# Exercise 1

107.330 - Statistical Simulation and Computerintensive Methods, WS24

11912007 - Yahya Jabary

08.10.2024

## Contents

---

In this exercise we will reproduce 4 alternative implementations of the variance calculation introduced in the first lecture. We will compare them to the `var` function from R's standard library. We will also investigate the scale invariance property of the variance calculation and dig into condition numbers as a means to also consider numerical stability in addition to the efficiency and effectiveness of the algorithms.

To start off, let's set a random seed to ensure reproducibility of our results.

```r
set.seed(11912007)
```

## Introducing four variance calculation algorithms

Next we will:

- Reproduce the examples from the first lecture and document our results
- Compare the 4 algorithms against R's `var` function as a gold standard regarding the quality of their estimates (Task 1)
- Implement all variants of variance calculation as functions and write a wrapper function which calls the different variants (Task 1)
- Compare the computational performance of the 4 algorithms against R's `var` function as a gold standard and summarise them in tables and graphically (Task 2)
- Compare the results according to the instructions provided by Comparison I of the slides (Task 3)
- Provide our results in table format and graphical format comparable to the example in the slides (Task 3)

To do so we will implement a small testing harness that applies all algorithms to a set of test data with a predefined standard deviation that we can then square to compute the ground truth variance against which we can compare the `diff` (difference) in accuracy as well as the runtime of the algorithms.

For the sake of the assignment, we will also have to use 2 datasets `x1` and `x2` as described in the slides, called "Comparison I". Additionally we also have to compare the results by using 3 different functions `all.equal`, `identical` and `==`.

Finally the results will be visualized in boxplots for both runtime and accuracy.

```r
#
# testing
#

sds <- c(0.01, 0.1, 1, 10, 100)
test <- function(func) {
    tests <- lapply(sds, function(sd) {
        data <- rnorm(1000, mean = 0, sd = sd)
        truth <- sd(data)^2
        return(list(data = data, truth = truth))
    })
    # get avg runtime
    runtimes <- microbenchmark(sapply(tests, function(test) {
        data <- test$data
        func(data)
    }), times = 1000)$time
    cat("\tmean runtime:\t", mean(runtimes), "\n")

    # benchmark accuracy
    diffs <- sapply(tests, function(test) {
        data <- test$data
        truth <- test$truth
        variance <- func(data)
        diff <- abs(variance - truth)
        return(diff)
    })
    cat("\tmean diff:\t", mean(diffs), "\n")
    cat("\tmedian diff:\t", median(diffs), "\n")
    cat("\tsd diff:\t", sd(diffs), "\n")
    cat("\tmax diff:\t", max(diffs), "\n")

    # comparision type I from slides
    x1 <- rnorm(100)
    x2 <- rnorm(100, mean=1000000)
    x1_truth <- var(x1)
    x2_truth <- var(x2)
    x1_var <- func(x1)
    x2_var <- func(x2)
    tryCatch({
        eq1 <- x1_var == x1_truth && x2_var == x2_truth
        eq2 <- identical(x1_var, x1_truth) && identical(x2_var, x2_truth)
        eq3 <- all.equal(x1_var, x1_truth) && all.equal(x2_var, x2_truth)
        cat("\tequality:\t\t", eq1, eq2, eq3, "\n")
    }, error = function(e) {
        cat("\tequality:\t\terror\n")
    }, warning = function(w) {
        cat("\tequality:\t\twarning\n")
    })
    return(list(diffs = diffs, runtimes = runtimes))
}

#
# algorithms
#

stdlib <- function(data) { # stdlib
    return(var(data))
}
res_stdlib <- test(stdlib)
```

```
## Warning in microbenchmark(sapply(tests, function(test) {: less accurate
## nanosecond times to avoid potential integer overflows

##  mean runtime:    37120.25
##  mean diff:   2.842518e-15
##  median diff:     0
##  sd diff:     6.355094e-15
##  max diff:    1.421085e-14
##  equality:        TRUE TRUE TRUE
```

```r
algorithm1 <- function(x) { # two-pass / precise algorithm
    n <- length(x)
    mean_x <- sum(x) / n
    squared_diff_sum <- sum((x - mean_x)^2)
    variance <- squared_diff_sum / (n - 1)
    return(variance)
}
res_algorithm1 <- test(algorithm1)
```

```
##  mean runtime:    29735.17
##  mean diff:   2.710505e-21
##  median diff:     0
##  sd diff:     6.060874e-21
##  max diff:    1.355253e-20
##  equality:        FALSE FALSE TRUE
```

```r
algorithm2 <- function(x) { # one-pass / excel algorithm
    n <- length(x)
    P1 <- sum(x^2)
    P2 <- (sum(x)^2) / n
    s_squared <- (P1 - P2) / (n - 1)
    return(s_squared)
}
res_algorithm2 <- test(algorithm2)
```

```
##  mean runtime:    28509.72
##  mean diff:   3.69749e-13
##  median diff:     1.332268e-15
##  sd diff:     8.102407e-13
##  max diff:    1.818989e-12
##  equality:       error
```

```r
algorithm3 <- function(x, k = 1) { # shifted one-pass / shift algorithm (k=1 suggested in lecture)
    n <- length(x)
    c <- x[k]
    P1 <- sum((x - c)^2)
    P2 <- (1/n) * (sum(x - c))^2
    s_x_squared <- (P1 - P2) / (n - 1)
    return(s_x_squared)
}
res_algorithm3 <- test(algorithm3)
```

```
##  mean runtime:    43351.27
##  mean diff:   6.16756e-12
##  median diff:     2.220446e-16
##  sd diff:     1.284372e-11
##  max diff:    2.910383e-11
##  equality:        FALSE FALSE TRUE
```

