This memory corresponds to the evaluation of the results obtained in the first project of Numerical Linear Algebra, as well as, the solutions of the theoretical questions made in the document of the explanation of the project, hence, the memory will have two big parts. Let us start solving the theoretical questions.

1 Theoretical questions

T1. *Show that the predictor steps reduces to solve a linear system with matrix* M_{KKT}

Proof. The predictor step corresponds to solve $F(z) = 0$ by Newton's method, where $F : \mathbb{R}^N \to \mathbb{R}^N$ is a function defined as

$$F(z) = F(x, \gamma, \lambda, s) = (Gx + g - A\gamma - C\lambda, b - A^T x, s + d - C^T x, s\lambda) \tag{1.1}$$

Let us see how Newton's method works for a function of multiple variables. Assume that $\bar{z}$ is a zero of F, i.e., $F(\bar{z}) = 0$. Doing Taylor approximation in $z^{(k)}$ near $\bar{z}$, we have

$$0 = F(\bar{z}) \approx F(z^{(k)}) + J_F(z^{(k)})(\bar{z} - z^{(k)}).$$

Solving the system $0 = F(z^{(k)}) + J_F(z^{(k)})(z - z^{(k)})$ in z, we obtain $z^{(k+1)}$, a new approximation of $\bar{z}$. Calling $\delta_z = z^{(k+1)} - z^{(k)}$, the predictor step corresponds to solve the following linear system in δ_z

$$J_F(z^{(k)})\delta_z = -F(z^{(k)}) \tag{1.2}$$

To finish the proof, we have to see that the Jacobian of F is exactly the matrix M_{KKT}. Notice that the Jacobian matrix is given by

$$J_F = \begin{pmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial \gamma} & \frac{\partial F_1}{\partial \lambda} & \frac{\partial F_1}{\partial s} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial \gamma} & \frac{\partial F_2}{\partial \lambda} & \frac{\partial F_2}{\partial s} \\ \frac{\partial F_3}{\partial x} & \frac{\partial F_3}{\partial \gamma} & \frac{\partial F_3}{\partial \lambda} & \frac{\partial F_3}{\partial s} \\ \frac{\partial F_4}{\partial x} & \frac{\partial F_4}{\partial \gamma} & \frac{\partial F_4}{\partial \lambda} & \frac{\partial F_4}{\partial s} \end{pmatrix} = \begin{pmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix}$$

Therefore, the system (1.2) becomes to $M_{KKT}\delta_z = -F(z^{(k)})$. □

T2. *Explain the previous derivations of the different strategies and justify under which assumptions they can be applied.*

Proof. In this case, we are assuming $A = 0$, hence, the matrix M_{KKT} is given by

$$M_{KKT} = \begin{pmatrix} G & -C & 0 \\ -C^T & 0 & I \\ 0 & S & \Lambda \end{pmatrix}$$

Calling $F(z_0) = (r_1, r_2, r_3)$, the linear system that we have to solve is the following

$$\begin{pmatrix} G & -C & 0 \\ -C^T & 0 & I \\ 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\lambda \\ \delta_s \end{pmatrix} = \begin{pmatrix} -r_1 \\ -r_2 \\ -r_3 \end{pmatrix}$$

As an equation system, this becomes to

$$\begin{cases} G\delta_x - C\delta_\lambda = -r_1 \\ -C^T\delta_x + \delta_s = -r_2 \\ S\delta_\lambda + \Lambda\delta_s = -r_3 \end{cases}$$

Let us discuss the **strategy 1**. First, we isolate δ_s from the third row of the system, that is $\delta_s = \Lambda^{-1}(-r_3 - S\delta_\lambda)$. Then we substitute it into the second row. This lead to the system

$$\begin{cases} G\delta_x - C\delta_\lambda = -r_1 \\ -C^T\delta_x + \Lambda^{-1}(-r_3 - S\delta_\lambda) = -r_2 \end{cases} \iff \begin{cases} G\delta_x - C\delta_\lambda = -r_1 \\ -C^T\delta_x - \Lambda^{-1}S\delta_\lambda = -r_2 + \Lambda^{-1}r_3 \end{cases}$$

In the matrix representation form,

$$\begin{pmatrix} G & -C \\ -C^T & -\Lambda^{-1}S \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\lambda \end{pmatrix} = \begin{pmatrix} -r_1 \\ -r_2 + \Lambda^{-1}r_3 \end{pmatrix}$$

Notice that to do this argument, we need Λ to be a non singular matrix. Since $\Lambda = \text{Diag}(\lambda)$, Λ is a non singular matrix if and only if $\lambda_i \neq 0 \ \forall i$. The strategy 1 invite us to use LDL^T factorization to the reduced matrix M_{KKT} in order to solve the system. We need the reduced matrix M_{KKT} to be symmetric, i.e., calling this matrix A, it has to be fulfilled that $A = A^T$. This is clearly true, since G is a symmetric matrix and and $-\Lambda^{-1}S$ is a diagonal block[1].

