Project 1 NLA: direct methods in optimization with constraints

November 5, 2023

Manuel Andrés Hernández Alonso, mhernaal70.alumnes@ub.edu, **niub20274855**

Given a problem of minimization where the objective is to find an $x \in \mathbb{R}^n$ that solves

$$\begin{aligned} &\textit{minimize} \quad f(x) = \frac{1}{2}x^T G x + g^T x \\ &\textit{subject to} \quad A^T = b,\ C^T x \geq d \end{aligned}$$

where $G \in \mathbb{R}^{n \times n}$ is symmetric semidefinite positive, $g \in \mathbb{R}^n$, $A^{n \times p}$, $C \in \mathbb{R}^{n \times m}$, $b \in \mathbb{R}^p$ and $d \in \mathbb{R}^m$.

1 Solving the KKT System

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

Let us solve the problem by means of Lagrange multipliers. We introduce $s = C^T x - d \in \mathbb{R}^m$, $s \geq 0$. Then the Lagrangia is given by

$$L(x, \gamma, \lambda, s) = \frac{1}{2}x^T G x + g^T x - \gamma^T (A^T x - b) - \lambda^T (C^T x - d - s)$$

where $\gamma \in \mathbb{R}^p$ and $\lambda \in \mathbb{R}^m$ are the Lagrangian multipliers for the equality and inequality contraints respectively.

We can rewrite the optimality conditions as:

$$\begin{aligned} Gx + g - A\gamma - C\lambda &= 0 \\ b - A^T x &= 0 \\ s + d - C^T x &= 0 \\ s_i \lambda_i &= 0 \end{aligned}$$

We can convert this set of linear equations into a function $F : \mathbb{R}^N \to \mathbb{R}^N$, $N = n + p + 2m$ that takes in a $z = (x, \gamma, \lambda, s)$, so the system would be solved at $F(z) = 0$. To achieve this $F(z) = 0$ we may use a Newton method. We define the Newton directon as δ_z and finds the next point at $z_{k+1} = z_k + \delta_z$, and to compute it we need to perform a second order Taylor expansion of the Lagrangian and making sure that the point is still feasible.

By solving the δ as the system of optimality conditions $(F(z))$ we get that the next δ_z should follow:

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

This is assuming x and $x + \delta_z$ are feasible points, if we start in an infeasible point we have to make sure that the next point is feasible by adding constraints related to $A^T x = b$ and $C^T x \geq d$ to the right hand side of the system

$$\begin{pmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\gamma \\ \delta_\lambda \\ \delta_s \end{pmatrix} = \begin{pmatrix} -g \\ -Ax + b \\ -C^T x + d \\ 0 \end{pmatrix}$$

This translates the right hand vector into the evaluation of $-F(z)$, finding thus the M_{KKT} system.

C1: Write down a routine function that implements the step-size substep.

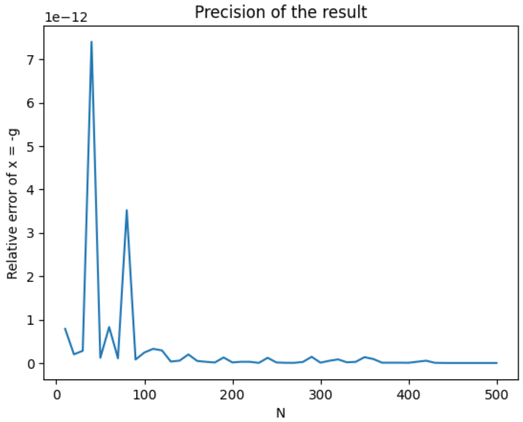
We find the step-size substep by means of the function `Newton_step` found in the `utils.py` file attached to this document.