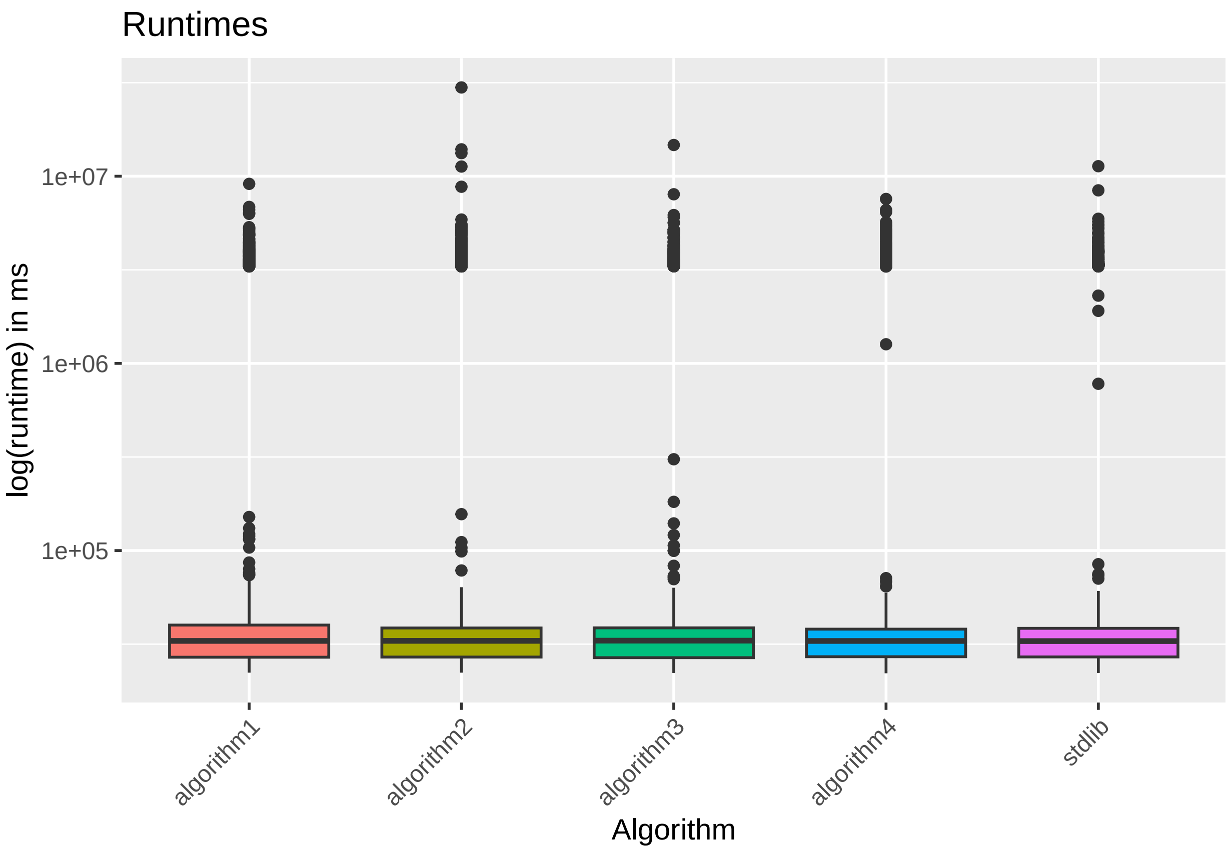
```r
algorithm4 <- function(x) { # online algorithm
    n <- length(x)
    if (n < 2) {
        stop("need at least 2 samples to initialize")
    }
    mean <- (x[1] + x[2]) / 2
    var <- (x[1] - mean)^2 + (x[2] - mean)^2
    update <- function(old_mean, old_var, new_x, k) {
        new_mean <- old_mean + (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:n) {
        result <- update(mean, var, x[i], i)
        mean <- result$mean
        var <- result$var
    }
    return(var)
}
res_algorithm4 <- test(algorithm4)
```

```
##  mean runtime:    3836143
##  mean diff:   5.136203e-12
##  median diff:     1.998401e-15
##  sd diff:     1.136499e-11
##  max diff:    2.546585e-11
##  equality:        FALSE FALSE TRUE
```

