Computational Statistics Lab1

Yuki Washio and Nicolas Taba

02/11/2020

Question 1: Be Careful when comparing

We are asked to evaluate the following code snippet

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

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

And this code snippet

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

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

In the first code snippet, the value of 1/3 is a infinite decimal number. In the floating point number representation, only a value that is similar to it is stored in the 64bits memory capacity. There is a rounding error called underflow that occurs when we operate on this value. In the second code snippet, no rounding error is made.

One possibility to improve the first code is doing the following:

```
x1 <- 1 ; x2 <- 1/2
if (isTRUE(all.equal( x1 - x2, 1/2))){
  print("substraction is correct")
}else{
  print("substraction is wrong")
}
```

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

This works because the **all.equal** method tests for “near equality” between the two quantities instead of exact results up to machine precision.

Question 2: Derivative

Here, we evaluate the derivative of $f(x) = x$ by writing out own R function.

```
epsilon = 10^(-15)

derivative <- function(x){
  stopifnot(is.numeric(x))
  dx <- ((x+epsilon) - x)/(epsilon)
  return(dx)
}
```

We now evaluate for $x = 1$ and $x = 100000$

```
x1 = 1
x2 = 100000
derivative(x1)
```

```
## [1] 1.110223
```

```
derivative(x2)
```

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

The true value of the derivative is 1. For $x = 1$, there is underflow in the calculation as we loose significant digits when dividing. For $x = 100000$, we have the difference in the nominator that is evaluated to be zero and the division by ϵ, which is small and doesn't get stored properly. We have underflow in this case

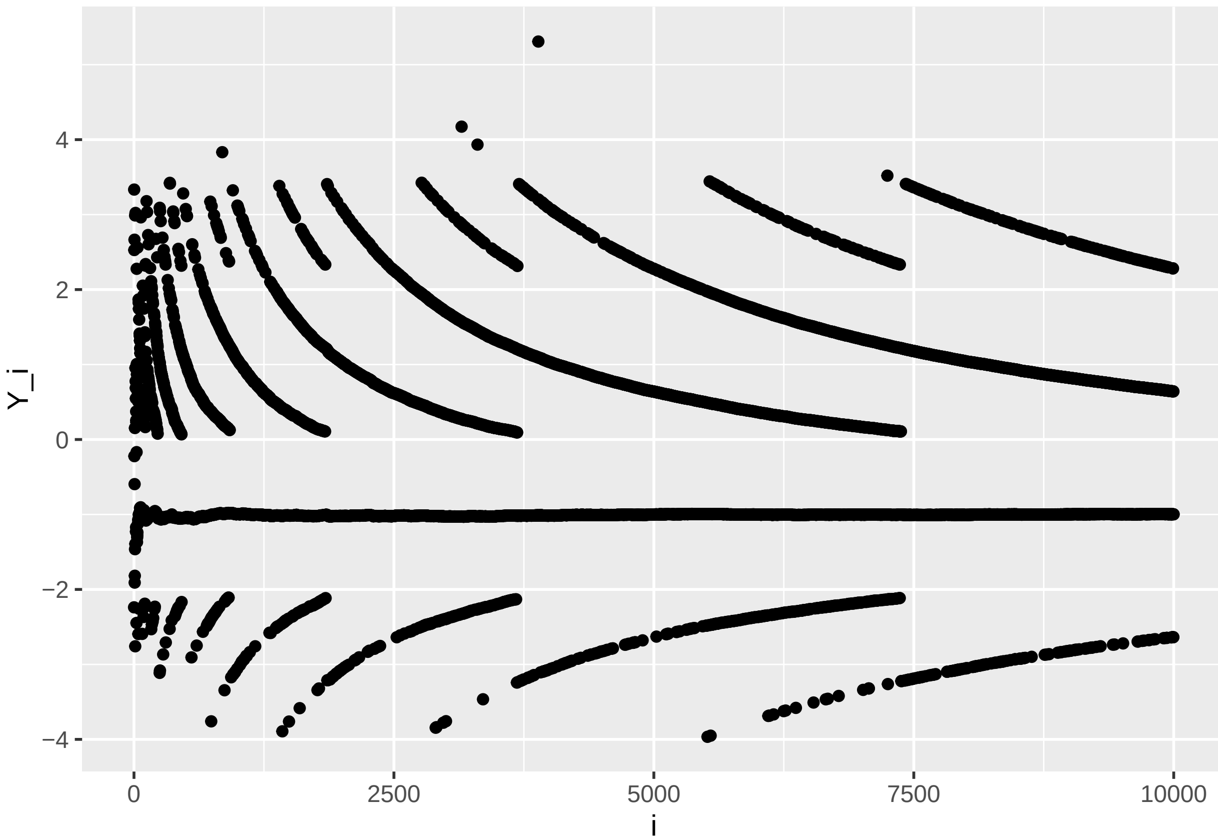
Question 3: Variance

This time, we are asked to write a **myvar** function to calculate the variance based on a vector given by the exercise. We then generate a vector with 10000 random numbers with mean 10^8 and variance 1. We then compare our function to the in-built function **var** in R. We plot the dependence of the difference with respect to the length of the vector.

```
myvar <- function(x){
  stopifnot(is.vector(x),
            is.numeric(x))
  n <- length(x)
  variance <- (1/(n-1))*(sum(x^2) - (1/n)*((sum(x))^2))
  return(variance)
}

vect_x <- rnorm(10000, mean = 10^8, sd = 1)

Y <- c()
for (i in 1:length(vect_x)){
  Y[i] <- myvar(vect_x[1:i]) - var(vect_x[1:i])
}

Y = Y[!is.na(Y)]
```

