# ComputationalStatistics-Lab01-Group 14

Akshath Srinivas, Samira Goudarzi

2022-11-14

# Assignment 1

## Question 1

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

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

### subquestion 1

1/3 is a recurring division. We get infinite number of digits on division. Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation . Therefore floating point approximation is rounded as finite representation which will result to the limited precision of mantissa and stored in computer memory. 1/4 results finite digits, computer memory can store it without loosing precision in mantissa. when we do floating point computation (1/3-1/4) we get a output which is a result of floating point rounding error of 1/3, but 1/12 is stored in computer memory in a different rounding approximation which is also shown below. This the reason we get subtraction is wrong in first case.

In the second code snippet both 1 and 1/2 are absolute values and has finite digits, computer memory can store it without loosing precision in mantissa.(Gentle 2009).

```
##  x1 = 1/3 is equal to 0.333333333333333148296
```

```
##  x2 = 1/4 is equal to 0.25
```

```
##  x1-x2 is equal to 0.0833333333333331482962
```

```
##  x = 1/12 is equal to 0.0833333333333333287074
```

### subquestion 2

We can improve the first case by using all.equal() function as used below. This function tests nearly equal values and outputs TRUE or FALSE.

we can use very small scale number as threshold to compare the values, then we can assume a-b equals c.

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

## Question 2

### subquestion 1

Our R function to calculate the derivative of f(x) = x when epsilon is 10**(-15):( please see the code in appendix)

### subquestion 3

1 is the true value for both the cases In the first case as epsilon is very small value by adding 1 yields a rounding approximation value as a result of limited precision of mantissa. 1+10^-15 yields 1.00000000000000011102230, when we subtract 1 from this value we get a value in numerator which little more than epsilon value in denominator i.e numerator greater than denominator which yields value little more than 1 on differentiating.

For second case, the magnitude of x is very large even when adding epsilon 10^-15(100000 + 10^-15), computer memory stores this value by loosing part of the mantissa and remains almost same as x. Hence the numerator becomes zero after subtracting x which will yield 0 as output.

```
## For x = 1, the derivative is : 1.110223024625156540424
```

```
## For x = 100000, the derivative is : 0
```

## Question 3

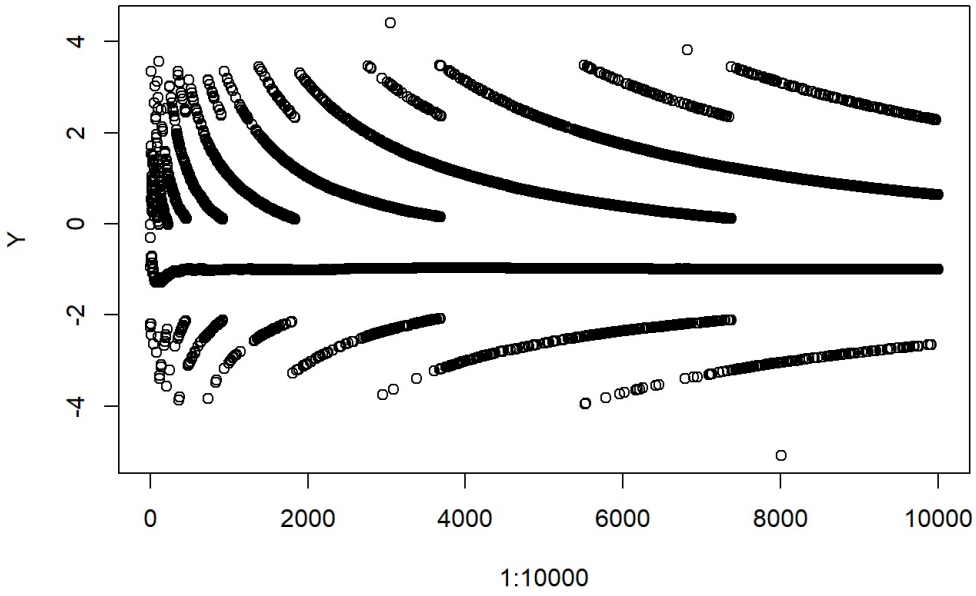
```
## My variance function value : 1.638563856385638617397
```

```
## Inbuilt var function value : 0.9996998992308180342903
```

## subquestion 3

We can see from the below plot with X axis as number of samples and Y axis as the difference between my_var function and inbuilt var function. We can also observe that one tends to zero, one tends to -2 and one is concentrated around -1.Our function does not work well, if it has worked well all of the result will be equal to zero.

