Homework 1

Santiago Ruiz, Nikita Karetnikov,

2025-03-27

1 Exercise 1

In this exercise, we are asked to calculate the limits for a system of number representations with the following parameters:

- $b = 10$ base
- $m = 3$ mantissa length
- $e_{\min} = -3$
- $e_{\max} = 4$

To calculate the required numbers we would need the following formula:

$$x = (-1)^{w_0} b^e \sum_{i=1}^{m} u_i b^{-i}$$

Let us implement it in R:

```
calculate <- function(w0, b, e, u,m){
  mantissa <- 0

  for (i in 1:m) {
    mantissa <- mantissa + u[i] * b^(-i)
  }

  x <- (-1)^w0 * b^e * mantissa

  return(x)
}
```

Good! Now we will be able to determine the required numbers.

1.1 Determine the largest floating point number

We start with the largest floating point number. We assume that the largest number should have:

- the largest possible exponent
- mantissa with the largest digits in this system (which should be equal to b-1)

```
b <- 10
m <- 3

e_min <- -3
e_max <- 4
```

```r
x_max <- calculate(w0=0, b=b, e=e_max, u= rep(b-1, m),m=m)
print(x_max)
```

```
## [1] 9990
```

1.2 Determine the smallest positive floating point number

Now we should determine the smallest positive floating point number. We assume that the largest number should have:

- the smallest possible exponent
- mantissa with the smallest digits in this system (which should be equal to 0)

But since mantissa should start with non-zero we set its digits to [1,0,0] :

```r
x_min <- calculate(w0 = 0, b = b, e = e_min, u = c(1,0,0), m = m)

print(x_min)
```

```
## [1] 1e-04
```

1.3 Determine the largest floating point number smaller than one.

Now we should find the number that approaches value of 1, but that is smaller then one.

We set: - exponent to 0 - mantissa to have largest digits in this system (which should be equal to b-1)

```r
x_max_2 <- calculate(w0 = 0, b = b, e = 0, u= rep(b-1, m), m = m)
print(x_max_2)
```

```
## [1] 0.999
```

1.4 Determine the smallest floating point number greater than one

Now similarly we should find the number that is closest to one, but is bigger than one.

We set: - exponent to 0 - mantissa digits to be equal to [1,0,1]

```r
x_min_2 <- calculate(w0 = 0, b = b, e = 1, u = c(1,0,1), m = m)
print(x_min_2)
```

```
## [1] 1.01
```

2 Exercise 2

In this exercise we need to understand the limits or R number representation system by running experiments with convergence of the series.

2.1 For which n does the loop stop? (practical part)

We first implement convergence in R. We run the loop until the difference between S_{n+1} and S_n becomes smaller than the smallest number that R can distinguish. .

```r
S_n_prev <- -1
S_n <- 0
n <- 0

while (abs(S_n - S_n_prev) > .Machine$double.eps){
```

```r
  n <-  n + 1
  S_n_prev  <- S_n
  S_n <- S_n_prev + 2^(-2 * n)
}

cat("Loop stops at n =",n)
```

```
## Loop stops at n = 26
```

2.2 For which n does the loop stop? (theoretical part)

Now we need to theoretically approximate the value of n at which the loop stops. We observe that the convergence of our series resembles the formula used to represent numbers in R.

$$\sum_{i=1}^{\infty} 2^{-2i} \qquad x = (-1)^{w_0} b^e \sum_{i=1}^{m} u_i b^{-i}$$

Let us remind ourselves what is mantissa length and base in R is:

```r
m <-  .Machine$double.digits
cat("Mantissa length (m):", m, "\n")
```

```
## Mantissa length (m): 53
```

```r
b <-  .Machine$double.base
cat("Base (b):", b, "\n")
```

```
## Base (b): 2
```

With each iteration, the summation approaches its limit more closely, but the increment decreases rapidly (each term is 2^{-2i}). Due to floating-point representation limits in R (with a mantissa length $m = 53$ bits for double precision), the smallest distinguishable difference around 1 is about 2^{-53}.

The summation stops updating numerically when the next term 2^{-2n} becomes smaller than this limit:

$$2^{-2n} \approx 2^{-m} \quad \Rightarrow \quad n = \frac{m}{2} = \frac{53}{2} = 26.5$$

26.5 is quite close to the practical stopping point we found earlier.