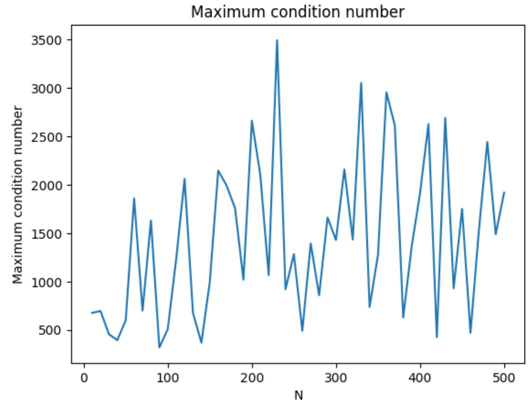
[ ]:
```
def Newton_step(lamb0,dlamb,s0,ds):
    alp=1
    idx_lamb0=np.array(np.where(dlamb<0))
    if idx_lamb0.size>0:
        alp = min(alp,np.min(-lamb0[idx_lamb0]/dlamb[idx_lamb0]))
    idx_s0=np.array(np.where(ds<0))
    if idx_s0.size>0:
        alp = min(alp,np.min(-s0[idx_s0]/ds[idx_s0]))
    return alp
```

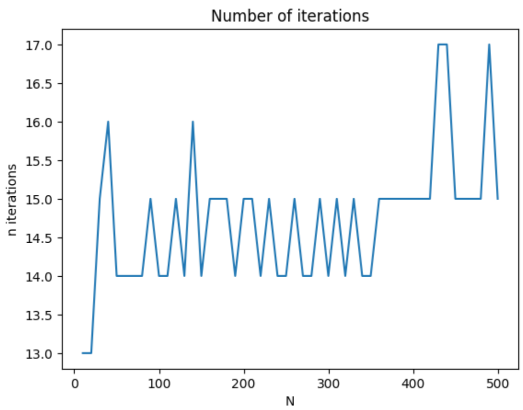
1.1 Inequality constraints case (i.e. with $A = 0$)

C2: Write down a program that, for a given n, implements the full algorithm for the test problem. Use the numpy.linalg.solve function to solve the KKT linear systems of the predictor and corrector substeps directly

The implementation of this algorithm can be found in `C2.py`. I performed the algorithm on different dimensions of n ranging from 10 to 500 in increments of 10 and obtained the following precisions, condition numbers and number of iterations.

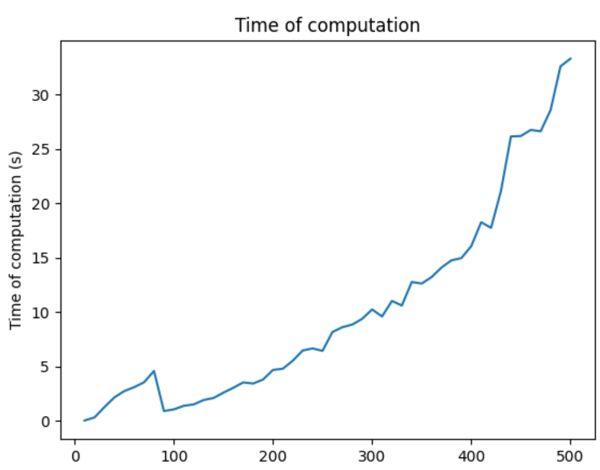






C3: Write a modification of the previous program **C2** to report the computation time of the solution of the test problem for different dimensions n

The modification of the previous program **C2** can be found in `C3.py`. I again performed the algorithm on different dimensions of n ranging from 10 to 500 in increments of 10 and obtained the following computation times:



T2: Explain the previous derivation of the different strategies and justify under which assumptions they can be applied

Stategy 1: We can apply this strategy under the assumption that the following matrix is symmetric due to the LDL^T factorization being a method used only on symmetric matrices.

$$\begin{pmatrix} G & -C \\ -C^T & -\Lambda^{-1}S \end{pmatrix}$$

This can be proven due to G already being a symmetric matrix and C and C^T being mirrored on the diagonal, lastly $-\Lambda^{-1}S$ is a diagonal matrix, so it is already symmetric

Strategy 2: We can apply this strategy under the assumption that $\hat{G} = (G + CS^{-1}\Lambda C^T)$ is symmetric definite positive. Since we are working in convex problems and G is a symmetric semidefinite positive matrix, when we add CC^T we get another symmetric semidefinite positive matrix due to CC^T being a symmetric matrix. Finally, we can ignore Λ^{-1} and S due to them being diagonal matrices.

