Computational statistics - Lab 1

Alexander Karlsson (aleka769), Lennart Schilling(lensc874)

2019-01-29

Contents

Question 1: Be careful when comparing

Subtraction results

```
x1 = 1/3
x2 = 1/4
if(x1-x2 == 1/12) {
  print("Subtraction is correct")
} else {
  print("Subtraction is wrong")
}
```

```
## [1] "Subtraction is wrong"
```

In this first case, the subtraction is described as wrong. Adjusting the *options* within R helps to explain why the difference between the two values is not as expected.

```
options(digits = 22)
paste0("x1 = ", x1)
```

```
## [1] "x1 = 0.333333333333333"
```

```
paste0("x2 = ", x2)
```

```
## [1] "x2 = 0.25"
```

```
paste0("Subtraction result = ", x1 - x2)
```

```
## [1] "Subtraction result = 0.0833333333333333"
```

```
paste0("Expected value = ",  1/12)
```

```
## [1] "Expected value = 0.0833333333333333"
```

It becomes clear that the the value for *x1* cannot be stored as the exact specified number (1/3). Instead, the computer was only able to save a very close approximation of the value. Even if the substraction result equals the expected value, the function prints *Subtraction is wrong*. The reason behind this could lie in the fact that ==- operator does not refer to the value of 1/12 that R can store but to the true value of 1/12.

```
x1 = 1
x2 = 1/2
if(x1-x2 == 1/2) {
  print("Subtraction is correct")
} else {
  print("Subtraction is wrong")
}
```

```
## [1] "Subtraction is correct"
```

```
options(digits = 22)
paste0("x1 = ", x1)
```

```
## [1] "x1 = 1"
```

```
paste0("x1 = ", x2)
```

```
## [1] "x1 = 0.5"
```

```
paste0("Subtraction result = ", x1 - x2)
```

```
## [1] "Subtraction result = 0.5"
```

```
paste0("Expected value = ",  1/2)
```

```
## [1] "Expected value = 0.5"
```

In the second case, the subtraction of the two values returns the expected value, because - compared to the first case - both values for x could be stored in the exact way as desired. The substraction result also exactly reflects the value 1/2.

Improvements

A possible improvement for the first case would be to avoid the ==-operator and to use the *isTrue*-function in combination with the *all.equal*-function instead.

```
x1 = 1/3
x2 = 1/4
if(isTRUE(all.equal(x1-x2, 1/12))) {
  print("Subtraction is correct")
} else {
  print("Subtraction is wrong")
}
```

```
## [1] "Subtraction is correct"
```

Question 2: Derivative

Derivative function

```
calculateDerivative = function(x) {
  derivative = (x + 10^(-15) - x) / (10^(-15))
  return(derivative)
}
```

Evaluation of function

```
calculateDerivative(1)
```

```
## [1] 1.1102230246251565
```

```
calculateDerivative(100000)
```

```
## [1] 0
```

Comments on results

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon} = \frac{f(x+10^{-15}) - f(x)}{10^{-15}} = \frac{x + 10^{-15} - x}{10^{-15}} = 1$$

Adding a number to the value of 10^{-15} and then subtracting this same number does not lead to the original value of 10^{-15}. Since x will be first added to 10^{-15} and then directly subtracted within the numerastor of

the formula, the value of the numerator will not stay 10^{-15} but it will be changed. Since the demoniator value is indeed 10^{-15}, the true result of 1 will not be obtained.

A proof for the wrong result based on an addition and a following subtraction of the same value can be found here:

```
(10^-15)
```

```
## [1] 1.0000000000000001e-15
```