3 Exercise 3

In this exercise we are asked to approximate exp(x) using Taylor series.

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

3.1 (i)

We implement an algorithm in R that approximates the exponential function $\exp(x)$ using the Taylor series expansion. The algorithm iteratively adds terms from the Taylor series until the absolute value of the next summand is smaller than `.Machine$double.eps^(1/2)` times the absolute value of the current approximation.

We run it to identify how many summands we need to approximate $\exp(x = 10)$

```r
calculate_exp <- function(X){
  term <- 1
  sum <- term
  n <- 0
  epsilon <- .Machine$double.eps^(1/2)


  while (abs(term) > epsilon * abs(sum)) {

    n <- n + 1
    term <- (X^n) / factorial(n)
    sum <- sum + term


  }
  return(sum)

}

X <- 40



cat("Our function:", calculate_exp(X), "\n")
```

```
## Our function: 2.353853e+17
```

```r
cat("R actual function", exp(X), "\n")
```

```
## R actual function 2.353853e+17
```

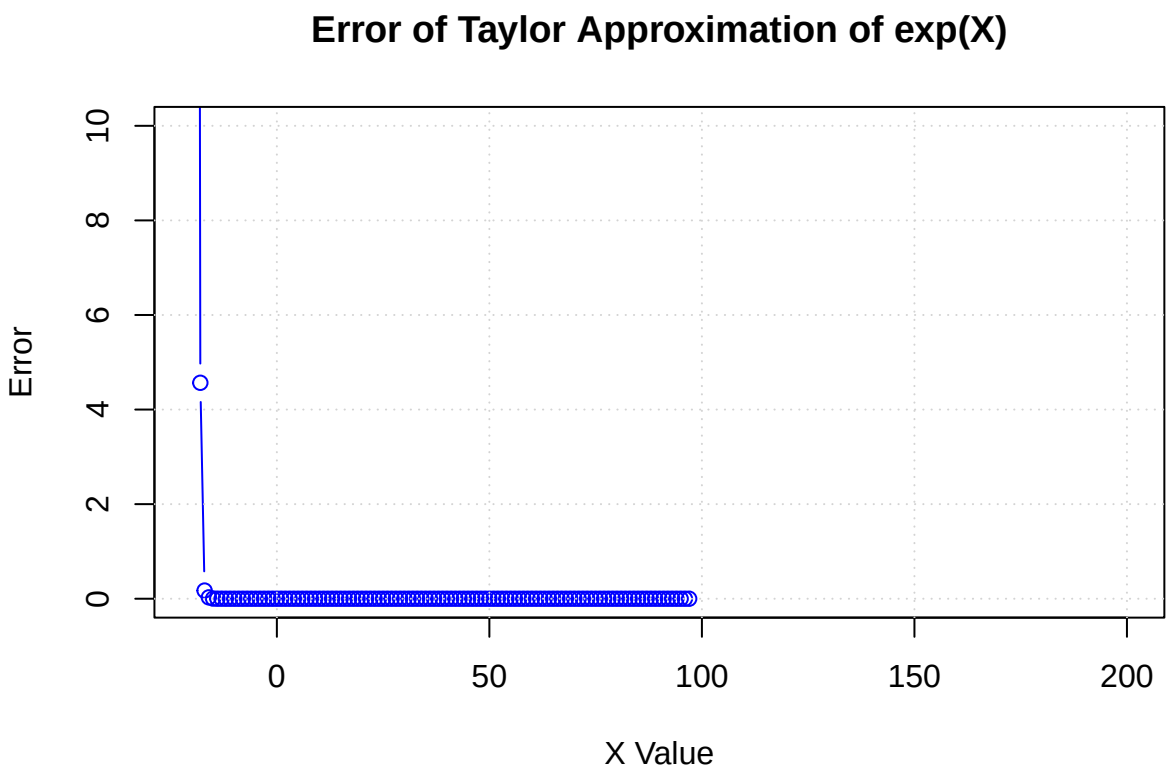
Looks that our implemented function closely matches R's built-in calculation of the exponential function. the exponent.