Let us continue discussing **strategy 2**. In this case, we isolate δ_s from the second row of the system. That is $\delta_s = -r_2 + C^T\delta_x$ and then, we substitute to the third row to get $S\delta_\lambda + \Lambda(-r_2 + C^T\delta_x) = -r_3$, i.e., $\delta_\lambda = S^{-1}(-r_3 + \Lambda r_2) - S^{-1}\Lambda C^T\delta_x$. Finally, we substitute into the first row to obtain the linear system $\hat{G}\delta_x = -r_1 - \hat{r}$, where $\hat{G} = G + CS^{-1}\Lambda C^T$ and $\hat{r} = -CS^{-1}(-r_3 + \Lambda r_2)$. Notice that to do this argument, we need S to be a non singular matrix. Since $S = \text{Diag}(s)$, S is a non singular matrix if and only if $s_i \neq 0 \ \forall i$. The strategy 2 invite us to us Cholesky factorization to the matrix $\hat{G}$ in order to solve the system. We need the matrix $\hat{G}$ to be symmetric and positive definite. Let us prove first that $\hat{G}$ is symmetric.

$$\hat{G}^T = (G + CS^{-1}\Lambda C^T)^T = G^T + (C^T)^T\Lambda^T S^{-T}C^T = G + C\Lambda S^{-1}C^T = G + CS^{-1}\Lambda C^T = \hat{G}$$

The last two equality holds because G is symmetric and Λ and S are diagonal matrices. Now, let us proof that $\hat{G}$ is positive definite. We have to check that for all $x \neq 0$, $x^T\hat{G}x > 0$, i.e., $x^T(G + CS^{-1}\Lambda C^T)x > 0$. Since G is positive definite, we only have to check that for all $x \neq 0$, $x^T(CS^{-1}\Lambda C^T)x > 0$. Due to the structure of C, if we define $\hat{x} = (x, -x)$, the previous expression can be written as

$$\hat{x}^T S^{-1}\Lambda\hat{x} > 0$$

Since $S = \text{Diag}(s)$ and $\Lambda = \text{Diag}(\lambda)$ with positive numbers[2], the expression becomes to

$$\sum_i t_i(\hat{x_i})^2, \ \ t_i > 0$$

which is clearly positive for all $x \neq 0$. This proof that $\hat{G}$ is a positive definite matrix.

□

[1] The product of two diagonal matrix is a diagonal matrix.

[2] This is said in the document of the explanation of the project, section 1.2.

T3. *Isolate δ_s from the fourth row of M_{KKT} and substitute into the third row. Justify that this procedure leads to a linear system with a symmetric matrix.*

Proof. Notice now that A is different from 0. Calling $F(z_0) = (r_L, r_A, r_C, r_s)$, the linear system that we have to solve is the following

$$\begin{pmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\gamma \\ \delta_\lambda \\ \delta_s \end{pmatrix} = \begin{pmatrix} -r_L \\ -r_A \\ -r_C \\ -r_s \end{pmatrix}$$

As an equation system, this becomes to

$$\begin{cases} G\delta_x - A\delta_\gamma - C\delta_\lambda = -r_L \\ -A^T\delta_x = -r_A \\ -C^T\delta_x + \delta_s = -r_C \\ S\delta_\lambda + \Lambda\delta_s = -r_s \end{cases}$$

Isolating δ_s from the fourth row, we obtain that $\delta_s = -\Lambda^{-1}(r_s + S\delta_\lambda)$. Now, we substitute this to the third row and we obtain the following system

$$\begin{cases} G\delta_x - A\delta_\gamma - C\delta_\lambda = -r_L \\ -A^T\delta_x = -r_A \\ -C^T\delta_x - \Lambda^{-1}(r_s + S\delta_\lambda) = -r_C \end{cases} \iff \begin{cases} G\delta_x - A\delta_\gamma - C\delta_\lambda = -r_L \\ -A^T\delta_x = -r_A \\ -C^T\delta_x - \Lambda^{-1}S\delta_\lambda = -r_C + \Lambda^{-1}r_s \end{cases}$$

In the matrix representation form,

$$\begin{pmatrix} G & -A & -C \\ -A^T & 0 & 0 \\ -C^T & 0 & -\Lambda^{-1}S \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\gamma \\ \delta_\lambda \end{pmatrix} = \begin{pmatrix} -r_L \\ -r_A \\ -r_C + \Lambda^{-1}r_s \end{pmatrix}$$