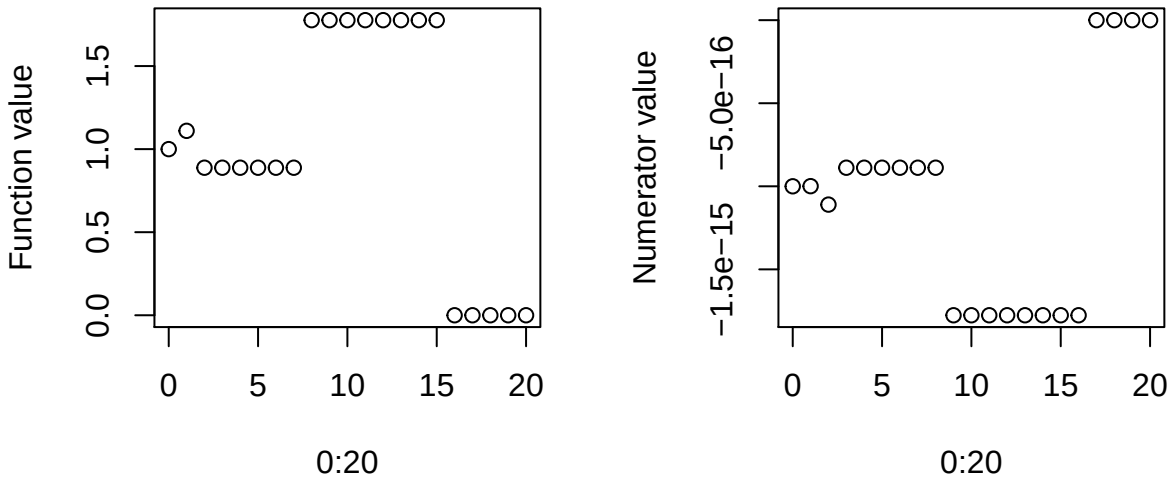
```
(10^-15)+1-1
```

```
## [1] 1.1102230246251565e-15
```

The reason for that is $(10^{-15}) + 1$ can not be stored by the true value but only by an approximation. That is why subtracting 1 from this approximated value will not return the original value.

When x increases, there seems to be a point when adding a really small number (ϵ) to x no longer makes a difference ('underflow') and the returned value converges against 0. This happens when $x = 16$, as can be seen below.

```
fun_ret = sapply(X = 0:20, function(i){
  calculateDerivative(x = i)
})
numerator_ret = function(x, eps){return( (x - eps) - x ) }
fun_ret2 = sapply(X = 0:20, function(i){
  numerator_ret(x = i, eps = 1e-15)
})
par(mfrow = c(1,2))
plot(0:20, fun_ret, ylab = "Function value")
plot(0:20, fun_ret2, ylab = "Numerator value")
```



For x $>$ 16 the value 0 is returned for the numerator.

Question 3: Variance

Function `myvar`

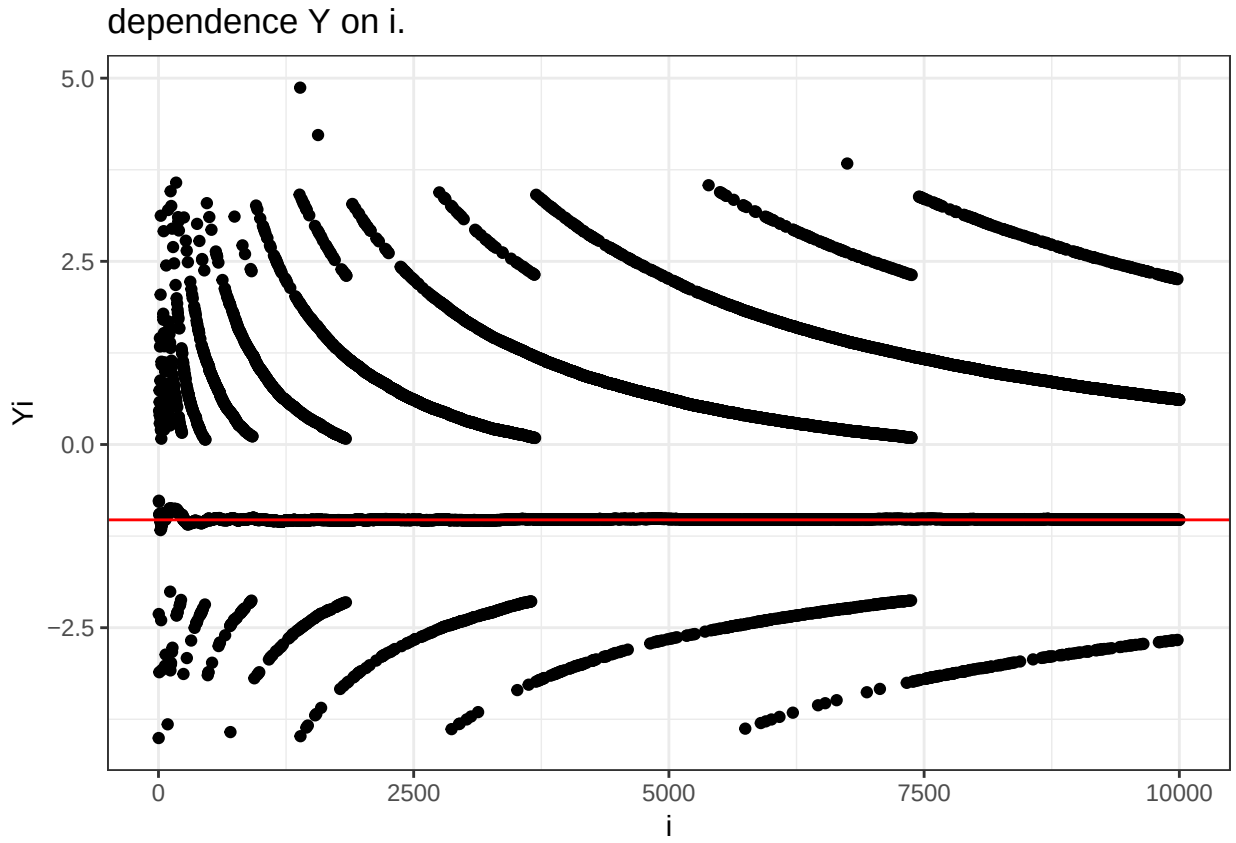
```
myvar = function(x) {
  n = length(x)
  var = 1/(n-1)*(sum(x^2)-(1/n)*(sum(x))^2)
  return(var)
}
```