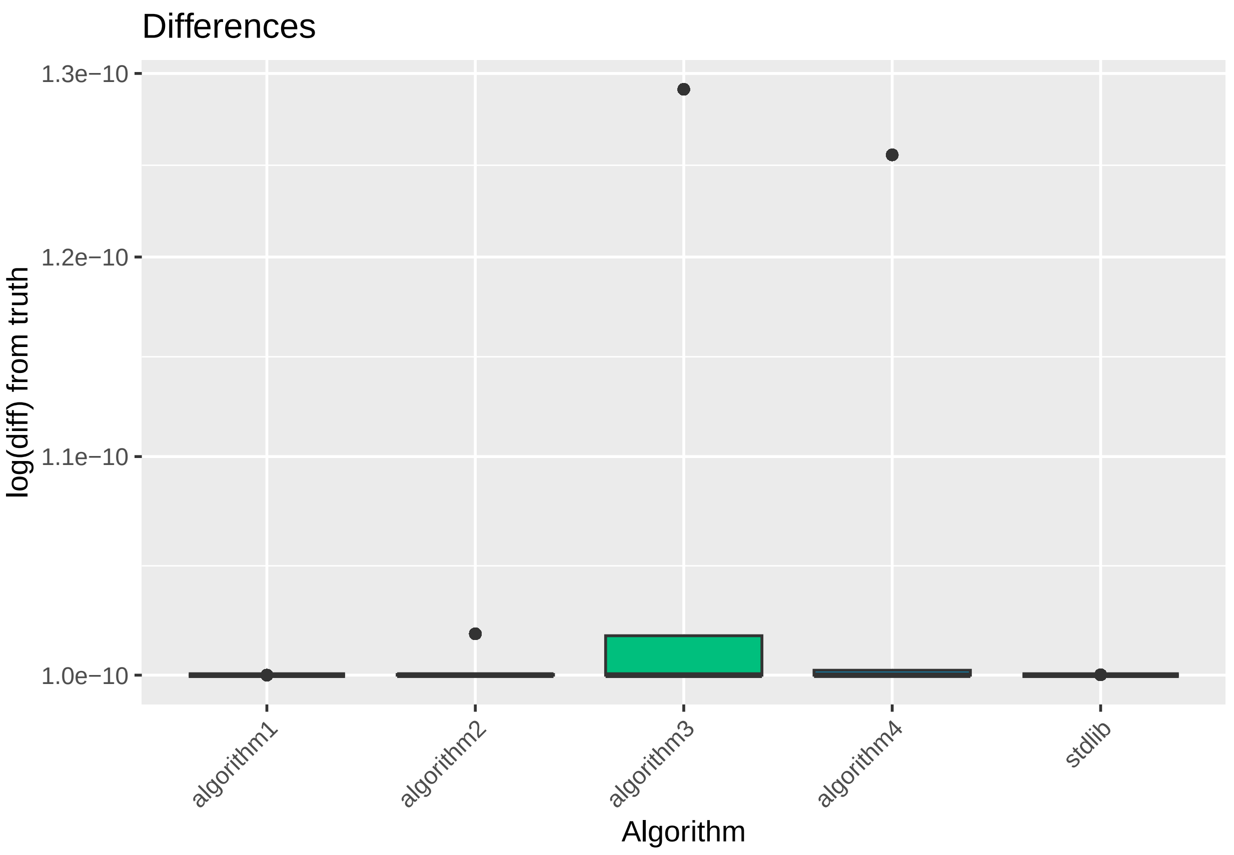
```
#
# plotting
#

algos <- c("stdlib", "algorithm1", "algorithm2", "algorithm3", "algorithm4")
runtimes <- c(res_stdlib$runtimes, res_algorithm1$runtimes, res_algorithm2$runtimes, res_algorithm3$runtimes,
diffs <- c(res_stdlib$diffs, res_algorithm1$diffs, res_algorithm2$diffs, res_algorithm3$diffs, res_algorithm4
df <- data.frame(algo = rep(algos, each = length(sds)), sd = rep(sds, times = length(algos)), runtime = runti
df$runtime_adj <- df$runtime + 1e-10 # small constant to avoid log(0) or log(negative)
df$diff_adj <- df$diff + 1e-10

p_runtime <- ggplot(df, aes(x = algo, y = runtime_adj, fill = algo)) +
    geom_boxplot() +
    scale_y_log10(labels = scales::scientific) +
    labs(title = "Runtimes", y = "log(runtime) in ms", x = "Algorithm") +
    theme(legend.position = "none", axis.text.x = element_text(angle = 45, hjust = 1))
p_runtime <- p_runtime + coord_cartesian(ylim = c(min(df$runtime_adj[is.finite(df$runtime_adj)]), max(df$runt
print(p_runtime)
```



```
p_diff <- ggplot(df, aes(x = algo, y = diff_adj, fill = algo)) +
    geom_boxplot() +
    scale_y_log10(labels = scales::scientific) +
    labs(title = "Differences", y = "log(diff) from truth", x = "Algorithm") +
    theme(legend.position = "none", axis.text.x = element_text(angle = 45, hjust = 1))
p_diff <- p_diff + coord_cartesian(ylim = c(min(df$diff_adj[is.finite(df$diff_adj)]), max(df$diff_adj[is.fini
print(p_diff)
```