This is clearly a linear system. Moreover the matrix involve is symmetric, since G is a symmetric matrix and $\Lambda^{-1}S$ is a diagonal block[3]. □

2 The Delivery

Before discussing the results obtained in the project, I would like to comment how the delivery is made and how I implement the python code. The delivery consists of three python scripts and this *pdf* document. As it is said at the beginning, the goal of this document is clear, let us discuss the python scripts. The first script, *Inequality_case.py*, corresponds to the case in which $A = 0$, the inequality constrains case. Here, we can find the solution of the **C2**, **C3** and **C4** questions. In order to answers this questions, I elaborate a program with six functions and the main. Let us discuss the functions. The first one, *KKT*, given the matrices G and C creates the matrix M_{KKT}. Then, we can find a function called F that returns the value of the mathematical function defined in (1.1) for a given z. Later on, we find the given code *Newton_step*, the solution of **C1**, that computes the step-size correction. To finish, we have three function that computes the modification of the Newton method algorithm depending on

[3]The product of two diagonal matrix is a diagonal matrix.

the strategy we use to solve the linear system. We have *Newton_mod* for the *np.linalg.solve* method, *Newton_mod_strat1* for the LDL^T method and *Newton_mod_strat2* for the Cholesky method.

In the main, we do a loop on the dimension, from 3 to 100 and we call this three functions to see if the method works. In the screen, we print if the method works for every dimension and we create three external files to print the results. More precisely, in the first file, *C2,C3-Problem.txt*, we find the time, the number of iterations and the precision for every n for the *np.linalg.solve* method. The rest, *C4-Problem_strat1.txt* and *C4-Problem_strat2.txt* prints the same but for LDL^T and Cholesky method respectively. Finally, we plot some charts that will be useful for the next section to discuss the results. Some of them are: The execution time for every method depending the dimension, the number of iterations depending the dimension for every method, the precision depending the dimensions for every method, and the condition number of the matrices depending iterations of Newton in one particular dimension.

In the second script, *General_case1.py*, we can find the solutions of **C5** and **C6** questions for the first dataset. There are the same functions than before, except the one corresponding to the Colesky method, because it is not required here. Moreover we can find two more functions, *read_matrix* and *read_vectors* that read the matrices and vectors given in the dataset. The big difference is in the main. We do not have to make any loop now, we set the dimensions, read the matrices and we print in the screen the obtained, i.e., execution time and iterations. Finally, the last script, *General_case2.py* is a copy of the previous one, but setting different values to the dimensions because we answer the questions **C5** and **C6** regarding to the second dataset.

3 Results

3.1 Inequality constrains case (i.e. with $A = 0$)

First of all, notice that the results are subject to the random variable g. More precisely, the components of the vector g are random, generated by a Gaussian distribution $N(0, 1)$ and evidently, this can affect the computational time, the number of iterations and the precision. Nevertheless, the behaviour is always similar and since the goal of the project is to compare the three methods, this is not an inconvenience.

Let us start discussing about the computational time that the three methods required for dimensions 3 to 100. As we can see in Figure 1, the method that use the python function *np.linalg.solve* to solve the linear system is the most expensive in the computational sense and the method that use Cholesky factorization to solve the linear system is the most cheaper. This is not an unexpected result, as we have seen in theory, GEPP (*np.linalg.solve* method) solve the linear system with an overall time complexity of $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops, while LDL^T and Cholesky factorization solve the linear system with half of them, because of the structure of the matrices. Nevertheless, notice that for low dimensions, up to 30, all three methods required a similar computational time, but when dimension grows, there is a huge difference. For example, for $n = 99$, the *np.linalg.solve*, LDL^T and Cholesky method required 0.868, 0.219 and 0.069 seconds respectively.

Let us continue reporting information about the iterations of Newton's modified algorithm for the three different methods to solve linear systems. For the first method, *np.linalg.solve*, as we can see in Figure 2, we usually get 13-18 iterations for dimensions less than 20 and 17-19 iterations for bigger dimensions. On the other hand, with a similar behaviour, for the LDL^T