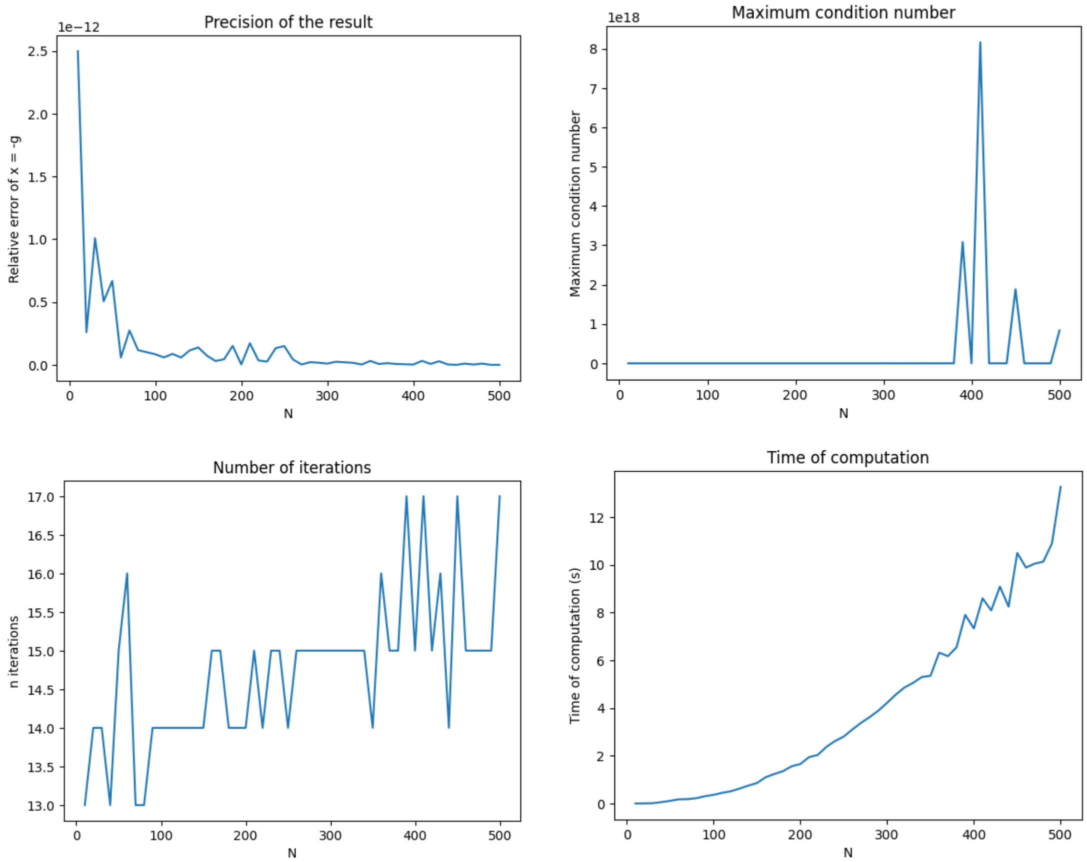
This follows as we can apply the Cholesky factorization on $\hat{G}$ to solve $\hat{G}\delta_x = -r_1 - \hat{r}$ where $\hat{r} = -CS^{-1}(-r_3 + \Lambda r_2)$ due to the assumption of symmetric positiveness.

