Exercise 1 - Introduction to Simulation with Variance Estimation

12433732 - Stefan Merdian

2024-10-09

In this exercise, we will implement four alternative methods for calculating variance, as introduced in the first lecture, and compare them to R's built-in var() function. Additionally, we will explore the scale invariance property of variance and examine the condition number to assess both the numerical stability and the computational efficiency of the algorithms.

Our tasks include:

1. - Implement all variance calculation methods as functions and create a wrapper function to call each variant.
   - Compare the four variance calculation algorithms against R's `var()` function for the quality of their estimates.
2. - Compare the computational performance of the algorithms against R's `var()` function, summarizing the results in tables and graphs.
3. - Investigate the *scale invariance* property and explain why using the mean minimizes the condition number.
4. - Compare condition numbers for the two simulated datasets, and a third dataset that does not meet the required conditions.

Task 1: Implementing Variance Calculation Algorithms

We will start by implementing all four variance algorithms covered in the lecture. The first one is the **precise two-pass algorithm**, which calculates the variance by first computing the mean, and then using it to calculate the variance in the second pass over the data.

Algorithm 1: Precise Two-Pass Algorithm

The two-pass algorithm proceeds in two steps:

1. **First pass**: Compute the sample mean.
2. **Second pass**: Compute the variance using the mean from the first pass.

```r
algorithmOne <- function(data) {
    n <- length(data)
    sample_mean <- sum(data) / n
    squared_diff_sum <- sum((data - sample_mean)^2)
    variance <- squared_diff_sum / (n - 1)

    return(variance)
}
```

Algorithm 2: One-Pass Algorithm (Excel-like)

The second algorithm we will implement is the **one-pass algorithm**. This approach calculates the variance in a single pass over the data. It is simpler and faster than the two-pass algorithm but is more prone to numerical instability, especially with large datasets or when the values are widely spread.

This algorithm proceeds in three steps:

1. Compute $P1 = \sum_{i=1}^{n} x_i^2$, the sum of squares of the data.

2. Compute $P2 = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2$, the square of the sum of the data divided by the number of observations.
3. Compute the sample variance using $s_x^2 = \frac{P1-P2}{n-1}$.

```r
algorithmTwo <- function(data) {
  n <- length(data)
  P1 <- sum(data^2)
  P2 <- (sum(data)^2) / n
  variance <- (P1 - P2) / (n - 1)

  return(variance)
}
```

Algorithm 3: Shifted One-Pass Algorithm

In this algorithm, we use a **shifted one-pass algorithm** to calculate variance. This approach improves numerical stability by introducing a shift value c, which helps reduce the risk of catastrophic cancellation during the computation, especially when the data values are large.

This algorithm proceeds in three steps:

1. Compute $P1 = \sum_{i=1}^{n}(x_i - c)^2$, the sum of squared differences between each data point and the shift value c.
2. Compute $P2 = \frac{1}{n}\left(\sum_{i=1}^{n}(x_i - c)\right)^2$, the square of the sum of shifted values divided by n.
3. Compute the sample variance using $s_x^2 = \frac{P1-P2}{n-1}$.

Note: The shift value c can be chosen to improve numerical stability. Initially, we will use the first data point x_1 as the shift value.

```r
algorithmThree <- function(data) {
  n <- length(data)
  c <- data[1]

  P1 <- sum((data - c)^2)
  P2 <- (sum(data - c))^2 / n
  s_x_squared <- (P1 - P2) / (n - 1)

  return(s_x_squared)
}
```

Algorithm 4: Online Algorithm

In this section, we will implement the **online algorithm** for variance calculation. This algorithm updates the mean and variance incrementally, as each new observation is added. It is particularly useful for streaming data or cases where we need to update statistics dynamically without storing the entire dataset.

