Introduction to Simulation with Variance Estimation

Exercise 1

Bosse Behrens, st.id 12347333

2024W

Task 1

First the four algorithms need to be implemented as shown on the lecture.
Algorithm 1 (precise method):

```
precise <- function(x) {
  n <- length(x)
  x_mean <- (1/n)*sum(x)
  s_X_squared = sum(((x - x_mean)^2))/(n-1)
  return(s_X_squared)
}
```

Algorithm 2 (excel method):

```
excel <- function(x){
  n <- length(x)
  P_1 <- sum(x^2)
  P_2 <- ((sum(x))^2)/n
  s_X_squared <- (P_1 - P_2)/(n-1)
  return(s_X_squared)
  }
```

Algorithm 3 (scale invariant shift) with setting a second argument for the scale invariant shift we need later. Default value is 0, which results in the same algorithm as the excel method:

```
shift <- function(x, c_shift=0){
  n <- length(x)
  P_1 <- sum((x - c_shift)^2)
  P_2 <- ((sum(x - c_shift))^2)/n
  s_X_squared <- (P_1 - P_2)/(n-1)
  return(s_X_squared)
}
```

Algorithm 4 (incremental updating; online) with first calculating the mean of first two elements in x and then the variance. The variance of the first 2 elements in x needs to be divided by n-1 = 2-1 = 1 (therefore not explicitly shown):

```
online <- function(x){
  n <- length(x)
  x_mean <- (x[1] + x[2])/2
  s_X_squared <- ((x[1]-x_mean)^2 + (x[2]-x_mean)^2)
  for (i in 3:n) {
    x_mean_old <- x_mean
    s_X_squared_old <- s_X_squared
```

```
    x_mean <- x_mean_old + (x[i]-x_mean_old)/i
    s_X_squared <- (s_X_squared_old * ((i-2)/(i-1))) + ((x[i]-x_mean_old)^2)/i
  }
  return(s_X_squared)
}
```

Now we implement a wrapper function that simply creates a vector out of the values for the variance calculated by the four implemented algorithms, as well as the value of the built in R function var(). For the shift algorithm we set x[1] as the second argument (as specified in the task).

```
wrapper <- function(x){
  values <- c(precise(x), excel(x), shift(x, x[1]), online(x), var(x))
  return(values)
}
```

The two datasets (vectors of 100 elements each, normally distributed around different means) are created. The seed is set to my student id.

```
set.seed(1234733)
x1 <- rnorm(100)
set.seed(1234733)
x2 <- rnorm(100, mean=1000000)
```

Now we create a function that checks for the equality of the values in the implemented algorithms and the var() function. The three methods used are '==', which checks in a vector element-wise if the items are the same as the one compared to. The 'all.equal' function checks if the two arguments have the same value, but within a certain error-threshold. If not within that threshhold, it shows the numeric difference. The 'identical' function simply checks if the two arguments are the same, making it in this implementation the same as the '==' method.

```
equal_check <- function(x){
  alg_names <- c("precise", "excel", "shift", "online", "R built-in var")
  equal_signs <- wrapper(x) == var(x) # '==' checks element-wise, so no need for a loop
  all_equal <- c()
  identical_call <- c()
  for (i in 1:5){  # 'all.equal' and 'identical' only compare arguments, therefore looping
    all_equal <- append(all_equal, all.equal(wrapper(x)[i], var(x)))
    identical_call <- append(identical_call, identical(wrapper(x)[i], var(x)))
  }

  results <- data.frame(   # building the table to show the results
    Method = alg_names,
    Equal_Signs = equal_signs,
    All_Equal = all_equal,
    Identical = identical_call
  )
  return(results)
}
```

Now we execute the function on x1 and x2.

```
equal_check(x1)
```

```
##           Method Equal_Signs All_Equal Identical
## 1        precise        TRUE      TRUE      TRUE
## 2          excel        TRUE      TRUE      TRUE
## 3          shift        TRUE      TRUE      TRUE
```

```
## 4         online       FALSE      TRUE     FALSE
## 5 R built-in var        TRUE      TRUE      TRUE
```

```
equal_check(x2)
```

```
##            Method Equal_Signs                                   All_Equal Identical
## 1         precise        TRUE                                        TRUE      TRUE
## 2           excel       FALSE Mean relative difference: 0.000254866     FALSE
## 3           shift        TRUE                                        TRUE      TRUE
## 4          online       FALSE                                        TRUE     FALSE
## 5 R built-in var         TRUE                                        TRUE      TRUE
```

