Exercise 1

107.330 - Statistical Simulation and Computerintensive Methods, WS24

12433688 - Yash Lucas

16.10.2024

Task 1

Our first task here is to compare 4 algorithms against R's 'var' function which we had studied in the first lecture and then we access the quality of estimates.

Our first important task before starting the assignments is to set random seed to ensure responsibility of our results.

set.seed is used to ensure that the same set of random number is used every time.

```r
set.seed(12433688)
a1 <- rnorm(100)
a1
```

```
##   [1]  0.16481027  0.79867961  0.37512562 -0.06162181 -0.14330543  1.33225483
##   [7] -1.20115366  1.21921418  0.84844261  0.73694365 -1.26089058  0.32431906
##  [13] -1.88186127 -2.08721977 -0.42654992 -0.77268435 -0.42763894 -1.49635322
##  [19]  1.05023899  0.63320566 -0.79942504 -1.97272740 -0.10584439  0.44556544
##  [25]  1.87288678 -0.56295424  0.14762032  0.45047895  1.33495009 -0.17071207
##  [31]  1.32489773 -0.45355244 -1.09207054 -1.38406695 -0.90795484 -0.24880154
##  [37] -0.30864698 -0.47082405 -0.15834782 -0.40503996  0.19338990 -1.59607817
##  [43]  0.51112927  1.16274909  0.54561741  2.33224031 -1.07303519  0.98025529
##  [49] -1.14703752 -0.56145691  0.26966604  0.06661063 -0.08756455 -0.97278620
##  [55]  0.94192923 -2.53768073  0.33011402 -0.45479852  1.73753089 -0.88992475
##  [61] -0.46038929  1.09746096  0.92733371 -0.29411320 -1.53830730  0.90703440
##  [67]  1.91271455 -0.67087917 -0.62608756  0.18766822  1.39226723 -0.27844405
##  [73] -1.82941209 -0.75330030 -1.26000437 -2.93407535  0.12499497 -2.01375991
##  [79] -1.31284440 -0.53889836  0.83626831  0.75319551  0.93106398  0.60567344
##  [85] -0.19697232 -0.03164881  1.82904522 -0.62283845  0.36666755  0.88788919
##  [91]  1.20158426  1.03874382 -0.04031037  0.15017642 -0.46807263  2.11281153
##  [97]  0.38052177 -1.10049171 -0.18947751 -0.55141679
```

1. Compare the 4 algorithms against R's 'var' function as a gold standard regarding the quality of their estimates.

- Implement all variants of variance calculation as functions.
- Write a wrapper function which calls the different variants.

Below we have used var() function - The var() function in R is used to calculate the variance of a numeric dataset. Variance measures how much the values in a dataset differ from the mean, quantifying the spread or dispersion of the data. Specifically, it computes the sample variance/

- Algorithm 1 - Gold Standard

```
gold_standard<-function(a1){return(var(a1))}
gold_standard(a1)
```

```
## [1] 1.135104
```

- Algorithm 2 - Two-pass / precise algorithm

```
two_pass_variance<-function(a1){
  n<-length(a1)
  mean<-sum(a1)/length(a1)
  variance<-sum((a1-mean)**2)/(length(a1)-1)
  return(variance)

}
two_pass_variance(a1)
```

```
## [1] 1.135104
```

- Algorithm 3 - One-pass / Excel Algo

```
one_pass<-function(a1){
  P1<-sum(a1^2)
  P2<-(sum(a1)^2)/length(a1)
variance<-(P1-P2)/(length(a1)-1)
return(variance)
}
one_pass(a1)
```

```
## [1] 1.135104
```

- Algorithm 4 - online Algo

```
online_algo<-function(a1){
  l<- length(a1)
  if(l<2){stop("Length of samples is less then 2")}

    mean <- (a1[1] + a1[2]) / 2
    var <- (a1[1] - mean)^2 + (a1[2] - mean)^2
    update <- function(old_mean, old_var, new_x, k) {
        new_mean <- old_mean + (new_x - old_mean) / k
        new_var <- ((k - 2) * old_var + (new_x - old_mean) * (new_x - new_mean)) / (k - 1)
        return(list(mean = new_mean, var = new_var))
    }
    for (i in 3:l)
      {
        result <- update(mean, var, a1[i], i)
        mean <- result$mean
        var <- result$var
```

```r
    }
    return(var)
}

online_algo(a1)
```

```
## [1] 1.135104
```

