Lab 1

Problem 1

1. Check the results of the snippet. Comment on whats going on

First Snippet:

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

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

If we examine the problem by each variable separately we see that the computed values are as follows:

```
x1 <-  1/3
x2 <- 1/4
x3 <- 1/12

x1
```

```
## [1] 0.33333333333333331
```

```
x2
```

```
## [1] 0.25
```

```
x3
```

```
## [1] 0.083333333333333329
```

```
x1-x2
```

```
## [1] 0.083333333333333315
```

We can see that since x1 has recurring decimals, it gets stored with a rounding error. Whereas, x2 is a number with a denominator that is a power of 2 and hence it doesn't have a mantissa and hence no rounding error. In the case for x3 i.e 1/12, it is the same as x1 where it has a recurring decimal case and get stored with a rounding error. When we perform the subtraction operation of x1-x2, the dedicated higher precision register where this operation takes place first lines up the decimals and performs the operations and then only after the operation is complete, it performs the rounding which leads to different values getting carried forward due to rounding. Hence we see the difference in the results of x1-x2 and 1/12.

Second Snippet:

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

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

We don't run into the same problem here as the first snippet because both x1 and x2 can be expressed as powers of 2 and hence dont encounter any rounding error when being stored in the machine. Therefore, here the result is accurate.

2. If there are any problems suggest solutions

One possible way to correct the error in the first snippet can be to use the equivalent algebraic expression (x - y)(x + y) = (x^2 - y^2) Sometimes this kind of an algebraic trick can help to reduce the loss of significance. We can see the results of this change below

```
x1 <-  1/3
x2 <- 1/4
x_res <- (x1^2 - x2^2)/(x1+x2)
if (x_res == 1/12) {
  print("Subtraction is correct")
  } else {
    print("Subtraction is wrong")
  }
```

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

Problem 2

1. Write your own R function to calculate the derivative of f(x) = x in this way with $\epsilon = 10^{-15}$

2. Evaluate your derivative function at x = 1 and x = 100000

3. What values did you obtain? What are the true values? Explain the reasons behind the discovered differences

Solution:

```
derivative_function <- function(x = 2, epsilon = 10**-15) {
  derivative_1 <- ((x + epsilon) - x) / epsilon
  return(derivative_1)
}

derivative_function(x=1)
```

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

```
derivative_function(x=1000)
```

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

We get the following values 1.110223 and 0. The correct answer in both of the cases should be 1. The numerator ((x+epsilon) - x) decides the outcome in this case. 1 + epsilon - 1 gives a value close to 1 with an error whilst 100000 + epsilon - 100000 gives 0. epsilon is thus disregarded when being summed or subtracted by relatively large values with 18 decimal places difference. This occurs because the computer want to add the mantissas of both numbers and it does it in a way where it will increase the exponent of the smallest number until it matches the large value. This is where the approximation happens and in the worse cases like when x is 100000, the small value will become 0. We know that the derivative of a of X with regards to X should == 1.

Problem 3

1. Write your own R function, myvar, to estimate the variance in this way

The function for calculating the variance from the given expression is as follows:

```r
myvar <- function(x) {
  n <- length(x)
  variance <- 1/(n-1)*(sum(x^2) - 1/n*(sum(x)^2))
  return(variance)
}

vec_x <- rnorm(n = 10000, mean = 5, sd = 1)

myvar(vec_x)
```

```
## [1] 0.98306450134734091
```

```r
var(vec_x)
```