and now we plot the dependence of Y_i on i.

```
data_Y <- data.frame(values = Y, len = 1:length(Y))
plot_dependence <- ggplot(data = data_Y) +
  geom_point(aes(x = len, y = values)) +
  xlab("i") +
  ylab("Y_i")

print(plot_dependence)
```



If myvar was a good estimate and close to the var function in R, we would expect to see all values of their difference to be equal to 0. This graph shows that the values are concentrated around -1. The substraction between large elements in the summation may lead to catastrophic cancellation and lead to overflow. It is not immediately clear if the summations themselves are conducted properly as we are summing from larger to smaller values. We must find a way to prevent this issue. Summing in reverse order might lead to a better result as small terms could better contribute to the calculation, but it would still be innacurate.

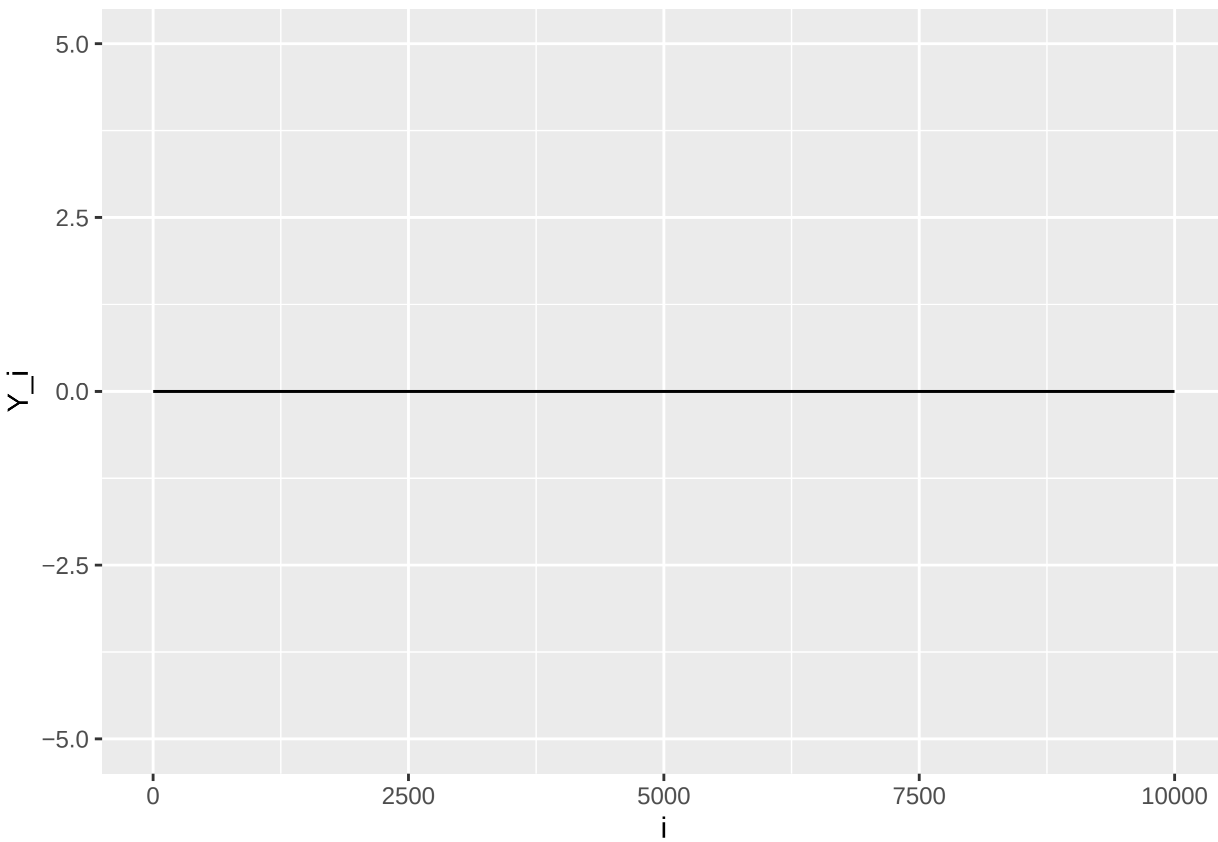
We can improve the calculation of the variance by remembering that the sample variance is

$$V[X] = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$$

. This is a good estimate of the variance of a population if the sample is large enough (which is our case). We can thus improve our variance calculation in the following way:

```
myvar_improved <- function(x){
  stopifnot(is.vector(x),
            is.numeric(x))
  mu <- mean(x)
  n <- length(x)
  variance_improved <- sum((x - mu)^2)/(n-1)
  return(variance_improved)
}

Y_improved <- c()
for (i in 1:length(vect_x)){
  Y_improved[i] <- myvar_improved(vect_x[1:i]) - var(vect_x[1:i])
}
```

```r
Y_improved = Y_improved[!is.na(Y_improved)]

data_Y_improved <- data.frame(values = Y_improved, len = 1:length(Y_improved))

plot_dependence_improved <-
  ggplot(data = data_Y_improved) +
  geom_line(aes(x = len, y = values)) +
  xlab("i") +
  ylab("Y_i") +
  ylim(-5,5)

print(plot_dependence_improved)
```