The results show us that the mathematically precise first algorithm is in both cases equal to R's var() function, meaning it gives the same values. The online algorithm that starts with only the first two elements of the data and updates the estimations for every new elements yields in both cases FALSE for the identical check, but TRUE for the 'all.equal' check. This means that the error is within the default setting of all.equal's threshold (1.490116e-08). The excel and shift algorithms both yield TRUE for all comparison methods checking for equality given x1. Given x2 though the shift method still returns TRUE for all comparisons, while the excel method returns FALSE for all three with the all.equal method also showing the mean relative difference that was higher than the internal error threshold. This is curious as the only difference between the excel and shift algorithms is the scale invariant shift by the first element of the argument x. Also it seems to make a difference in this case between the mean of 0 for x1 and the mean of 1,000,000 in x2. This will be further explored in the next tasks.

Task 2

After comparing the variance estimate equality of the different algorithms, we now want to explore the computation time, for which we are using the microbench package.

```
library(microbenchmark)

comp_time_fun <- function(x){
  comp_time <- microbenchmark(
  precise = precise(x),
  excel = excel(x),
  shift = shift(x, x[1]), # using the first as the shift parameter
  online = online(x),
  var = var(x),
  unit = "ns"
  )
  return(comp_time)
}
c1 <- comp_time_fun(x1)
c2 <- comp_time_fun(x2)
print(c1)
```

```
## Unit: nanoseconds
##      expr   min    lq  mean median    uq   max neval
##   precise   900  1000  1175   1100  1200  3500   100
##     excel   900  1000  1231   1100  1200  6600   100
##     shift  1300  1400  1960   1500  1700 15600   100
##    online 10300 10400 10655  10500 10600 14900   100
##       var  5500  5900  6387   6000  6250 22700   100
```

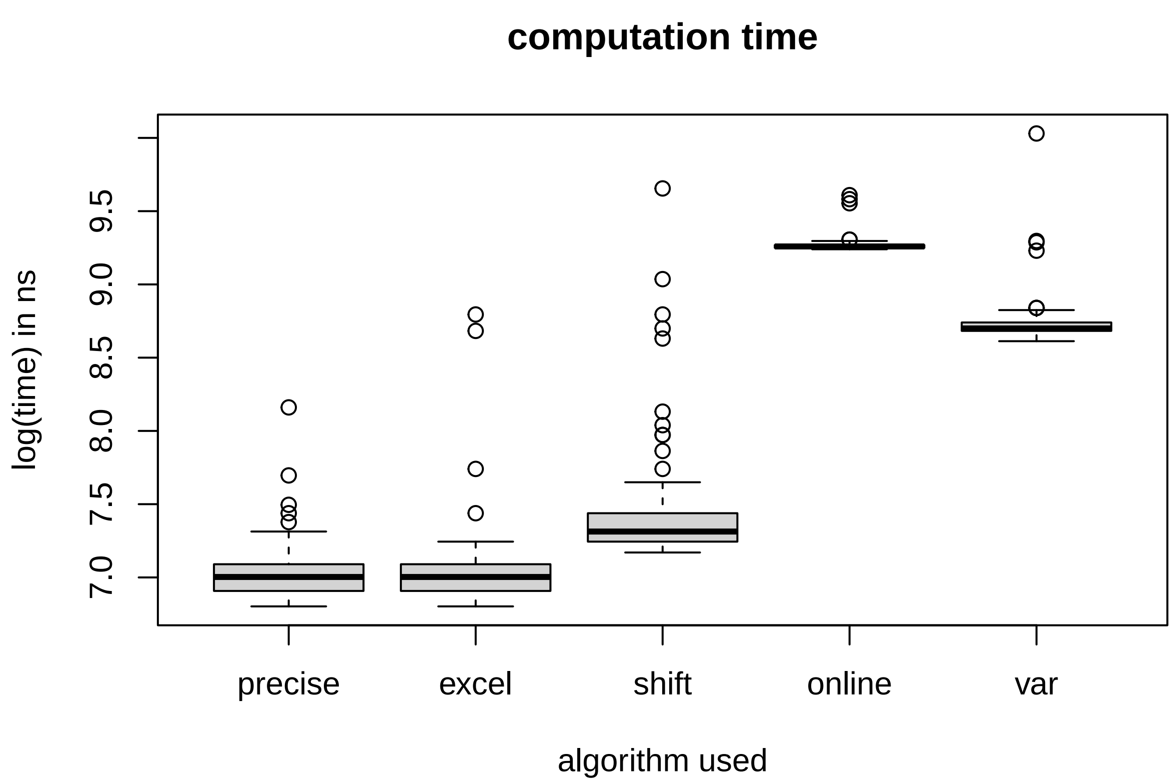
```
print(c2)
```

```
## Unit: nanoseconds
```