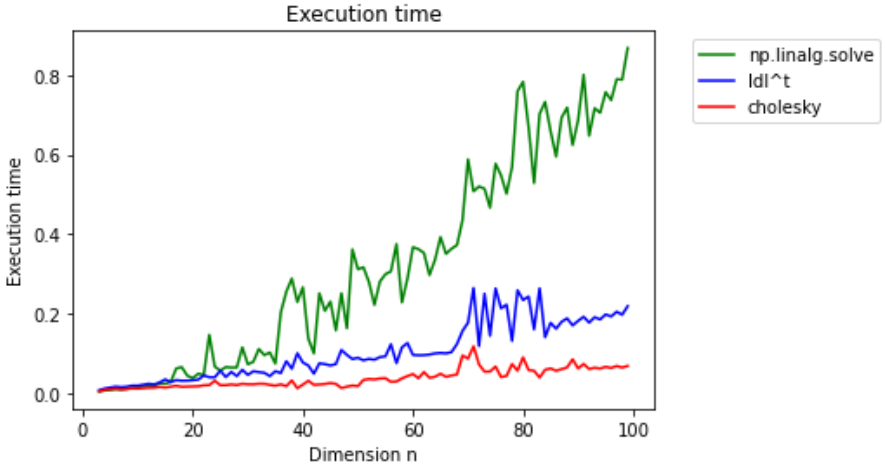

Figure 1: Computational time for the three methods depending the dimension.

and Cholesky methods, we usually get 14-19 iterations for dimensions less than 20 and 18-20 iterations for bigger dimensions (see Figures 3 and 4).

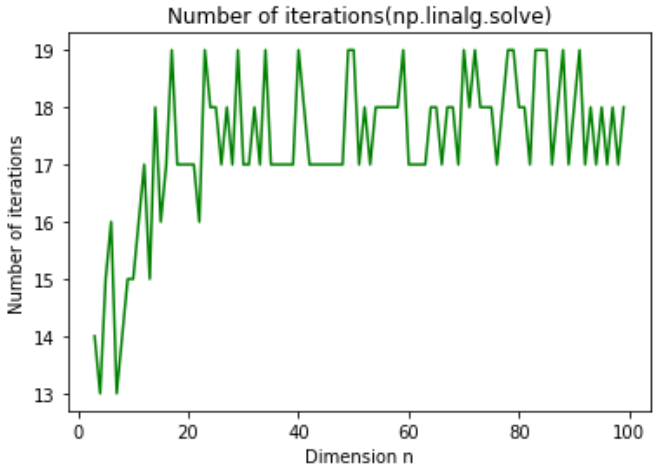

Figure 2: Number of iterations depending the dimension for *np.linalg.solve* method.

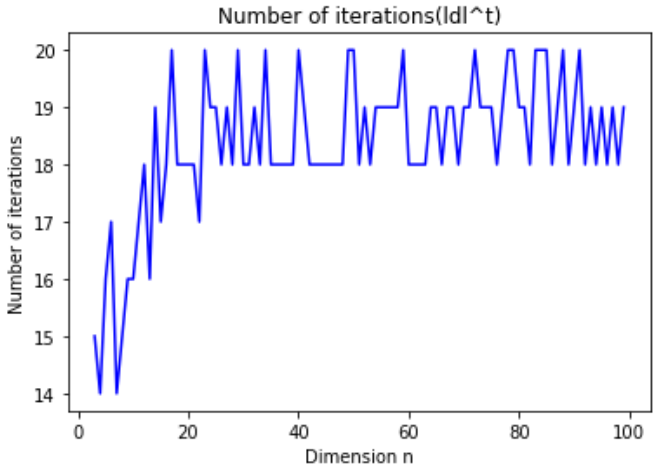

Figure 3: Number of iterations depending the dimension for LDL^T method.

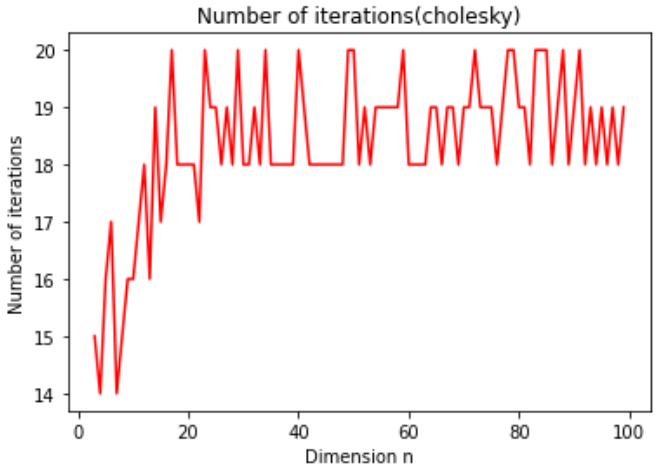

Figure 4: Number of iterations depending the dimension for Cholesky method.

Let us see now some information about the precision of the results for each method. As we can see in Figure 5, for the *np.linalg.solve* method, we can guarantee precision up to the twelfth decimal, even though, as we can see in the chart, for bigger dimensions the precision is perfect. The same behaviour is presented using the LDL^T and Cholesky methods. Observe now (Figures 6 and 7) that for this methods we can guarantee precision up to the thirteenth decimal.

