1 Project 1: direct methods in optimization with constraints

The goal of this project is to investigate some of the basic numerical linear algebra ideas behind optimization problems. For simplicity consider the convex optimization problem for $x \in \mathbb{R}^n$:

$$\begin{cases} Min & f(x) = \frac{1}{2}x^TGx + g^Tx \\ subject\ to & A^Tx = b,\ C^Tx \geq g, \end{cases}$$

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

Note that in order to find this constrained optimization problem we can consider the Lagrange multipliers so we obtained the Lagrangian:

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

where $s = C^Tx - d \in \mathbb{R}^m, s \geq 0$, and $\gamma \in \mathbb{R}^p$ and $\lambda \in \mathbb{R}^m$ are the vectors containing the Langrangian multipliers for the equality and inequality constrains respectively.
The optimality conditions can be computed by conducting the corresponding partial derivatives of the Lagrangian, i.e.

$$\nabla L(x, \gamma, \lambda, s) = \Big(\frac{\partial L}{\partial x}(x, \gamma, \lambda, s), \frac{\partial L}{\partial \gamma}(x, \gamma, \lambda, s), \frac{\partial L}{\partial \lambda}(x, \gamma, \lambda, s), \frac{\partial L}{\partial s}(x, \gamma, \lambda, s)\Big) = (0, 0, 0, 0)$$

and can be written as follows:

$$\begin{aligned} Gx + g - A\gamma - C\lambda &= 0 \\ b - A^Tx &= 0 \\ s + d - C^Tx &= 0 \\ s_i\lambda_i &= 0, i = 1, \dots, m \end{aligned}$$

where we also require the components v_i of $v = (\lambda, s) \in \mathbb{R}^{2m}$ to verify $v_i \geq 0$, so we are in the feasibility region.

We introduce $z = (x, \gamma, \lambda, s)$ and $F : \mathbb{R}^N \to \mathbb{R}^N$, $N = n + p + 2m$ such that the previous system translates into $F(z) = 0$. Note that this is equivalent to consider the aforementioned condition $\nabla L(z) = 0$, and thus, we can consider $F := \nabla L$.

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

Note now that, in order to compute the Newton step of size $\delta_z = (\delta_x, \delta_\gamma, \delta_\lambda, \delta_s)$ for z such that $z_1 = z_0 + \delta_z$ we consider the Newton step:

$$z_1 = z_0 - \nabla^2 L(z_0)^{-1}\nabla L(z_0),$$

Note that an alternative formulation for this Newton step is to find a direction δ_z such that $\nabla L(z_0 + \delta_z) = 0$. Considering now the linearization of this expression in terms of the Taylor's expansion of second order we have:

$$\nabla L(z_0 + \delta_z) \approx \nabla L(z_0) + \nabla^2 L(z_0)\delta_z$$

Thus, we are left with the following system:

$$\nabla^2 L(z_0)\delta_z = -\nabla L(z_0) = -F(z_0)$$

in terms of the observation considered previously. Note now that the corresponding Hessian is trivially given as follows

$$\nabla^2 L(z) = \begin{pmatrix} G & -A & -C & 0 \\ -A^T & 0 & 0 & 0 \\ -C^T & 0 & 0 & I \\ 0 & 0 & S & \Lambda \end{pmatrix} = M_{KKT}$$

where S, Λ is the diagonal matrix with s and λ in the diagonal, respectively. Thus, computing the aforementioned Newton's step is equivalent to solving the linear system defined by the KKT matrix, M_{KKT}, as wanted to prove.

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

The corresponding code is attached on the file `C1.py` and contains a function implementing the Newton substep size correction, returning as output the new step size α.