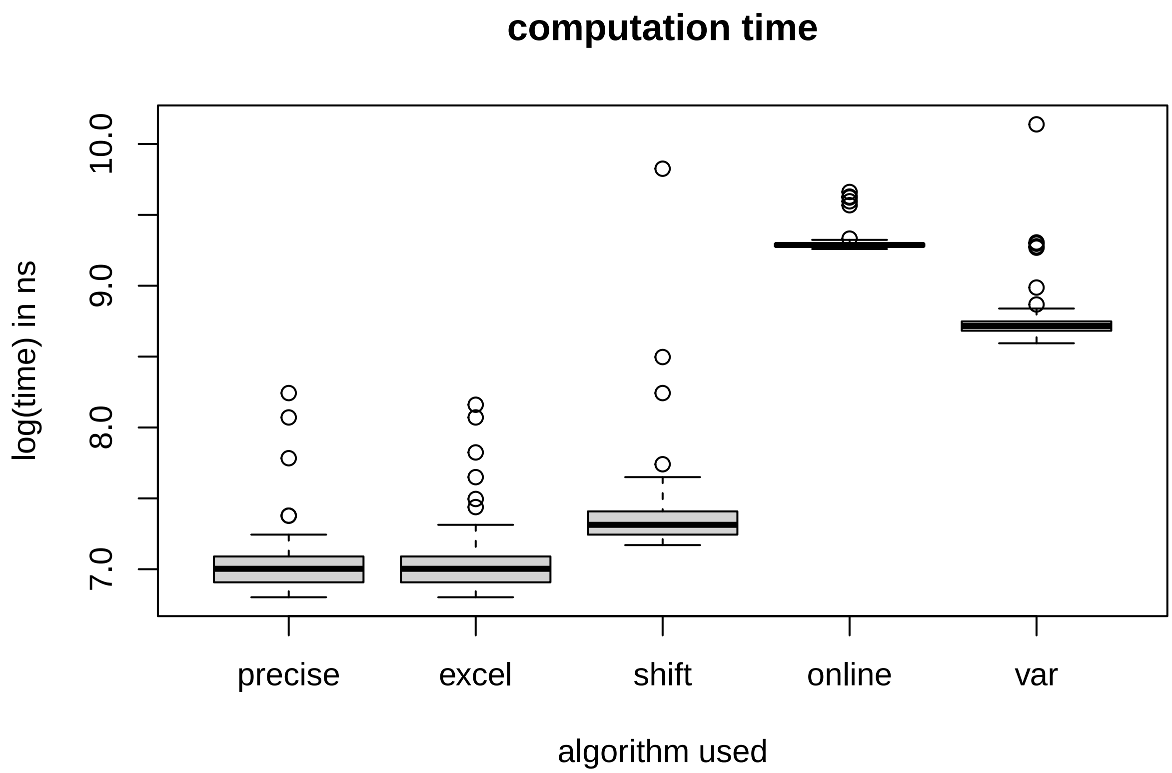
```
##      expr   min    lq  mean median    uq   max neval
##   precise   900  1000  1180   1100  1200  3800   100
##     excel   900  1000  1189   1100  1200  3500   100
##     shift  1300  1400  1772   1500  1650 18500   100
##    online 10500 10700 11014  10800 10900 15700   100
##       var  5400  5900  6488   6100  6300 25300   100
```

Now we create the boxplots out of the tables.

```
comp_boxplot <- function(x){
  comp_time_df <- as.data.frame(x)
  boxplot(log(time) ~ expr, data = comp_time_df,
        main = "computation time",
        xlab = "algorithm used",
        ylab = "log(time) in ns")
}
comp_boxplot(c1)
```




```
comp_boxplot(c2)
```



We can observe that the precise and excel algorithms are the fastest, performing very similar to each other. Both have a los median and mean, with some few outliers being higher. The shift algorithm is a bit slower on average by about a third of th time. A reason could be, that it first needs to calculate the shifted dataset, before performing the same algorithm as the excel method. The online algorithm that updates the estimates each element after another is as expected the slowest, since it needs calculate the variance and mean anew for every added element. Only very few outliers of the other methods are higher than its mean and median. The built-in var() functions is also on average more than four times slower than the fastest algorithms (precise, excel), but still faster than the online algorithm. Remembering the first task, there seems to be a trade-off, since the shift and online algorithms performed well on both datasets (only some very small deviation for the online method) and being slower, while the excel method was the fastest but performed poorly on x2.