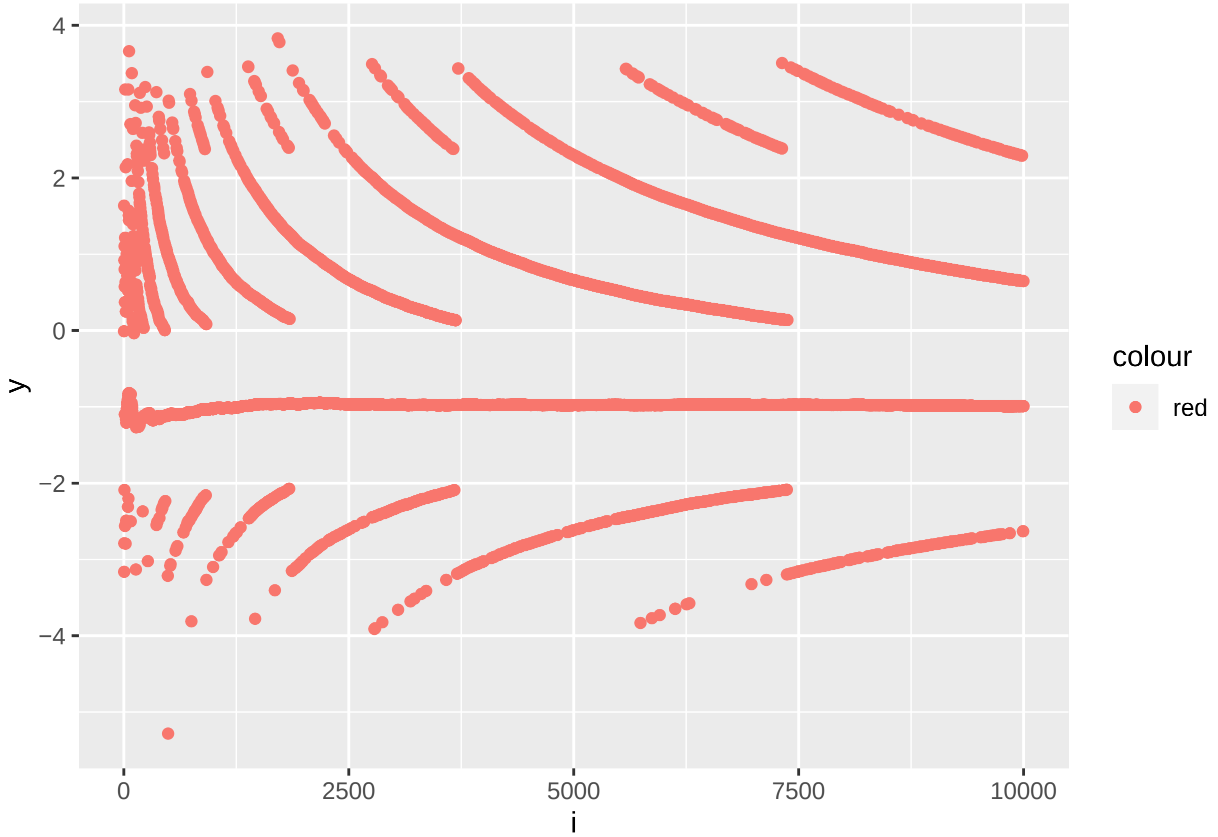
```
## [1] 0.98306450134734258
```

2. Generate a vector $x = (x_1, \ldots, x_{10000})$ with 10000 random numbers with mean 10^8 and variance 1

3. For each subset $X_i = \{x_1, \ldots\ , x_i\}$, $i = 1, \ldots, 10000$ compute the difference Y_i = myvar(X_i)-var(X_i), where var(X_i) is the standard variance estimation function in R. Plot the dependence Y_i on i. Draw conclusions from this plot. How well does your function work? Can you explain the behaviour?

```r
myvar <- function(x) {
  n <- length(x)
  variance <- 1/(n-1)*(sum(x^2) - 1/n*(sum(x)^2))
  return(variance)
}
x <- rnorm(n = 10000, mean = 10^8, sd = 1)
y<- c()
for (i in 1:length(x)) {
  y[i] = myvar(x[1:i]) - var(x[1:i])
}
i = 1:10000
ggplot() +
  geom_point(aes(x = i, y = y, color = "red"))
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



When we pass very large numbers to our function myvar(), there is a chance for overflow because of the square terms in the expression. As a result, some numbers maybe stored in the memory with wrong sign bit resulting in a very large variability in the calculated y term.

4. How can you better implement a variance estimator? Find and implement a formula that will give the same results as var()?

The expression that we utilize in statistics for calculating sample variance is as follows:

s = (1/(n-1)) * sum(x - xmean) ^2

```
myvar_corrected <- function(x) {
  n = length(x)
  variance = 1/(n-1) * sum((x - mean(x))^2)
  return(variance)
}
vec_x <- rnorm(n = 10000, mean = 5, sd = 1)
myvar_corrected(vec_x)
```