This algorithm proceeds in three steps:

1. Compute the mean $\bar{x}_2$ and variance s_2^2 using the first two observations.
2. For each new observation x_i (from $i = 3$ to n), update the mean $\bar{x}_i$ and variance s_i^2.
3. Return the final variance s_n^2.

```r
algorithmFour <- function(data) {

n <- length(data)
  if (n < 2) {
    stop("Need at least two data points.")
  }

  mean <- (data[1] + data[2] ) / 2
  variance <- (((data[1]- mean)^2) + (data[2] - mean)^2) / 1

  for (i in 3:n) {
    old_mean <- mean
    mean <- old_mean + ((data[i] - old_mean) / i)
    variance <- ((i - 2) / (i - 1)) * variance + ((data[i] - old_mean)^2)/i
  }

 return(variance)

}
```

Wrapper Function

We will create a **wrapper function** that calls each of the four variance calculation algorithms as well as R's built-in `var()` function. This will allow us to compare the results of each algorithm more easily.

```r
variance_comparison <- function(data) {
  r_var <- var(data)
  algo1_var <- algorithmOne(data)
  algo2_var <- algorithmTwo(data)
  algo3_var <- algorithmThree(data)
  algo4_var <- algorithmFour(data)
  return(list(
    r_standard = r_var,
    precise = algo1_var,
    excel = algo2_var,
    shift = algo3_var,
    online = algo4_var
  ))
}
```

Generate the Datasets

We begin by generating two datasets:

- `x1`: A dataset of 100 random values generated from a standard normal distribution (mean = 0, standard deviation = 1).
- `x2`: A dataset of 100 random values generated from a normal distribution with a very large mean of 1,000,000.

These datasets will be used to compare the performance and accuracy of the variance algorithms, particularly to examine how they handle datasets with small and large means.

```r
set.seed(12433732)
x1 <- rnorm(100)

set.seed(12433732)
x2 <- rnorm(100, mean = 1000000)
```

Print Variance Results in Table

We print the the results for each algorithm and data set in a table.

```r
formatted_df <- data.frame(
  Algorithm = c("r_standard", 'precise', 'excel', 'shift', 'online'),
  x1_data = sprintf("%.20f", variance_comparison(x1)),
  x2_data = sprintf("%.20f", variance_comparison(x2))
)
library(knitr)
kable(formatted_df)
```

Algorithm	x1_data	x2_data
r_standard	0.81056581996153287406	0.81056581995641763250
precise	0.81056581996153287406	0.81056581995641763250
excel	0.81056581996153287406	0.81060606060606055223
shift	0.81056581996153287406	0.81056581995641763250
online	0.81056581996153331815	0.81056582000349552963

Comparison

To compare the results of the variance algorithms against R's built-in `var()` function, we define the `compare_variances()` function. This function checks whether the variance results from the custom algorithms are identical to, or approximately equal to, the result from `var()`. Three comparison methods are used:

1. **`identical()`**: Strict equality check.
2. **`all.equal()`**: Checks if two values are nearly equal, allowing for small numerical differences (useful for floating-point comparison).
3. **`==` (equal operator)**: A straightforward comparison that checks element-wise equality.

To show that, we will write a helper function, that takes all results and compare than.

```
compare_variances_wrapper <- function(variance_results) {
  r_var <- variance_results$r_standard
  comparison_results <- list()

  for (algo_name in names(variance_results)[-1]) {
    algo_var <- variance_results[[algo_name]]
    comparison_results[[algo_name]] <- c(
      Identical = identical(r_var, algo_var),
      All_Equal = all.equal(r_var, algo_var),
      Equal_Operator = all(r_var == algo_var)
    )
  }
  comparison_df <- do.call(rbind, comparison_results)
  comparison_df <- as.data.frame(comparison_df)

  kable(comparison_df)
}
```

Results for x1 data set

```
compare_variances_wrapper(variance_comparison(x1))
```

	Identical	All_Equal	Equal_Operator
precise	TRUE	TRUE	TRUE
excel	TRUE	TRUE	TRUE
shift	TRUE	TRUE	TRUE
online	FALSE	TRUE	FALSE

