Problem Set 7: STAT243

Cristobal Pais - cpaismz@berkeley.edu

November 17, 2017

Problem 1: Simulation

Based on the information and values provided, we can determine if the standard error properly characterizes the uncertainty of the estimated regression coefficient as follows (clearly, we have far more ways of addressing this):

i) Using our knowledge from the theory behind a regression model, we can compare the distribution of the standard errors from the statistical method and the ones obtained from the simulation approach. Clearly, we will expect them to be very similar when the uncertainty of the estimated regression coefficient is well characterized, as well as a convergence toward a normal distribution. This is based on the fact that the standard errors from a regression model are assumed to be distributed following a normal distribution (as a main assumption of the regression models: residuals come from a normal distribution, usually tested via the Shapiro test) and thus, we should expect the simulated errors to converge to this distribution.

   Therefore, we can perform some statistical test and/or develop a visual analysis of both distributions and conclude about the good/bad characterization of our simulation study.

ii) Like in any simulation study, we should be aware of the fact that we are estimating values and we do not have the REAL value of the variables of interest since we are taking a sampling (a subset from the total Universe) from a particular distribution. Therefore, we should construct confidence intervals for both the statistical method and the simulation study results in order to be able to establish the values of the regression coefficients with a certain level of significance, where the standard errors distribution plays a fundamental role in order to characterize these intervals.

   Thus, we would be able to check if the intervals are overlapping between both methods (statistical and simulation) and perform some extra tests such as t-median tests in order to compare the regression values. In this case (as an example), we would expect to not reject the null hypothesis of equal means in the case that both the averages and the standard error values are well characterized by the simulation study.

iii) In addition, it is very important to measure and check if the number of simulations is enough for getting significant results for a selected level of significance/accuracy. As we already saw during the lectures (and our simulation background), there are several ways to asset the number of replications needed in order to satisfy certain study requirements.

iv) As an extra analysis, depending on the kind of model and data that is being simulated, it is important to check if the errors satisfy the homoscedasticity/heteroscedasticity assumption.

Problem 2

We have A a symmetric matrix and we want to find the following quantity:

$$||A||_2 = \sup_{z:||z||_2=1} \sqrt{Az^TAz} \quad (1)$$

$$= \max_{z:||z||_2=1} ||Az|| \quad (2)$$

Based on our optimization background, we can easily solve this quadratic optimization problem using the classic lagrangian multipliers approach where the constrains ($||z||_2 = 1$ in this case) can be added to the objective function as a penalized term, using a multiplier λ in order to quantify the magnitude of the penalty term (deviations from the optimal/feasible value). Therefore, we proceed as follows:

i) We penalize the constrain $||z||_2 = 1$ with a penalty value λ.

ii) We construct the Lagrangian function $L(z, \lambda)$ using the squares of both the original objective function and the constraint in order to have a very easy problem (classic trick/manipulation when dealing with norms):

$$L(z, \lambda) = ||Az||^2 - \lambda(||z||^2 - 1) \quad (3)$$

$$= z^TA^2z - \lambda(z^Tz - 1) \quad (4)$$

iii) Now, we optimize the Lagrangian function by taking the derivative with respect to the z vector and equalizing the equation to 0, obtaining:

$$A^2z - \lambda z = 0 \quad (5)$$

iv) Clearly, the solution of this system is:

$$\lambda^* = \text{Eigenvector\_ of\_ } A^2 \quad (6)$$

v) Since A is symmetric, we know that all its eigenvalues are real numbers and thus, the maximum value reached by z^TA^2z will be achieved when $||z||_2 = 1$ with a maximal eigenvalue of the matrix A^2. Following the same logic, we can easily decompose the A matrix (since it is symmetric) by its eigenvectors and eigenvalues, as we saw in class:

$$A = \Gamma\Lambda\Gamma^T \quad (7)$$

Where Γ is the matrix containing the eigenvectors (orthogonal matrix) and Λ is a diagonal matrix containing the eigenvalues of A.

vi) Similarly, for A^2 we have (remembering that Γ is orthogonal):

$$A^2 = \Gamma\Lambda^2\Gamma^T \quad (8)$$

Therefore, the eigenvalues of A^2 are equivalent to the squares of the eigenvalues of A.

Hence, the simple euclidean norm $||A||_2$ consists of the square root attached from the supremum (maximum) of z^TA^2z when we have an unitary vector norm $||z||_2 = 1$ and it is exactly the square root of the maximal eigenvalue that A^2 can reach, that is equal to the maximal absolute eigenvalue of the original matrix A. Thus, we have proven the statement (remember that we optimized the square version of the problem).

Problem 3

a) Eigenvectors & Eigenvalues

We have a regular matrix $X_{n\times p}$ where $n > p$ and we want to show that the right singular vectors of the X matrix are exactly the eigenvectors of the matrix X^TX and that the eigenvalues of this new matrix are the squares of the singular values of the original X matrix. In addition, we will show that the X^TX matrix is, in fact, a positive semi-definite matrix.

In order to prove the first statement, we proceed as follows:

i) We can decompose the X matrix using the known *Singular Value Decomposition* (SVD) method (also known as spectral decomposition) where we have a non-negative diagonal matrix in the center $D_{k\times k}$ and two orthogonal matrices $U_{n\times k}$ (left singular vectors) and $V_{p\times k}$ (right singular vectors) as multiplying factors as follows:

$$X = UDV^T \tag{9}$$

ii) Based on the previous expression, we can easily compute X^TX using the decomposition:

$$\begin{aligned} X^TX &= (UDV^T)^TUDV^T && (10)\\ &= VD^TU^TUDV^T && (11)\\ &\underbrace{=}_{U \text{ orthogonal}} VD^TDV^T && (12)\\ &\underbrace{=}_{D \text{ diagonal}} VD^2V^T && (13) \end{aligned}$$

From the previous expression (and remembering that V is an orthogonal matrix), we have that the elements of the original diagonal D from the decomposition of X, $d_ii\ \forall i \in \{1,..,k\}$ satisfy:

$$\begin{aligned} d_{ii} &= \sqrt{\lambda_i(X^TX)} && (14)\\ &\underbrace{=}_{\text{by symmetry}} \sqrt{\lambda_i(XX^T)} && (15) \end{aligned}$$

Therefore, we can easily check that the non-zero right singular vectors of the X matrix (V matrix) are the eigenvectors associated with X^TX and that the non-singular eigenvalues of X are the square root of the non-zero eigenvalues of X^TX, in other words: the eigenvalues of X^TX are the squares of the singular values of the original X matrix.

As an extra comment notice that this property shows the equivalence between the SVD and the PCA techniques.

iii) For completeness, we can easily obtain the following expression by multiplying both sides of the equation (13) by the corresponding V^{-1} matrices:

$$V^TX^TXV = D^2 \tag{16}$$

Here, we can clearly see the fact that the squares of the eigenvalues of X correspond to the eigenvalues of X^TX (crossproduct() matrix) and the fact that the right singular vectors (V) of X are in fact the eigenvectors of X^TX.

Based on the previous analysis, we can easily check that the matrix X^TX is positive semi-definite. By definition, a positive semi-definite matrix satisfies the following equation:

$$x^TAx \geq 0 \Leftrightarrow z^TX^TXz \geq 0 \tag{17}$$

Now, based on our previous analysis we know that X^TX can be expressed as:

$$X^TX = VD^2V^{T]\Leftrightarrow V^TX^TXV=D^2} \tag{18}$$

From these equations, we can easily check that replacing V by any other vector/matrix will lead us to a resulting matrix $D^2 \geq 0$ because all the values in the diagonal are square numbers and thus each of them satisfies $d_{ii}^2 \geq 0$. Therefore, the matrix X^TX is clearly positive semi-definite (the diagonal of the D matrix can have null entries).

b) Efficient eigenvalues calculations

In this case, we have an $n \times n$ positive semi-definite matrix Σ. We already computed the eigendecomposition of the Σ matrix and thus, we have access to its eigenvectors u and eigenvalues λ.