This behavior is because that all the sample values are approximated and we are doing arithmetic operations like sum(x)^2 yields large values causing integer overflow as the values are huge. So, the answer(variance value) will loose accuracy.

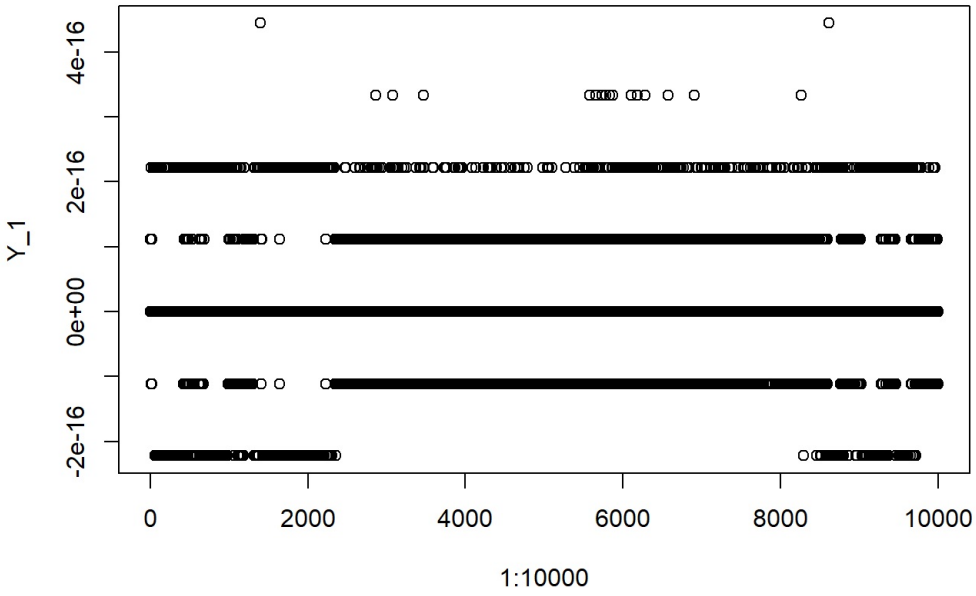



## subquestion 4

Improving the function by calculating the mean of the vector and inputting this value to variance formula.(please refer to the formula in appendix code)

```
## My variance function improved value : 0.9996998992308180342903
```

```
## Inbuilt var function value : 0.9996998992308180342903
```



# Question 4

```
##    n  k        a                         b                          c   actual
## 1 25  6   177100   177100.00000000000000000   177100.00000000002910383   177100
## 2 26 22    14950    14950.00000000000000000    14950.00000000000181899    14950
## 3 27 13 20058300 20058300.00000000000000000 20058300.00000000000000000 20058300
## 4 28  5    98280    98279.99999999998544808    98280.00000000000000000    98280
## 5 29  1       29       29.00000000000000000       29.00000000000000000       29
## 6 30 11 54627300 54627300.00000000000000000 54627300.00000002980232239 54627300
```

## subquestion 1

**Problem with A:** when k==0,k==n, it will compute prod(1:0) which will lead to inf as result

**Problem with B:** when k==n, it will compute prod(1:0) in denominator which will lead to inf as result

**Problem with C:** when k==n, it will compute prod(1:0) in denominator which will lead to inf as result

## subquestion 2

For A overflow happens when value of n is equal or greater than 171.

For B overflow happens when value of n is equal or greater than 171,k=1.our observation is when value of n is very high and value of k is less we can observe overflow

For C overflow happens when value of n is equal 1170, k is equal to result of dividing n by 3 and rounding off.