C2: *Consider the inequality constrains case, i.e. $A = 0$. 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 system of the predictor and corrector substeps directly.*

The corresponding code is presented on the file `C2.py`. In this program the corresponding functions needed for the algorithm are defined. The function `testProblem(n)` solves the inequality constrains case for the optimization test problem defined by

$$m = 2n,\ G = I_{n\times n},\ C = (I_{n\times n}\ \ -I_{n\times n}),\ d = (-10, \dots, -10)$$

g given by a Gaussian distribution $N(0,1)$, $x_0 = (0,\dots,0)$, and $s_0 0 \lambda_0 = (1,\dots,1)$, which has the analytical exact solution of $x = -g$.
The stopping criterion used is that either $F(z_k) = (r_L, r_A, r_C, r_s)$ verifies $|r_C| < \epsilon, |r_L| < \epsilon, |r_A| < \epsilon$, or $|\mu| < \epsilon$, from the 3rd step, with $\epsilon = 10^{-16}$, or either we exceed a maximum of 100 iterations.

The program can be implemented for different values of n so we can study the iterations performed, the precision of the results (used by computing the mean squared error between the found solution and the analytical $x = -g$) and the condition number of the matrices: $k_A = ||A^{-1}|| \cdot ||A||$.

For the particular case of $n = 5$ we have after 100 iterations a condition number of the M_{KKT} matrix of 21.711571338319008 after the last iteration, which is itertation 13, and a mean square error of the order of $1.1443916996305594e-16$, which is quite impressive. The evolution of the MSE is shown on Figure 1, to check its minimization throughout the optimization progress of the algorithm.

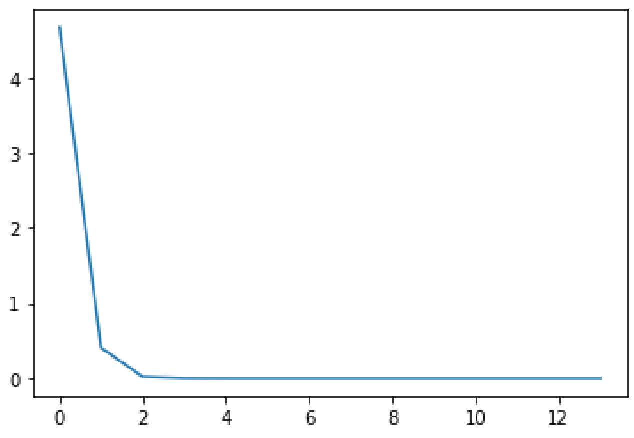


Figure 1: MSE throughout each iteration of the algorithm for the program `C2.py`.

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 corresponding program is present on the file `C3.py`, in which a modification is introduced that runs the `testProblem(n)` algorithm for different values of n, and then gives as an output the MSE, the computation time, the solution found and the number of iterations spent for each n, and then a plot of the corresponding computation time in terms of n. A particular example is presented for dimension until $n = 100$ in Figure 2. The number of iterations, mean square error, and computing time are printed for each dimension size. The corresponding condition numbers can be computed by changing the function's parameters. In general, the condition number of the KKT matrix is around 24, however, sometimes it gets higher indicating that small perturbances on the initial condition could affect significantly the solution found. However, the algorithm used generally solves this problem ending up converging to a solution that approximates quite satisfactory the real solution (to the order of 10^{-15} generally). In case it would not converge, since the convergence of the Newton number is conditioned to the initial point, a good way to solve this problem could be considering a small noise modifying the initial point and checking the error obtained. If wished, a loop that carries out this procedure could ensure the convergence of the method, when the condition number is high.

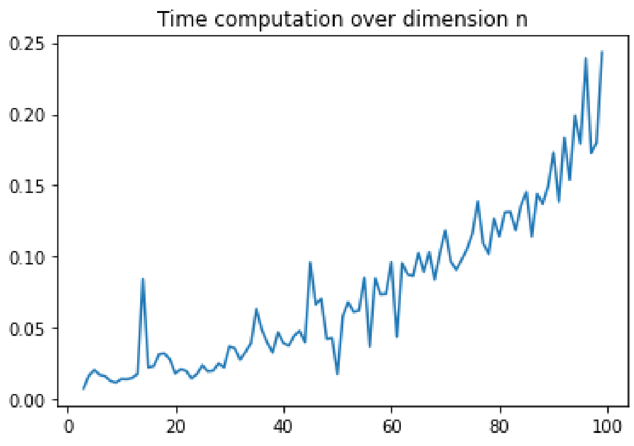


Figure 2: Computation time versus n for $n = 1, \ldots, 100$.