3.2 (ii)

Next, we evaluate the approximation for various values of X, both positive and negative. We calculate the MAPE for values ranging from -20 to 200.

```r
X_values <- seq(-20, 200, by=1)
errors <- numeric(length(X_values))

for (i in seq_along(X_values)) {
  X <- X_values[i]


  mape <- mean(abs((exp(X) - calculate_exp(X)) / exp(X))) * 100



  errors[i] <- mape
}


plot(X_values, errors, type = "b", col = "blue",
```

```
     xlab = "X Value", ylab = "Error",
     main = "Error of Taylor Approximation of exp(X)",
     ylim = c(0, 10))  # Adjust these limits as needed for your data
grid()
```



We notice that for lower values, MAPE increases starting approximately from -10.
For values of X higher than 100, MAPE is absent as the approximation of the exponent is probably equal to ∞.

Let us check:

```
X <- 101

cat("Our function:", calculate_exp(X), "\n")
```

```
## Our function: Inf
```

```
cat("R actual function", exp(X), "\n")
```

```
## R actual function 7.30706e+43
```

```
X <- -20

cat("Our function:", calculate_exp(X), "\n")
```

```
## Our function: 4.992704e-09
```

```
cat("R actual function", exp(X), "\n")
```

```
## R actual function 2.061154e-09
```

Indeed, either our approximations are too far from the actual values of the function, or our approximations are equal to infinity.

3.3 (iii)

Let us suggest a modification how to fix the numerical instability. We use the following property:

$$e^x = \left(e^{x/n}\right)^n$$

We implement it in our updated function as a wrapper:

```
calculate_exp_2 <- function(x, n = 10) {
    x_n <- x / n
    result <- calculate_exp(x_n)^n
    return(result)
}
```

Let's first check it for single values:

```
X <- -20

cat("Our function:", calculate_exp_2(X), "\n")
```

```
## Our function: 2.061154e-09
```

```
cat("R actual function", exp(X), "\n")
```

```
## R actual function 2.061154e-09
```

```
X <- 101

cat("Our function:", calculate_exp_2(X), "\n")
```

```
## Our function: 7.30706e+43
```

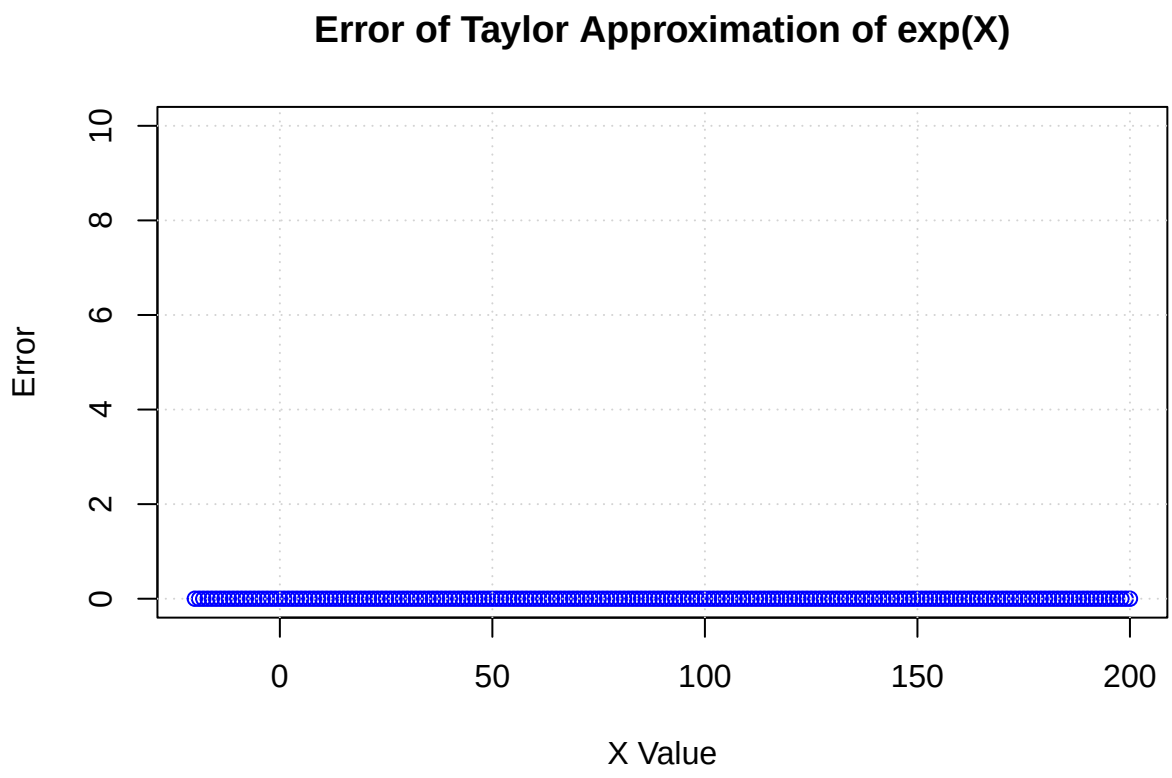
```
cat("R actual function", exp(X), "\n")
```

```
## R actual function 7.30706e+43
```