```
## [1] 0.99057031207955459
```

```
var(vec_x)
```

```
## [1] 0.99057031207955448
```

```
myvar(vec_x)
```

```
## [1] 0.99057031207955204
```

As it is clearly seen, the precision of the new function is closer to the actual value of the variance. We were able to achieve this because we were able to compute differences before we squared any potential rounding errors.

Problem 4

Consider the three below R expressions for computing the binomial coefficient. They all use the prod() function, which computes the product of all the elements of the vector passed to it.

A) prod (1:n) / (prod (1:k) * prod (1:(n-k)))

B) prod((k+1):n) / prod (1:(n-k))

C) prod (((k+1):n) / (1:(n-k)))

1. Even if overflow and underflow would not occur these expressions will not work correctly for all values of n and k. Explain what is the problem in A, B and C respectively

The 3 expressions are mathematically manipulated expression for the same result but for computer calculations they are treated differently as the order of calculation is different in each calculation.

Consider the following case where n = 1 and k = 0, where the expected result should be 1,

```
n = 1
k = 0

A <- prod(1:n) / (prod(1:k) * prod(1:(n-k)))
B <- prod((k+1):n) / prod(1:(n-k))
C <- prod(((k+1):n) / (1:(n-k)))

A
```

```
## [1] Inf
```

```
B
```

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

```
C
```

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

As you can see, the first expression returns an incorrect result i.e. Inf. The reason is that because we are using the prod() and not factorial(), we get a element wise multiplication of the vector which includes 0. hence the denominator goes to 0 and the final result is incorrect as Inf. The same problem is avoided in B & C because the 0 gets eliminated in the vector before prod() is called.

Now similarly consider the following case where n = 0 and k = 0 where the expected result should again be 1,

```
n = 0
k = 0

A <- prod(1:n) / (prod(1:k) * prod(1:(n-k)))
B <- prod((k+1):n) / prod(1:(n-k))
C <- prod(((k+1):n) / (1:(n-k)))

A
```

```
## [1] NaN
```

```
B
```

```
## [1] NaN
```

```
C
```

```
## [1] NaN
```

Here we run into the trouble of getting a result which NaN for all 3 expressions. This is because in all 3 expressions a vector containing 0 is passed to the prod(). Hence in all 3 cases, the resulting expression is 0/0 resulting in NaN as the result for all 3.

2. In mathematical formulae one should suspect overflow to occur when parameters, here n and k, are large. Experiment numerically with the code of A, B and C, for different values of n and k to see whether overflow occurs. Graphically present the results of your experiments.

3. Which of the three expressions have the overflow problem? Explain why.

We'll tackle 2 & 3 together as follows:

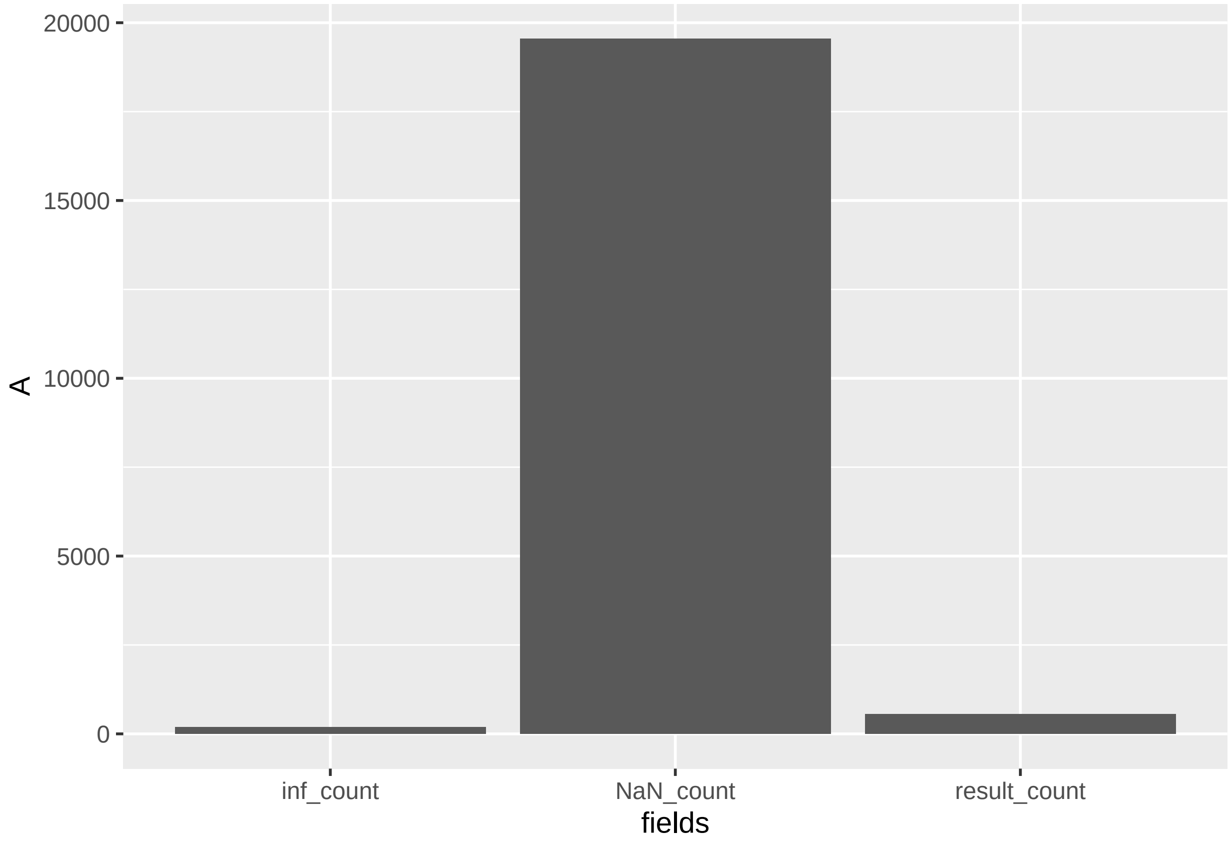
In order to simulate results with large values of k & n, we created 3 matrices for A, B & C computations for values upto 1000 with a step of 5. Thus giving us a large set of values to observe (200x200 matrix)