Results for x2 data set

```
compare_variances_wrapper(variance_comparison(x2))
```

	Identical	All_Equal	Equal_Operator
precise	TRUE	TRUE	TRUE

	Identical	All_Equal	Equal_Operator
excel	FALSE	Mean relative difference: 4.964514e-05	FALSE
shift	TRUE	TRUE	TRUE
online	FALSE	TRUE	FALSE

Task 2: Comparing Computational Performance of Variance Algorithms

In this task, we will evaluate the computational performance of the four variance calculation algorithms and compare them to R's built-in var() function, which will serve as the gold standard.

First we will get the `microbenchmark()` library.

```
if (!require(microbenchmark)) {
  install.packages("microbenchmark")
}
```

```
## Loading required package: microbenchmark
```

```
library(microbenchmark)
```

We will create a helper function to streamline the benchmarking process. This function will run the benchmarks and then produce a print-ready data frame summarizing the results for easy analysis and comparison. We will test each algorithm **10000** times.

```
benchmark_results <-function(data){
  microbenchmark(
    precise = algorithmOne(data),
    excel = algorithmTwo(data),
    shift = algorithmThree(data),
    online = algorithmFour(data),
    R_var = var(data),
    times = 10000
  )}

benchmarktable <- function(results) {
  benchmark_summary <- summary(results)
  benchmark_df <- as.data.frame(benchmark_summary)
  result <- kable(benchmark_df)
  result
}
```

Benchmarkresult x1_data set

```
x1_result <- benchmark_results(x1)
benchmarktable(x1_result)
```

expr	min	lq	mean	median	uq	max	neval
precise	1.9	2.2	3.00536	2.3	2.5	3781.8	10000
excel	1.8	2.0	2.48536	2.2	2.4	27.1	10000
shift	2.3	2.5	3.21782	2.7	3.5	48.4	10000
online	22.8	24.2	24.74202	24.3	24.6	80.2	10000
R_var	7.3	8.0	9.20959	8.3	8.7	3840.1	10000

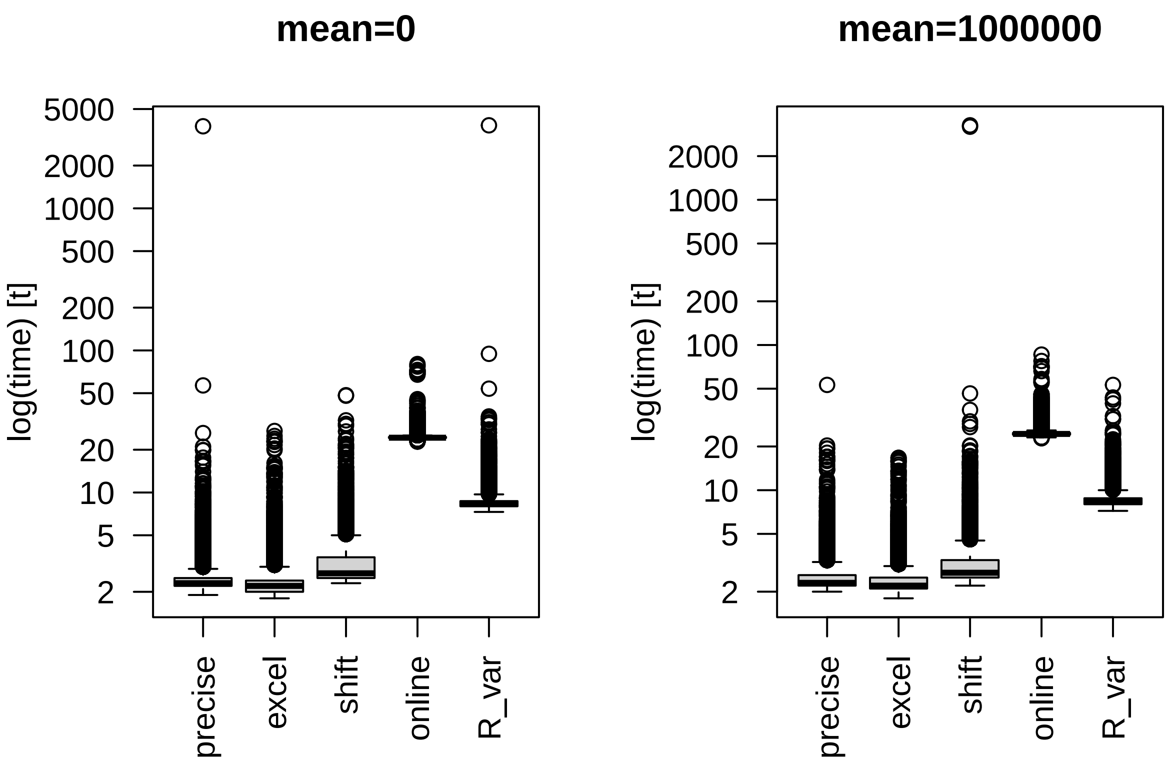
Benchmarkresult x2_data set

```r
x2_result <- benchmark_results(x2)
benchmarktable(x2_result)
```