Now, we want to calculate the eigenvalues from a perturbed $\hat{\Sigma}$ matrix Z, such that $Z = \Sigma + cI$ where c is a scalar and I is the identity matrix. The idea is to use our previous computations in order to find the eigenvalues of Z in $O(n)$ arithmetic calculations.

In order to perform this, we can proceed as follows:

i) By definition, we know that if u is an eigenvector of Σ with a corresponding eigenvalue λ, we have the following equality:

$$\Sigma u = \lambda u \tag{19}$$

ii) Now, we can create the matrix $Z = \Sigma + cI$ as stated above. Let's suppose that we take the vector u and we multiply Z by it as follows:

$$\begin{aligned} Zu &= (\Sigma + cI)u && (20)\\ &= \Sigma u + cu && (21)\\ &= \lambda u + cu && (22)\\ &= (\lambda + c)u && (23) \end{aligned}$$

From here, we can see that the vector u corresponding to the λ eigenvalue of the Σ matrix is in fact the eigenvector corresponding to $\lambda + c$, for the $Z = \Sigma + cI$ matrix.

iii) Using the previous property, we can easily compute all the n eigenvalues of Z by simply perform the n arithmetic operations $(\lambda_i + c)u_i \ \ \forall i \in \{1, .., n\}$, where λ_i, u_i are the ith eigenvalue and eigenvectors of the original Σ matrix.

Hence, we can clearly compute the eigenvalues of Z in $O(n)$ arithmetic operations if we already have the eigendecomposition of the Σ matrix.

Problem 4: Constrained OLS

The general approach for this problem consists of following the tricks and efficient steps studied in class in order to optimize the linear algebra operations inside our code such as:

- The optimal order of operations: save computational time and use of memory (RAM).
- Possible useful factorization/decomposition (like Cholesky's).
- Avoiding the computation of inverse matrices (back/forward-solve approach)
- Vectorized operations

In this particular case, we have a constrained OLS model where the matrix A represents the coefficients of these constraints and the vector b represents the right-hand side of them.

The solution from the classic Lagrange multiplier approach gives us:

$$\hat{\beta} = C^{-1}d + C^{-1}A^T(AC^{-1}A^T)^{-1}(-AC^{-1}d + b) \tag{24}$$

where $A \in R^{mp}$, $X \in R^{n\times p}, C = X^TX \in R^{p\times p}, b \in R^m, d = X^TY \in R^p$.

Therefore, we can analyze each component and term of the global expression in terms of its complexity/amount of calculations in order to determine the most efficient approach:

i) $C^{-1} = (X^TX)^{-1}$: In this case, we have that the total amount of memory stored is the order of p^2 numbers and the calculations for obtaining C are $O(np^2)$. We would like to avoid the explicit computation of C^{-1} since, as we study in class, the numerical issues associated with the inversion can affect our results and it is clearly not the most optimal way (actually it is a very naive way) of obtaining the desire expression value.

   In this case (in R context), we can start by computing $C = X^TX$ using the $crossprod(X)$ function instead of the naive $t(X)\%*\%X$ implementation that is far slower than the $crossprod()$ approach. Once we have the value of C ($O(np^2)$ operations) we can apply a Cholesky's decomposition to C such that we have $U_C = chol(C)$ where U is an upper-triangular matrix. Once we have it, we can use it for computing expressions involving C^{-1} via the $backsolve()$ function.

ii) $d = X^TY$: The computation of d involves $O(np)$ arithmetic calculations and it is a very useful term due to the fact that its final dimensions make it a scalar instead of a matrix ($d \in R^p$). Hence, we should start multiplying some expressions by this vector in order to save computations.

   In the R implementation context for part (b), we can easily compute d as the $crossprod(X, Y)$. A very efficient and simple way of obtaining its value.

iii) $C^{-1}d$: Looking at the expressions, we can see that this term is used twice in the calculations for $\hat{\beta}$. Hence, we can take advantage of this and pre-calculate it. Since we want to avoid explicit inverse matrix calculations, we can again apply a Cholesky's decomposition in order to obtain the value of the expression by solving a series of linear equations. The order of arithmetic operations is $O(p^2)$

   Important is to notice that in the case of the $-AC^{-1}d$ term it is better to previously compute $C^{-1}d$ and then multiply it by $-A$ since we will multiply a matrix with a scalar instead of two matrices ($-(AC^{-1})d$ vs $-A(C^{-1}d)$).

For our function in part (b), we will use the fact that $x = C^{-1}d \Leftrightarrow Cx = d$. Therefore, we can use our previously calculated U_C upper-triangular matrix for solving the system via two calls of the *backsolve*() function: $backsolve(U_C, backsolve(U_C, d, transpose = TRUE))$.

iv) $C^{-1}A^T$: Another term that is used twice in the final computation of $\hat{\beta}$. Here we can use the Cholesky's decomposition approach again in order to avoid the calculation of the inverse of C. The order of calculations is $O(p^2m)$ if we have the inverse matrix.

In our function, we will compute its value as in the previous term: using the consecutive *backsolve*() calls by re-using our U_C matrix.

v) In order to calculate the $(AC^{-1}A^T)^{-1}(-AC^{-1}d + b)$ expression we can again use the Cholesky's decomposition as follows: using the fact that $x = (AC^{-1}A^T)^{-1}(-AC^{-1}d + b) \Leftrightarrow (AC^{-1}A^T)x = (-AC^{-1}d)$ (notice that if A has inverse, we can multiply if on the left side of both equations), we can just calculate the Cholesky's decomposition of the $(AC^{-1}A^T)$ term, where we already know the value of $C^{-1}A^T$ from the previous point. Therefore, we just multiply if by A on the left side and we obtain $U_{AC^{-1}A^T}$.

Once we have the upper-triangular matrix, we can simply use the *backsolve*() approach in R by multiplying the known values of the right side $(C^{-1}d)$ by $-A$ on the left while adding the vector b as follows: $backsolve(U_{AC^{-1}A^T}, backsolve(U_{AC^{-1}A^T}, (-AC^{-1}d + b), transpose = TRUE))$.

vi) Using all the previous elements, we can easily compute the value of $\hat{\beta}$ by simply adding the value of $C^{-1}d$ (known) to the previous expression.

Hence, we will implement the previous steps in our R function in part (b), as well as two Naive approaches in order to perform a formal comparison between different approaches.

a) Pseudo-Code

Based on all the previous analysis, the Pseudo-code for our implementation is as follows:

Algorithm 1 Pseudo-Code: $\hat{\beta}$ Constrained OLS

```
1: procedure COMPUTE BETACOLS(X, Y, A, B
2:     d = X^T Y                                   # Computes vector d
3:     U_X = cholesky_decomp(X^t X)                # Computes the Cholesky's decomp. of X^T X
4:     R_1 = Backsolve(U_X, d)                     # Returns (X^T X)^{-1} d
5:     R_2 = Backsolve(U_X, A^T)                   # Returns X^T X^{-1} A^T
6:     U_{AR_2} = cholesky_decomp(AR2)             # Computes the Cholesky's decomp. of AR_2
7:     R_3 = -AR_1 + b                             # Pre-computes value for backsolving
8:     R_4 = Backsolve(UAR_2, R_3)                 # Returns (A(X^T X)^{-1} A^T)(-A(X^T X)^{-1} d + b)
9:     β̂ = R_1 + R_2 R_3                           # Computes the final Beta
10:    return β̂
```

In this case, the *cholesky_decomp*() function is the equivalent to *chol*() in R while the *Backsolve*() method is a contraction of the nested *backsolve*() implementation in R. Therefore, we are following exactly the steps we already discussed above when we were analyzing the most efficient implementation for the algebraic operations needed to compute $\hat{\beta}$.

b) R functions

In order to perform a formal comparison between a Naive implementation and our already described more efficient approach taking advantage of the Cholesky's decomposition and the use of consecutive *backsolve*() calls for solving the linear equations instead of calculating the explicit inverse matrices (or using the *solve*() function), we developed three functions: (1) A naive approach computing everything as-is (no special orders or decomposition), (2) a second Naive approach where computations are performed in an iterative way, and (3) the more efficient approach described at the beginning of this section, using Cholesky's decomposition as the main tool for improving the performance of the function.