And now for our range:

```
X_values <- seq(-20, 200, by=1)
errors <- numeric(length(X_values))

for (i in seq_along(X_values)) {
  X <- X_values[i]


 mape <- mean(abs((exp(X) - calculate_exp_2(X)) / exp(X))) * 100



  errors[i] <- mape
}

# plotting

plot(X_values, errors, type = "b", col = "blue",
```

```
     xlab = "X Value", ylab = "Error",
     main = "Error of Taylor Approximation of exp(X)",
     ylim = c(0, 10))  # Adjust these limits as needed for your data
grid()
```



4 Exercise 4

We want to calculate

$$\frac{\sqrt{1+x}-1}{x}$$

4.1 Determine the values of R for x =1, 10^-4, 10^-32

```
function_x <- function(x) {
  return((sqrt(1 + x) - 1) / x)
}

function_x_rationalisation <- function(x) {
  return(1 / (sqrt(1 + x) + 1))
}


function_x(1)
```

```
## [1] 0.4142136
```

```
function_x(10^(-4))
```

```
## [1] 0.4999875
```

```
function_x(10^(-32))
```

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

4.2 Use the Taylor series of

$$\sqrt{1+x}$$

to approximate the result for small values of x. And Implement an approximate calculation of the result based on the Taylor series.

```
taylor_expansion_at_zero <- function(y) {
  return(1 + y / 2 - y^2 / 8 + y^3 / 16 - 5 * y^4 / 128 + 7 * y^5 / 256)
}

approximation_result_function_x_at_zero <- function(x) {
  return((taylor_expansion_at_zero(x) - 1) / x)
}

approximation_result_function_x_at_zero(1)
```

```
## [1] 0.4257812
```

```
approximation_result_function_x_at_zero(10^(-4))
```

```
## [1] 0.4999875
```

```
approximation_result_function_x_at_zero(10^(-32))
```