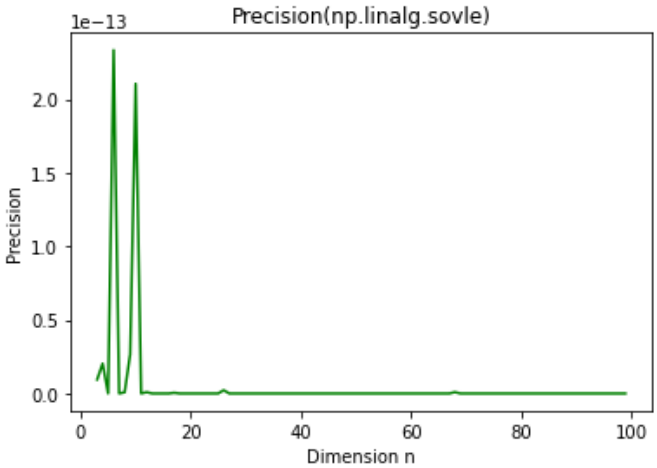

Figure 5: Error of the result depending the dimension for *np.linalg.solve* method.

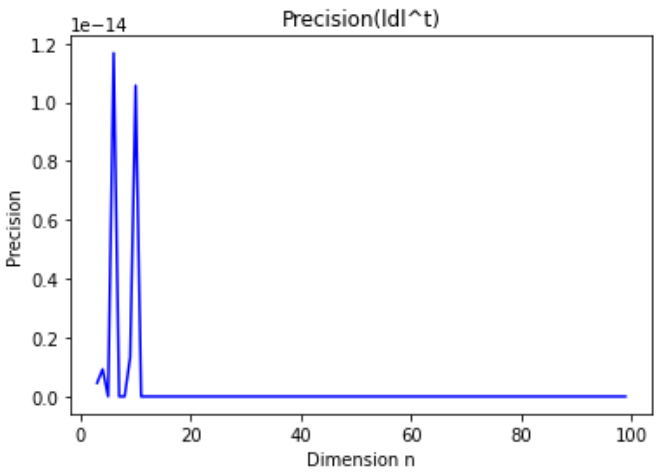

Figure 6: Error of the result depending the dimension for LDL^T method.

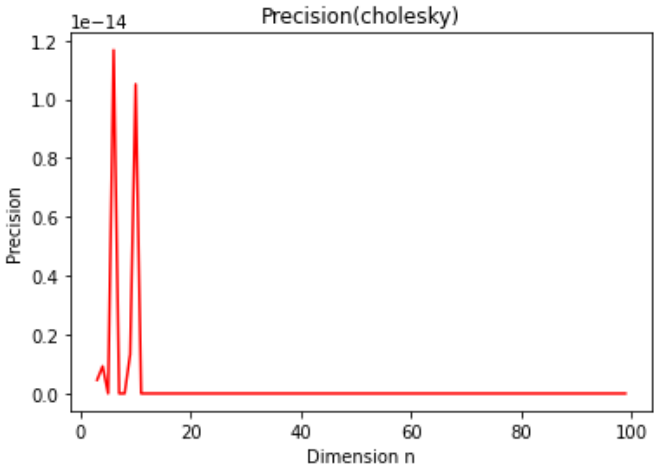

Figure 7: Error of the result depending the dimension for Cholesky method.

To finish this first part, let us discuss about the condition number of the matrices in which we are applying the methods. We are interested in how matrices changes over the Newton's iterations and how bad or good conditioned are this matrix. We recall that the closer to one is the condition number of the matrix, the better conditioned is the matrix. In order to study that, we have fixed three values of n, $40, 60$ and 80. For the first method, *np.linalg.solve*, as we can see in Figures 8, 9 and 10, the condition number of the matrix in which we apply *np.linalg.solve* is "big" at the beginning and then it stabilizes to one making this strategy well-conditioned.

For the LDL^T method, we can observe in Figures 11, 12 and **??**, that the matrices start being well-conditioned, but since we are inverting a matrix in which their coefficients may be "small" in the final iterations, the condition number grows a lot, up to 10^{20}, making the matrix ill-conditioned. Nevertheless, since Newton's method convergence is quadratic, the method is converging to the right solution, even with more precision and less time than before, thanks to the facility of solving the system due to the LDL^T factorization.