expr	min	lq	mean	median	uq	max	neval
precise	2.0	2.2	2.61405	2.3	2.6	53.1	10000
excel	1.8	2.1	2.47913	2.2	2.5	16.7	10000
shift	2.2	2.5	3.80505	2.7	3.3	3258.9	10000
online	22.8	24.1	25.13456	24.4	24.8	85.9	10000
R_var	7.2	8.0	8.73397	8.3	8.8	53.1	10000

Boxplot of the results

```r
par(mfrow = c(1, 2))
boxplot(x1_result, main = "mean=0", las=2,xlab="")
boxplot(x2_result, main = "mean=1000000", las=2, xlab="")
```



Task 3: Investigate scale invariance

We will now investigate the scale invariance property for different values and argue why the mean is performing best as mentioned with the condition number.

Steps:

1. Writing functions to calculate the condition number for the shifted algorithm.
2. Modify the shifted algorithm to accept a shift value cc.
3. Test the algorithm with different values of c (shifted value).
4. Compute the condition number for each cc.
5. Compare and choose the best value of c that minimizes the condition number.

First, we define a function to calculate the condition number. This calculation differs from the non-shifted version, as it takes into account adjustments for shifting the dataset.

```r
calculate_condition_number_shifted <- function(data, shift) {
  n <- length(data)
  mean_data <- mean(data)
  S <- sum((data - mean_data)^2)

  if (S == 0) {
    return(Inf)
  } else {
    condition_number <- sqrt(1 + (n / S) * (mean_data - shift)^2)
    return(condition_number)
  }
}
```

Next, we modify the third algorithm, also known as the shifted algorithm, to allow it to accept different shift values (cc). This enables us to test how various shift values impact the condition number and overall numerical stability.

```r
algorithmThree_modified <- function(data, c) {
  n <- length(data)
  P1 <- sum((data - c)^2)
  P2 <- (sum(data - c)^2) / n
  variance <- (P1 - P2) / (n - 1)
  return(variance)
}
```

Now, we will create a helper function to compare the results for different shift values (cc) and store them in a data frame. This will allow us to visualize the results and gain better insights into how the choice of cc impacts the condition number and stability of the variance calculations.

```r
getResults_cValue <- function(data, c_values) {
  comparison_results <- data.frame(
    Shift_Value = numeric(),
    Condition_Number = numeric()
  )

  for (c in c_values) {
    variance <- algorithmThree_modified(data, c)

     condition_number <- calculate_condition_number_shifted(data, c)

    comparison_results <- rbind(
      comparison_results,
      data.frame(
        Shift_Value = c,
        Condition_Number = condition_number
      )
    )
  }

  return(comparison_results)
}
```

We generate a range of different cc-values, starting from slightly below to slightly above the mean of the dataset, to test the impact of these shifts on the condition number.