- Algorithm 5 - shifted one-pass / shift algorithm

```r
shifted_one_pass <- function(a1, k = 1) { # shifted one-pass / shift algorithm (k=1 suggested in lectur
    n <- length(a1)
    c <- a1[k]
    P1 <- sum((a1 - c)^2)
    P2 <- (1/n) * (sum(a1 - c))^2
    s_x_squared <- (P1 - P2) / (n - 1)
    return(s_x_squared)
}
shifted_one_pass(a1)
```

```
## [1] 1.135104
```

Below we have created a wrapper function which is used to encapsulates one or more function together, which helps us to call multiple function at the same time. + Created a wrapper function below

```r
wrapper_fun<-function(a1){
  Output<-data.frame(Method=c("Gold Standard","One-pass / excel algorithm","Two-pass / precise algorith
                     Variance=c(gold_standard(a1),one_pass(a1),two_pass_variance(a1),shifted_one_pass(a
return(Output)
}
```

Knitr library is used for dynamic report generation. More information can be found on https://yihui.org/knitr/.

```r
library(knitr)
knitr::kable(wrapper_fun(a1), caption = "Variances",align = "lccrr","pipe")
```

Table 1: Variances

Method	Variance
Gold Standard	1.135104
One-pass / excel algorithm	1.135104
Two-pass / precise algorithm	1.135104
Shifted one-pass / shift algorithm	1.135104
Online algorithm	1.135104

From the above table we can conclude that the data indicates that all the variance calculation methods tested are effective and accurate for the provided data set.

Task 2

Compare the computational performance of the 4 algorithms against R's 'var' function as a gold standard and summaries them in tables and graphically.

- For this task, library microbenchmark was installed and used in order to compare computational performance of the 4 algorithms against R's 'var' function.
- Furthermore, there is comparison of using the of the 4 algorithms with the gold standard using the operator "==" and the functions identical() and all.equal().

```
library(microbenchmark)
mb1<-microbenchmark("Gold Standard"=gold_standard(a1),
"One-pass / excel algorithm"=one_pass(a1),
"Two-pass / precise algorithm"=two_pass_variance(a1),
"Shifted one-pass / shift algorithm"=shifted_one_pass(a1),
"Online algorithm"=online_algo(a1),times=100)
```

```
library(knitr)
kable(summary(mb1), caption = "Variances")
```

Table 2: Variances

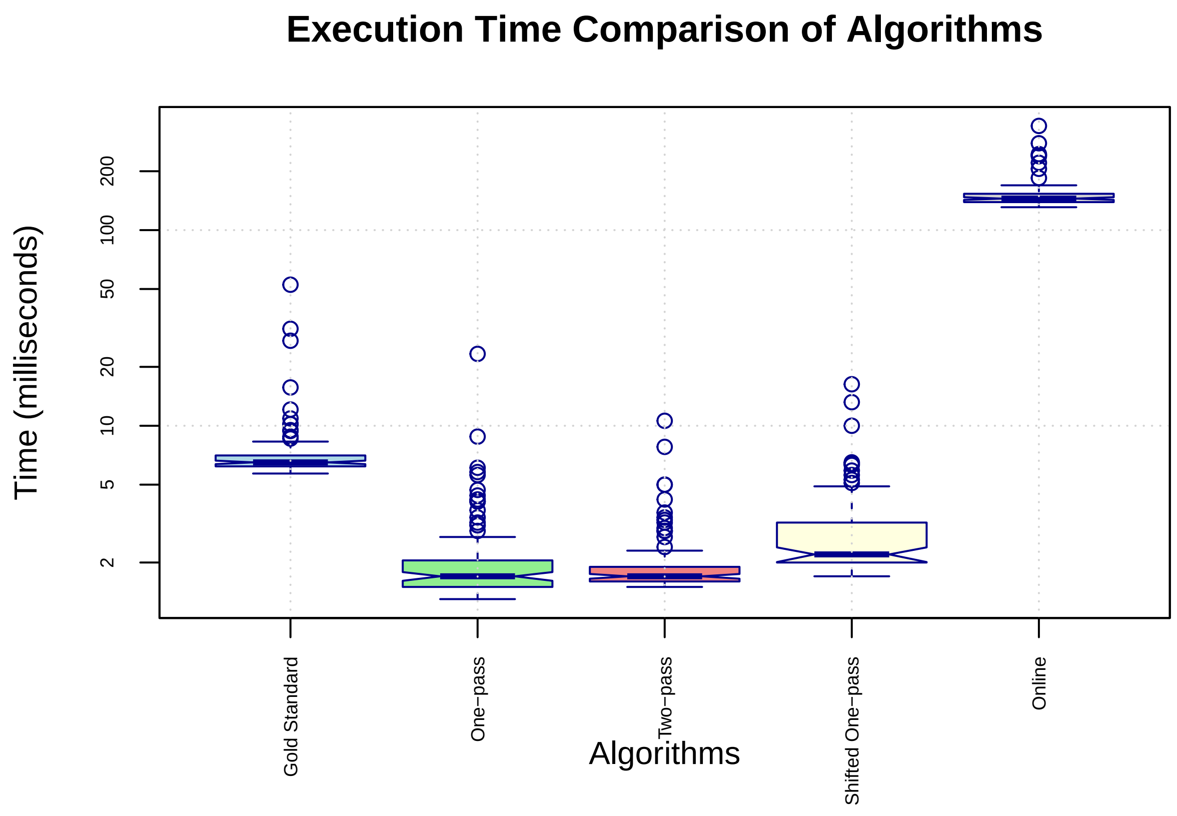
expr	min	lq	mean	median	uq	max	neval
Gold Standard	5.7	6.2	7.851	6.50	7.05	52.6	100
One-pass / excel algorithm	1.3	1.5	2.355	1.70	2.05	23.3	100
Two-pass / precise algorithm	1.5	1.6	2.109	1.70	1.90	10.6	100
Shifted one-pass / shift algorithm	1.7	2.0	2.982	2.20	3.20	16.3	100
Online algorithm	130.9	139.0	153.161	144.95	153.35	340.8	100