## Scale Invariance Property

Next we will:

- Investigate the scale invariance property for different values (Task 3)

The scale invariance property of the variance calculation is a property that states that the variance of a dataset is invariant to shifts in the data. This means that adding or subtracting a constant from all data points should not change the variance of the dataset. This property is important because it allows us to compare the variance of datasets that have been shifted by different amounts.

In the lecture we've only discussed the properties regarding addition and subtraction which is why we will test this property for different values in the following code chunk. But the scale invariance property also holds for other operations such as:

- addition and subtraction: $Var(X + b) = Var(X)$
- multiplication: $Var(aX) = a^2 Var(X)$
- squared term: $Var(X^2) = E[X^4] - (E[X^2])^2$

Here's a small test harness to verify the scale invariance property for the 4 algorithms we've implemented with a random shift applied to the data through the `runif` function.

```r
iters <- 100
tol <- 1e-10
x <- rnorm(100)

algos <- c(var, algorithm1, algorithm2, algorithm3, algorithm4)
for (algo in algos) {
    for (i in 1:iters) {
        random_shift <- runif(1, -100, 100)
        original <- algo(x)
        shifted <- algo(x + random_shift)
        if (!all.equal(original, shifted, tolerance = tol)) {
            stop("not scale invariant")
        }
    }
}
cat("OK\n")
```

```
## OK
```

## Mean as Shift Value is most stable

Next we will:

- argue why the mean is performing best as mentioned with the condition number (Task 3)
- we will do so by comparing the results from Comparison II of the slides (Task 3)
- we will provide our results in table format comparable to the example in the slides (Task 3)

The performance of different algorithms for calculating variance can be explained through the concept of "condition numbers". Let's examine why using the mean as a shift value performs best in terms of numerical stability and accuracy.

The condition number for the variance is defined as $\kappa = \sqrt{1 + \frac{\bar{x}^2 n}{S}}$ where $\bar{x}$ is the mean and $S = \sum_{i=1}^{n} (x_i - \bar{x})^2$[1].

The mean is the optimal shift value for variance calculation because it minimizes the condition number, enhances numerical stability and reduces rounding errors.