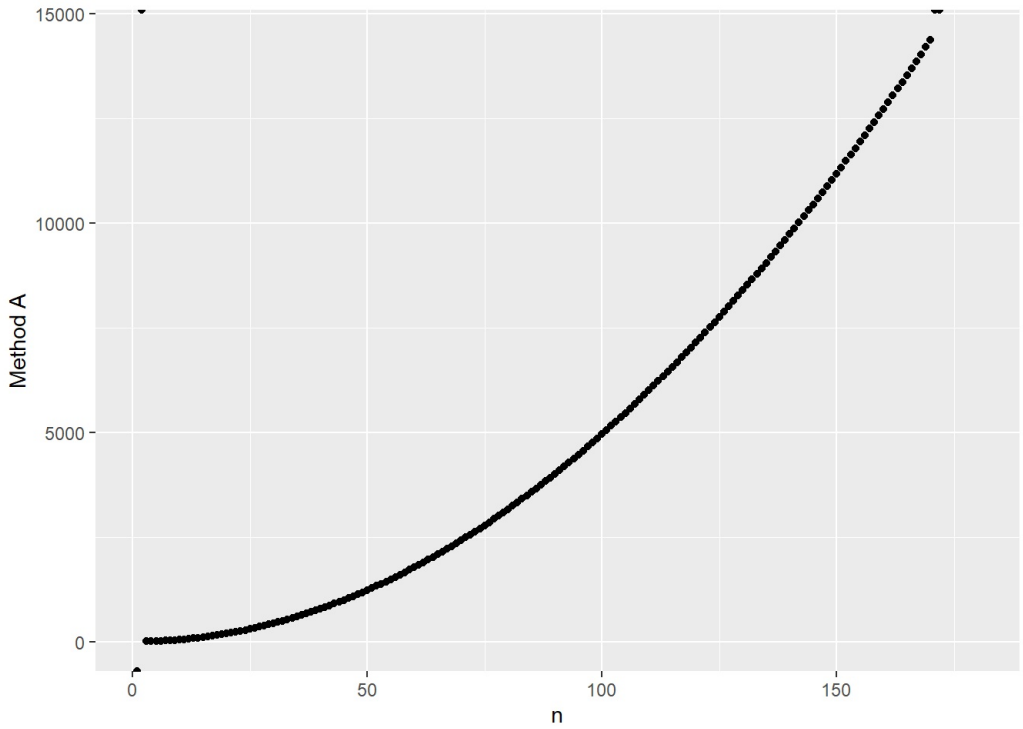
```
##  For (n = 171 , k=2) ,a= Inf
```

```
##  For (n = 171 , k=1) ,b= Inf
```

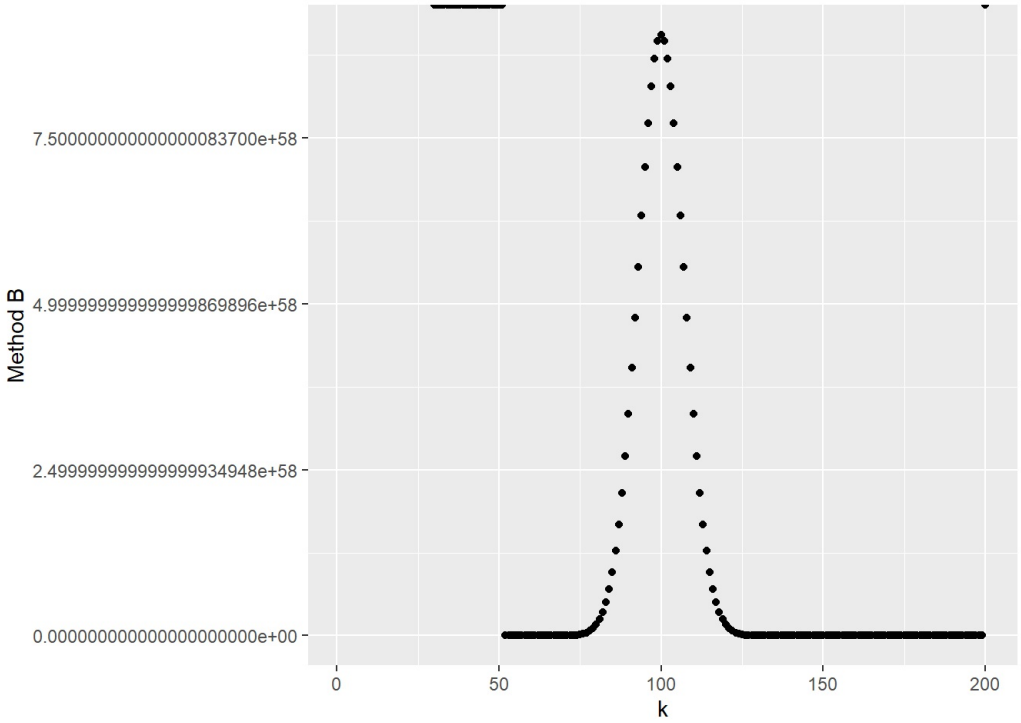
```
##  For (n = 1170 , k=390) ,c= Inf
```