Random draws

```
set.seed(123456789)
x = rnorm(10000, 10^8, 1)
```

Plot dependence

```
# creating Y
Y = c()
for (i in 1:length(x)) {
  Xi = x[1:i]
  Yi = myvar(Xi) - var(Xi)
  Y = c(Y, Yi)
}
# plotting dependence Yi on i
plotData = as.data.frame(cbind(i = 1:length(x),
                               Yi = Y))
library(ggplot2)
ggplot(data = plotData) +
  geom_point(aes(x = i, y = Yi)) +
  geom_abline(intercept = mean(Yi), slope = 0, color = "red") +
  theme_bw() +
  ggtitle("dependence Y on i.")
```




```
mean(Yi)
```

```
## [1] -1.0282732214275443
```

The function we created does not return the same values as the built-in function *var* as we have both smaller and larger values for different iterations. The red line represents the mean difference between variances calculated by the two different functions. It shows that our implemented function averagely returns slighlty higher variances than the built-in function *var*.

The function does not work well, although the variance and also difference in variance is small in relation to the mean of x (10^8). Ideally, the points should all lie on a horizontal line at y = 0.

On top of that, there is also no general pattern which describes how our returned function values developes in comparison to the returned values of the *var*-function when the input vector increases or decreases in length.

Alternative approach

Another way of estimating the sample variance is:

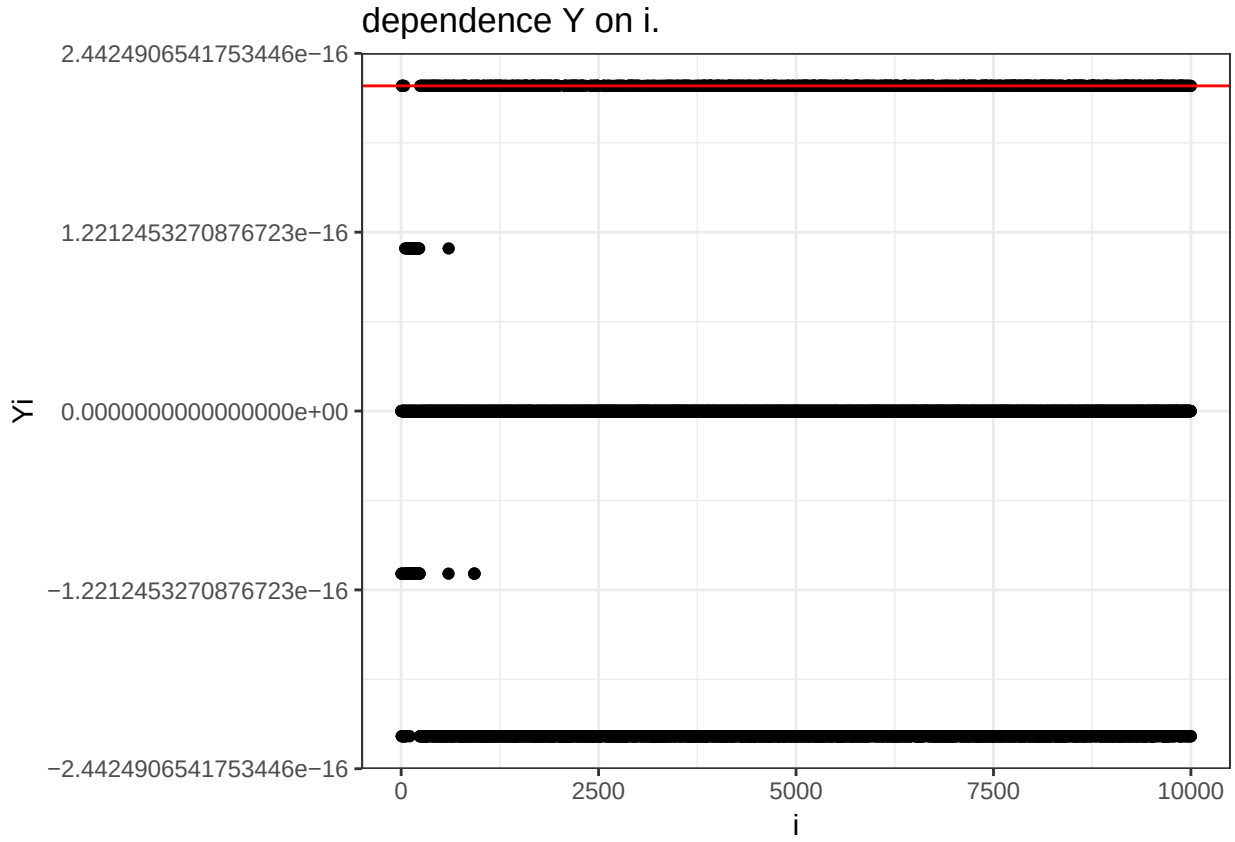
$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$

```
mynewvar = function(x) {
  n = length(x)
  # var = sum(x^2)/n - (sum(x)/n)^2
  var = (1/(n-1))*sum((x - mean(x))^2)
```

```
  return(var)
}
```