1. The first implementation (Naive approach) consists of calculating the final expression in one line by using the *solve*() function and explicit multiplications without taking into account the order of the operations:

```r
# Naive first implementation
NaiveOLS <- function(A, b, X, Y){
  # Compute final beta expression (all together)
  FinalBeta = solve(t(X) %*% X) %*% t(X) %*% Y +
              solve(t(X) %*% X) %*% t(A) %*% solve(A %*% solve(t(X) %*% X) %*%
              t(A)) %*% (-A %*% solve(t(X) %*% X) %*% t(X) %*% Y + b)

  return(FinalBeta)
}
```

2. In the second implementation we compute the C matrix and its explicit inverse by using the *solve*() function. Then, each part of the final expression is computed separately and then the final $\hat{\beta}$ vector is computed:

```r
# Naive second implementation
Naive2OLS <- function(A, b, X, Y){
  # C = X^{t} X
  C = t(X) %*% X

  # Explicit inverse
  Cinv = solve(C)

  # d = X^{t} Y
  d = t(X) %*% Y

  # Compute relevant parts of Beta
  Beta_Naive1 = Cinv %*% d
  Beta_Naive2_1 = Cinv %*% t(A)
  Beta_Naive2_2 = solve(A %*% Beta_Naive2_1)
  Beta_Naive2_3 = (-A %*% Cinv %*% d + b)

  # Compute final beta expression
  FinalBeta = Beta_Naive1 + Beta_Naive2_1 %*% Beta_Naive2_2 %*% Beta_Naive2_3
  return(FinalBeta)
}
```

3) Finally, the Cholesky's efficient implementation is declared. In this approach, we follow the steps indicated at the beginning of the section, taking advantage of the *crossprod*(), *backsolve*(), and *chol*() functions in order to avoid the computation of the inverse matrices and we take into account the optimal order (in terms of number of operations and memory usage) of the operations.

```
# Cholesky decomposition implementation
CholOLS <- function(A, b, X, Y){
  # Cholesky implementation
  # d = X^{t} Y
  d = crossprod(X, Y)

  # C^{-1} %*% d
  U1 = chol(crossprod(X))
  R1 = backsolve(U1, backsolve(U1, d, transpose = TRUE))

  # C^{-1} t(A)
  R2 = backsolve(U1, backsolve(U1, t(A), transpose = TRUE))

  # (AC^{-1}t(A)) %*% (-AC^{-1}d + b)
  U2 = chol(A %*% R2)
  R3 = backsolve(U2, backsolve(U2, -A %*% R1 + b, transpose = TRUE))

  # Compute final beta
  FinalBeta = R1 + R2 %*% R3
  return(FinalBeta)
}
```

4) In addition to the previous functions, we define an auxiliary function for generating the data needed to run the code. Hence, all matrices and vectors are randomly generated by giving the main dimensions as the inputs of the function:

```
# Generate data function
GenerateData <- function(n, m, p){
  # Data
  A = matrix(as.integer(runif(n = m * p, 0, 100)), m, p)
  b = as.integer(runif(n = m, 0, 100))
  X = matrix(as.integer(runif(n * p, 0, 100)), n, p)
  Y = as.integer(runif(n = n, 0, 100))

  return(list(A, b, X, Y))
}
```

5) In order to perform the comparison, we use the *microbenchmark* package for checking the summary statistics regarding the performance of each function.

```r
# Main Script
# Loading libraries
library(microbenchmark)

# Dimensions
n = 800
p = 600
m = 400

# Generate data
Data = GenerateData(n, m, p)
```

```r
# Benchmark Comparisons
microbenchmark(NaiveOLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]]),
               Naive2OLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]]),
               CholOLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]]),
               times = 100)

## Unit: milliseconds
##                                                 expr       min        lq      mean    median
##   NaiveOLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]]) 1749.8828 1809.1558 1897.9179 1852.2142
##  Naive2OLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]])  563.9098  589.4101  618.0459  603.4677
##    CholOLS(Data[[1]], Data[[2]], Data[[3]], Data[[4]])  350.1354  355.5136  373.9600  360.4070
##         uq       max neval
##  1930.7944 2442.5337   100
##   622.9787  901.6513   100
##   389.4656  486.1582   100
```

Based on the results, we can easily see that the *CholOLS*() function is the most efficient implementation taking about 1/6 of the time taken by the first Naive implementation and about half of the second (improved) Naive implementation. Important is to note that the first Naive implementation is by far the worse in terms of performance (one line of computations without any particular order) while the second Naive implementation, although still not efficient, improves its performance in a significant way.

Problem 5: Two-stage least squares

Similarly to the previous problem, in this case, we want to solve a series of algebraic operations that can be very slow and, in particular in this case, extremely high memory consuming, leading to out of memory errors unless we have access to a computer with a very large amount of RAM memory. This is due to the fact that the dimensions of the matrices involved in the calculations are extremely large: $X \in R^{60e6 \times 600}$, $Z \in R^{60e6 \times 630}$, and $y \in R^{60e6}$.

Looking at the expressions we have:

$$\hat{X} = Z(Z^T Z)^{-1} Z^T X \tag{25}$$

$$\hat{\beta} = (\hat{X}^T \hat{X})^{-1} \hat{X}^T y \tag{26}$$

I. Approach 1

1. In the case of $\hat{X}$ we have that the multiplication between $Z^T X$ will give us a 630×600 matrix (orders of operations $O(2.268e13)$, allowing us to reduce the dimensionality of the problem by performing this operation first. Then, for the computation of the inverse matrix, we can use (again) the Cholesky's decomposition approach in order to use the nested *backsolve*() functions with *crossprod*(Z) (630×630 matrix, $O(2.3814e13)$ calculations).

   After this, we will have the solution for $(Z^T Z)^{-1} Z^T X$, with dimensions 630×600. Finally, we just need to multiply it by Z on the right side, obtaining a $6e7 \times 600$ matrix ($\hat{X}$). Clearly, if we are not keeping it exploiting its sparsity, a very large amount of memory will be needed in order to keep it for the second stage (more comments in section (a)).

2. In the second stage, we can start by using the *crossprod*() function with the $\hat{X}$ matrix and the y vector, obtaining a 600×1 vector ($O(3.6e10)$ calculations). Then, we can again use the Cholesky's decomposition approach for obtaining the final value for $\hat{\beta}$ without explicitly calculating the inverse of the crossproduct of the $\hat{X}$ matrix. Thus, we obtain the final 600×1 vector.

After this analysis, we are able to answer the main questions of the problem.

II. Approach 2

1. In a second approach for solving the equations, we can just replace the value of $\hat{X}$ in the $\hat{\beta}$ expression in order to take advantage of the symmetry of the terms inside of it. This way, we obtain the following expression:

$$\hat{\beta} = (X^T Z(Z^T Z)^{-1} Z^T X)^{-1} X^T Z(Z^T Z)^{-1} Z^T y \tag{27}$$

$$= [X^T P_Z X]^{-1} [X^T P_Z y] \tag{28}$$

   where $P_Z = Z(Z^T Z)^{-1} Z^T$, satisfying $P_Z = P_Z^T$, $P_Z P_Z^T = P_Z$, $P_Z Z = Z$, and $Z^T P_Z = Z^T$

   Based on this approach, we can see that if we are able to compute P_Z, we would only need to multiply it by X^T on the left and then we just need to adjust one expression by X and the second one by y. However, looking at the dimension of P_Z we can see that we will end up performing far more operations than the ones we want, although the expressions are very useful for understanding the value of $\hat{\beta}$ and storing objects.

a) Memory issues

Based on the previous discussion, some of the main reasons why we cannot perform this calculation as given in the equations are:

i) At this point, it is clear that one of the main issues for performing the previous calculations consist of the fact that the dimensions of the matrices/vectors are extremely large, with a number of 60 million rows: each object will take/need a huge amount of memory just to be initialized and stored.