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

Consider the alternative strategies in order to solve the KKT system more efficiently.

1. From the inequality constrains case, isolate δ_s from the 3rd row of M_{KKT} so we have:

   $$\delta_s = \Lambda^{-1}(-r_s - S\delta_\lambda)$$

   Note that in order to compute this step we need the existence of Λ^{-1}, i.e. we need Λ to be non singular. This is equivalent to have no coefficient of λ vanishing. Substituting this value into the 2nd row we have the following system:

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

   which can be solved using the LDL^T factorization of the matrix M_1 if it is symmetric. Then in order to apply this Strategy we need M_1 to be symmetric. By its construction, this is equivalent to have G symmetric.

2. Isolate now δ_s from the 2nd row obtaining

   $$\delta_s = -r_c + C^T \delta_x$$

   Thus, from 3rd row we have

   $$\delta_\lambda = S^{-1}(-r_s + \Lambda r_c) - S^{-1}\Lambda C^T \delta_x$$

   Note this time, that we need S to be nonsingular, i.e. we need to have s without any zero coefficient. Using this expression into the 1st row we obtain the linear system

   $$\hat{G}\delta_x = -r_L - \hat{r}$$

   where $\hat{G} = (G + CS^{-1}C^T)$ and $\hat{r} = -CS^{-1}(-r_s + \Lambda r_c)$. We can now solve this system applying Cholesky factorization to $\hat{G}$. Note that in order to apply Cholesky we need $\hat{G}$ to be symmetric and positive definite, which again is conditioned on the symmetry of G. In order to check whether the matrix is positive definite it is enough to compute the eigenvalues of $\hat{G}$.

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 it with the results.*

The attached file `C4.py` contains the corresponding functions that implement the strategies aforementioned to solve the KKT linear system that gives the Newton step for each iteration. Then a main function, `compareTimes(n)` carries out the algorithm using the 3 different strategies and gives as an output the plot that represents the different computation times for the 3 strategies and in terms of n.

In this file the functions `strategy1` and `strategy2` solve the KKT system using the strategies previously defined. In order to conduct the LDL^T factorization, function `scipy.linalg.ldl` is applied to the symmetric matix. Then we have $M_1 = LDL^T$, with PL triangular, where P is a permutation matrix given as an output of the aforementioned function. Therefore, the corresponding system $M_1\delta_{x,\lambda} = r'$ corresponds to solving the following systems:

1. We have $LDL^T\delta_{x,\lambda} = r'$. So we first solve the system $Ly_1 = r'$, that can be considered as the triangular system by using the permutation matrix as follows $PLy_1 = Pr'$, that can be solved more efficiently with `scipy.linalg.solve_triangular`.

2. Then we can solve $Dy_2 = y_1$, by simply dividing the coefficients of y_1 by the diagonal vector of D, $y_2 = y_1/diag(D)$.

3. We are left with a system we wish to be upper triangular. Note we have $L^T\delta_{x,\lambda} = y_2$, with PL lower triangular, then $(PL)^T = L^T P^T$ is upper triangular. Note now that P is a permutation matrix and therefore it is orthogonal, i.e. $P^T \cdot P = Id$. Thus, we have the equivalent upper triangular system: $L^T P^T P\delta_{x,\lambda} = y_2$. From this linear system we get $P\delta_{x,\lambda}$, and we can easily compute $\delta_{x,\lambda}$ by multiplying this matrix by P^T again.

Similarly, the same arguments are used to solve the system we get with the Strategy 2, keeping in mind that the Choleskey factorization gives us $M_2 = M_C M_C^T$, where M_C is also triangular. This is imlemented using the function `np.linalg.cholesky`. Note that this factorization is conducted when M_2 is in fact symmetric and positive definite. Therefore, the corresponding checks are conducted in order to ensure the convergence of the method.