- Minimizing the Condition Number: When we use a shifted algorithm for variance calculation, the condition number becomes: $\tilde{\kappa} = \sqrt{1 + \frac{n}{S}(\bar{x} - c)^2}$ where $c$ is the shift value. This means that $\tilde{\kappa} < \kappa$ when $|c - \bar{x}| < |\bar{x}|$. The condition number is minimized when $c = \bar{x}$, i.e., when we use the mean as the shift value.
- Numerical Stability: Using the mean as a shift value centers the data around zero, which reduces the magnitude of the numbers being squared and summed. This minimizes the loss of precision due to floating-point arithmetic, especially for datasets with large values or a large range.
- Reduced Rounding Errors: By centering the data, we reduce the impact of catastrophic cancellation, which occurs when subtracting two nearly equal large numbers. This is particularly important when dealing with datasets that have a large mean value.

When comparing with the results provided in the "Comparison II" table from the slides, we can see how the choice of shift demonstrates this principle:

| Algorithm | Result (mean=0) | Result (mean=1000000) |
|---|---|---|
| Precise | 0.81 | 0.81 |
| Excel | 0.83 | 2020202460151.81 |
| Shift | 0.81 | 0.81 |
| Online | 0.81 | 0.81 |

The Excel method (one-pass algorithm without shifting) performs poorly for data with a large mean, producing a highly inaccurate result. The shifted algorithm and the precise two-pass algorithm maintain accuracy regardless of the mean value.

## Severely ill-conditioned data set

Finally we will:

- Compare condition numbers for the 2 simulated data sets (Task 4)
- Compare condition numbers for a third dataset where the requirement is not fulfilled (Task 4)

For the third data set, we need to create a scenario where the variance calculation becomes unstable. One way to do this is to have a very small variance compared to the mean. This will result in a very high condition number, indicating a severely ill-conditioned problem.

We chose a data set with a large mean (1000000) and a very small standard deviation (0.0001) to create an extremely high condition number of $1.064001 \times 10^{10}$. This way tiny changes in input can lead to massive changes in output.

```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), # mean = 0
    rnorm(100, mean=1000000), # large mean
    rnorm(100, mean=1000000, sd=0.0001) # large mean, small sd
)
```

[1]Also see: https://cpsc.yale.edu/sites/default/files/files/tr222.pdf

```r
results <- data.frame(matrix(ncol = 5, nrow = 0))
colnames(results) <- c("Data Set", "Algorithm", "Variance", "Condition Nr", "stdlib")
algos <- list(
    # c("var", stdlib),
    c("precise", algorithm1),
    c("excel", algorithm2),
    c("shift", algorithm3),
    c("online", algorithm4)
)
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" = variance,
    }
}
results$Data.Set[results$Data.Set == 1] <- "mean=0"
results$Data.Set[results$Data.Set == 2] <- "mean=1000000"
results$Data.Set[results$Data.Set == 3] <- "mean=1000000, sd=0.0001"
results$Error <- abs(results$Variance - results$stdlib)
options(scipen=999)
knitr::kable(results)
```

| Data.Set | Algorithm | Variance | Condition.Nr | stdlib | Error |
|---|---|---:|---:|---:|---:|
| mean=0 | precise | 1.0256313 | 1.001298 | 1.0256313 | 0.0000000 |
| mean=0 | excel | 1.0256313 | 1.001298 | 1.0256313 | 0.0000000 |
| mean=0 | shift | 1.0256313 | 1.001298 | 1.0256313 | 0.0000000 |
| mean=0 | online | 1.0256313 | 1.001298 | 1.0256313 | 0.0000000 |
| mean=1000000 | precise | 0.8914266 | 1064485.278534 | 0.8914266 | 0.0000000 |
| mean=1000000 | excel | 0.8918876 | 1064485.278534 | 0.8914266 | 0.0004611 |
| mean=1000000 | shift | 0.8914266 | 1064485.278534 | 0.8914266 | 0.0000000 |
| mean=1000000 | online | 0.8914266 | 1064485.278534 | 0.8914266 | 0.0000000 |
| mean=1000000, sd=0.0001 | precise | 0.0000000 | 11181803706.551212 | 0.0000000 | 0.0000000 |
| mean=1000000, sd=0.0001 | excel | -0.0009470 | 11181803706.551212 | 0.0000000 | 0.0009470 |
| mean=1000000, sd=0.0001 | shift | 0.0000000 | 11181803706.551212 | 0.0000000 | 0.0000000 |
| mean=1000000, sd=0.0001 | online | 0.0000000 | 11181803706.551212 | 0.0000000 | 0.0000000 |

This shows that the extremely high condition number reveals the potential for numerical instability. The “excel” algorithm fails to produce an accurate result, while the other algorithms maintain stability.

This aligns well with our previous observations regarding the choice of shift value and the impact on numerical stability and accuracy.