Universitat de Barcelona

Numerical Linear Algebra

master in fundamental principles of data science

Project 1

Direct methods in optimization with constraints

DATA SCIENCE @ UNIVERSITAT DE BARCELONA

Author
Vladislav Nikolov Vasilev

Faculty of Mathematics and Computer Science

Academic year 2021-2022

Contents

1 Introduction

The main goal of this project is to study the basic numerical linear algebra behind optimization problems. In this case, we are going to consider a convex optimization problem which has equality and inequality constraints. The goal is to find a value of $x \in \mathbb{R}^n$ that solves the following problem:

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

This constrained minimization problem can be solved using Lagrange multipliers. In order to do so, we have to transform the inequality constraints into equality constraints. Therefore, we introduce the slack variable $s = C^Tx - d \in \mathbb{R}^m, s \geq 0$. The Lagrangian is given by the following expression:

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

The previous expression can be rewritten as:

$$\begin{aligned} Gx + g - A\gamma - C\lambda &= 0 \\ b - A^Tx &= 0 \\ s + d - C^Tx &= 0 \\ s_i\lambda_i &= 0, \quad i = 1, \dots, m \end{aligned} \tag{3}$$

To solve this problem, we are going to use the Newton's method. Additionally, each step of the method is going to have two correction substeps that that will help us stay in the feasible region of the problem.

T1: In order to solve this problem, let us define $z = (x, \gamma, \lambda, s)$ and $F : \mathbb{R}^N \to \mathbb{R}^N$, where $N = n + p + 2m$. The function F can be defined as follows:

$$F(z) = F(x, \gamma, \lambda, s) = (Gx + g - A\gamma - C\lambda, b - A^Tx, s + d - C^Tx, s_i\lambda_i)$$

Our goal is to solve $F(z) = 0$ using Newton's method. This involves computing a Newton step δ_z so that for a given point z_k we have that $z_{k+1} = z_k + \delta_z, \forall k \in \mathbb{N}$. Knowing that $F(z_{k+1}) = F(z_k) + J_F\delta_z$, we can see that this is equivalent to solving the system $J_F \delta_z = -F(z_k)$, where J_F is the Jacobian matrix. This matrix is defined as follows:

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

where I is a $m \times m$ identity matrix and S and Λ are $m \times m$ diagonal matrices containing the values of s and λ, respectively.

Thus, in order to obtain δ_z at each step, we have to solve a linear system of equations defined by the matrix M_{KKT} and the right hand vector $-F(z_k)$.

2 Solving the KKT system without inequalities

The first case that we are going to study is the KKT system without inequalities. This allows us to represent the matrix M_{KKT} as a 3×3 block matrix:

$$M_{\text{KKT}} = \begin{pmatrix} G & -C & 0 \\ -C^T & 0 & I \\ 0 & S & \Lambda \end{pmatrix} \tag{4}$$

There are some strategies that we can follow in order to solve the linear system of equations. First, we will start by discussing a *naive* approach, and later on we will present some more advanced strategies that will allow us to solve the linear system faster.

2.1 Naive approach

The first approach that we are going to discuss is the one that we have called *naive*. It is named this way because it is the first one that one can think of and is the most direct one. This method consists in solving the system using the `numpy.linalg.solve` function. This function uses one of `LAPACK`'s `gesv` routines, which basically solves the system by computing the LU factorization using GEPP under the hood.

C1, C2, C3: To solve this exercises, we have defined a function called `lu_solver` which can be found in the `kktsolvers/inequality.py` script. It uses the provided `Newton_method` function to compute the step-size correction and one more function called `solve_system` which computes $F(z_k)$.

As we already know, this method is quite slow, since it solves the system with $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops. Moreover, it is used twice per iteration of the Newton's method, which means that the factorization has to be computed two times. This implies that big sized problems will take quite some time until they are solved. There are some improvements that can be made, like computing the LU factorization just once and storing it. However, it might be a better idea to explore other alternatives.

Fortunately, we can make some changes to M_{MKKT} matrix that will allow us to solve the problem faster.

2.2 Other strategies

In this section we are going to discuss two strategies that we can use to speed up the Newton's method.

2.2.1 LDL^T factorization

The first strategy that we are going to talk about is the LDL^T factorization. Given a symmetric matrix A, we can express this matrix as $A = LDL^T$, where L is a unitriangular lower matrix and D is a diagonal matrix. This method is faster than the LU factorization, it takes up less space (we only need to store one lower triangular and one diagonal matrix) and is more stable in general.