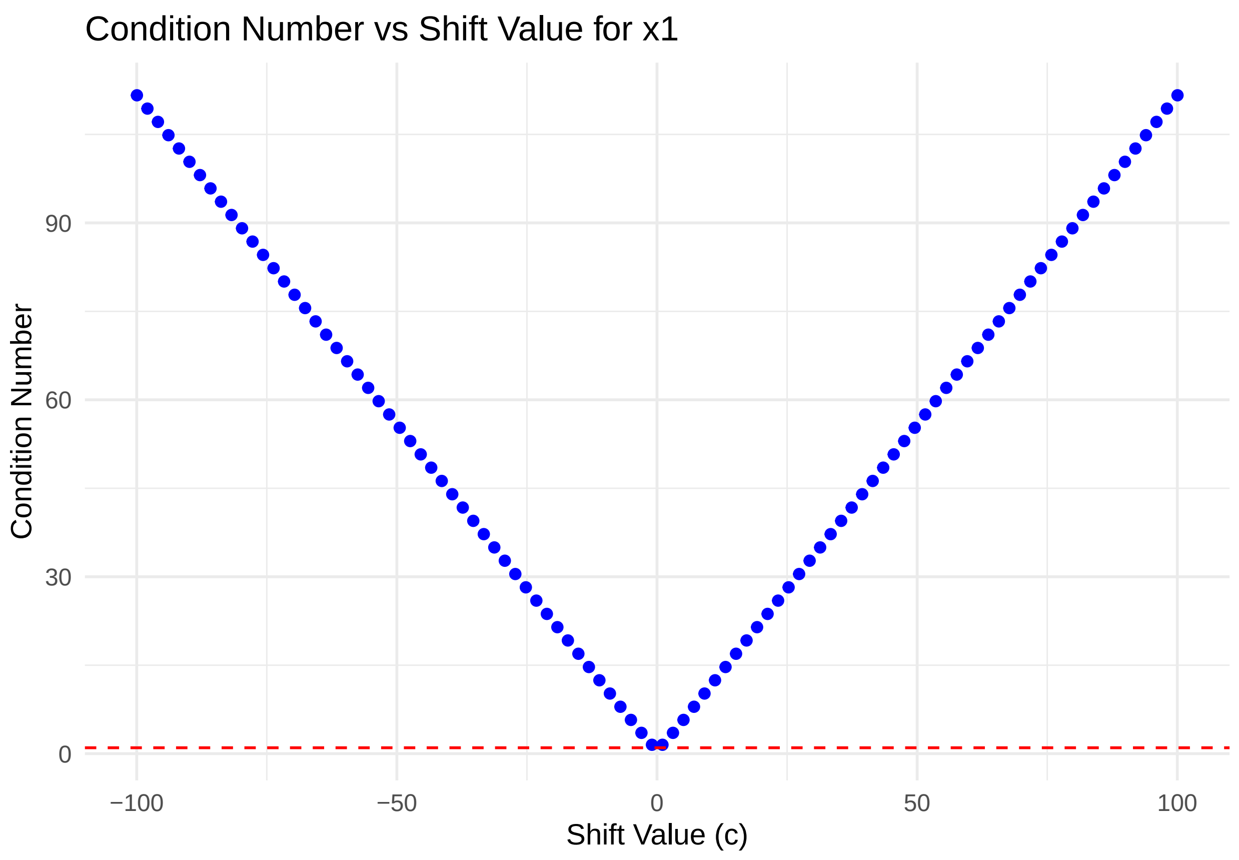
```r
c_values_x1 <- seq(mean(x1) - 100, mean(x1) + 100, length.out = 100)
c_values_x2 <- seq(mean(x2) - 10000, mean(x2) + 10000, length.out = 100)
```

```r
results_x1 <- getResults_cValue(x1, c_values_x1)
results_x2 <- getResults_cValue(x2, c_values_x2)
```