   If the matrices (in particular the sparse ones) are not manipulated using special packages such as *spam*, the amount of memory just wasted with 0s will be very high, demanding more resources from the machine. In addition, each computation/multiplication will require a great amount of memory and CPU usage in order to be performed due to the high dimensionality of the objects: the order of the computations will play a fundamental role in this case.

   As an example, if instead of multiplying Z^TX, and then calculate $(Z^TZ)^{-1}Z^TX$ via the Cholesky's decomposition we calculate $(Z^TZ)^{-1}Z^T$, we will end up with a lower performance (and higher memory usage) since we will generate larger matrices instead of reducing the dimensionality of the computations.

ii) A naive implementation of each of the two stages will lead us to an "out of memory" error without any doubts: calculations such as computing the explicit inverse matrices or not using the *crossprod*() function whenever we can, will lead us to very poor results and out of memory errors (no results will be obtained).

iii) In addition, assuming that we have access to a computer with a huge amount of memory, we can still have problems such as numerical imprecision due to inversions and the performance will be still a problem if the implementation is not well developed.

b) Sparse matrices: efficient code

In order to perform a full formal comparison, we developed three main approaches:

i) **Direct Two-stage approach**: Using the equations stated, we simply perform the algebraic operations both using a naive and a sparse implementation using the *spam* package.

ii) **Step-by-Step approach**: Taking advantage of the structure of the expressions, we perform a series of operations in order to keep the lightest elements in memory while performing the minimum number of arithmetic operations. In this case, we show how important is the order of the operations by comparing three specific implementations, two of them including the use of the *spam* package.

iii) **Cholesky's approach:** As with the previous problem, in this third implementation we perform a series of Cholesky's decomposition in order to avoid the explicit calculation of inverse matrices via the known *solve*() function by replacing it by a nested *backsolve*() calls.

   Note: After performing some tests, we realized that in the case of the *spam* package, we need to use a $forwardsolve()$ nested inside a *backsolve*() in order to obtain the same result as with two *backsolve*() calls (when no sparsity is exploited). This comes from the fact that the *spam* objects are treated in a different way in R and thus, adding the $transpose = TRUE$ flag does not have any effect on them.

Hence, we have the following code:

1. We start by generating random data based on the main dimensions of the matrices. Each matrix is generated in the classic and sparse version using the *spam* package. The dimensions and the sizes of each object are reported in order to have an initial impression of the advantage (in terms of memory usage) of the *spam* package when the elements are sparse.

```
# Sparse matrices and 2SLS
# Suppress warnings
options(warn = -1)

# Loading libraries
library(spam)
library(dplyr)
library(pryr)

# Matrices Dimensions
r = 1e6  # (60e6)
m = 55   # (630)
p = 50   # (600)
```

```
# X matrix
X = matrix(sample(c(0,1,5,2,6), size = p * r,
                  replace = TRUE,
                  prob = c(0.9,0.05,0.02,0.02,0.01)),
           nrow = r, ncol = p)
dim(X)

## [1] 1000000      50

object_size(X)

## 400 MB

# Convert to sparse
Xsparse = as.spam(X)
object_size(Xsparse)

## 64 MB
```

Based on the sizes of both objects, we can clearly understand the advantage of working with the *spam* package when the objects are sparse.

```
# Z matrix
Z = matrix(sample(c(0,1,5,2,6), size = m * r,
                  replace = TRUE,
                  prob = c(0.9,0.05,0.02,0.02,0.01)),
           nrow = r, ncol = m)
dim(Z)

## [1] 1000000      55

object_size(Z)
```

```
## 440 MB

# Convert to sparse
Zsparse = as.spam(Z)
object_size(Zsparse)

## 70 MB
```

Again, here we can see how we can obtain a very good rate of compression when the matrix is Sparse.

```
# y vector
y = as.integer(runif(r, 0, 100))
dim(y)

## NULL

object_size(y)

## 4 MB

# Convert to sparse
ysparse = as.spam(y)
object_size(ysparse)

## 15.9 MB
```

Notice that in this case, the y vector is larger (requires more memory) when we convert it to the spam class. This happens because it is not an exploitable sparse object and the conversion adds extra elements (such as pointers and indexes) to the object.

2. Using our first implementation of a direct calculation of both expressions, we use the following code:

```
# 2-stage LS calculations
# Original approach
X_hat = Z %*% solve(t(Z) %*% Z) %*% t(Z) %*% X
object_size(X_hat)

Beta_hat = solve(t(X_hat) %*% X_hat) %*% t(X_hat) %*% y
object_size(Beta_hat)

# Original approach SPARSE
X_hat = Zsparse %*% solve.spam(t(Zsparse) %*% Zsparse) %*% t.spam(Zsparse) %*% Xsparse
object_size(X_hat)

Beta_hat = solve(t.spam(X_hat) %*% X_hat) %*% t.spam(X_hat) %*% y
object_size(Beta_hat)
```

Based on the output of the previous code (not included here in order to be able to compile the file), we can easily check that the operations are not carried out by R since the total allocation of memory exceeds the amount of 12 GB of RAM (our computer has 12GB) and thus, all the code fails. The Same situation happens when using the *spam* package. Therefore, we should move to a different implementation.

3. In this case, three different implementations are coded. The main difference with the Naive implementation lies in the fact that operations are performed in an iterative way, taking into account the importance of the order of the operations as well as the size of each object kept in memory.

   The first implementation does not exploit the sparse objects and computes explicit inverse matrices for the computations. The code is the following:

```
# Auxiliary vector for recording sizes
Sizes1 = integer(7)
MSizes1 = integer(3)
MSizes1[1] = object_size(X)
MSizes1[2] = object_size(Z)
MSizes1[3] = object_size(y)

# Original approach improved 1
# Unused objects are removed every step
## First Stage
# (Z^{t} Z)^{-1}
R0 = solve(crossprod(Z))
Sizes1[1] = object_size(R0)
object_size(R0)

## 24.4 kB

# Z (Z^{t} Z)^{-1}
R01 = Z %*% R0
rm(R0)
object_size(R01)

## 440 MB

Sizes1[2] = object_size(R01)

# Z^{t} X
R02 = t(Z) %*% X
object_size(R02)

## 22.2 kB

Sizes1[3] = object_size(R02)

# Final value X_hat
X_hat = R01 %*% R02
rm(R01)
rm(R02)
object_size(X_hat)

## 400 MB

Sizes1[4] = object_size(X_hat)
```

   At this point, we have finished the first stage computations without memory issues as with the Naive approach. Now, we solve the second stage:

```
## Second stage
# (X^{t} X) ^{-1}
R1 = solve(crossprod(X_hat))
object_size(R1)

## 20.2 kB

Sizes1[5] = object_size(R1)

# X_hat^{t} y
R12 = crossprod(X_hat, y)
object_size(R12)

## 600 B

Sizes1[6] = object_size(R12)

# Beta_hat final value
Beta_hat = R1 %*% R12
rm(R1)
rm(R12)
object_size(Beta_hat)

## 600 B

Sizes1[7] = object_size(Beta_hat)
```