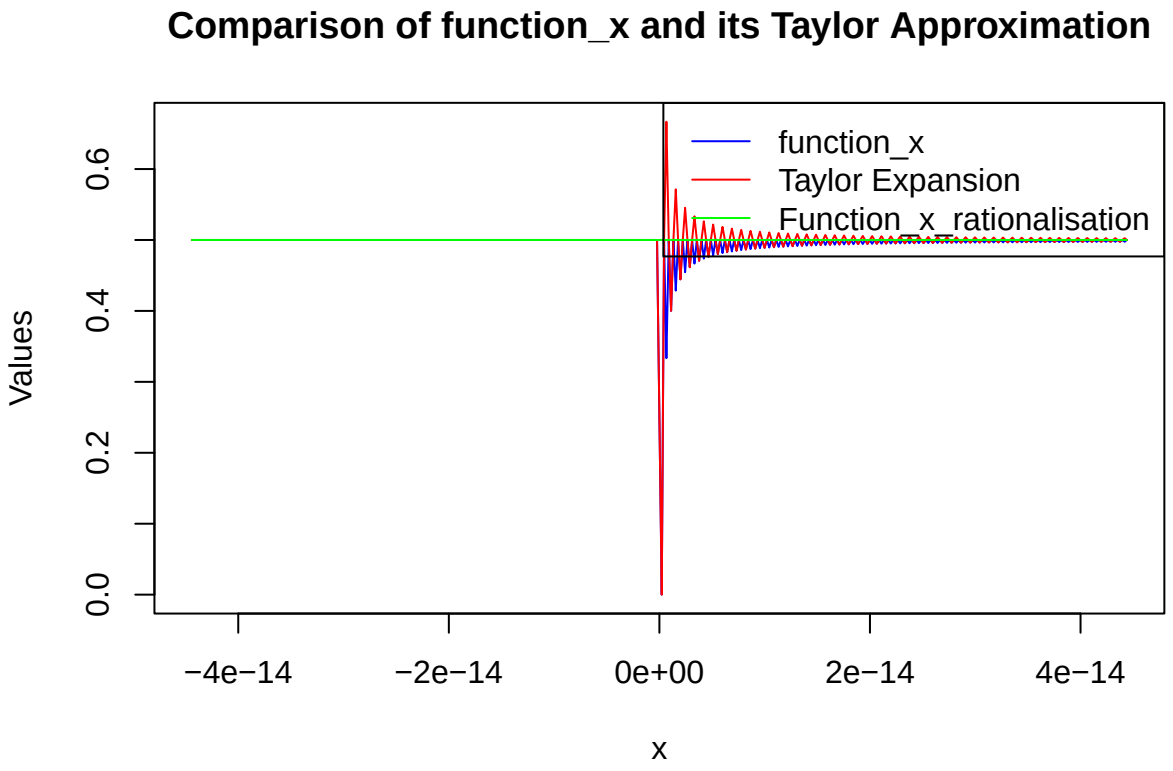
```
## [1] 0
```

```
# Create a loop that stores the results of a multiplication in a vector
multiplication_results_negative <- numeric(100)
multiplication_results_positive <- numeric(100)

for (i in seq_len(200)) {
  multiplication_results_negative[i] <- .Machine$double.eps*i + 0
  multiplication_results_positive[i] <- -.Machine$double.eps*i + 0
}

combined_results <- c(multiplication_results_negative, multiplication_results_positive)
sorted_results <- sort(combined_results)
```

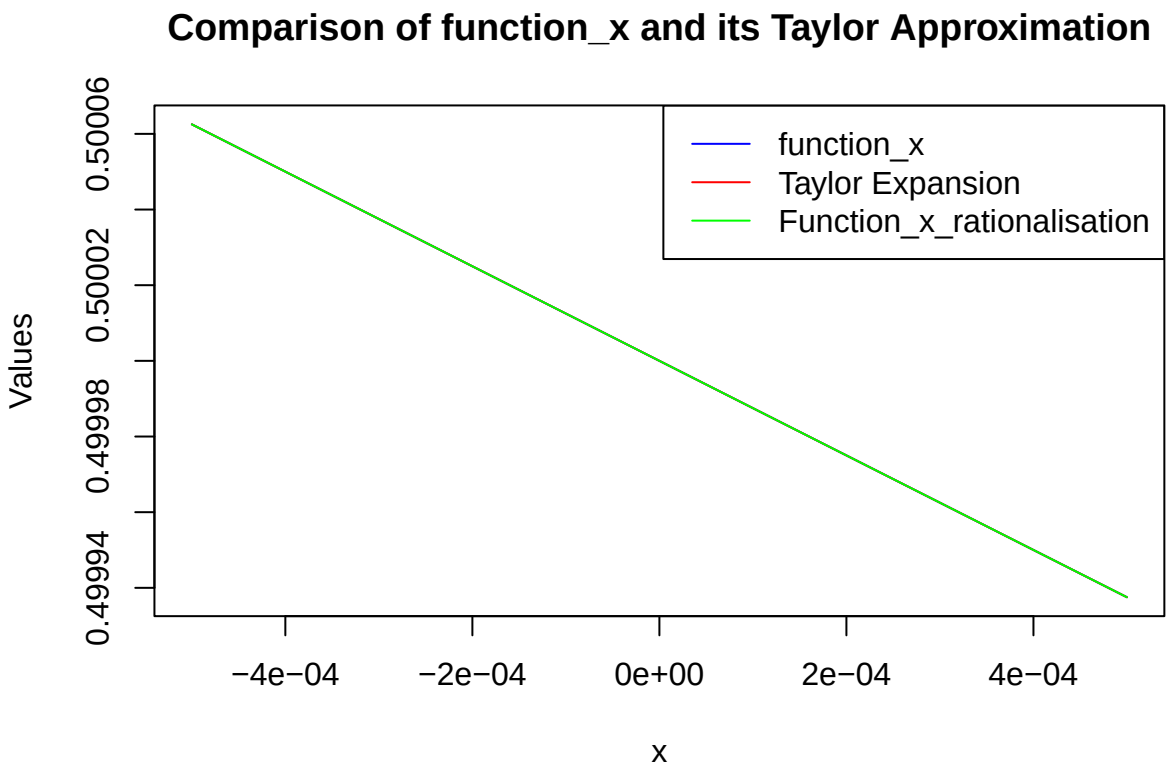
4.3 Compare the results for the two different calculations visually with a focus on small values of x. Explain what the problem is.

```
function_x_values <- sapply(sorted_results, function_x)
approximation_values <- sapply(sorted_results, approximation_result_function_x_at_zero)
function_x_rationalisation_values <- sapply(sorted_results, function_x_rationalisation)

plot(sorted_results, function_x_values, type = "l", col = "blue", ylim = range(c(function_x_values, appr
lines(sorted_results, approximation_values, col = "red")
```

```
lines(sorted_results, function_x_rationalisation_values, col = "green")
legend("topright", legend = c("function_x", "Taylor Expansion", "Function_x_rationalisation"), col = c(
```




```
# With greater values of around o
increment_values <- seq(-0.0005, 0.0005, length.out = 1000)

# Calculate function values for the new vector
function_x_increment_values <- sapply(increment_values, function_x)
approximation_increment_values <- sapply(increment_values, approximation_result_function_x_at_zero)
function_x_rationalisation_increment_values <- sapply(increment_values, function_x_rationalisation)

# Plot the new values
plot(increment_values, function_x_increment_values, type = "l", col = "blue", ylim = range(c(function_x_
lines(increment_values, approximation_increment_values, col = "red")
lines(increment_values, function_x_rationalisation_increment_values, col = "green")
legend("topright", legend = c("function_x", "Taylor Expansion", "Function_x_rationalisation"), col = c(
```




```
print(function_x_increment_values[1])
```

```
## [1] 0.5000625
```

```
print(approximation_increment_values[1])
```

```
## [1] 0.5000625
```

```
print(function_x_rationalisation_increment_values[1])
```

```
## [1] 0.5000625
```