Using this new variance estimator, the differences to the built-in function *var* are plotted.

```
# creating Y
Y = c()
newvar = c()
actvar = c()
for (i in 1:length(x)) {
  Xi = x[1:i]
  Yi = mynewvar(Xi) - var(Xi)
  Y = c(Y, Yi)
}
# plotting dependence Yi on i
plotData = as.data.frame(cbind(i = 1:length(x),
                               Yi = Y))
library(ggplot2)
ggplot(data = plotData) +
  geom_point(aes(x = i, y = Yi)) +
  geom_abline(intercept = mean(Yi), slope = 0, color = "red") +
  theme_bw() +
  ggtitle("dependence Y on i.")
```



We see now that the variance estimator is much better, as it follows the built-in function `var` almost exactly. The differences to the function `var` are extremely small. The mean difference confirms this as

```
## [1] "mean(Yi): 2.22044604925031e-16"
```

Furthermore, since we draw values from a distribution with standard deviations $s = 1 \Rightarrow s^2 = 1$ the variance should be 1 (at least for large samples). This is also true, as the yellow line converges to y = 1 with larger samples.

Question 4: Linear Algebra

Import data

```
data = read.csv("tecator.csv")
```

Linear equation system

```
library(dplyr)
X = data[,-c(which(colnames(data) == "Sample"), which(colnames(data) == "Protein"))] %>%
  as.matrix()
y = data[,which(colnames(data) == "Protein")]
A = t(X) %*% X
b = t(X) %*% y
```

Inverse of A using `solve`

$$A\beta = b \Rightarrow \beta = A^{-1}b$$

```
solve(A)
```

When using the *solve*-function we get an error message with the following printout: `system is computationally singular`. The linear system has either zero or infinite solutions and the equation cannot be solved within the 64-bit system..

Kappa value

```
kappa(A)
```

```
## [1] 1157834236871692.2
```

A kappa value of size infinity means that the matrix is singular and thus non-inversible. Since our kappa value is very large, the solution of the linear system is 'prone to large numerical errors'. The reasoning fits with previous conclusions.

Repeat of 2-4 with scaled data

We scale the data to a distribution with mean 0 and variance 1. For the `channel-` variables it did little difference, but `Fat`and `Moisture` were significantly changed.

```r
X_ = apply(X, 2, scale)

A = t(X_) %*% X_
b = t(X_) %*% y

is.matrix(solve(A))
```

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

```r
kappa(A)
```

```
## [1] 490471520662.05011
```

With scaled variables, it is possible to calculate the inverse of A. The kappa value remains high, but small in comparison to the previous one.

Since the linear system is close to impossible to solve with highly correlated variables we check the correlation. Most variables have a correlation larger than 0.9.

The scaling does not change pairwise correlation between the X-variables, only the range/scale. The reason for non-scaled matrix `X` being computationally singular while the scaled matrix `X_` is not has to do with the size of the measurements. The scaled data has made the (approximate) calculations possible, and the matrix has an inverse.

Appendix

All relevant code is shown in code chunks in the document.