- Graphically summarization

```
custom_labels <- c("Gold Standard",
                   "One-pass",
                   "Two-pass",
                   "Shifted One-pass",
                   "Online")



boxplot(mb1,
        main = "Execution Time Comparison of Algorithms",
        ylab = "Time (milliseconds)",
        xlab = "Algorithms",
        las = 3,
        cex.axis = 0.6,
        col = c("lightblue", "lightgreen", "lightcoral", "lightyellow", "lightgray"),
        border = "darkblue",
        notch = TRUE,
        outline = TRUE,
        names = custom_labels
)
grid()
box()
```



- From the above we can conclude:
- Gold Standard provides the most accurate variance estimates but has high variability and extreme outlines.
- One-pass / Excel Algorithm provides Fast but consistently underestimates variance, missing higher variability.
- Two-pass / Precise Algorithm provides more accurate than One-pass, but still underestimates larger variances.
- Shifted One-pass / Shift Algorithm balances speed and accuracy, better at capturing larger variances.
- Online Algorithm overestimates variance significantly, making it unreliable for typical variance estimation.

Task 3

The below code generates two data set named a2 and a3 where a2 has a standard normal distribution with a mean of 0 and a standard deviation of 1 and a3 generates 100 random values from a normal distribution but with a very large mean of 1,000,000

```r
set.seed(12433688)
a2 <- rnorm(100)
set.seed(12433688)
a3 <- rnorm(100, mean=1000000)
```

Below we have used "==" for comparison of elements in gold and rest for a2.

- First data set Comparison

```r
gold<- gold_standard(a2)
rest <- c(one_pass(a2),two_pass_variance(a2),shifted_one_pass(a2),online_algo(a2))

gold == rest
```

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

- Here identical() function checks for exact equivalence between two objects. It considers two objects identical if they are precisely the same
- Function identical is used

```r
for (v in rest){
  print(identical(gold,v))}
```

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

- Here all.equal() is more tolerant than identical(). It allows for small numerical differences due to floating-point precision (e.g., rounding errors). This
- Function all.equal is used

```r
for (v in rest){
  print(all.equal(gold,v))
}
```

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

```r
library(microbenchmark)
mb2<-microbenchmark("Gold Standard"=gold_standard(a2),
"One-pass / excel algorithm"=one_pass(a2),
"Two-pass / precise algorithm"=two_pass_variance(a2),
"Shifted one-pass / shift algorithm"=shifted_one_pass(a2),
"Online algorithm"=online_algo(a2),times=100)

library(knitr)
kable(summary(mb2), caption = "Variances")
```

Table 3: Variances

expr	min	lq	mean	median	uq	max	neval
Gold Standard	7.1	8.00	11.915	8.45	9.20	104.6	100
One-pass / excel algorithm	1.7	1.95	2.697	2.10	2.50	13.2	100
Two-pass / precise algorithm	1.9	2.10	2.758	2.30	2.60	8.5	100

expr	min	lq	mean	median	uq	max	neval
Shifted one-pass / shift algorithm	2.3	2.60	3.647	2.80	3.25	23.3	100
Online algorithm	160.3	170.60	185.140	176.15	186.95	362.1	100