We have obtained better result this time by performing only one summation and comparing the numbers within the summation operation.

Question 4: Binomial coefficient

For expression A, an issue occurs if k = 0 or when n-k = 0 as R will be unable to compute the division. A similar issue occurs in the other sets of expressions for n-k = 0. (Are there other mechanisms?)

```r
# Formulae A
binom_a <- function(n,k){
  stopifnot(is.numeric(n), is.numeric(k))
  coeff_a <- prod(1:n) / (prod(1:k) * prod(1:(n-k)))
  return(coeff_a)
}
```

```r
# Formulae B
binom_b <- function(n,k){
  stopifnot(is.numeric(n), is.numeric(k))
  coeff_b <- prod((k+1):n) / prod(1:(n-k))
  return(coeff_b)
}
# Formulae C
binom_c <- function(n,k){
  stopifnot(is.numeric(n), is.numeric(k))
  coeff_c <- prod(((k+1):n) / (1:(n-k)))
  return(coeff_c)
}


# What a surprise!! R doesn't like vectors that have a length of 10^12
n <- c(2, 10, 10^2, 10^4)
k_rand <- sample(1:2, 4, replace = TRUE)
k1 <- n - k_rand
k2 <- c(0, 1, 2, 4)

small_a <- c()
small_b <- c()
small_c <- c()

large_a <- c()
large_b <- c()
large_c <- c()

for(i in 1:4){

  # small difference between n and k
  small_a[i] <- binom_a(n[i], k1[i])
  small_b[i] <- binom_b(n[i], k1[i])
  small_c[i] <- binom_c(n[i], k1[i])

  #big difference between n and k
  large_a[i] <- binom_a(n[i], k2[i])
  large_b[i] <- binom_b(n[i], k2[i])
  large_c[i] <- binom_c(n[i], k2[i])
}

print(small_a)
```

```
## [1]   2  10 100 NaN
```

```r
print(small_b)
```

```
## [1]     2    10   100 10000
```

```r
print(small_c)
```

```
## [1]     2    10   100 10000
```

```r
print(large_a)
```

```
## [1]  Inf   10 4950  NaN
```

```r
print(large_b)
```

```
## [1]    1   10 4950  NaN
```

```r
print(large_c)
```

```
## [1] 1.000000e+00 1.000000e+01 4.950000e+03 4.164167e+14
```

Formulae A

If either $k = 0$ or $n - k = 0$, then *denominator* $= 0$ and if *numerator* $= 0$, then *result* $= NaN$. For every other *numerator* > 0 we will get *result* $= Inf$, because any positive integer or *Inf* divided by *0* will become *Inf*.

If $n > 170$, then $n! = Inf$. This behavior is called *overflow*. Off course the same applies for $k > 170$, then in the denominator we will get $k! = Inf$. Therefore, if *numerator* $= Inf$ and *denominator* $= Inf$ we will get *result* $= NaN$.

If $n = k$ and $n, k <= 170$, then the right side of the denominator will equal to *0*. Therefore, *denominator* $= 0$ and *result* $= Inf$. Else if $n = k$ and $n, k > 170$, then $(prod(1 : k) = Inf$ and $prod(1 : (n-k)) = 0$. Therefore, $Inf * 0 = NaN$ and *result* $= \frac{Inf}{NaN} = NaN$.

Formulae B

For this implementation *overflow* can be detected if $n-k > 170$. This is because when calling $prod((k+1) : n)$ with a big n and a small k R can't handle this calculation and evaluates this expression to *Inf*. Same applies for the the *denominator*, $prod(1 : (n - k))$ will evaluate to *Inf*, too. Therefore, *result* $= NaN$.

If $n - k < 170$ and *numerator* $= Inf$, we get *result* $= \frac{prod((k+1):n)}{prod(1:(n-k))} = \frac{Inf}{0<denominator<Inf} = Inf$

If $n = k$, then $\frac{prod((k+1):n)}{prod(1:(n-k))} = \frac{Integer}{0}$. Therefore, *result* $= Inf$

Formulae C

This formulae delivers the best results for calculating diverse n and k values. This is because the critical *prod()* function is only called at the end after calculating the *quotient* for $\frac{((k+1):n)}{(1:(n-k))}$.

The only issue that might come up is *underflow* by dividing values from the *numerator-vector* with the according values from the *denominator-vector*.