Project 1 NLA: direct methods in optimization with constraints

This document contains the explanation for the optimization problem with inequality constraints and for the general case (with equality and inequality constraints). The purpose of the project is to study the basic numerical linear algebra ideas behind optimization problems.

The problem to consider is the following:

$$minimize\ f(x) = \frac{1}{2}\, x^T G x + g^T x$$

$$subject\ to\ A^T x = b,\ C^T x \geq d$$

$$where\ G \in \mathbb{R}^{nxn}\ is\ symmetric\ positive\ definite, g \in \mathbb{R}^n, A \in \mathbb{R}^{nxp}, C \in \mathbb{R}^{nxm}, b \in \mathbb{R}^p\ and\ d \in \mathbb{R}^m$$

The solution to this problem will be computed using the interior point algorithm, which is based on the solution of the KKT system. The project itself consists in the implementation of specific routines to solve the problem based on different ways to solve the KKT system.

Inequality Constraints (A = 0)

In this first case different strategies to solve optimization problems with inequality constraints are studied. To test the developed solutions a test problem configuration, provided in the statement of the exercise, is used.

$$m = 2n, G = I_{nxn}, C = (I_{nxn}, -I_{nxn}), d = (-10, \dots, -10), g \sim N(0,1), x_0 = (0, \dots, 0)\ and\ s_0 = \lambda_0 = (1, \dots, 1)$$

The solution of this test problem is **–g**.

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 *numpy.linalg.solve* function solves the system applying LU factorization. Due that the central path algorithm requires to apply *linalg.solve* twice, it would be advisable to keep the LU factorization and apply it the second time instead of computing it twice. For this reason, and in order to probe later the time efficiency of the previous advice, two different programs have been implemented in this part; **C2.py** solves the two KKT system applying *numpy.linalg.solve*, which implies LU factorization twice, and **C2_with_LU.py** obtains the LU factorization of the KKT matrix just once and save it using *scipy.linalg.lu_factor,* then it solves the system twice (applying *scipy.linalg.lu_solve* function) taking benefit of having this factorization already computed.

Both codes take as argument the value of *n*, so this value should be passed when calling these routines.

Example:

If n = 4

> python C2.py 4

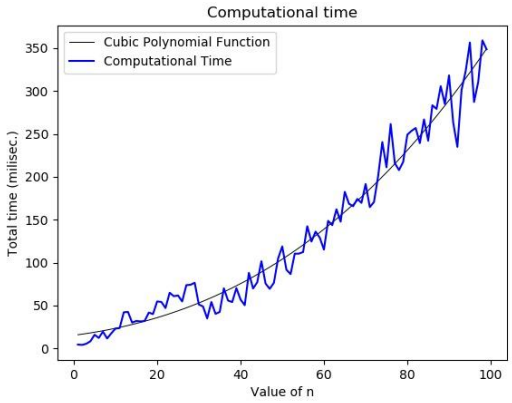
> python C2_with_LU.py 4

As it is said in the statement the solution to this problem is **x = -g**, and the norm of the gradient of the function **(F)** is almost zero at the given solution.

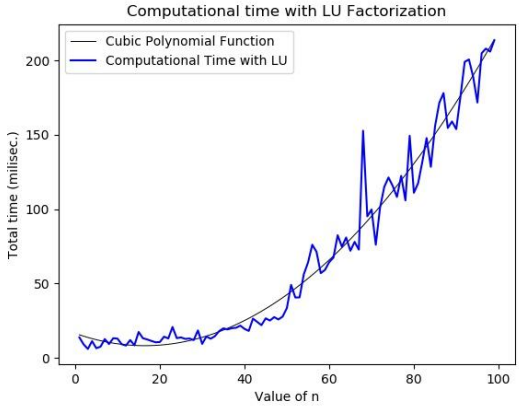
C3: Write a modification of the previous program C2 to report the computation time of the solution of the test problem for different dimensions n.

Modifying the two previous codes we write a new program which reports the computation time in function of n. The code plots a graph with the computational time for each value of n. Moreover, for each case a cubic polynomial function is also plotted in order to verify the cubic polynomial behaviour of the time required with respect dimension n. We use the *numpy* function *polyfit* to estimate the coefficients of the cubic polynomial function and *poly1d* to construct it.

Run **C3.py** to visualize the results for the code which uses *numpy.linalg.solve* to solve the KKT system.



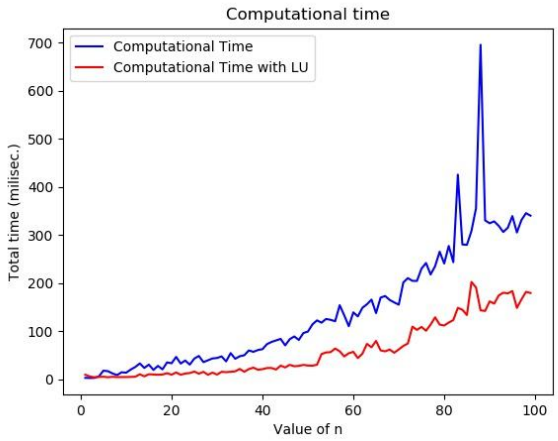
Run **C3_with_LU.py** to visualize the results for the code which uses *numpy.linalg.solve* to solve the KKT system.