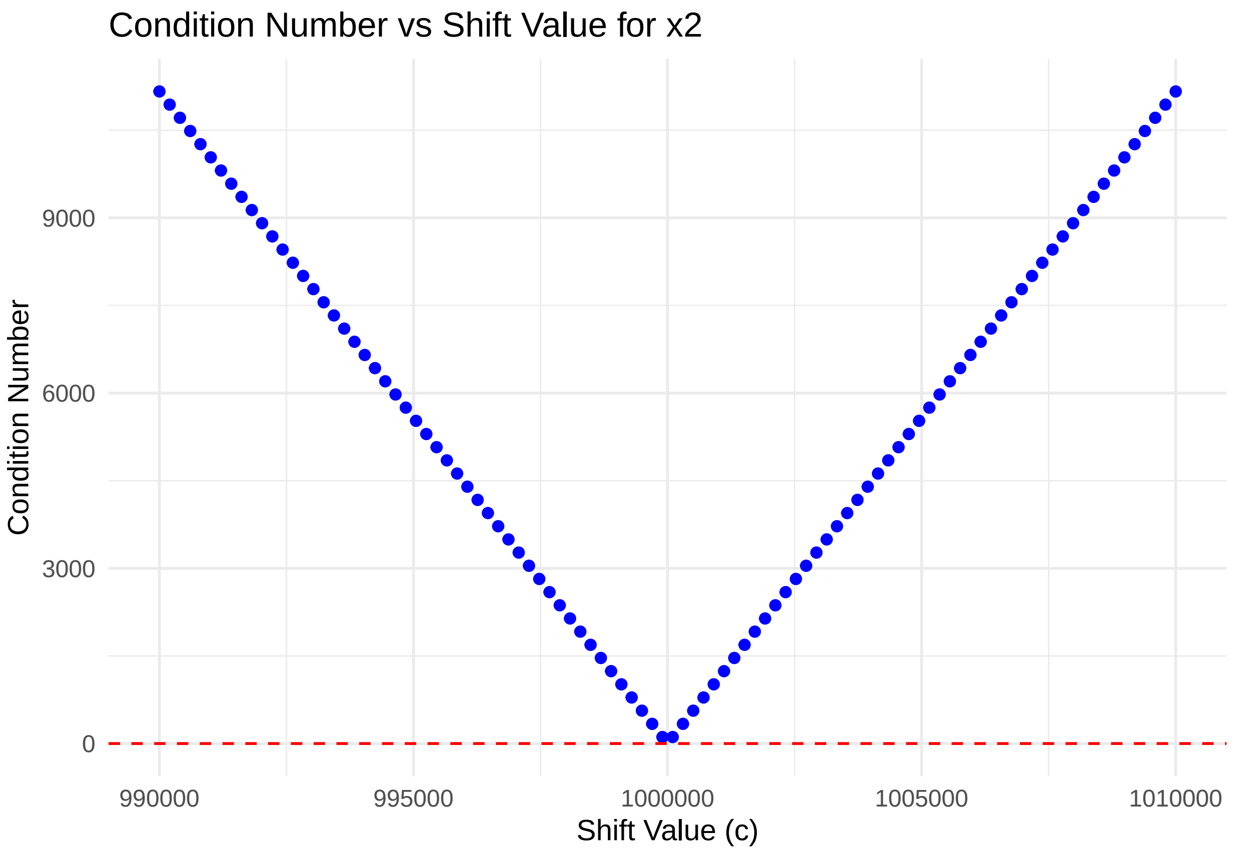
```r
library(ggplot2)

plotter_c <- function(results, dataset_name) {
  ggplot(results, aes(x = Shift_Value, y = Condition_Number)) +
    geom_point(color = "blue") +
    ggtitle(paste("Condition Number vs Shift Value for", dataset_name)) +
    xlab("Shift Value (c)") +
    ylab("Condition Number") +
    theme_minimal() +
    geom_hline(yintercept = 1, linetype = "dashed", color = "red")
}
```

```r
plotter_c(results_x1,"x1")
```



```r
plotter_c(results_x2,"x2")
```



The plot effectively illustrates why shifting by the mean is optimal—it minimizes the condition number, thereby ensuring that the variance calculation is most stable and invariant to input perturbations.