Answer: As shown in the first graph, the divergences between the values of the taylor approximation, and the function in its two forms differ when the values of zero are very small of the order of .Machine$double.eps that represents the minimal differences stored in doubles so that 1 + .Machine$double.eps is recognized as different than 1. According to what we saw in class we can think on two possible answers. Similar to computing the mean either through the sum of squares of deviations or as the differences between the mean of the squared values and the squared value of the mean, it may be R uses different process to solve (sqrt) and polynomial fractions that yield different results for very small values of x. It can be also that the taylor approximation is not accurate enough for very small values of x.

5 Exercise 5

```
data("KNex", package = "Matrix")
X <- as.matrix(KNex$mm)
Y <- KNex$y
```

```r
# It may be more accurate ways to compute the type of execution.
start_time <- Sys.time()
fit <- lm.fit(X, Y)
end_time <- Sys.time()
execution_time_lmfit <- end_time - start_time
print(paste("Execution time:", execution_time_lmfit ))
```

```
## [1] "Execution time: 0.499016046524048"
```

```r
# Formula of Least Square estimator B = (X^T X)^-1 X^T Y
# Let us try to find this (X^T X)^-1 part with

#####################################################
### Cholesky decomposition
#####################################################
# We find the product of the matrix X^TX (This matrix is symetric)
# Crossprod makes it so that it computes only the lower triangular product, it does then only
# half of the work, by computing only the upper triangular part of the matrix.


R <- chol(crossprod(X))

# Using Solve to find the inverse of both matrix and to multiply the right upper triangular matrix

start_time <- Sys.time()
chol_betas <- backsolve(R, forwardsolve(R, crossprod(X, Y), upper.tri = TRUE, transpose = TRUE))
end_time <- Sys.time()
execution_time_chol <- end_time - start_time
print(paste("Execution time:", execution_time_chol))
```

```
## [1] "Execution time: 0.00500702857971191"
```

```r
#####################################################
### QR decomposition
#####################################################
QR <- qr(X)
qr_betas <- backsolve(qr.R(QR), crossprod(qr.Q(QR), Y))
solve_r_builtin <- qr.solve(QR, Y)


start_time <- Sys.time()
qr_betas_solve <- solve(QR, Y)
end_time <- Sys.time()
execution_time_qr <- end_time - start_time
print(paste("Execution time:", execution_time_qr))
```

```
## [1] "Execution time: 0.00663185119628906"
```

```r
# The betas are identical, but the execution time is different.

fit$coefficients[2]
```

```
##       x2
## 340.1156
```

```r
chol_betas[2]
```

```
## [1] 340.1156
```

```
qr_betas[2]
```

```
## [1] 340.1156
```