It can be seen that at the same time the dimension n increases, the computational time increases as a cubic function.

Finally, and as a corroboration of the fact that saving the KKT matrix LU factorization is more time efficient, the computational times for both methods are plotted together.

Run **C3_Computational_Time_Comparison.py** to see the comparison of computational times.



As expected, if the LU factorization of KKT matrix is saved and used later in predictor and corrector substeps, the computational time is lower than if the systems are solved using *numpy.linalg.solve*.

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.

Two different strategies to solve the KKT system more efficiently are explained in the project statement.

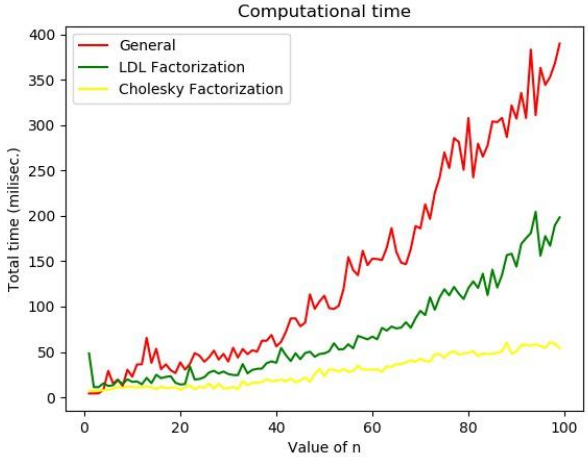
Strategy 1 reduces the KKT matrix and applies the LDL^T factorization to this new matrix, and uses it to solve the system. This time, due the LDL^T factorization we are in front of two triangular systems, for this reason we used *scipy.linalg.solve_triangular* instead of *numpy.linalg.solve* in order to save computational time.

Run **C4_Strategy1.py** to check its ability to find the solution. This code takes as argument the value of *n*, so invoke it as we did in C2 part.

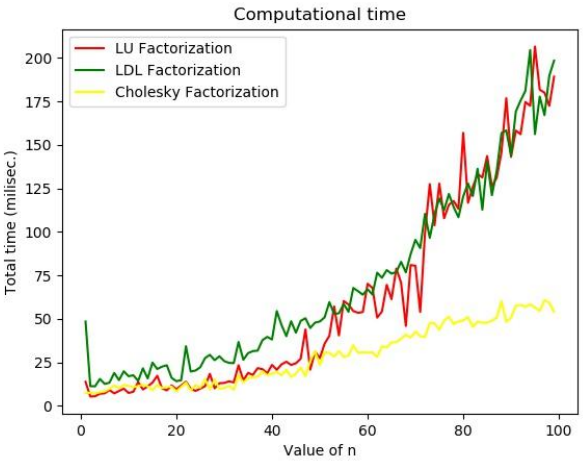
Strategy 2 modifies the KKT system in a way that Cholesky factorization can be done over the matrix. This brings us to solve two triangular systems. The *scipy.linalg.cholesky* function is used to obtain the factorization, and as before, the system is solved with *scipy.linalg.solve_triangular* in order to be more efficient in computational terms.

Run **C4_Strategy2.py** to check its ability to find the solution. Like strategy 1 code, it takes as argument the value of *n*, so invoke it as we did in C2 part.

In the statement it is asked to report the computational time of these new strategies and compare the values with the results in **C3**. The **C4_Computational_Time_Comparison.py** code generates two plots; one that represents computational time of these two new strategies against the method developed in **C2** (solving the system using *numpy.linalg.solve*), and another which compares the new strategies with the proposal of saving the LU factorization of KKT matrix to solve the system. Below it is shown the results obtained.



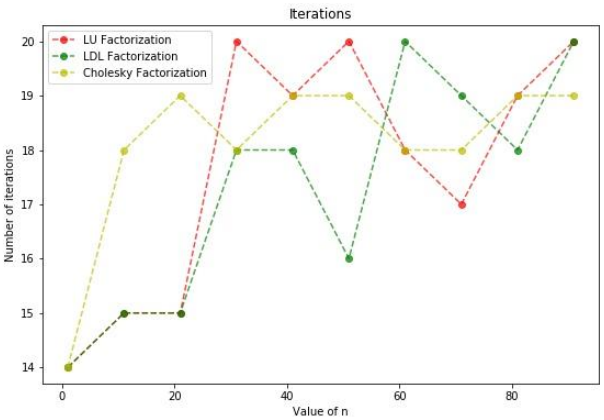
It can be appreciated that the initial method (General) needs more computational effort than the new strategies (LDL Factorization and Cholesky), and then the premise that the KKT system is solved more efficiently is accomplished.