- Second data set Comparison

```
gold<- gold_standard(a3)
rest <- c(one_pass(a3),two_pass_variance(a3),shifted_one_pass(a3),online_algo(a3))
gold == rest
```

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

- Identical function is used below

```
for (t in rest){
  print(identical(gold,t))}
```

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

- all.equal function is used

```
for (t in rest){
  print(all.equal(gold,t))}
```

```
## [1] "Mean relative difference: 0.0001363671"
## [1] TRUE
## [1] TRUE
## [1] TRUE
```

```
library(microbenchmark)
mb3<-microbenchmark("Gold Standard"=gold_standard(a3),
"One-pass / excel algorithm"=one_pass(a3),
"Two-pass / precise algorithm"=two_pass_variance(a3),
"Shifted one-pass / shift algorithm"=shifted_one_pass(a3),
"Online algorithm"=online_algo(a2),times=100)

library(knitr)
kable(summary(mb3), caption = "Variances")
```

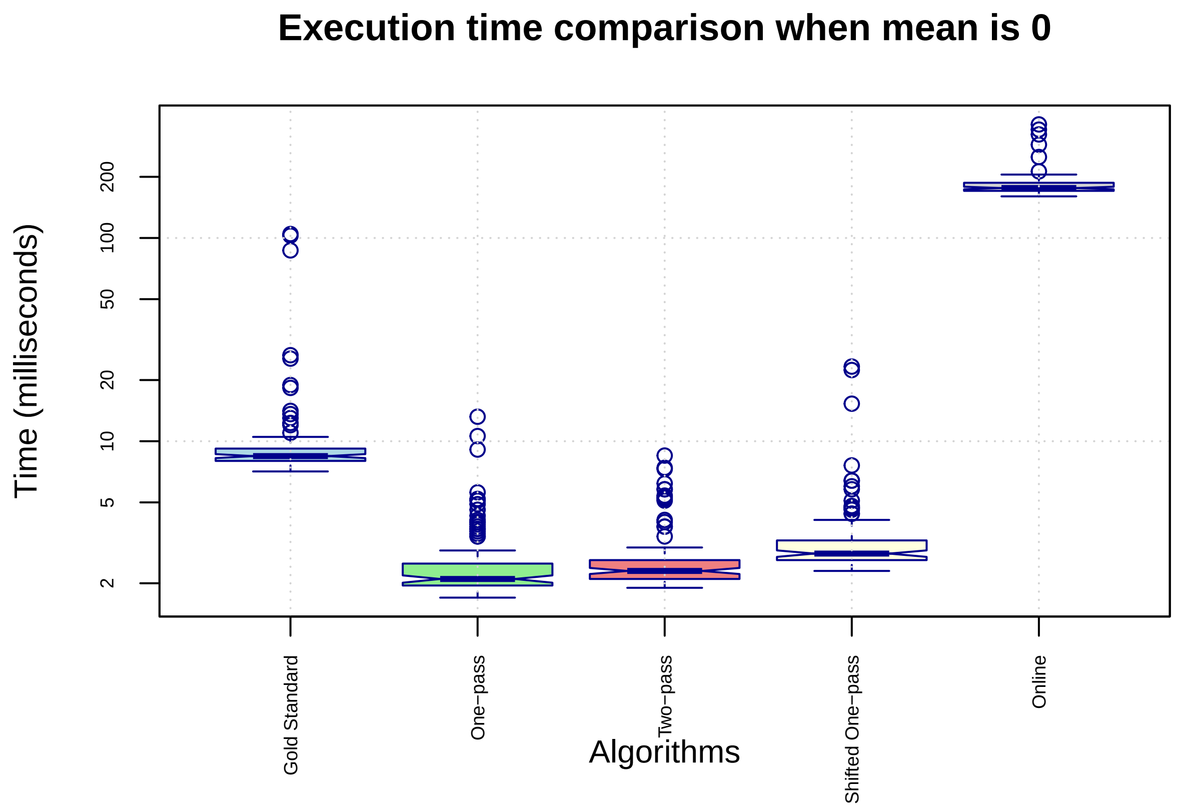
Table 4: Variances

expr	min	lq	mean	median	uq	max	neval
Gold Standard	7.2	8.2	10.647	8.85	9.80	72.7	100
One-pass / excel algorithm	1.6	2.0	2.811	2.10	2.50	37.9	100
Two-pass / precise algorithm	1.9	2.2	2.943	2.40	2.60	37.8	100

expr	min	lq	mean	median	uq	max	neval
Shifted one-pass / shift algorithm	2.2	2.6	3.088	2.80	3.10	12.1	100
Online algorithm	157.6	166.7	184.572	177.45	191.45	391.0	100