Explaination

When investigating the scale invariance property of variance calculation, we found that shifting by the mean gives the best numerical stability for both datasets (x1 with mean   0, and x2 with mean = 1,000,000).

For x1, when the shift value (cc) was equal to the mean (0), the condition number remained close to 1, indicating optimal stability. Increasing the shift away from the mean led to a higher condition number, implying less stability.

Similarly, for x2, the lowest condition number was achieved when shifting by the mean of the dataset (1,000,000). Using other shift values caused the condition number to increase, again suggesting reduced stability.

Overall, the results confirm that shifting by the mean minimizes the condition number and yields the most stable outcome, making the variance calculation robust to changes in the input. This is why shifting by the mean performs best in terms of numerical stability.

Task 4

Finally, we will:

- Compare condition numbers for the two simulated datasets
- Compare condition numbers for a third dataset where the requirement is not fulfilled

For the third dataset, 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 dataset with a large mean (1,000,000) and a very small standard deviation (0.0001) to create an extremely high condition number. This illustrates that tiny changes in input can lead to massive changes in output.

First we create a function for the not shifted algorithm.

```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))
}
```

We are generating three different datasets with varying properties to analyze the condition numbers associated with each:

We calculate the condition number for each dataset using the function get_condition_number(), which provides an indication of numerical stability. The results are then compiled into a table for easy comparison:

- A dataset with mean = 0.
- A dataset with a large mean = 1,000,000.
- A dataset with a large mean = 1,000,000 and a very small standard deviation = 0.0001

This analysis helps us understand the impact of dataset properties, such as mean and variance, on the numerical stability of variance calculations.

```r
  set.seed(12433732)
datasets <- list(
  rnorm(100),
  rnorm(100, mean = 1000000),
  rnorm(100, mean = 1000000, sd = 0.0001)
)

results <- data.frame(matrix(ncol = 2, nrow = 0))
colnames(results) <- c("Data_Set", "Condition_Nr")


for (i in 1:length(datasets)) {
  x <- datasets[[i]]
  cond_num <- get_condition_number(x)

  results <- rbind(
    results,
    data.frame(
      "Data_Set" = i,
      "Condition_Nr" = cond_num
    )
  )
}
```

At the end we get th result and print it out

```r
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"

library(knitr)
kable(results, caption = "Condition Numbers with Different Algorithms and Datasets")
```

Table 6: Condition Numbers with Different Algorithms and Datasets

Data_Set	Condition_Nr
mean=0	1.000904e+00
mean=1000000	9.375879e+05
mean=1000000, sd=0.0001	1.039383e+10

Conclusion

The **condition number** (κ) measures the **sensitivity** of variance calculations to changes in the input data. It is given by:

$$\kappa = \sqrt{1 + \frac{n \cdot \bar{x}^2}{S}}$$

Where: - n: Number of data points. - $\bar{x}$: Mean of the dataset. - S: Sum of squared deviations from the mean.

A **low condition number** (close to **1**) implies **stability** in variance calculations, meaning that small input changes have minimal impact on the result. When the **mean** ($\bar{x}$) is **small** or comparable to the **variance** (S), the term $\frac{n \cdot \bar{x}^2}{S}$ remains low, keeping κ close to **1**.

However, as the **mean** becomes significantly **larger** than the **variance**, $\frac{n \cdot \bar{x}^2}{S}$ increases, leading to a **higher condition number** and reduced **numerical stability**. This means that datasets with a **large mean** and **small variance** are **ill-conditioned**, making the variance calculation highly sensitive to even slight perturbations, resulting in potentially large errors.