Task 3

We now want to explore the shift algorithm some more, since it was only slightly slower than the fastest algorithms, but still producing good estimates. The only difference from the excel algorithm, which seemed more unstable on different datasets, is the scale-invariant shift in the data. Previously we simply used the first element of the data, but considering this could also be 0, resulting in the excel algorithm this doesn't seem like a good choice. Therefore we now explore different values for the shift parameter. I decided on using the mean, the median and the maximum and minimum values of the data. For this we implement a function that just like with the estimation quality of the different algorithms in task 1 now does the same for different shift values in the shift algorithm.

```
compare_shift <- function(x){
  value_names <- c("mean", "median", "max", "min", "first element of x")
  values <- c(shift(x, mean(x)), shift(x,median(x)),
              shift(x, max(x)), shift(x, min(x)), shift(x, x[1]))
  equal_signs <- values == var(x)
```

```
  all_equal <- c()
  identical_call <- c()
  for (i in 1:5){
    all_equal <- append(all_equal, all.equal(values[i], var(x)))
    identical_call <- append(identical_call, identical(values[i], var(x)))
  }
  results <- data.frame( # creating a table out of the comparisons
    Value = value_names,
    Equal_Signs = equal_signs,
    All_Equal = all_equal,
    Identical_Call = identical_call
  )
  return(results)

}
```

Executing the function on both datasets.

```
compare_shift(x1)
```

```
##                Value Equal_Signs All_Equal Identical_Call
## 1               mean        TRUE      TRUE           TRUE
## 2             median        TRUE      TRUE           TRUE
## 3                max       FALSE      TRUE          FALSE
## 4                min       FALSE      TRUE          FALSE
## 5 first element of x        TRUE      TRUE           TRUE
```

```
compare_shift(x2)
```

```
##                Value Equal_Signs All_Equal Identical_Call
## 1               mean        TRUE      TRUE           TRUE
## 2             median        TRUE      TRUE           TRUE
## 3                max       FALSE      TRUE          FALSE
## 4                min       FALSE      TRUE          FALSE
## 5 first element of x        TRUE      TRUE           TRUE
```

As we can observe, the mean and median both perform well and deliver estimates that are deemed identical by the identical function which is the most strict of the three comparison methods, only returning TRUE if the arguments are the exact same. The estimates using the max and min values are still within the all.equal error threshold and therefore do not deviate by much, but are still worse than median and mean since the comparison using '==' and 'identical' both return FALSE. The first element of the data also performs well also performs well in this case, but as mentioned before this is random and can therefore also be the max or min element.

Now we are benching the computation times using the different values by making use of the microbench package again.