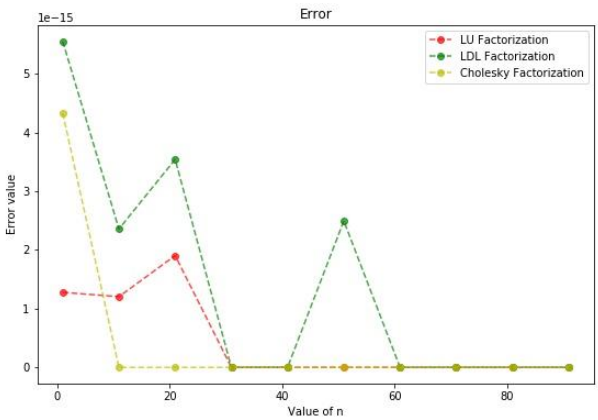


Whereas, if new strategies are compared with the method that saves LU factorization, the last one shows a good behaviour really similar to the strategy 1 (LDL Factorization). Given these results, it can be concluded that Cholesky is the method that performs fastest.

In addition, information about the number of iterations needed and the error of the solution is provided. Regarding the number of iterations, despite the three methods do not coincide for all the values of *n,* the difference is not so remarkable. On the other hand, taking a look at the values obtained for the error it can be appreciated that for many values of *n* (for those upper than 30) the error value is zero. This does not mean that the algorithms are more precise when the dimension *n* is higher; the explanation for these cases is that the difference between the solution found and the expected one is lower than the tolerance of the machine (typically 1e-16), and then the computer is not able to perceive the difference between both numbers.

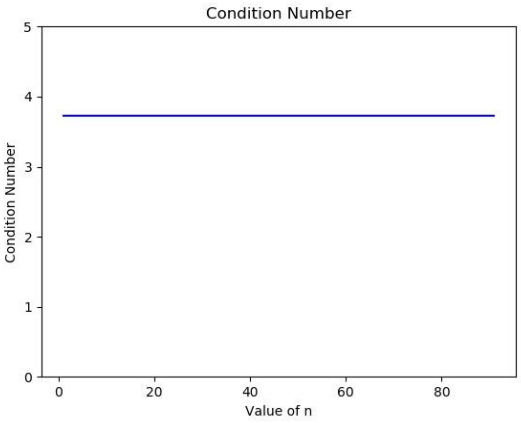
Run **C4_Iterations_Error_Comparison.py** to obtain the plots below.





Now it would be interesting to talk about the stability of the solutions found. First, the condition number is calculated for each value of *n,* and as it can be seen it is a constant value, 3.7259692503155892 **.**

Run **C4_Condition_Number.py**.



Condition number *K(A)* measures the accuracy of the computed solution. If small changes in the matrix A of our system can cause large changes in the solution, then the matrix is ill-conditioned. Otherwise, our matrix is well-conditioned. When *K(A)* is low, the matrix is well-conditioned, but when *K(A)* is high, the matrix is ill-conditioned.

Recalling the Backward Error Analysis Formula:

$$\frac{\|\delta x\|}{\|\hat{x}\|} \leq K(A)\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\|\|\hat{x}\|}\right)$$

The part inside the parenthesis is the relative error introduced by the input data, in the literature it is usually approximated by the precision of the machine $\varepsilon_{machine}$:

$$\frac{\|\delta x\|}{\|\hat{x}\|} \leq K(A)\varepsilon_{machine}$$

So, the relative changes in our solutions $\frac{\|\delta x\|}{\|\hat{x}\|}$ are bounded by the quantity $K(A)\varepsilon_{machine}$, aprox. 3.7256e-16 if we consider $\varepsilon_{machine}$ = 1e-16. In other words, two solutions of our system may differ in the 16th decimal number. If the condition number increased, the variation from one solution to the other would vary considerably, obtaining two very different solutions of the same system.

General Case (with equality and inequality constraints)

In this part it is asked to solve two problems for the general case. The configuration of each problem (**G, g, A, b, C** and **d**) consist on a collection of files available at the Campus Virtual.

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 = γ_0 = λ_0 = (1, ..., 1) for all problems.

To obtain the results for the first problem run **C5_General_Case_Problem_1.py**. The algorithm for this first general case finds a solution with f(x) = 11590.718119426763, which coincides with the value provided in the statement *f(x) = 1.15907181 x 10^4*.

Then, we can proceed to execute the code for the second problem **C5_General_Case_Problem_2.py**. In this second case the dimensions are ten times larger than before (**pr1:** n=100, p = 50, m = 200, **pr2:** n=1000, p = 500, m = 2000), and the time the algorithm takes to find the solution increases considerably. The solution found has a value f(x) = 1087511.5673215, which coincides with the value provided in the statement *f(x) = 1.08751157 x 10^6*.

If we wanted to know about the stability of the solutions found, we could compute the condition number for each problem, **C5_Condition_Numbers.py**.

The condition number for the problem 1 is 1246556.3476140387, and for the problem 2 is 8759727449.648342. Both are large numbers, which means that both system matrices are ill-conditioned. This leads us to infer that small changes in its input values may cause large changes in the computed solution.

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

Two codes have been implemented, one for each problem: **C6_General_Case_LDLT_Pr1.py** and **C6_General_Case_LDLT_Pr2.py**, and in both cases the algorithm fails because at some point the LDL^T factorization of the reduced matrix is singular and then the system can not be solved. In conclusion, for this problem, the LDL^T factorization without pivoting is not feasible.