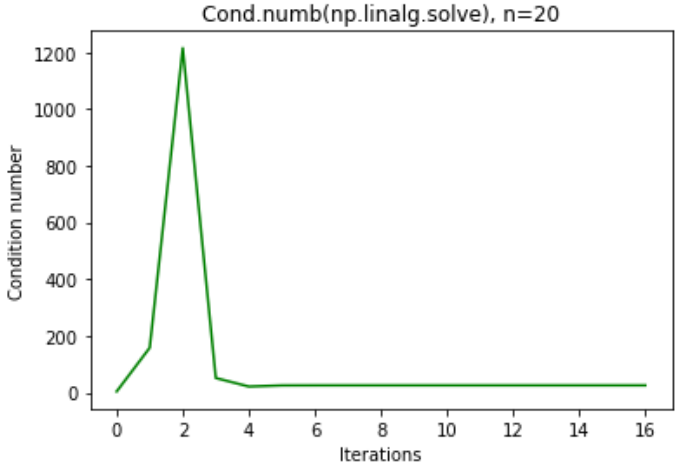

Figure 8: Condition number of the matrices depending the iteration of Newton's modified algorithm using *np.linalg.solve* method for $n = 20$.

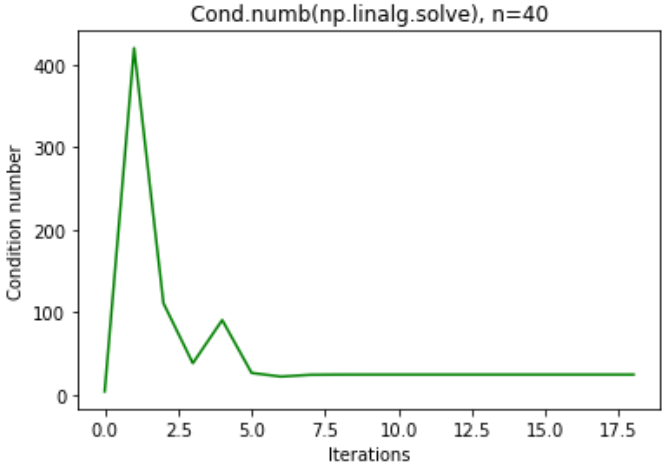

Figure 9: Condition number of the matrices depending the iteration of Newton's modified algorithm using *np.linalg.solve* method for $n = 40$.

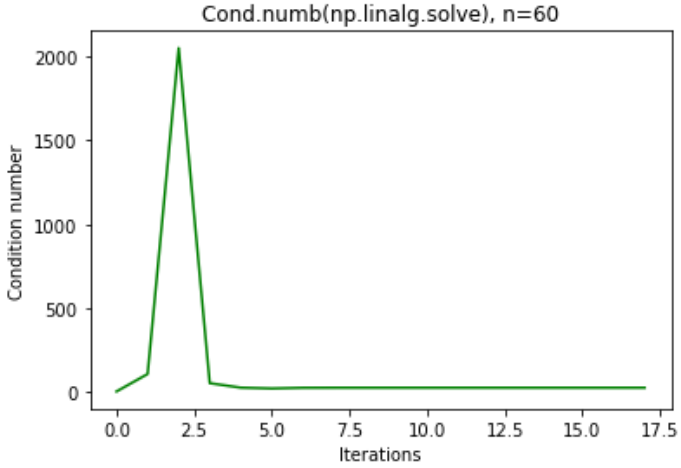

Figure 10: Condition number of the matrices depending the iteration of Newton's modified algorithm using *np.linalg.solve* method for $n = 60$.

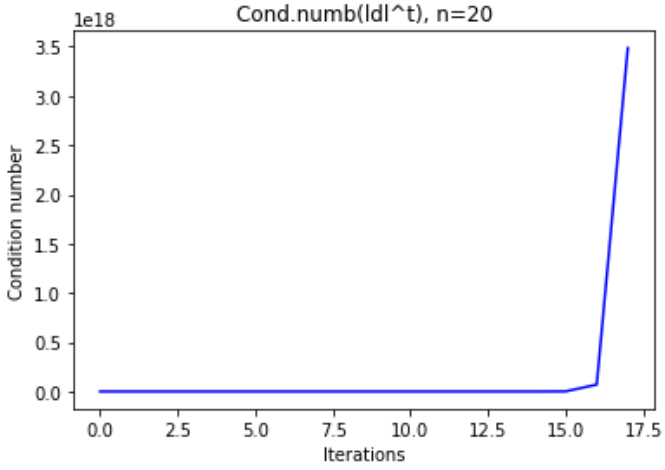

Figure 11: Condition number of the matrices depending the iteration of Newton's modified algorithm using LDL^T method for $n = 20$.