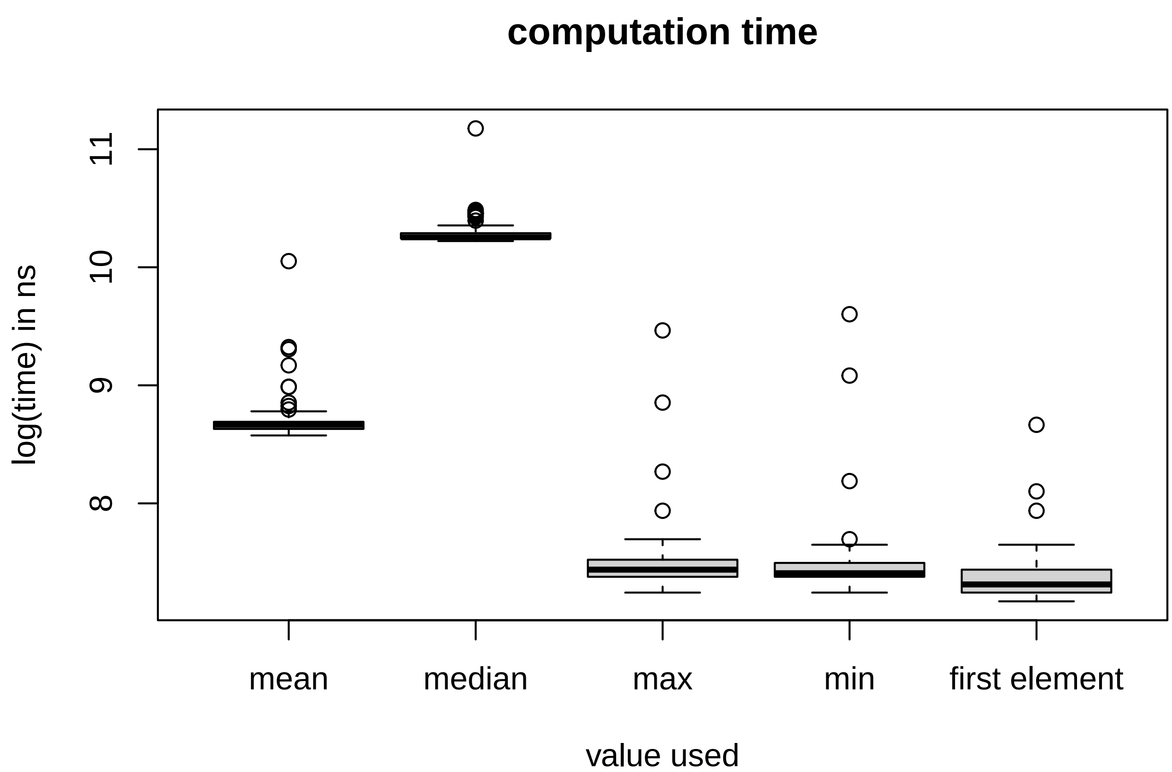
```
library(microbenchmark)
scale_value_time <- function(x){
  comp_time_scale <- microbenchmark(
  "mean" = shift(x, mean(x)),
  "median" = shift(x, median(x)),
  "max" = shift(x, max(x)),
  "min" = shift(x, min(x)),
  "first element" =shift(x, x[1]),
  unit = "ns"
  )
```

```r
  print(comp_time_scale) # showing the data table
  comp_time_scale_df <- as.data.frame(comp_time_scale)
  boxplot(log(time) ~ expr, data = comp_time_scale_df, # plotting boxplots
          main = "computation time",
          xlab = "value used",
          ylab = "log(time) in ns")
}
```

Executing on x1.

```r
scale_value_time(x1)
```

```
## Unit: nanoseconds
##           expr   min    lq  mean median    uq   max neval
##           mean  5300  5600  6132   5800  5950 23200   100
##         median 27500 28100 29671  28400 29400 71400   100
##            max  1400  1600  1907   1700  1850 12900   100
##            min  1400  1600  1908   1650  1800 14800   100
##  first element  1300  1400  1604   1500  1700  5800   100
```