C4: Write down two programs (modifications of **C2**) that solve the optimization problem for the test problem using the previous strategies. Report the computational time for different values of n and compare with the results in **C3**

Strategy 1 The first modification applies the first strategy where $\delta_s = \Lambda^{-1}(-r_3 - S\delta_\lambda)$ andthe system to solve is

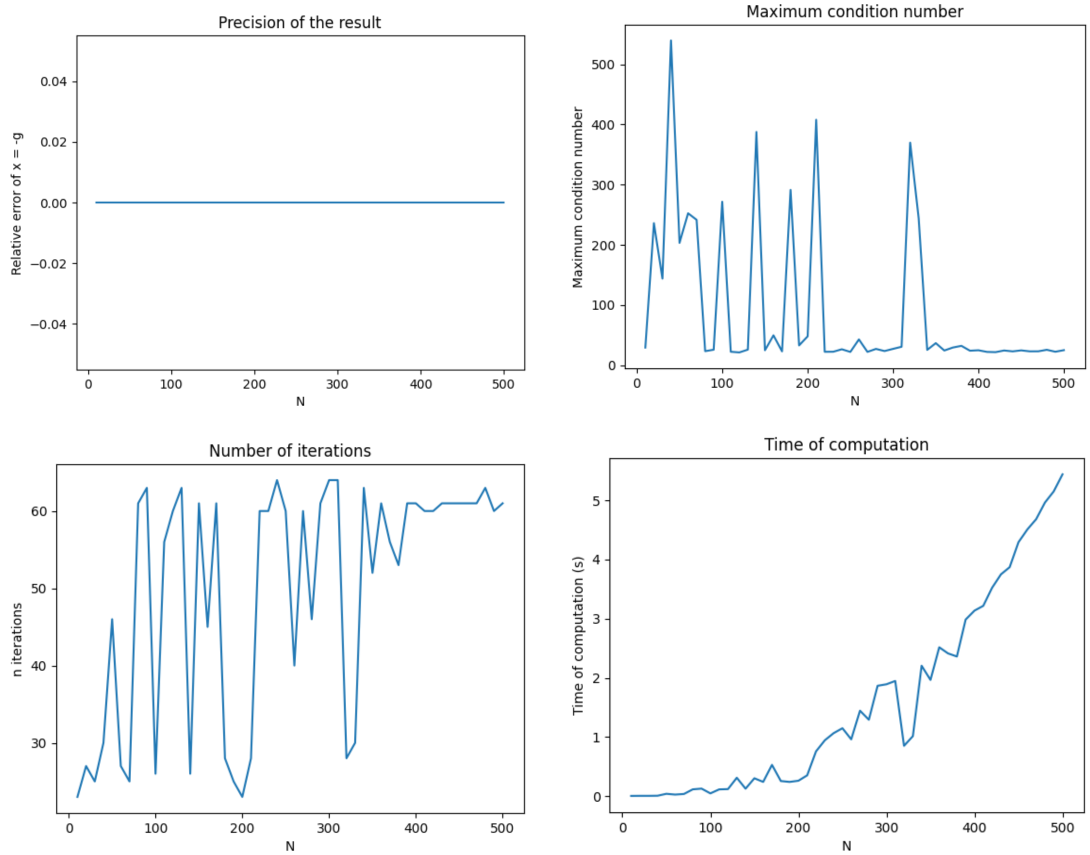
$$\begin{pmatrix} G & -C \\ -C^T & -\Lambda^{-1}S \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_\lambda \end{pmatrix} = -\begin{pmatrix} r_1 \\ r_2 - \Lambda^{-1}r_3 \end{pmatrix}$$

We can find the implementation in `C4.py`. I performed the algorithm on different dimensions of n ranging from 10 to 500 in increments of 10 and obtained the following information:



As we can see the times are greatly reduced compared to the use of the full system, from around 30 seconds at $n = 500$ with the full system to around 12 seconds with the strategy 1.

Strategy 2 The second modification applies the second strategy where $\delta_s = -r_2 + C^T\delta_x$, $\delta_\lambda = S^{-1}(-r_3 + \Lambda r_2) - S^{-1}\Lambda C^T\delta_x$ and the system to solve is $\hat{G}\delta_x = -r_1 - \hat{r}$, where $\hat{G} = (G + CS^{-1}\Lambda C^T)$ and $\hat{r} = -CS^{-1}(-r_3 + \Lambda r_2)$. Then applying Cholesky factorization to $\hat{G}$ we solve the system. We can find the implementation also in `C4.py`. Finally, I performed the algorithm on different dimensions of n ranging from 10 to 500 in increments of 10 and obtained the following information:



As we can see the times are greatly reduced again compared to the use of the second strategy, from around 12 seconds at $n = 500$ with the strategy 1 to around 5.5 seconds with the strategy 2. We can also see that here we get full precision, meaning $x = -g$ is completely precise up to machine ϵ. Additionally, the number of iterations jump from ~15 to ~45. The condition numbers for the matrices also seem to jump between values more than the strategy 1 or the full system.