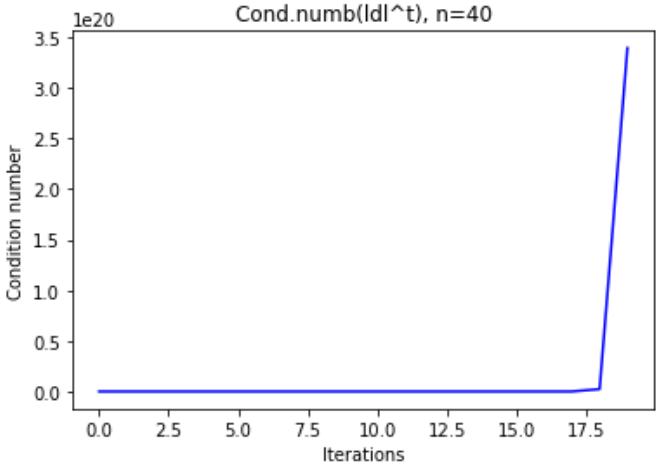

Figure 12: Condition number of the matrices depending the iteration of Newton's modified algorithm using LDL^T method for $n = 40$.

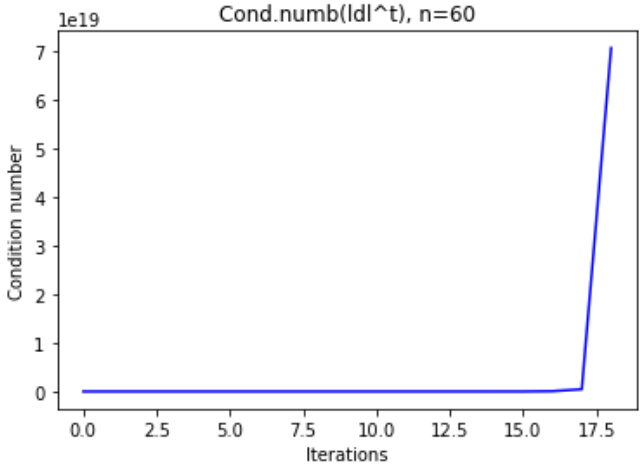

Figure 13: Condition number of the matrices depending the iteration of Newton's modified algorithm using LDL^T method for $n = 60$

Finally, in Figures 14, 15 and 16, we can observe the results for Cholesky method. As we can see, the behaviour is similar to the *np.linalg.solve* method. At the beginning the condition number of the matrix in which we apply Cholesky is "big", but less than *np.linalg.solve* method and later on, on the final iterations, the condition number stabilizes to one, making the strategy well-conditioned. Although the behaviour in that case is similar to the *np.linalg.solve* case, notice that the computational time is much smaller (see Figure 1), this is due to the facility of solving the system with the Cholesky factorization.

To sum up, we conclude that regarding to iterations and precision, the three methods are really similar. Respecting the computational time and the condition number, we have seen that although *np.linalg.solve* and Cholesky methods have a similar behaviour in the condition number (are well-conditioned), the computational time are really different due to way of solving the linear system. In addition, we have observed, that the LDL^T method has an advantage with respect the computational time regarding to *np.linalg.solve* method, but has the disadvantage that is ill-conditioned.

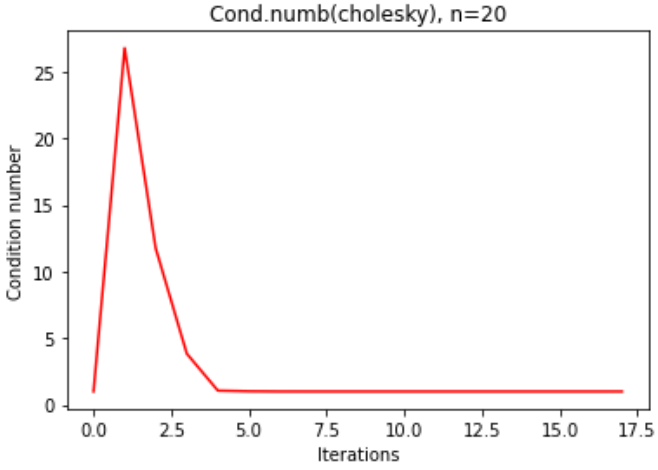

Figure 14: Condition number of the matrices depending the iteration of Newton's modified algorithm using Cholesky method for $n = 20$.

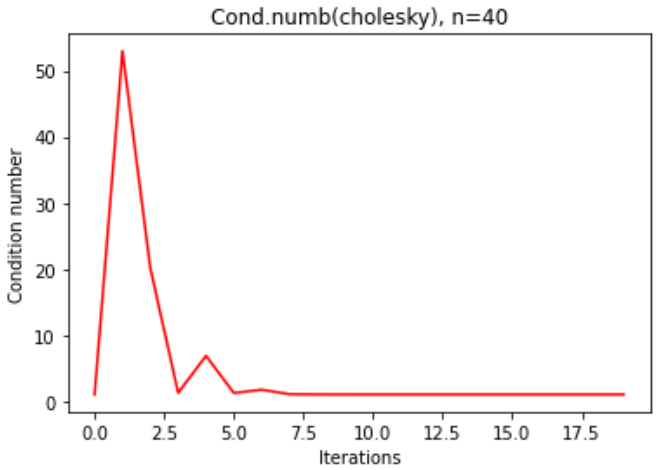

Figure 15: Condition number of the matrices depending the iteration of Newton's modified algorithm using Cholesky method for $n = 40$.

