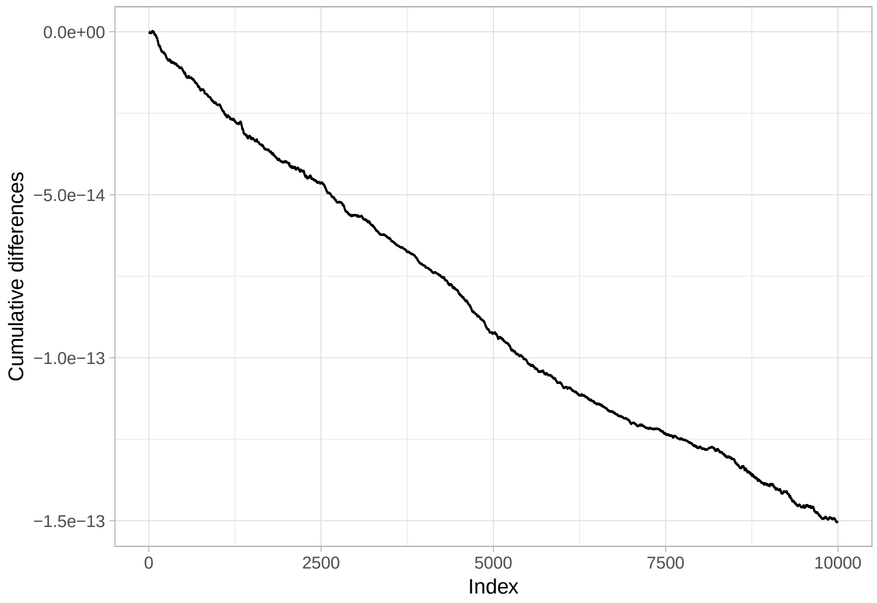

Figure 4: Behaviour of the cumulative differences between the variance function implemented according to the R formula and the actual default function from R.

Question 4

The data is read with the following script:

```r
# We read the data and delete the first column, that is just the ID of the observations
data <- read.csv("../datasets/tecator.csv")[,-1]
y <- data[["Protein"]]
x <- as.matrix(data[, names(data)!="Protein"])
```

First of all we compute the quantities needed to solve the linear system of the type $A\overrightarrow{\beta} = \overrightarrow{b}$ in order to determine the coefficients of the linear model. The quantities we are interested in are $A = X^TX$ and $\overrightarrow{b} = X^T\overrightarrow{y}$.
The quantities are computed with the following code:

```r
A <- t(x) %*% x
b <- t(x) %*% y
```

In order to obtain the solution of $A\overrightarrow{\beta} = \overrightarrow{b}$ we will now try to invert the matrix A to then estimate the coefficients $\overrightarrow{\beta}$ exploiting the relationship $\overrightarrow{\beta} = A^{-1}\overrightarrow{b}$.
The `R` calculation however lead to the following error:

```r
solve(A)
```

```
## Error in solve.default(A): system is computationally singular: reciprocal condition number = 7.13971
```

The matrix A we previously obtained is singular, in other words contains columns which are linear dependently. For instance the correlation between `Channel1` and `Channel2` is equal to 0.9999908, meaning that every single observation of one column can be almost perfectly approximated with a linear combination of the observations from the other column.

We therefore check the *condition number* of the matrix A, that is giving us a measure of how sensitive the inversion is to perturbations in the input data and to round-off errors made during the inversion process (so is telling us if the output does not change much with small variations of the input). The function that achieve it in `R` is the `kappa()` function (the product of the norm of the matrix and the norm of its inverse). For this case the value returned is 1.15783e+15. Since it's of the order of 10^{15}, it means that even the smallest change in the input will have a huge impact on the output. In fact, supposing we are changing one of the many columns that is highly correlated with another regressor, the output will lead to totally different results. In our case small changes on the input may even lead to the possibility of actually inverting the matrix A (we should try to decrease the correlation of each single pair of columns).

```
# Scaling
scaled_x <- scale(x)
scaled_y <- scale(y)
scaled_A <- t(scaled_x) %*% scaled_x
scaled_b <- t(scaled_x) %*% scaled_y

inv_scaled_A <- solve(scaled_A)
scaled_beta <- inv_scaled_A
```

Scaling the dataset change substantially the situation. With a mean of 0 and a standard deviation of each feature of just 1, the matrix A can finally be inverted. The value of the `kappa()` function is still high (4.90472e+11) but we managed to render the matrix stable enough to be solved. This happened because even though the correlation between the rows remains the same 0.9999908 and the computed determinant of the matrix is still 0, the decreasement of the scale brings all the involved quantities in a dense interval around zero, allowing to decrease all the rounding errors since more decimal points can be exactly represented by the machine.