Finally, we display the $\hat{\beta}$ vector for visualizing purposes.

```
# Display the vector (sample)
Beta_hat[1:5]

## [1]  11.762885  -2.617291  -8.527377  25.480514 -14.639191

# Sizes statistics
cat("Minimum object size (computed): \t", min(Sizes1), "[bytes]")

## Minimum object size (computed):   600 [bytes]

cat("Maximum object size (computed): \t ", max(Sizes1), "[bytes]")

## Maximum object size (computed):    440000200 [bytes]

cat("Total memory used: \t\t\t", sum(Sizes1), "[bytes]")

## Total memory used:   840068400 [bytes]

cat("Total memory used including matrices: \t", sum(Sizes1) + sum(MSizes1), "[bytes]")

## Total memory used including matrices:   1684068840 [bytes]
```

From the previous results, we can see the summary statistics in terms of memory usage of the entire function. Now, we will compare this previous implementation with the use of the *spam* package.

```
# Auxiliary vector for recording sizes
Sizes2 = integer(7)
MSizes2 = integer(3)
MSizes2[1] = object_size(Xsparse)
MSizes2[2] = object_size(Zsparse)
MSizes2[3] = object_size(ysparse)

# Original approach improved 1: SPARSE
## First Stage
# (Z^{t} Z)^{-1}
R0 = solve(crossprod.spam(Zsparse))
Sizes2[1] = object_size(R0)
object_size(R0)

## 24.4 kB

# Z (Z^{t} Z)^{-1}
R01 = Zsparse %*% R0
rm(R0)
Sizes2[2] = object_size(R01)
object_size(R01)

## 440 MB

# Z^{t} X
R02 = t.spam(Zsparse) %*% Xsparse
Sizes2[3] = object_size(R02)
object_size(R02)

## 34.2 kB

# Final value X_hat
X_hat = R01 %*% R02
rm(R01)
rm(R02)
object_size(X_hat)

## 602 MB

Sizes2[4] = object_size(X_hat)
```

As before, we do not have problems after calculating the first stage expressions and we proceed to the second stage of the estimation.

```
## Second stage
# (X^{t} X) ^{-1}
R1 = solve.spam(crossprod.spam(X_hat))
object_size(R1)
```

```
## 20.2 kB

Sizes2[5] = object_size(R1)

# X_hat^{t} y
R12 = crossprod.spam(X_hat, ysparse)
object_size(R12)

## 1.82 kB

Sizes2[6] = object_size(R12)

# Beta_hat final value
Beta_hat = R1 %*% R12
rm(R1)
rm(R12)
object_size(Beta_hat)

## 1.82 kB

Sizes2[7] = object_size(Beta_hat)
```

Finally, we compute and display the final estimation. Based on the results, we can easily see that the results are identical by both methods.

```
# Display the vector (sample)
Beta_hat[1:5]

## [1]  11.762885  -2.617291  -8.527377  25.480514 -14.639191

# Sizes statistics
cat("Minimum object size (computed): \t", min(Sizes2), "[bytes]")

## Minimum object size (computed):   1824 [bytes]

cat("Maximum object size (computed): \t ", max(Sizes2), "[bytes]")

## Maximum object size (computed):    602187224 [bytes]

cat("Total memory used: \t\t\t", sum(Sizes2), "[bytes]")

## Total memory used:   1042269912 [bytes]

cat("Total memory used including matrices: \t", sum(Sizes2) + sum(MSizes2), "[bytes]")

## Total memory used including matrices:   1192166048 [bytes]
```

Looking at the total amount of memory needed for performing the operations, we can clearly see that the sparse approach is better (lower amount of memory is needed). At the end of this section, we will compare all the implementations in order to conclude which approach is the most efficient for the current problem.

Finally, the third implementation of the current approach is exactly the same as the previous one but with a specific detail that impacts its performance in a very significant way: In this case, we first multiply $(Z^tZ)^{-1}$ by Z^tX instead of performing $Z(Z^tZ)^{-1}$. Clearly, we are saving a fair amount of operations since in the first one we will end up multiplying a scalar with a matrix instead of two matrices like in the second approach.

Therefore, we have the following results:

```
# Auxiliary vector for recording sizes
Sizes22 = integer(7)

# Original approach improved 1.2: SPARSE with different order
## First Stage
# (Z^{t} Z)^{-1}
R0 = solve(crossprod.spam(Zsparse))
Sizes22[1] = object_size(R0)
object_size(R0)

## 24.4 kB

# Z^{t} X
R01 = t.spam(Zsparse) %*% Xsparse
Sizes22[2] = object_size(R01)
object_size(R01)

## 34.2 kB

# (Z^{t} Z)^{-1} Z^{t} X  (Order change step)
R02 = R0 %*% R01
rm(R0)
rm(R01)
Sizes22[3] = object_size(R02)
object_size(R02)

## 34.2 kB

# Final value X_hat
X_hat = Z %*% R02
rm(R02)
object_size(X_hat)

## 602 MB

Sizes22[4] = object_size(X_hat)
```

```
## Second stage
# (X^{t} X) ^{-1}
R1 = solve.spam(crossprod.spam(X_hat))
object_size(R1)

## 20.2 kB
```

```r
Sizes22[5] = object_size(R1)

# X_hat^{t} y
R12 = crossprod.spam(X_hat, ysparse)
object_size(R12)

## 1.82 kB

Sizes22[6] = object_size(R12)

# Beta_hat final value
Beta_hat = R1 %*% R12
rm(R1)
rm(R12)
object_size(Beta_hat)

## 1.82 kB

Sizes22[7] = object_size(Beta_hat)
```

```r
# Display the vector (sample)
Beta_hat[1:5]

## [1]  11.762885  -2.617291  -8.527377  25.480514 -14.639191

# Sizes statistics
cat("Minimum object size (computed): \t", min(Sizes22), "[bytes]")

## Minimum object size (computed):   1824 [bytes]

cat("Maximum object size (computed): \t ", max(Sizes22), "[bytes]")

## Maximum object size (computed):    602187224 [bytes]

cat("Total memory used: \t\t\t", sum(Sizes22), "[bytes]")

## Total memory used:   602303952 [bytes]

cat("Total memory used including matrices: \t", sum(Sizes22) + sum(MSizes2), "[bytes]")

## Total memory used including matrices:   752200088 [bytes]
```

Looking at the total memory usage, we can clearly see the significant impact of the minor change in the order of the operations. In this case, the total amount of memory is about 60% of the previous implementation.

4. In our final implementation, we take advantage of the Cholesky's decomposition for avoiding the explicit inverse calculations, using the already discussed *backsolve*() and $forwardsolve()$ functions.

   The first version (not using the *spam* package) is as follows:

```r
# Auxiliary vector for recording sizes
Sizes3 = integer(5)

# Original improved 2
## First stage
# Cholesky decomposition of Z^{t} Z
U1 = chol(crossprod(Z))
object_size(U1)

## 24.4 kB

Sizes3[1] = object_size(U1)

# Compute (Z^{t}Z)^{-1} Z^{t}X via Cholesky backsolve
R1 = backsolve(U1, backsolve(U1, t(Z) %*% X, transpose = TRUE))
rm(U1)
object_size(R1)

## 22.2 kB

Sizes3[2] = object_size(R1)

# Compute X_hat
X_hat = Z %*% R1
rm(R1)
object_size(X_hat)

## 400 MB

Sizes3[3] = object_size(X_hat)
```