```
sample_n <- seq(0, 1000, by = 5)
sample_k <- seq(0, 1000, by = 5)

mat_A <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_A[i,j] = prod(1:sample_n[i]) / (prod(1:sample_k[j]) * prod(1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_A) <- sample_k
rownames(mat_A) <- sample_n

mat_B <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_B[i,j] = prod((sample_k[j] + 1):sample_n[i]) / (prod(1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_B) <- sample_k
rownames(mat_B) <- sample_n

mat_C <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_C[i,j] = prod(((sample_k[j] + 1):sample_n[i]) / (1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_C) <- sample_k
rownames(mat_C) <- sample_n
```

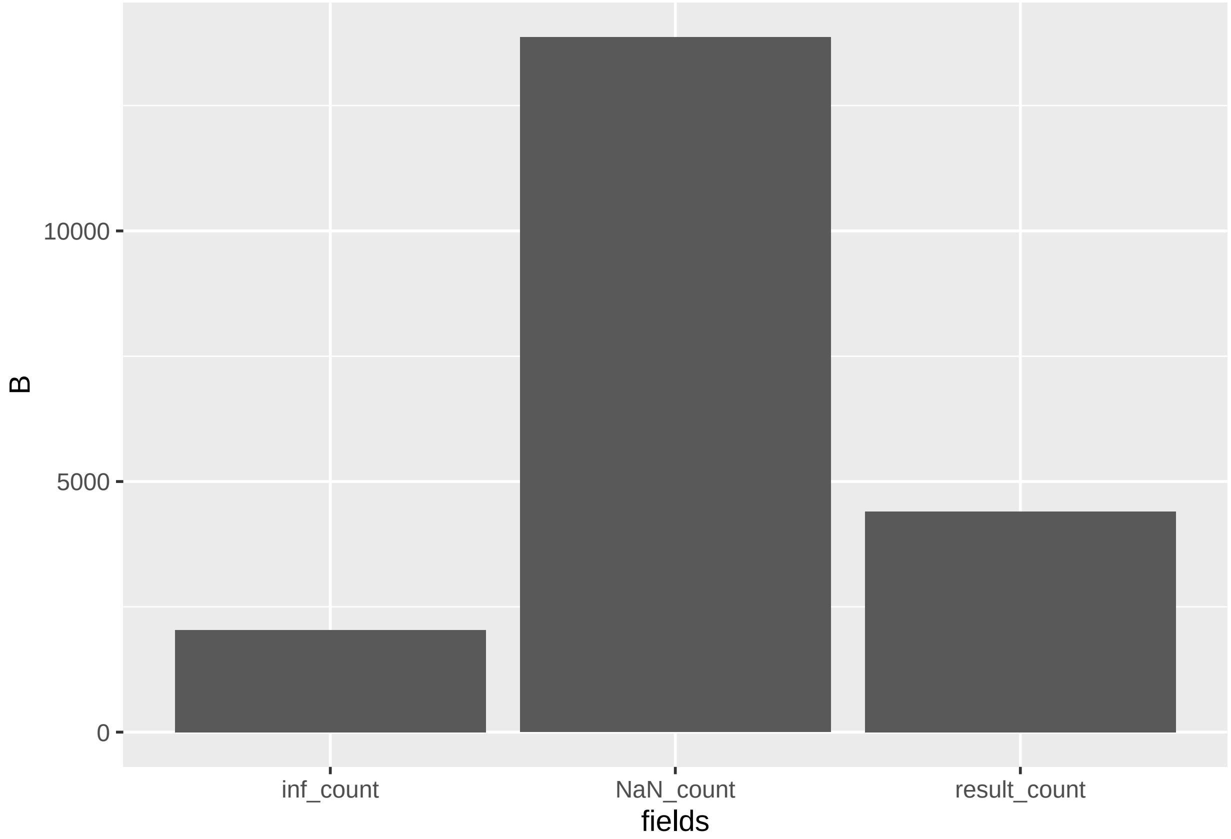
To represent this better, we use the following results from our 3 computations:

```
sample_n <- seq(0, 1000, by = 5)
sample_k <- seq(0, 1000, by = 5)

mat_A <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_A[i,j] = prod(1:sample_n[i]) / (prod(1:sample_k[j]) * prod(1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_A) <- sample_k
rownames(mat_A) <- sample_n

mat_B <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_B[i,j] = prod((sample_k[j] + 1):sample_n[i]) / (prod(1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_B) <- sample_k
rownames(mat_B) <- sample_n

mat_C <- matrix(NA, nrow = length(sample_n), ncol = length(sample_k))
for (i in 1:length(sample_n)) {
  for (j in 1:length(sample_k)) {
    if (sample_k[j] <= sample_n[i]) {
      mat_C[i,j] = prod(((sample_k[j] + 1):sample_n[i]) / (1:(sample_n[i] - sample_k[j])))
    }
  }
}
colnames(mat_C) <- sample_k
rownames(mat_C) <- sample_n

mat_elements = 201*201

na_count_A <- sum(is.na(mat_A))
inf_count_A <- sum(is.infinite(mat_A))
nan_count_A <- sum(is.nan(mat_A))
result_count_A <- mat_elements - na_count_A - inf_count_A
# n_max_A <- 170
# k_max_A <- 165

na_count_B <- sum(is.na(mat_B))
inf_count_B <- sum(is.infinite(mat_B))
```

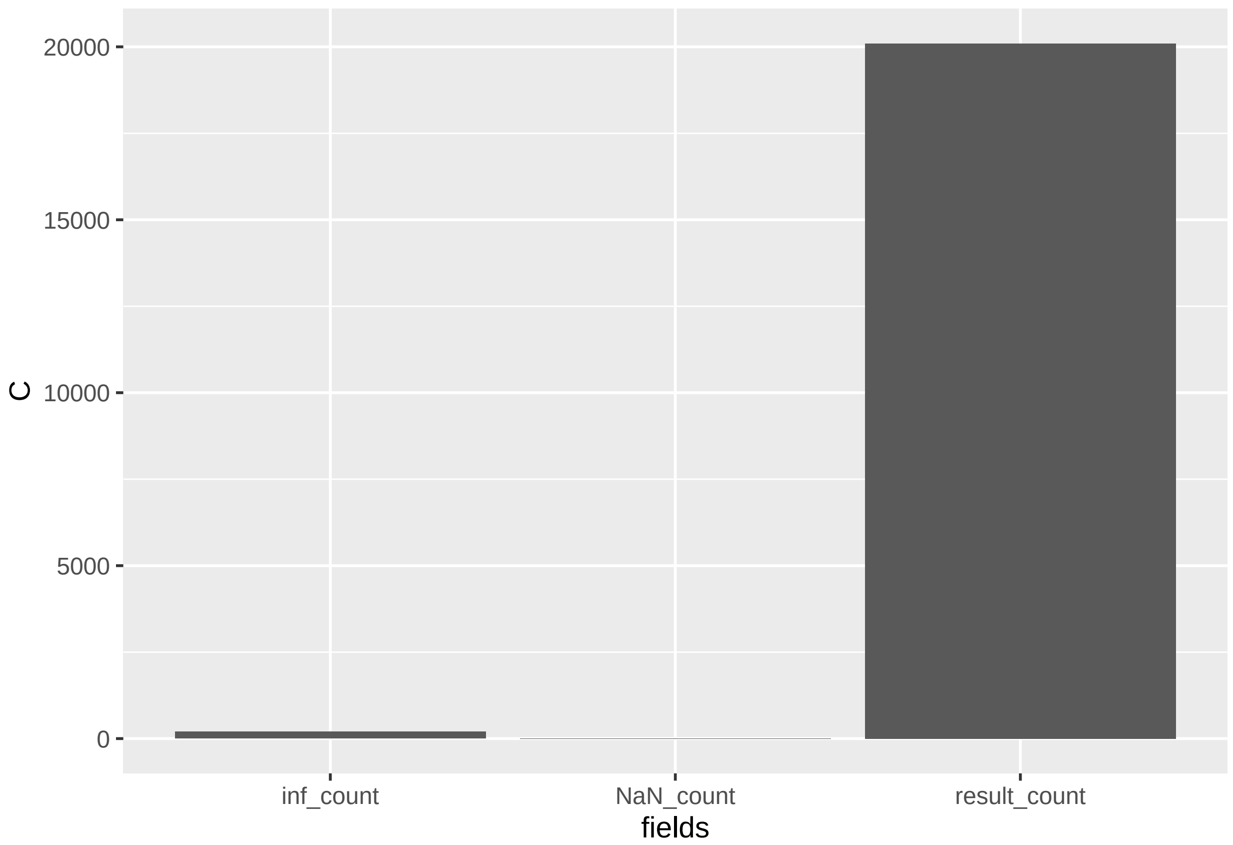
```
nan_count_B <- sum(is.nan(mat_B))
result_count_B <- mat_elements - na_count_B - inf_count_B
# n_max_B <- 1000
# k_max_B <- 995

na_count_C <- sum(is.na(mat_C))
inf_count_C <- sum(is.infinite(mat_C))
nan_count_C <- sum(is.nan(mat_C))
result_count_C <- mat_elements - na_count_C - inf_count_C
# n_max_C <- 1000
# k_max_C <- 995

vis_df <- data.frame(fields = c("result_count", "inf_count", "NaN_count"),
                     A = c(result_count_A, inf_count_A, nan_count_A),
                     B = c(result_count_B, inf_count_B, nan_count_B),
                     C = c(result_count_C, inf_count_C, nan_count_C))