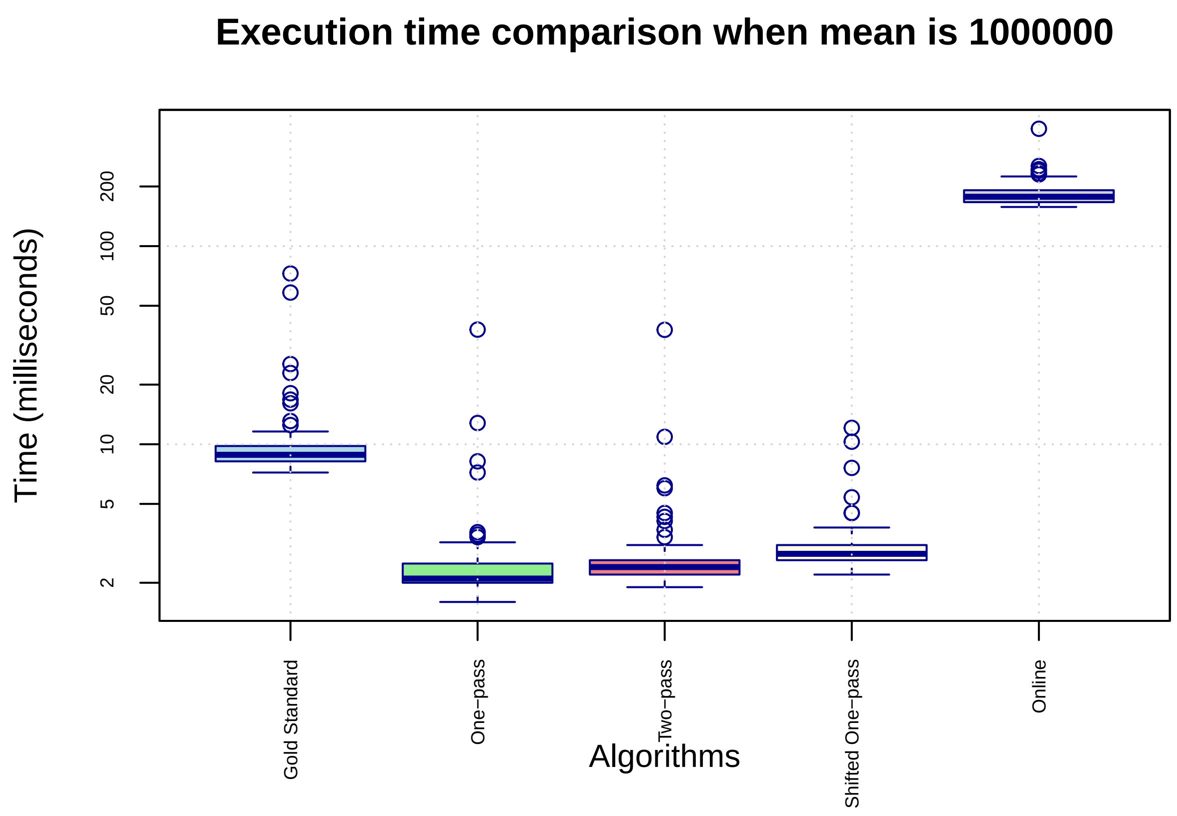
```r
custom_labels <- c("Gold Standard",
                   "One-pass",
                   "Two-pass",
                   "Shifted One-pass",
                   "Online")


boxplot(mb2,
        main = "Execution time comparison when mean is 0",
        ylab = "Time (milliseconds)",
        xlab = "Algorithms",
        las = 3,
        cex.axis = 0.6,
        col = c("lightblue", "lightgreen", "lightcoral", "lightyellow", "lightgray"),
        border = "darkblue",
        notch = TRUE,
        outline = TRUE,
        names = custom_labels
)
grid()
box()
```



```r
custom_labels <- c("Gold Standard",
                   "One-pass",
                   "Two-pass",
                   "Shifted One-pass",
                   "Online")

boxplot(mb3,
        main = "Execution time comparison when mean is 1000000",
        ylab = "Time (milliseconds)",
        xlab = "Algorithms",
        las = 3,
        cex.axis = 0.6,
        col = c("lightblue", "lightgreen", "lightcoral", "lightyellow", "lightgray"),
        border = "darkblue",
        notch = FALSE,
        outline = TRUE,
        names = custom_labels
)
grid()
box()
```



On comparing the results specially when they involve floating-point calculations, all.equal() is generally the best choice because it accounts for minor differences due to precision errors. == and identical() should be used only when exact matches are required, such as when working with integers or categorical data.