Important is to note the size of the objects, remembering that since the U matrices are sparse, we can obtain a very good compression rate when using the *spam* package.

```r
## Second stage
# Cholesky decomposition of X_hat^{t} X_hat
U2 = chol(crossprod(X_hat))
object_size(U2)

## 20.2 kB

Sizes3[4] = object_size(U2)

# Compute Beta_hat via Cholesky backsolve
Beta_hat = backsolve(U2, backsolve(U2, crossprod(X_hat, y), transpose = TRUE))
rm(U2)
object_size(Beta_hat)

## 600 B

Sizes3[5] = object_size(Beta_hat)
```

```
# Display the vector (sample)
Beta_hat[1:5]

## [1]  11.762885  -2.617291  -8.527377  25.480514 -14.639191

# Sizes statistics
cat("Minimum object size (computed): \t", min(Sizes3), "[bytes]")

## Minimum object size (computed):   600 [bytes]

cat("Maximum object size (computed): \t ", max(Sizes3), "[bytes]")

## Maximum object size (computed):    400000200 [bytes]

cat("Total memory used: \t\t\t", sum(Sizes3), "[bytes]")

## Total memory used:   400067600 [bytes]

cat("Total memory used including matrices: \t", sum(Sizes3) + sum(MSizes1), "[bytes]")

## Total memory used including matrices:   1244068040 [bytes]
```

Finally, we implement the Cholesky's decomposition method while working with sparse matrices by using the *spam* package:

```
# Auxiliary vector for recording sizes
Sizes4 = integer(5)

# Original improved 2: SPARSE
## First stage
# Cholesky decomposition of Z^{t} Z
U1 = chol.spam(crossprod.spam(Zsparse), pivot = FALSE)
object_size(U1)

## 15.6 kB

Sizes4[1] = object_size(U1)

# Compute (Z^{t}Z)^{-1} Z^{t}X via Cholesky backsolve
# and forwardsolve (spam package difference)
R1 = backsolve.spam(U1, forwardsolve.spam(U1,
                                          t.spam(Zsparse) %*% Xsparse, transpose = T))
rm(U1)
object_size(R1)

## 22.2 kB

Sizes4[2] = object_size(R1)

# Compute X_hat
X_hat = as.spam(Zsparse %*% R1)
rm(R1)
object_size(X_hat)
```

```
## 602 MB

Sizes4[3] = object_size(X_hat)
```

```
## Second stage
# Cholesky decomposition of X_hat^{t} X_hat
U2 = chol.spam(crossprod.spam(X_hat), pivot = FALSE)
object_size(U2)

## 13.4 kB

Sizes4[4] = object_size(U2)

# Compute Beta_hat via Cholesky backsolve and forwardsolve (spam package difference)
Beta_hat = backsolve.spam(U2, forwardsolve.spam(U2,
                                                crossprod.spam(X_hat, ysparse),
                                                transpose = T)
                          )
rm(U2)
object_size(Beta_hat)

## 440 B

Sizes4[5] = object_size(Beta_hat)
```

```
# Display the vector (sample)
Beta_hat[1:5]

## [1]  11.762885  -2.617291  -8.527377  25.480514 -14.639191

# Sizes statistics
cat("Minimum object size (computed): \t", min(Sizes4), "[bytes]")

## Minimum object size (computed):   440 [bytes]

cat("Maximum object size (computed): \t ", max(Sizes4), "[bytes]")

## Maximum object size (computed):    602187224 [bytes]

cat("Total memory used: \t\t\t", sum(Sizes4), "[bytes]")

## Total memory used:   602238928 [bytes]

cat("Total memory used including matrices: \t", sum(Sizes4) + sum(MSizes2), "[bytes]")

## Total memory used including matrices:   752135064 [bytes]
```

In order to compare the memory usage across all the implementations, we create a DataFrame containing all the relevant information and then we display it to the command line:

```r
# Final comparison
FComp = data.frame("min" = c(min(Sizes1), min(Sizes2), min(Sizes22),
                             min(Sizes3), min(Sizes4)),
                   "max" = c(max(Sizes1), max(Sizes2), max(Sizes22),
                             max(Sizes3), max(Sizes4)),
                   "Total" = c(sum(Sizes1), sum(Sizes2), sum(Sizes22),
                               sum(Sizes3), sum(Sizes4)),
                   "Total_with_Matrices" = c(sum(Sizes1) + sum(MSizes1),
                                             sum(Sizes2) + sum(MSizes2),
                                             sum(Sizes22) + sum(MSizes2),
                                             sum(Sizes3) + sum(MSizes1),
                                             sum(Sizes4) + sum(MSizes2)),
                   row.names = c("Method 1", "Method 1 SP", "Method 1 SP+OR",
                                 "Method 2", "Method 2 SP")
                   )

# Display final comparison table
FComp
```

```
##                  min       max      Total Total_with_Matrices
## Method 1         600 440000200  840068400          1684068840
## Method 1 SP     1824 602187224 1042269912          1192166048
## Method 1 SP+OR  1824 602187224  602303952           752200088
## Method 2         600 400000200  400067600          1244068040
## Method 2 SP      440 602187224  602238928           752135064
```

Based on the final results, we can easily check that both the first Sparse implementation approach with the special order of operations and the second Sparse implementation that takes advantage of the Cholesky's decomposition uses a very similar amount of total memory during the entire execution (including the matrices), reaching the highest place in terms of performance (solving time and memory usage). On the other hand, we can clearly see that the worst implementation consists of the first Naive approach without using the *spam* package, followed by the second approach without sparse matrices and then the first sparse approach that is not taking care of the order of operations.

Therefore, we have (deeply) described how we would be able to implement the calculations of a two-stage least squares coefficient estimation by taking advantage of all the relevant techniques we have studied during the linear algebra unit during the course.

Problem 6: Empirical study eigendecomposition

In this problem, we would like to empirically explore the condition number of the eigendecomposition and how it affects the computations such as determining the eigenvalues and properties of the original matrix, checking if it is symmetric and/or positive semi-definite. Following the instructions of the statement of the problem, we have the following steps:

1. We create an arbitrary matrix $Z \in R^{n \times n}$ using a random function

```
#PS6 Eigendecomposition and condition value
# Supress warning
options(warn = -1)

# Loading libraries
library(matrixcalc)

# Dimension
n = 100

# Random matrix Z
Z = matrix(runif(n * n, 0, 100), n, n)
Z = matrix(runif(n * n), n, n)
```

2. A symmetric matrix A is generated from $Z^T Z$. Then, we extract the eigenvectors of this matrix using the $eigen()$ function and taking out the eigenvectors object, obtaining the matrix Γ

```
# A = Z^{T}Z
A = crossprod(Z)

# Compute Gamma using eigen function
Gamma = eigen(A)$vectors
```

3. We define a $Tester()$ function that takes the desired eigenvalues, dimension n, and the eigenvectors Γ matrix as the main inputs in order to perform all the operations needed for computing the matrix $A_2 = \Gamma \Lambda \Gamma^T$, its condition number, check if the values obtained via the call of the $eigen()$ function match the given λ values, and compute the square error of this computation:

```
# Function: Tester
Tester <- function(evalues, n, Gamma){
  # Generate the diagonal matrix with eigenvalues
  Lambda <- matrix(0, ncol = n, nrow = n)
  diag(Lambda) <- evalues

  # Generate the matrix A by explicit multiplication
  NewA = Gamma %*% Lambda %*% t(Gamma)

  # Display eigenvalues by function and reference
  cat("Eigenvalues by engen()[top4]:", eigen(NewA)$values[1:4], "\n")
```

```
  cat("Actual eigenvalues[top4]:    ", sort(evalues, decreasing = TRUE)[1:4], "\n")

  # Check condition number
  Condition = max(evalues) / min(evalues)
  cat("Condition number:", Condition, "\n")

  # Calculate the square error (between eigenvalues)
  Error = sum((sort(evalues, decreasing = TRUE) - eigen(NewA)$values)^2)
  cat("Square error:", Error, "\n")

  # Check if positive semi-definite (uncomment if wanted, from package matrixcalc)
  #cat("Positive semi-definite:", is.positive.semi.definite(NewA,tol = 1e-3))

  # Return the square error
  return(Error)
}
```