1.2 General case (with equality and inequality constraints)

C5: Write down a program that solves the optimization problem for the general case. Use numpy.linalg.solve function. Read the data of the optimization problems from the files (available at the Campus Virtual). Each problem consists on a collection of files: `G.dad`, `g.dad`, `A.dad`, `b.dad`, `C.dad` and `d.dad`. They contain the corresponding data in coordinate format. Take as initial condition $x_0 = (0, ..., 0)$ and $s_0 = \gamma_0 = \lambda_0 = (1, ..., 1)$ for all problems.

Firstly, I defined two functions to read the `.dad` files: `open_matrix(path, n, m)` and `open_vector(path, n)` *(Note that since I was working in windows I couldn't have two files in the same folder with names `G.dad` and `g.dad`, so I renamed the second one to `g_vector.dad`).*

After loading them I had to convert G into symmetric matrix by adding G^T and substracting the diagonal of G. The results I got for the files in `optpr1` was a solution vector x such that $f(x) = 1.15907181 \times 10^4$ in 1.4311 seconds and for `optpr2` was a solution vector x such that $f(x) = 1.08751157 \times 10^6$ in 59.8986 seconds. The implementation for this algorithm can be found in `C5.py` and the functions for reading the files in `utils.py`

T3: Isolate δ_s from the 4th row of M_{KKT} and substitute into the 3rd row. Justify that this procedure leads to a linear system with a symmetric matrix.

Firstly, we start with the M_{KKT} system:

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

We then isolate δ_s in the 4th row:

$$\begin{aligned} S\delta_\lambda + \Lambda\delta_s &= -r_s \\ \Lambda\delta_s &= -r_s - S\delta_\lambda \\ \delta_s &= \Lambda^{-1}(-r_s - S\delta_\lambda) \end{aligned}$$

Subsequently, we substitute δ_s in the 3rd row

$$\begin{aligned} -C^T\delta_\gamma + \Lambda^{-1}(-r_s - S\delta_\lambda) &= -r_C \\ -C^T\delta_\gamma - \Lambda^{-1}r_s - \Lambda^{-1}S\delta_\lambda &= -r_C \\ -C^T\delta_\gamma - \Lambda^{-1}S\delta_\lambda &= -r_C + \Lambda^{-1}r_s \end{aligned}$$

And as matrix 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 final system has a symmetric matrix M_{KKT} due to G already being a symmetric matrix and $-\Lambda^{-1}S$ being a multiplication of two diagonal matrices. Finally, A and C follow that A^T and C^T are mirrored on the diagonal since they are sharing symmetric blocks $[i, j]$ and $[j, i]$.

C6: Implement a routine that uses LDL^T to solve the optimizations problems (in **C5**) and compare the results.

To implement the algorithm I used the resulting symmetric system from **T3**, I had some problems with the numerical stability when using `scipy.linalg.ldl` and `scipy.linalg.solve_triangular` so I had to change it to `scipy.linalg.lapack.dsysv`, that takes the matrix and the right hand vector of the system and automatically applies LDL^T factorization and solves it. I also used the same functions to read the files in **C5**, then applied the algorithm to both `optpr1` and `optpr2`. The results I got for the files in `optpr1` was a solution vector x such that $f(x) = 1.15907181 \times 10^4$ in 0.4229 seconds and for `optpr2` was a solution vector x such that $f(x) = 1.08751157 \times 10^6$ in 21.8845 seconds. The implementation for this algorithm can be found in `C6.py`.

We can note that the time for computation has been significantly reduced to around a third of the original full system computation time.