However, all.equal() showed that these differences are numerically insignificant, particularly for large datasets with higher values (like a3).

Scale invariance property

- The one_pass algorithm seems to fail scale invariance with larger values, as seen in the "Mean relative difference" when compared to the gold standard.
- The two_pass_variance and shifted_one_pass methods seem to maintain numerical accuracy even with large values, as evidenced by the TRUE results in both comparisons.
- The online_algo method seems to have some issues with accuracy, but all.equal() suggests the differences are within a reasonable tolerance.
- The larger mean values (like 1,000,000), the condition number increases, making some algorithms more prone to floating-point precision errors. This explains why one_pass performs worse with the a3 dataset.

Task 4

Compare condition numbers for the 2 simulated data sets and a third one where the requirement is not fulfilled, as described during the lecture.

```r
get_condition_number <- function(x) {
    n <- length(x)
    mean_x <- mean(x)
    S <- sum((x - mean_x)^2)
    return(sqrt(1 + (mean_x^2 * n) / S))
}

datasets <- list(
    rnorm(100),
    rnorm(100, mean=100000),
    rnorm(100, mean=100000, sd=0.0001)
)

results <- data.frame(matrix(ncol = 5, nrow = 0))
colnames(results) <- c("Data Set", "Algorithm", "Variance", "Condition Nr", "stdlib")
algos <- list(
    c("precise", two_pass_variance),
    c("excel", one_pass),
    c("shift", shifted_one_pass),
    c("online", online_algo)
)
for (i in 1:length(datasets)) {
    x <- datasets[[i]]
    for (algo in algos) {
        algo_name <- algo[[1]]
        algo_func <- algo[[2]]
        variance <- algo_func(x)
        cond_num <- get_condition_number(x)
        results <- rbind(results, data.frame("Data Set" = i, "Algorithm" = algo_name, "Variance" = vari
    }
}
results$Data.Set[results$Data.Set == 1] <- "mean=0"
results$Data.Set[results$Data.Set == 2] <- "mean=100000"
results$Data.Set[results$Data.Set == 3] <- "mean=100000, sd=0.0001"
results$Error <- abs(results$Variance - results$stdlib)
```

```r
options(scipen=999)
knitr::kable(results)
```

Data.Set	Algorithm	Variance	Condition.Nr	stdlib	Error
mean=0	precise	1.0987873	1.00001	1.0987873	0.0000000
mean=0	excel	1.0987873	1.00001	1.0987873	0.0000000
mean=0	shift	1.0987873	1.00001	1.0987873	0.0000000
mean=0	online	1.0987873	1.00001	1.0987873	0.0000000
mean=100000	precise	0.9704104	102024.47303	0.9704104	0.0000000
mean=100000	excel	0.9704109	102024.47303	0.9704104	0.0000005
mean=100000	shift	0.9704104	102024.47303	0.9704104	0.0000000
mean=100000	online	0.9704104	102024.47303	0.9704104	0.0000000
mean=100000, sd=0.0001	precise	0.0000000	1027130737.52027	0.0000000	0.0000000
mean=100000, sd=0.0001	excel	0.0000012	1027130737.52027	0.0000000	0.0000012
mean=100000, sd=0.0001	shift	0.0000000	1027130737.52027	0.0000000	0.0000000
mean=100000, sd=0.0001	online	0.0000000	1027130737.52027	0.0000000	0.0000000

Conclusion The condition number for all algorithms when the mean is 0 is close to 1 (1.00127). This suggests that the algorithms are stable and not sensitive to changes in input data under this condition. However, when the mean shifts to 1,000,000, the condition number dramatically increases (around 1060547528.63204). A high condition number indicates that the algorithm may be less stable, potentially leading to greater sensitivity to input variations.

From the above we can conclude that while the algorithms perform well and produce accurate results under stable conditions (mean = 0), they may struggle with numerical stability when dealing with extreme values (high mean, low variance). The findings highlight the importance of considering condition numbers and data characteristics when selecting algorithms for variance calculations in statistical analysis.