## subquestion 3

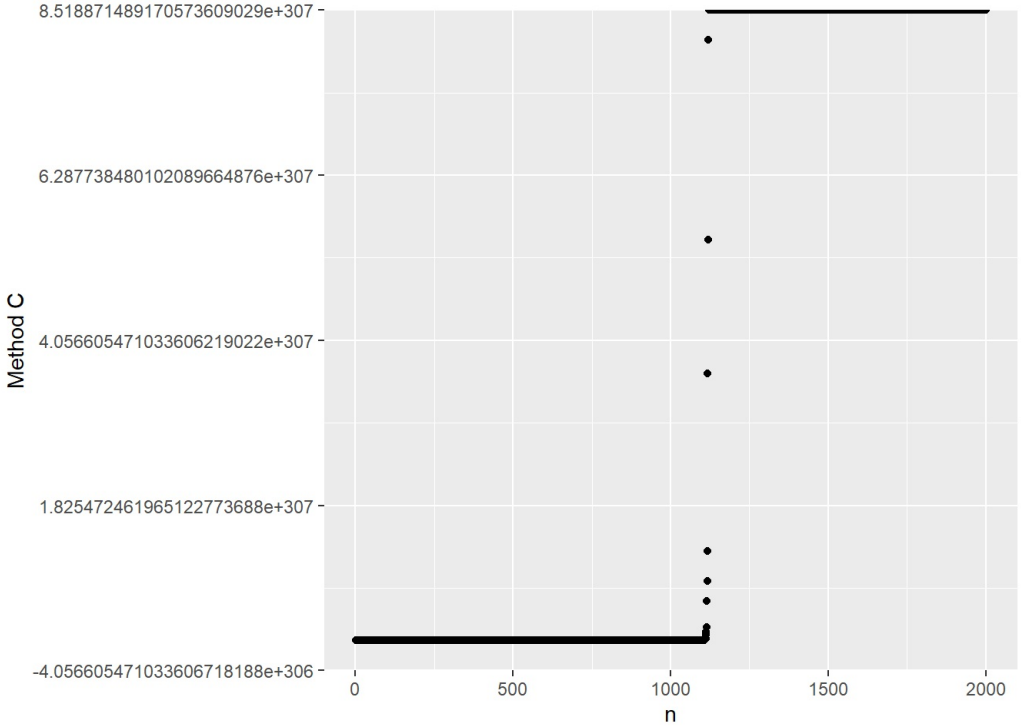
**Method A** : serious overflow problem happens when n grows as it will compute prod(1:n) first, no matter the magnitude of K, it will overflow when n is increased. From below plot we can see as k is fixed as 2 and we increase n value overflow occurs.




**Method B** is better than A, but it will also face overflow problem as we increase the n. Also when k is reduced and keeping high value of n fixed, the vector length inside the prod( ( k+1) : n ) increases which will lead to overflow. From below plot we can see when n is fixed as 200 and k is increased,we can see overflow when k values are less below 52.




**Method c** is better than both method a and method b, as the vector division is happening inside prod() function, which will lower the magnitude and risk of overflow. But, it will eventually overflow when n is large and k is 1/3rd of n value.