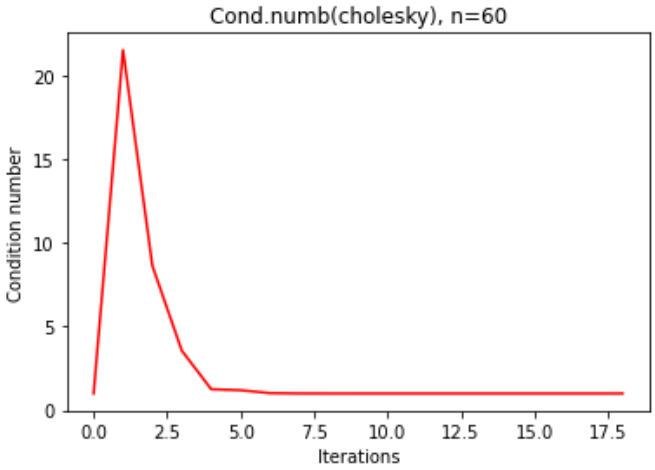

Figure 16: Condition number of the matrices depending the iteration of Newton's modified algorithm using Cholesky method for $n = 60$.

3.2 General case (with equality and inequality constrains)

To finish this document, let us share the results for the general case. The results obtained are the ones expected, a vector x such that $f(x) = 1.15907181 \times 10^4$ for the first dataset and a vector x such that $f(x) = 1.08751157 \times 10^6$ for the second dataset. The dimensions are $n = 100, p = 50, m = 200$ and $n = 1000, p = 500, m = 2000$ respectively. For the execution time and the iterations, see Tables 1 and 2.

Method	Time (s)	Iterations
np.linalg.solve	1.43	29
LDL^T	0.39	28

Table 1: Results for the first dataset.

Method	Time (s)	Iterations
np.linalg.solve	186.10	29
LDL^T	23.93	29

Table 2: Results for the second dataset.

Observe the improvement of the computational time for LDL^T method. Amazing!

4 Difficulties

I would like to say that the project was quite interesting and it allows to deepen in practical aspects of the studied methods in theoretical lessons. In this section, I would like to discuss the difficulty I found while doing this project. I would like to thanks Arturo Viero for helping me to solve it.

The problem was when using the LDL^T python function for the given datasets to solve the linear systems. In that case, the diagonal matrix D was not strictly diagonal, but diagonal

per blocks and the function introduce a permutation matrix that is not the identity matrix as the inequality constrain case. In the previous case, the inequality constrain case, as the permutation matrix was the identity, the solution of the linear system was immediate. We had $A = LDL^T$, so if we want to solve $Ax = b$, call $y = DL^T x$ and solve $Ly = b$ taking into account that L is a lower triangular matrix. Then, call $z = L^T x$ and solve $Dz = y$ taking into account that D is a diagonal matrix. Finally, we can find the final solution solving $L^T x = z$ taking into account that L^T is an upper triangular matrix.[4]

For the given datasets, the function added a permutation matrix P in the sense that if we had the decomposition $A = LDL^T$, the lower triangular matrix is $\hat{L} = PL$ and D is diagonal per blocks. Now, the solution of the system is not as immediate as the previous one, let us check it. We want to solve $Ax = b$ and we have the decomposition $A = LDL^T$. Call $y = DL^T x$ and solve $Ly = b$, but no L is not triangular. Nevertheless, we know that $\hat{L}$ is lower triangular, so we multiply on both sides by P to obtain $\hat{L}y = Pb$ and solve it taking advantage of the strcutre of the $\hat{L}$ matrix. Now, call $z = L^T x$ and solve $Dz = y$. D is a diagonal per block matrix, hence, what we have done is go over the sub-diagonal of the matrix checking if the value was zero or no. If it was zero, then we have a strictly diagonal part, if not, we solve the 2×2 block system. Finally, we can find the final solution solving $L^T x = z$, but as before, L is not triangular. We know that $\hat{L}$ is lower triangular, so we solve $\hat{L}^T x = z$ taking advantage of the structure of the $\hat{L}^T$ matrix, in fact, notice that $\hat{L}^T = L^T P^T$ is an upper triangular matrix and we can easily solve the system. Then, the final solution is given by $\bar{x} = P^T x$.

[4]When we have to solve a system of the form $Bx = b$ and B is lower or upper triangular, we can take advantage of this and use *solve_triangular* python function of *scipy.linalg* library instead of *np.linalg.solve*.