Using the 3 different strategies for $n = 2, \dots 100$, the computation times are computed and plotted, as can be observed in Figure 3.

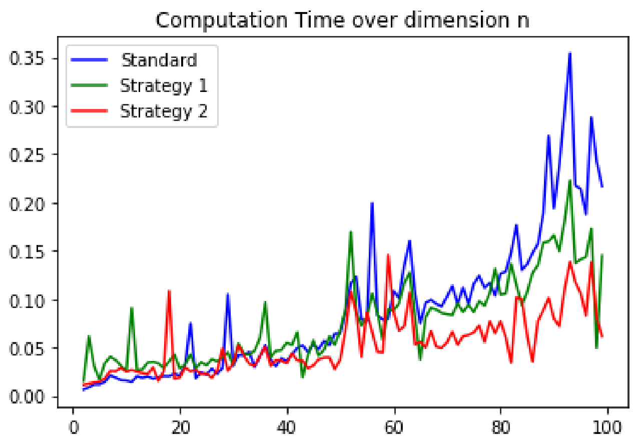


Figure 3: Computation time for different values of $n = 2, \dots, 100$ and for the 3 different strategies considered.

In order to print the corresponding details on each method, it is enough to execute `testProblem(n, cnumber = False, method = 1, show_prints = True, make_plot=True)` for a given n and fix `show_prints=True` to print the corresponding results, `cnumber = True` to print the condition numbers, and `make_plot = True` to show an error plot with the same format as for `C3.py` in the arguments of the function.

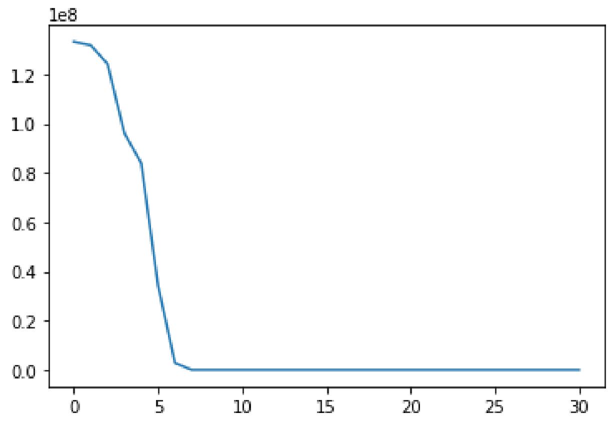
C5: *Write down a program that solves the optimization problem for the general case. Use* `numpy.linalg.solve` *function. Read the data of the optimization problems from the files. Each problem consists on a collection of files* `G.dad`, `g.dad`, `A.dad`, `b.dad`, `C.dad`, `d.dad`. *They contain the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

The corresponding program is attached in the file `C5.py` with the corresponding functions that load the demanded matrices `load_matrix(file_name, n, m)`.
In the initial implementation just the original strategy, with `np.linalg.solve` function as the system's solver is applied. For each problem the corresponding results are listed on the following table, Table 1:

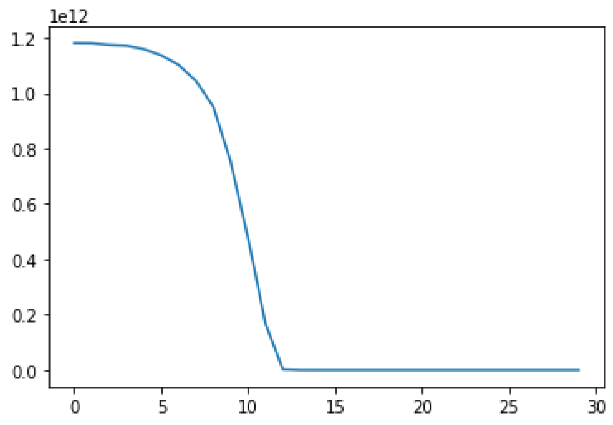
Data set	Error	Computation Time (s)	Iterations
1	3.77e-10	0.76	30
1 LDL^T	3.77e-10	0.73	26
2	7.17e-06	207.15	29
2 LDL^T	7.17e-06	75.65	28

Table 1: Results of the implementation of the algorithm for the given datasets.