# Appendix

```
knitr::opts_chunk$set(echo = TRUE)
library(dplyr)
library(tidyr)
library(ggplot2)


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


x1<-1
x2<-1/2
if ( x1-x2==1/2 ) {
```

```
  print ( "Subtraction is correct")
} else {
  print ( "Subtraction is wrong" )
}

options(digits=22)
x1<-1/3

cat(" x1 = 1/3 is equal to", x1,"\n")
x2<-1/4
cat(" x2 = 1/4 is equal to", x2,"\n")
diff<-x1-x2
cat(" x1-x2 is equal to", diff,"\n")
val<-1/12
cat(" x = 1/12 is equal to", val,"\n")


x1<- 1/3
x2<- 1/4
if (all.equal(x1-x2,1/12)) {
  print ( "Subtraction is correct " )
} else {
  print ( "Subtraction is wrong" )
}
dev <- function(x){
  d = ((x+10^(-15))-x)/(10^(-15))
  return(d)
}
#Question 2

dev <- function(x){
  d = ((x+10^(-15))-x)/(10^(-15))
  return(d)
}
cat("For x = 1, the derivative is :", dev(1),"\n")

cat("For x = 100000, the derivative is :", dev(100000),"\n")


#Question 3, subquestion 1
my_var<-function(x){
  var<-1/(length(x)-1)*(sum(x^2)-1/length(x)*(sum(x))^2)
  return(var)
}
set.seed(12345)
x<-rnorm(10000,mean = 10^8,sd=1)
cat("My variance function value :", my_var(x),"\n")
cat("Inbuilt var function value :", var(x),"\n")

#Question 4, subquestion 3
Y<-c()
for(i in 1:length(x)){
  xi<-x[1:i]
  yi<-my_var(xi)-var(xi)
  Y<-c(Y,yi)
}
plot(1:10000,Y)


#Question 3, subquestion 4
my_var_imp<-function(x){
  mean<-sum(x)/length(x)
  var<-sum((x-mean)^2)/(length(x)-1)
  return(var)
}
cat("My variance function improved value :", my_var_imp(x),"\n")
cat("Inbuilt var function value :", var(x),"\n")
Y_1<-c()
for(i in 1:length(x)){
  xi_1<-x[1:i]
  yi_1<-my_var_imp(xi_1)-var(xi_1)
  Y_1<-c(Y_1,yi_1)
}
plot(1:10000,Y_1)


#Question 4
df<-as.data.frame(matrix(nrow=0,ncol=6))
colnames(df)<-c('n','k','a','b','c','actual')
```

```
for(n in 25:100 ){
  k<-sample(1:n-1,1)
  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 ) ) )
  ac<-choose(n,k)
  df<-rbind(df,data.frame(n=n,k=k,a=a,b=b,c=c,actual=ac))
}
# plot<-df %>% pivot_longer(c('a','b','c','actual'),names_to='prod_type',
#                           values_to='prod')
# ggplot(df,aes(x=n,y=prod))+
#   geom_point(aes(color=prod_type))
head(df)

n<-171
k<-2
a<-prod ( 1 : n ) / (prod ( 1 : k ) * prod ( 1 : ( n-k ) ) )
cat(" For (n = 171 , k=2) ,a=", a,"\n")

n<-171
k<-1
b<-prod( ( k+1) : n ) / prod ( 1 : ( n-k ) )
cat(" For (n = 171 , k=1) ,b=", b,"\n")

n<-1170
k<-390
c<-prod ( ( ( k+1) : n ) / ( 1 : ( n-k ) ) )
cat(" For (n = 1170 , k=390) ,c=", c,"\n")

#keeping k fixed for method A
k<-2
n<-c(1:180)
a_vec<-c()
for(i in 1:length(n)){
  a<-prod ( 1 : n[i] ) / (prod ( 1 : k ) * prod ( 1 : ( n[i]-k ) ) )
  a_vec<-c(a_vec,a)
}
library(ggplot2)
ggplot()+geom_point(aes(x=n,y=a_vec),na.rm = TRUE)+ylab('Method A')

#keeping high value n fixed for method b
n<-200
k<-c(1:200)
b_vec<-c()
for(i in 1:length(k)){
  b<-prod( ( k[i]+1) : n ) / prod ( 1 : ( n-k[i] ) )
  b_vec<-c(b_vec,b)
}
library(ggplot2)
ggplot()+geom_point(aes(x=k,y=b_vec),na.rm = TRUE)+ylab('Method B')

#changing n and k for method C
n<-1:2000
c_vec<-c()
for(i in 1:length(n)){
  c<-prod ( ( ( round(n[i]/3)+1) : n[i] ) / ( 1 : ( n[i]-round(n[i]/3) ) ) )
  c_vec<-c(c_vec,c)
}
library(ggplot2)
ggplot()+geom_point(aes(x=n,y=c_vec),na.rm = TRUE)+ylab('Method C')
```

# Reference

Gentle, James E. 2009. “Generation of Random Numbers.” In *Computational Statistics*, 305–31. Springer.