4. Then, a series of different Λ diagonal matrices are generated using the previous function by creating a series of different eigenvalue arrays. We start with a series of sequences where we vary the distance between the first (lower) eigenvalue and the largest one, in order to increase the condition number and check its impact in our computations.

```
# Square error and Condition number vector
SE = integer(17)
CN = integer(17)

##Sequences analysis
# First Test: all eigenvalues equal to 1
EigL1 <- rep(1, n)
SE[1] = Tester(EigL1, n, Gamma)

## Eigenvalues by engen()[top4]: 1 1 1 1
## Actual eigenvalues[top4]:     1 1 1 1
## Condition number: 1
## Square error: 1.697192e-26

CN[1] = max(EigL1) / min(EigL1)

# Second Test: sequence up to n by 1
EigL2 <- c(1:n)
SE[2] = Tester(EigL2, n, Gamma)

## Eigenvalues by engen()[top4]: 100 99 98 97
## Actual eigenvalues[top4]:     100 99 98 97
## Condition number: 100
## Square error: 1.052151e-25

CN[2] = max(EigL2) / min(EigL2)

# Third test: 1 to 100 total n elements
EigL3 <- seq(1, 100, length = n)
SE[3] = Tester(EigL3, n, Gamma)
```

```
## Eigenvalues by engen()[top4]: 100 99 98 97
## Actual eigenvalues[top4]:     100 99 98 97
## Condition number: 100
## Square error: 1.052151e-25

CN[3] = max(EigL3) / min(EigL3)

# Fourth test: 1 to 1000
EigL4 <- seq(1, 1000, length = n)
SE[4] = Tester(EigL4, n, Gamma)

## Eigenvalues by engen()[top4]: 1000 989.9091 979.8182 969.7273
## Actual eigenvalues[top4]:     1000 989.9091 979.8182 969.7273
## Condition number: 1000
## Square error: 1.049229e-23

CN[4] = max(EigL4) / min(EigL4)

# Fifth test: 1 to 10000
EigL5 <- seq(1, 10000, length = n)
SE[5] = Tester(EigL5, n, Gamma)

## Eigenvalues by engen()[top4]: 10000 9899 9798 9697
## Actual eigenvalues[top4]:     10000 9899 9798 9697
## Condition number: 10000
## Square error: 1.020741e-21

CN[5] = max(EigL5) / min(EigL5)
```

Based on the outputs, we can clearly see how the increase of the condition number is positively correlated with the square error obtained when computing the difference between the eigenvalues calculated by the *eigen*() function and the ones given as inputs.

```
# Sixth test: 1 to 1e5
EigL6 <- seq(1, 1e5, length = n)
SE[6] = Tester(EigL6, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+05 98989.91 97979.82 96969.73
## Actual eigenvalues[top4]:     1e+05 98989.91 97979.82 96969.73
## Condition number: 1e+05
## Square error: 9.599913e-20

CN[6] = max(EigL6) / min(EigL6)

# Seventh test: 1 to 1e6
EigL7 <- seq(1, 1e6, length = n)
SE[7] = Tester(EigL7, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+06 989899 979798 969697
## Actual eigenvalues[top4]:     1e+06 989899 979798 969697
## Condition number: 1e+06
## Square error: 1.156031e-17
```

```
CN[7] = max(EigL7) / min(EigL7)

# Eight test: 1 to 1e16
EigL8 <- seq(1, 1e16, length = n)
SE[8] = Tester(EigL8, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Actual eigenvalues[top4]:     1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Condition number: 1e+16
## Square error: 1267.544

CN[8] = max(EigL8) / min(EigL8)

# Ninth test: 0.001 to 1e16
EigL9 <- seq(1e-3, 1e16, length = n)
SE[9] = Tester(EigL9, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Actual eigenvalues[top4]:     1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Condition number: 1e+19
## Square error: 1039.134

CN[9] = max(EigL9) / min(EigL9)

# Tenth test: 1e-6 to 1e16
EigL10 <- seq(1e-6, 1e16, length = n)
SE[10] = Tester(EigL10, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Actual eigenvalues[top4]:     1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Condition number: 1e+22
## Square error: 1039.138

CN[10] = max(EigL10) / min(EigL10)
```

As before, we can notice how the magnitude of the error is clearly increasing with the value of the condition number, obtaining very large errors at this point.

```
# Eleventh test: 1e-12 to 1e16
EigL11 <- seq(1e-12, 1e16, length = n)
SE[11] = Tester(EigL11, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Actual eigenvalues[top4]:     1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Condition number: 1e+28
## Square error: 1039.138

CN[11] = max(EigL11) / min(EigL11)

# Twelth test: 1e-16 to 1e16
EigL12 <- seq(1e-16, 1e16, length = n)
SE[12] = Tester(EigL12, n, Gamma)
```

```
## Eigenvalues by engen()[top4]: 1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Actual eigenvalues[top4]:     1e+16 9.89899e+15 9.79798e+15 9.69697e+15
## Condition number: 1e+32
## Square error: 1039.138

CN[12] = max(EigL12) / min(EigL12)

# Thirteenth test: 1e-16 to 1e18
EigL13 <- seq(1e-16, 1e18, length = n)
SE[13] = Tester(EigL13, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+18 9.89899e+17 9.79798e+17 9.69697e+17
## Actual eigenvalues[top4]:     1e+18 9.89899e+17 9.79798e+17 9.69697e+17
## Condition number: 1e+34
## Square error: 12043124

CN[13] = max(EigL13) / min(EigL13)

# Fourteenth test: 1e-16 to 1e24
EigL14 <- seq(1e-16, 1e24, length = n)
SE[14] = Tester(EigL14, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+24 9.89899e+23 9.79798e+23 9.69697e+23
## Actual eigenvalues[top4]:     1e+24 9.89899e+23 9.79798e+23 9.69697e+23
## Condition number: 1e+40
## Square error: 8.554926e+18

CN[14] = max(EigL14) / min(EigL14)

# Fifteenth test: 1e-16 to 1e32
EigL15 <- seq(1e-16, 1e32, length = n)
SE[15] = Tester(EigL15, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+32 9.89899e+31 9.79798e+31 9.69697e+31
## Actual eigenvalues[top4]:     1e+32 9.89899e+31 9.79798e+31 9.69697e+31
## Condition number: 1e+48
## Square error: 1.195093e+35

CN[15] = max(EigL15) / min(EigL15)

# Sixteenth test: 1e-32 to 1e32
EigL16 <- seq(1e-32, 1e32, length = n)
SE[16] = Tester(EigL16, n, Gamma)

## Eigenvalues by engen()[top4]: 1e+32 9.89899e+31 9.79798e+31 9.69697e+31
## Actual eigenvalues[top4]:     1e+32 9.89899e+31 9.79798e+31 9.69697e+31
## Condition number: 1e+64
## Square error: 1.195093e+35

CN[16] = max(EigL16) / min(EigL16)

# Seventeenth test: 1e-64 to 1e64
EigL17 <- seq(1e-64, 1e64, length = n)
SE[17] = Tester(EigL17, n, Gamma)
```

```
## Eigenvalues by engen()[top4]: 1e+64 9.89899e+63 9.79798e+63 9.69697e+63
## Actual eigenvalues[top4]:     1e+64 9.89899e+63 9.79798e+63 9.69697e+63
## Condition number: 1e+128
## Square error: 1.675032e+99

CN[17] = max(EigL17) / min(EigL17)
```

At this point, the relation is clear: the magnitude of the square error is proportional to the value of the condition number.

```
# Display results
print("Square errors:")

## [1] "Square errors:"

SE[1:5]

## [1] 1.697192e-26 1.052151e-25 1.052151e-25 1.049229e-23 1.020741e-21

SE[6:10]

## [1] 9.599913e-20 1.156031e-17 1.267544e+03 1.039134e+03 1.039138e+03

SE[11:15]

## [1] 1.039138e+03 1.039138e+03 1.204312e+07 8.554926e+18 1.195093e+35

SE[16:17]

## [1] 1.195093e+35 1.675032e+99

print("Condition number:")

## [1] "Condition number:"

CN[1:11]

##  [1] 1e+00 1e+02 1e+02 1e+03 1e+04 1e+05 1e+06 1e+16 1e+19 1e+22 1e+28

CN[12:17]

## [1]  1e+32  1e+34  1e+40  1e+48  1e+64 1e+128
```

Note: For all the previous experiments, we checked if the A_2 matrix was positive semi-definite using the *matrixcalc* package, however, we always obtained a $FALSE$ value when testing them indicating that the matrix was not symmetric. We tested it with different tolerance levels and we got the same answer. Therefore, we were not able to perform a deeper empirical study in this field. It is possible that our implementation was not exactly what was expected in the statement of the problem or our code contains an error that we were not able to detect.