vis_df_1 <- tidyr::gather(vis_df, key = "fields", value =  "formula", 2:4)


p1 <- ggplot(vis_df, aes(fields, A)) + geom_bar(stat='identity')
p2 <- ggplot(vis_df, aes(fields, B)) + geom_bar(stat='identity')
p3 <- ggplot(vis_df, aes(fields, C)) + geom_bar(stat='identity')
p1
```



```
p2
```



```
p3
```



As we can see from the comparison of the results from the three computation expressions, A is the least flexible with a max limit of n upto 170. This is because beyond this, the numerator of the expression goes into overflow.

Next, we see the results of B and while B increases the flexibility of A and can compute results upto n = 1000, its is also only able to do so, as long as the numerator doesn't overflow. Therefore, it allows for every value of n, computations for all those values of k where prod((k+1):1) doesn't overflow. For the values chosen, it was observed that the max tolerance (n-k) ~150. but we get many more values which go to inf as compared to A.

Moving onto C, It is easily the most flexible of the 3, implying that the it gives a result for every combination of n & k. The reason for this is quite obvious that the vector being passed to prod() is never leading to an overflow as the division results in the cancellations that eliminate large numbers that can lead to overflow condition.

As a result we can conclude that computations due to A & B expressions have overflow issues whereas C doesn't. Between A & B, it is actually B where more cases of overflow are occurring.It is interesting to note though, that C does face the additional issues related to rounding. because, when the values of n & k become moderately large, there is a propagation of rounding errors and hence we are given an approximated value that is different from the expected value of the expression.