Note that here the error is computed by calculating the difference between the computed $f(x)$ for the optimal obtained x and the analytical value, given as $f(x) = 1.15907181 \times 10^4$ for problem 1, and $f(x) = 1.08751157 \times 10^6$ for problem 2.
Additionally, the plots of this error for each iteration performed are present on Figure 4:



(a) Problem 1



(b) Problem 2

Figure 4: Error per iteration for both datasets.

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

Note that in this case we must consider the general case, i.e. we have $A \neq 0$, and thus we have to include b and the corresponding rows (equations) to the system. Therefore, we have to slightly modify the approach followed previously in strategy 1: We isolate δ_s from the 4th row of M_{KKT}:

$$\delta_s = \Lambda^{-1}(-r_s - S\delta_\lambda)$$

Note that in order to compute this step we need the existence of Λ^{-1}. This is equivalent to have no coefficient of λ vanishing. Substituting this value into the 3rd row we have the following system:

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

which can be solved using the LDL^T factorization of the matrix M_1 if it is symmetric. Then in order to apply this Strategy we need M_1 to be symmetric. By its construction, this is equivalent to have G symmetric, which in both cases is given by the definition of the matrix that the datasets provide. More explicitly, if we compute the corresponding transpose, taking into consideration that $\Lambda^{-1}S$ is the diagonal matrix containing the values s_i/λ_i for each $i = 1, \dots, m$, and thus $(\Lambda^{-1}S)^T = (\Lambda^{-1}S)$ we trivially have:

$$M_1^T = \begin{pmatrix} G^T & -A & -C \\ -A^T & 0 & 0 \\ -C^T & 0 & -\Lambda^{-1}S \end{pmatrix} = M_1 \iff G^T = G$$

as aforementioned.

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

In order to implement this part of the project it is necessary to observe that we have some cases in which the regular LDL^T factorization explodes. More explicitly, consider the case in which we have:

$$A = \begin{pmatrix} \epsilon & 1 \\ 1 & \epsilon \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} \epsilon & 0 \\ 0 & 1/\epsilon \end{pmatrix}}_{D} \underbrace{\begin{pmatrix} 1 & 1/\epsilon \\ 0 & 1 \end{pmatrix}}_{L^T}$$

with $\epsilon \to 0$, thus we have infinite numbers both in D and L making the algorithm not converge. Thus we have to consider this fact when implementing the LDL^T factorization.

Observe that when applying the function `scipy.ldl` we obtain a block matrix D with blocks on the diagonal. Thus we may consider an algorithm that implements the regular LDL^T as specified on `C4.py` when we have D diagonal, and solve the system classically when we find a block.

This process of going through the diagonal matrix looking for the block matrices is implemented using the function `solveNotDiagonal`, implemented on the code `C6.py`. Using this function we find the results shown on table 1.

Note that clearly the new strategy considered is in general faster than solving directly the system. However, given the fact that D is built in blocks, the algorithm has to compute several linear systems by directly applying the `np.linalg.solve` function, which takes longer that simply computing the LDL^T factorization and solving triangular and diagonal system. This time difference is therefore more present in the test problem, since the diagonal matrix given as output from the LDL^T factorization is in fact diagonal.

As specified on the other parts, if wished, the condition number along the iterations can be printed changing the function's parameters. In particular, for the problem 1, the condition number of M_{KKT} increases with the iterations, becoming a more unstable matrix as the algorithm iterates. When implementing strategy 1, the condition number of the reduced matrix used also increases with the iterations. Note additionally, that since the computation of the condition number requires the calculation of the inverse matrix, the computation time increases significantly when this option is selected (thus, the default parameters set this option in `False`.