T2.1: We can factorize our M_{KKT} matrix provided that it is symmetric. In order to obtain one, we can perform some manipulations on it. From (4) we can obtain the following linear system:

$$\begin{aligned} G\delta_x - C\delta_\lambda &= -r_L \\ -C^T\delta_x + I\delta_s &= -r_C \\ S\delta_\lambda + \Lambda\delta s &= -r_s \end{aligned} \tag{5}$$

Now, if we isolate δ_s from the last equation, we get:

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

Substituting in the second row we get:

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

This has the following matrix form:

$$\begin{pmatrix} G & -C \\ -C^T & -\Lambda^{-1}S \end{pmatrix}\begin{pmatrix} \delta_x \\ \delta_\lambda \end{pmatrix} = -\begin{pmatrix} r_L \\ r_C - \Lambda^{-1}r_s \end{pmatrix} \tag{6}$$

Since G is a symmetric positive definite matrix, C appears both in its normal forms and transposed, and the matrix product $-\Lambda^{-1}S$ is a diagonal (which is symmetric by definition), we can clearly see that this version of the M_{KKT} is also symmetric. Thus, the LDL^T factorization can be applied to it.

C4.1: In the `kktsolvers/inequality.py` script you can find the implementation of this exercise. The function is called `ldlt_solver`. It computes the LDL^T factorization of the 2×2 M_{KKT} block matrix and then solves an upper triangular equation system using the `scipy.linalg.solve_triangular` function, which is way faster than solving a whole system like we were doing before. Then, it divides the resulting vector by the diagonal matrix D, which has been implemented as an element-wise division between the result vector and the diagonal vector. Finally, it solves an upper triangular system, using this time L^T. This time, the factorization is only computed once and is stored so that it can be reused later, which allows us to save some time.

2.2.2 Cholesky factorization

The second strategy that we want to discuss is the Cholesky factorization. Given a symmetric positive definite matrix A, instead of computing its factorization as $A = LU$, we can compute it as $A = GG^T$, where G is a lower triangular matrix whose diagonal is positive. Recalling the LDL^T factorization, we had a lower unitriangular matrix L and a diagonal matrix D. Since A is SPD, its diagonal entries are positive. Thus, we can define $D = \text{diag}(\sqrt{d_1}, \dots, \sqrt{d_n})^2$, and from here we can derive that $G = L \times \text{diag}(\sqrt{d_1}, \dots, \sqrt{d_n})$.

The Cholesky factorization performs better than the LU since it takes half the time to be computed, takes less space in memory (only one lower triangular matrix has to be stored) and is more stable in general, since it can be computed without pivoting, which allows us to get better solutions.

T2.2: In order to apply the Cholesky factorization to our matrix, we have to perform some transformations so that it becomes symmetric and positive definite. Once again, we can obtain the following linear system from the expression (4):

$$\begin{aligned} G\delta_x - C\delta_\lambda &= -r_L \\ -C^T\delta_x + I\delta_s &= -r_C \\ S\delta_\lambda + \Lambda\delta s &= -r_s \end{aligned}$$

Now, if we isolate δ_s in the second row, we get the following expression:

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

Substituting δ_s in the third row we get:

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

Substituting δ_λ into the first row we get:

$$\begin{aligned} G\delta_x - C(S^{-1}(-r_s + \Lambda r_C) - S^{-1}\Lambda C^T\delta_x) &= -r_L \\ G\delta_x - CS^{-1}(-r_s + \Lambda r_C) + CS^{-1}\Lambda C^T\delta_x &= -r_L \\ G\delta_x + CS^{-1}\Lambda C^T\delta_x &= -r_L + CS^{-1}(-r_s + \Lambda r_C) \\ (G + CS^{-1}\Lambda C^T)\delta_x &= -r_L + CS^{-1}(-r_s + \Lambda r_C) \end{aligned}$$

Let us define $\hat{G} = G + CS^{-1}\Lambda C^T$ and $\hat{r} = -CS^{-1}(-r_s + \Lambda r_C)$. Substituting in the previous expression we obtain the following system:

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

We can apply the Cholesky factorization to the matrix $\hat{G}$ if it is SPD. We know that the matrix G is SPD. If the product $CS^{-1}\Lambda C^T$ is also SPD, then we would have that $\hat{G}$ is also SPD and it can be factorized using this method.

C4.2: The `cholesky_solver` function (found in the same file as the previous two) implements the Cholesky factorization and solves the system. The biggest difference regarding the previous ones is that here we don't use the M_{KKT} matrix. Instead, we compute $\hat{G}$ and then compute its Cholesky factorization and replace its value (since its only used to compute the factorization). This allows us to save some extra space because we don't have to save both the factorization matrices and the M_{KKT} matrix at the same time.

2.3 Experimentation

3 Solving the general KKT system

After getting acquainted with the KKT system without inequalities restrictions, it is time that we try to solve the full system.

Just as we did before, we are going to try first a naive approach based on LU factorization, and then we will try to modify the system so that we can apply another type of factorization to reduce the execution time.

3.1 Naive approach

3.2 LDL^T factorization

3.3 Experimentation

References

[1] Texto referencia
`https://url.referencia.com`