For completeness, we include extra experiments with random numbers instead of sequences for generating the different eigenvalue vectors:

```r
## No sequences
# First test: 0 to 1
EigL1 <- runif(n, 0, 1)
Tester(EigL1, n, Gamma)

## Eigenvalues by engen()[top4]: 0.9879349 0.9644718 0.9530653 0.948584
## Actual eigenvalues[top4]:     0.9879349 0.9644718 0.9530653 0.948584
## Condition number: 428.6754
## Square error: 1.147697e-29
## [1] 1.147697e-29

# Second test: 0 to 10
EigL2 <- runif(n, 0, 10)
Tester(EigL2, n, Gamma)

## Eigenvalues by engen()[top4]: 9.991096 9.954444 9.907156 9.819712
## Actual eigenvalues[top4]:     9.991096 9.954444 9.907156 9.819712
## Condition number: 2361.694
## Square error: 1.551529e-27
## [1] 1.551529e-27

# Third test: 0 to 100
EigL3 <- runif(n, 0, 100)
Tester(EigL3, n, Gamma)

## Eigenvalues by engen()[top4]: 98.73587 98.7297 96.62922 95.24275
## Actual eigenvalues[top4]:     98.73587 98.7297 96.62922 95.24275
## Condition number: 167.3606
## Square error: 1.62479e-25
## [1] 1.62479e-25

# Fourth test: 04 to 1e4
EigL4 <- runif(n, 0, 1e4)
Tester(EigL4, n, Gamma)

## Eigenvalues by engen()[top4]: 9631.137 9191.88 9095.66 8803.892
## Actual eigenvalues[top4]:     9631.137 9191.88 9095.66 8803.892
## Condition number: 160.0006
## Square error: 1.453147e-21
## [1] 1.453147e-21

# Fifth test: 0 to 1e5
EigL5 <- runif(n, 0, 1e5)
Tester(EigL5, n, Gamma)

## Eigenvalues by engen()[top4]: 98560.41 98367.37 97737.7 96121.01
## Actual eigenvalues[top4]:     98560.41 98367.37 97737.7 96121.01
## Condition number: 51.01502
## Square error: 9.988371e-20
## [1] 9.988371e-20

# Sixth test: 0 to 1e6
EigL6 <- runif(n, 0, 1e6)
Tester(EigL6, n, Gamma)
```

```
## Eigenvalues by engen()[top4]: 989961.1 963046 961406 945096.9
## Actual eigenvalues[top4]:     989961.1 963046 961406 945096.9
## Condition number: 69.38904
## Square error: 1.335432e-17
## [1] 1.335432e-17

# Seventh test: 0 to 1e6
EigL7 <- runif(n, 0, 1e8)
Tester(EigL7, n, Gamma)

## Eigenvalues by engen()[top4]: 99468938 99170251 98488267 98168958
## Actual eigenvalues[top4]:     99468938 99170251 98488267 98168958
## Condition number: 65.21245
## Square error: 1.361862e-13
## [1] 1.361862e-13

# Eight test: 0 to 1e6
EigL8 <- runif(n, 0, 1e10)
Tester(EigL8, n, Gamma)

## Eigenvalues by engen()[top4]: 9861587165 9854531572 9758649867 9701726411
## Actual eigenvalues[top4]:     9861587165 9854531572 9758649867 9701726411
## Condition number: 118.1615
## Square error: 1.301087e-09
## [1] 1.301087e-09

# Ninth test: 0 to 1e6
EigL9 <- runif(n, 0, 1e12)
Tester(EigL9, n, Gamma)

## Eigenvalues by engen()[top4]: 993757676566 992208692944 984165197238 969178225845
## Actual eigenvalues[top4]:     993757676566 992208692944 984165197238 969178225845
## Condition number: 11582.55
## Square error: 1.436897e-05
## [1] 1.436897e-05

# Tenth test: 0 to 1e6
EigL10 <- runif(n, 0, 1e16)
Tester(EigL10, n, Gamma)

## Eigenvalues by engen()[top4]: 9.973097e+15 9.912241e+15 9.882936e+15 9.792566e+15
## Actual eigenvalues[top4]:     9.973097e+15 9.912241e+15 9.882936e+15 9.792566e+15
## Condition number: 99.9168
## Square error: 986.6711
## [1] 986.6711

# Eleventh test: 0 to 1e24
EigL11 <- runif(n, 0, 1e24)
Tester(EigL11, n, Gamma)

## Eigenvalues by engen()[top4]: 9.971233e+23 9.792281e+23 9.691045e+23 9.592693e+23
## Actual eigenvalues[top4]:     9.971233e+23 9.792281e+23 9.691045e+23 9.592693e+23
## Condition number: 582.7653
## Square error: 9.708536e+18
## [1] 9.708536e+18
```

```r
# Tenth test: 0 to 1e32
EigL12 <- runif(n, 0, 1e32)
Tester(EigL12, n, Gamma)

## Eigenvalues by engen()[top4]: 9.87822e+31 9.806384e+31 9.70587e+31 9.569755e+31
## Actual eigenvalues[top4]:     9.87822e+31 9.806384e+31 9.70587e+31 9.569755e+31
## Condition number: 98.87261
## Square error: 8.225032e+34
## [1] 8.225032e+34

# Thirteenth test: 0 to 1e64
EigL13 <- runif(n, 0, 1e64)
Tester(EigL13, n, Gamma)

## Eigenvalues by engen()[top4]: 9.983418e+63 9.969597e+63 9.930212e+63 9.914198e+63
## Actual eigenvalues[top4]:     9.983418e+63 9.969597e+63 9.930212e+63 9.914198e+63
## Condition number: 51.54087
## Square error: 1.059655e+99
## [1] 1.059655e+99

# Fourteenth test: 0 to 1e128
EigL14 <- runif(n, 0, 1e128)
Tester(EigL14, n, Gamma)

## Eigenvalues by engen()[top4]: 9.915761e+127 9.661892e+127 9.638349e+127 9.337359e+127
## Actual eigenvalues[top4]:     9.915761e+127 9.661892e+127 9.638349e+127 9.337359e+127
## Condition number: 4637.757
## Square error: 1.664741e+227
## [1] 1.664741e+227

# Fifteenth test: 0 to 1e256
EigL15 <- runif(n, 0, 1e256)
Tester(EigL15, n, Gamma)

## Eigenvalues by engen()[top4]: 9.86248e+255 9.64685e+255 9.578395e+255 9.555497e+255
## Actual eigenvalues[top4]:     9.86248e+255 9.64685e+255 9.578395e+255 9.555497e+255
## Condition number: 635.5122
## Square error: Inf
## [1] Inf
```

Looking at the results, we can clearly check how the Square error is increased in a very significant way every time the condition number is increased and also due to the numerical imprecision produced when we perform arithmetical calculations with numbers of very different magnitude. As an example, we can see that the last test obtains an infinite square error (overflow) while its condition number is not as large as in the previous test. The explanation is clear: since we are dealing with huge numbers, any small error will give us an overflow (inf) value.