Moreover the inversion algorithm `solve()` is probably an iterative one, based on numeric optimization and converging only when the change between successive steps are really small (according to a threshold value of magnitude probably $< 10^{-15}$). Standardizing the variables and bringing them to a scale where decimal numbers can be represented with enhanced precision may have brought the algorithm to reach this threshold within the maximum number of iterations allowed, something that was not possible on the original scale due to overflowing/rounding errors.

Appendix

```r
knitr::opts_chunk$set(echo = F, message = F, error = F, warning = F,
                      fig.align='center', out.width="70%")

# Load libraries


# --------------------------------------------------------------------------
# A1
# --------------------------------------------------------------------------


# Computation 1
x1 <- 1/3
x2 <- 1/4
if((x1-x2)==(1/12)) print("Subtraction is correct") else print("Subtraction is wrong")

# Computation 2
x1 <- 1
x2 <- 1/2
if((x1-x2)==(1/2)) print("Subtraction is correct") else print("Subtraction is wrong")


# Print the various quantities
sprintf("%.22f", 1/3)
sprintf("%.22f", 1/3 - 1/4)
sprintf("%.22f", 1/12)


# --------------------------------------------------------------------------
# A2
# --------------------------------------------------------------------------


my_derivative <- function(f, x, epsilon = 1e-15) {

    return( (f(x+epsilon)-f(x)) / epsilon )

}


my_lm <- function(x) {

  return( x )

}


epsilon = 1e-15

# Print the various quantities (1st case)
sprintf("%1.22e", epsilon)
sprintf("%.22f", my_lm(1+epsilon))
print(my_derivative(my_lm, 1))

# Print the various quantities (2nd case)
my_lm(100000)
my_lm(100000 + epsilon)
my_lm(100000 + epsilon) == my_lm(100000)  # TRUE
my_derivative(my_lm, 100000)


# ----------------------------------------------------------------------------
# A3
# ----------------------------------------------------------------------------


# Implement variance function
myvar <- function(x) {

  n <- length(x)
  return( (sum(x^2) - sum(x)^2/n)/(n-1) )

}

set.seed(12345)
data <- rnorm(1e+04, 1e+08)
my_diff <- rep(NA, 1e+04)
indexes <- c()
i <- 1

while(i<=1e+04) {

  indexes <- c(indexes, i)
  my_diff[i] <- myvar(data[indexes]) - var(data[indexes])
  i <- i+1

}
my_cum_diff <- cumsum(my_diff[-1])


library(ggplot2)
df_plot <- data.frame(Index = 1:1e+04, Differences = my_diff)

ggplot(aes(x = Index, y = Differences), data = df_plot) +
  geom_point() +
  # scale_y_continuous(labels = function(x) format(x, scientific = T, digits = 3)) +
  theme_light()


df_plot <- data.frame(Index = 2:1e+04, Differences = my_cum_diff)

ggplot(aes(x = Index, y = Differences), data = df_plot) +
  geom_line() +
  # scale_y_continuous(labels = function(x) format(x, scientific = T, digits = 3)) +
  ylab("Cumulative differences") +
  theme_light()


# Implement variance function
R_var <- function(x) {

  n <- length(x)
  mu <- mean(x)
  return( sum((x-mean(x))^2)/(n-1) )

}

R_diff <- rep(NA, 1e+04)
indexes <- c()
i <- 1

while(i<=1e+04) {

  indexes <- c(indexes, i)
  R_diff[i] <- R_var(data[indexes]) - var(data[indexes])
  i <- i+1

}


library(ggplot2)
df_plot <- data.frame(Index = 1:1e+04, Differences = R_diff)

ggplot(aes(x = Index, y = Differences), data = df_plot) +
  geom_point() +
  scale_y_continuous(labels = function(x) format(x, scientific = T, digits = 3)) +
  theme_light()


df_plot <- data.frame(Index = 2:1e+04, Differences = cumsum(R_diff[-1]))

ggplot(aes(x = Index, y = Differences), data = df_plot) +
  geom_line() +
  scale_y_continuous(labels = function(x) format(x, scientific = T, digits = 3)) +
  ylab("Cumulative differences") +
  theme_light()


# We read the data and delete the first column, that is just the ID of the observations
data <- read.csv("../datasets/tecator.csv")[,-1]
y <- data[["Protein"]]
x <- as.matrix(data[, names(data)!="Protein"])


A <- t(x) %*% x
b <- t(x) %*% y
```

```r
solve(A)


# Scaling
scaled_x <- scale(x)
scaled_y <- scale(y)
scaled_A <- t(scaled_x) %*% scaled_x
scaled_b <- t(scaled_x) %*% scaled_y

inv_scaled_A <- solve(scaled_A)
scaled_beta <- inv_scaled_A
```