Answer: LM FIT requires the longest execution time, followed by Cholesky decomposition and QR. The time seems to vary as if the algorithm used for solving all methods will depend on a random parameter.

5.1 Determine the class of the matrix mm and the vector y. Inspect the structure of mm and explain how the data is stored. Determine the sparsity level of the matrix, i.e., the proportion of non-zero entries in the matrix. Also determine an estimate of the memory being used by mm and as.matrix(mm) in MB (see ?object.size).

```
library(coop)
```

```
## Warning: package 'coop' was built under R version 4.2.2
```

```
library(Matrix)
```

```
## Warning: package 'Matrix' was built under R version 4.2.3
```

```
object.size(X)
```

```
## 10537816 bytes
```

```
X_sparse <- KNex$mm
object.size(X_sparse)
```

```
## 109416 bytes
```

```
X_sparse_matrix <- Matrix(KNex$mm, sparse = TRUE)
object.size(X_sparse_matrix)
```

```
## 109416 bytes
```

```
# Storing the matrix using the Matrix library and the sparse option makes it even a lower-bit file.

# The class is a dgCMatrix, which is a sparse matrix. Meaning that it
# saves some space by not saving the zeros.

class(KNex$mm)
```

```
## [1] "dgCMatrix"
## attr(,"package")
## [1] "Matrix"
```

```
class(X)
```

```
## [1] "matrix" "array"
```

5.2 Use QR decomposition and the Cholesky decomposition to solve the least squares problem using the sparse matrix. Compare the solutions and the timings.

```
# When storing the matrix as a "Matrix" object it requires more bits. (Almost 100 times more)
# Computing the sparsity (count(0 entries in the matrix)/nXm)
```

```r
sparsity <- function(mat, tol = .00001) {
    mat[abs(mat) < tol] <- 0
    100 - (100 * sum((mat != 0) * 1) / prod(dim(mat)))
}

# Computing the sparsity in two ways
sparsity(X)
```

```
## [1] 99.33541
```

```r
coop::sparsity(X)
```

```
## [1] 0.9933541
```

```r
# Around 99 % of the values are very near to zero


#######################################################
### Cholesky decomposition
#######################################################
# We find the product of the matrix X^TX (This matrix is symetric)
# Crossprod makes it so that it computes only the lower triangular product, it does then only
# half of the work, by computing only the upper triangular part of the matrix.


R <- chol(crossprod(X_sparse_matrix))
# Since crossprod only works with Matrixes of vectors, I transformed the mm dgCMatrix into a Matrix

# Using Solve to find the inverse of both matrix and to multiply the right upper triangular matrix

start_time <- Sys.time()
chol_betas <- backsolve(R, forwardsolve(R, crossprod(X_sparse_matrix , Y), upper.tri = TRUE, transpose =
end_time <- Sys.time()
execution_time_chol <- end_time - start_time
print(paste("Execution time:", execution_time_chol))
```

```
## [1] "Execution time: 0.00727319717407227"
```

```r
#######################################################
### QR decomposition
#######################################################
QR <- qr(X_sparse)
qr_betas <- backsolve(qr.R(QR), crossprod(qr.Q(QR), Y))



start_time <- Sys.time()
qr_betas_solve <- solve(QR, Y)
end_time <- Sys.time()
execution_time_qr <- end_time - start_time
print(paste("Execution time:", execution_time_qr))
```

```
## [1] "Execution time: 0.00205278396606445"
```

```
fit$coefficients[2]
```

```
##       x2
## 340.1156
```

```
chol_betas[2]
```

```
## [1] 340.1156
```

```
qr_betas[2]
```

```
## [1] -1028.976
```

The execution time for the QR is smaller for the sparse matrix. However, using the Cholesky method in a sparse matrix increases the computation time.