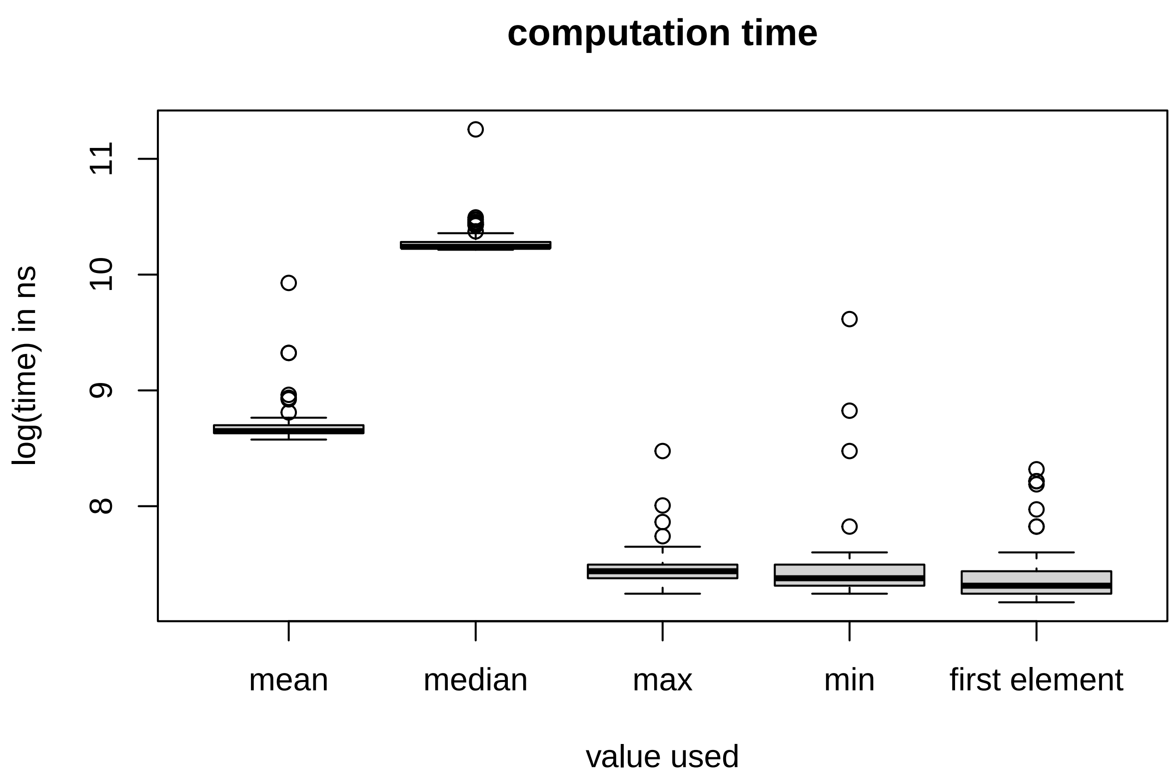


Executing on x2.

```r
scale_value_time(x2)
```

```
## Unit: nanoseconds
##           expr   min    lq  mean median    uq   max neval
##           mean  5300  5600  6082   5700  6000 20500   100
##         median 27300 27700 29551  28000 29200 77200   100
##            max  1400  1600  1764   1700  1800  4800   100
```

```
##            min  1400  1500  1886   1600  1800 15000   100
##  first element  1300  1400  1657   1500  1700  4100   100
```



As we can observe using the first value is obviously the fastest, since it doesn't need to be calculated. Min and max values are only slighjtly slwoer while using mean and median is slower with the mean being a bit faster than median. This probably simply results from the time needed to compute these different values. While max and min values are a simple search in the given array, mean and median are more complex to be calculated, mean a bit lesser so.
Overall the mean seems like the best choice, being only about 3 times slower than min and max values but having good estimates. The median also has good estimates but is slightly slower in computing. Also given the mean is the best choice (as stated in the lecture) and the median often being close to the mean this only makes sense.

Task 4

We now want to compare the different condition numbers of the data sets x1 and x2 as well as a third one where the approximation doesn't hold due to a very small mean (which we will set to 1/1,000,000).

```r
set.seed(12347333)
x3 <- rnorm(100, mean=1/1000000) # setting the third dataset with a very small mean

cond_numb <- function(x){
  cn <- sqrt(1 + ((mean(x)^2)*length(x))/var(x)) # generating the condition number
  return(cn)
}
cond_apprx <- function(x){
  cna <- mean(x)*sqrt(length(x)/var(x)) # generating the approximation
  return(cna)
```

```r
}

cond_names <- c("x1", "x2", "x3")
cond_values <- c(cond_numb(x1), cond_numb(x2), cond_numb(x3))
cond_apprx_values <- c(cond_apprx(x1), cond_apprx(x2), cond_apprx(x3))
results <- data.frame(   # creating a table out of the values
  vector = cond_names,
  "Condition Number" = cond_values,
  "Condition Number approximation" = cond_apprx_values)
print(results)
```

```
##   vector Condition.Number Condition.Number.approximation
## 1     x1     1.269179e+00                  -7.815468e-01
## 2     x2     1.067370e+07                   1.067370e+07
## 3     x3     1.628287e+00                   1.285036e+00
```

First we can see that the approximation of the condition number does not work for a mean of 0 (since it has to be nonzero for it to be legible) and also being a bad approximation for very data with a very small mean (x3). For x2 which has a big mean it is a very good approximation. For the condition number itself we can observe that it is close to 1 for x1 and very big (e+07) for x2. The condition number is an application of the derivative and formally the asymptotic worst-case relative change in output for a relative change in input. That means for x1 this worst-case change in output only deviates extremely few from the input while this asypmtotic relative worst-case change is very big in x2. The condition number measures robustness and stability with respect to input values and their perturbances. If we now compare this with our results from task 1 it makes sense that all algorithms performed well on x1. In these cases of well-defined data and problems fast algorithms like the excel method perform well. On the other hand for ill-defined data and problems, e.g. missing values, high condition number with small values, etc. it might be a better idea to use more robust and stable algorithms like the online method. They will perform better even if more slowly while for example the excel algorithm performs poorly. Also the shift algorithm with the mean as a parameter might be a good choice, since the shift by the mean will give the best condition number possible while the algorithm itself is still faster than an online algorithm.