

The Alternating Method of Multipliers

From the past, with Vengeance

Operator Splitting Quadratic Programming

We will tackle our QP using the OSQP solver by Oxford University

OSQP is a modern solver for Quadratic Programs in the form:

$$\arg \min_x \left\{ \frac{1}{2} x^T P x + q^T x \mid l \leq A x \leq u \right\}$$

The solver:

- is very fast, especially for problems with sparse matrices
- is available under a (very permissive) Apache 2.0 license
- has API for many programming languages

The solver relies on the Alternating Direction Method of Multipliers (ADMM)

- ...Plus a bunch of clever "tricks" to improve speed
- Here we will discuss only the basic ADMM, to provide **an intuition**

The Alternating Direction Method of Multipliers

The ADMM solves numerical constrained optimization problems in the form:

$$\begin{aligned} & \operatorname{argmin} f(x) + g(z) \\ & \text{subject to: } Ax + Bz = c \end{aligned}$$

- Where f and g are assumed to be convex

The method relies on a so-called **augmented Lagrangian**

This is a reformulation where the constraints are turned into **penalty terms**:

$$\mathcal{L}_\rho(x, z, \lambda) = f(x) + g(z) + \lambda^T (Ax + Bz - c) + \frac{1}{2} \rho \|Ax + Bz - c\|_2^2$$

- The algorithm idea is to **optimize the augmented Lagrangian**
- ...And to encourage constraint satisfaction via the penalty terms
- In practice, this is done by adjusting the **multiplier vector λ**

The Alternating Direction Method of Multipliers

The ADMM operates as follows

We start from an initial assignment $\mathbf{x}^0, \mathbf{z}^0, \boldsymbol{\lambda}^0$, then we iterate:

$$\mathbf{x}^{k+1} = \operatorname{argmin}_{\mathbf{x}} \mathcal{L}_{\rho}(\mathbf{x}, \mathbf{z}^k, \boldsymbol{\lambda}^k)$$

$$\mathbf{z}^{k+1} = \operatorname{argmin}_{\mathbf{z}} \mathcal{L}_{\rho}(\mathbf{x}^{k+1}, \mathbf{z}, \boldsymbol{\lambda}^k)$$

$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \rho(\mathbf{A}\mathbf{x}^{k+1} + \mathbf{B}\mathbf{z}^{k+1} - \mathbf{c})$$

In other words:

- We keep everything fixed and we optimize over \mathbf{x} to obtain \mathbf{x}^{k+1}
- We replace \mathbf{x}^k with \mathbf{x}^{k+1} , keep everything fixed and optimize over \mathbf{z}
- Finally, we update the multiplier vector

The switch between \mathbf{x} and \mathbf{z} optimization is the "alternating" part

...While the use of the multipliers $\boldsymbol{\lambda}$ explains the rest of the name

Multiplier Update

Let's try to understand better the multiplier update

...Which consists in the rule:

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - c)$$

- The term $Ax^{k+1} + Bz^{k+1} - c$ is just the current constraint violation
- ...In particular both its **amount** and **direction**

Multiplier Update

Let's try to understand better the multiplier update

...Which consists in the rule:

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - c)$$

- The term $Ax^{k+1} + Bz^{k+1} - c$ is just the current constraint violation
- ...In particular both its **amount** and **direction**

If $(Ax^{k+1} + Bz^{k+1})_i > c_i$ for some constraint i :

- Then we **increase** the corresponding multiplier λ_i
- So that the penalty term $\lambda_i(Ax^{k+1} + Bz^{k+1} - c)_i$ grows
- This will push the next iteration to reduce the degree of violation

Multiplier Update

Let's try to understand better the multiplier update

...Which consists in the rule:

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - c)$$

- The term $Ax^{k+1} + Bz^{k+1} - c$ is just the current constraint violation
- ...In particular both its **amount** and **direction**

If $(Ax^{k+1} + Bz^{k+1})_i < c_i$ for some constraint i :

- Then we **decrease** the corresponding multiplier λ_i
- So that the penalty term $\lambda_i(Ax^{k+1} + Bz^{k+1} - c)_i$ grows (again)
- This will push the next iteration to reduce the degree of violation (again)

Multiplier Update

Let's try to understand better the multiplier update

...Which consists in the rule:

$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - c)$$

- The term $Ax^{k+1} + Bz^{k+1} - c$ is just the current constraint violation
- ...In particular both its **amount** and **direction**

If $(Ax^{k+1} + Bx^{k+1})_i = c_i$ for some constraint i :

- Then we **keep** the corresponding multiplier λ_i **as it is**
- The constraint is not violated, so there is nothing to do

Main Advantages of the Method

The method has two major advantages:

1) The x and z variables can be handled in isolation

- This results into simpler problems
- ...And in some cases enables massive parallelization

2) The ADMM converges under relatively mild conditions

- In the classical formulation, f and g need to be closed, proper, convex functions
 - They do not need to be differentiable
 - They can take the value $+\infty$
 - We will see why that matters in the next slides
- The second condition is that $\mathcal{L}_0(x, z, \lambda)$ should have a saddle point
 - This one is way trickier to check...

The full convergence proof can be found e.g. [here](#)

The ADMM and QP

That was the whole point, right?

QP Reformulation

Let's see these advantages at work on Quadratic Programs

We need to solve:

$$\operatorname{argmin}_x \left\{ \frac{1}{2} x^T P x + q^T x \mid l \leq A x \leq u \right\}$$

...Which we reformulate to:

$$\begin{aligned} & \operatorname{argmin} x^T P x + q^T x \\ & \text{subject to: } z = A x \\ & \quad l \leq z \leq u \end{aligned}$$

- We have introduced **a new variables z**
- ...And posted the inequality constraints over that

QP Reformulation

Then, we turn the inequality constraints into a function

$$\begin{aligned} & \operatorname{argmin} x^T P x + q^T x + \chi_{l \leq z \leq u}(z) \\ & \text{subject to: } z = A x \end{aligned}$$

Where:

- $\chi_{l \leq z \leq u}$ is the **characteristic function** of $l \leq z \leq u$
 - It's value is $+\infty$ when the constraint is violated and 0 elsewhere
 - In this case, it is non-differentiable, but closed, proper, and convex!
- $x^T P x + q^T x$ is our usual cost term
 - It is differentiable
 - ...And closed, proper, and convex if P is semi-definite positive

We can now proceed to apply the ADMM!

The ADMM Steps for QP

We need to start from a feasible x^0, z^0, λ^0 :

- That's easy, we get it by setting $\lambda^0 = 0, z^0 = l$, then solving $Ax^0 = l$

The x minimization step for $\hat{z} = z^k$ becomes:

$$\operatorname{argmin}_x x^T P x + q^T x + \chi_{l \leq z \leq u}(\hat{z}) + \lambda^T (\hat{z} - Ax) + \frac{1}{2} \rho \|\hat{z} - Ax\|_2^2$$

And then, since \hat{z} is fixed and feasible:

$$\operatorname{argmin} x^T P x + q^T x + \lambda^T (\hat{z} - Ax) + \frac{1}{2} \rho \|\hat{z} - Ax\|_2^2$$

This is a convex, differentiable, quadratic minimization problem

- It can be tackled via gradient descent
- ...Or by solving a linear system of equations

The ADMM Steps for QP

The z minimization step for $\hat{x} = x^{k+1}$ becomes

$$\operatorname{argmin}_z \hat{x}^T P \hat{x} + q^T \hat{x} + \chi_{l \leq z \leq u}(z) + \lambda^T (z - A \hat{x}) + \frac{1}{2} \rho \|z - A \hat{x}\|_2^2$$

Since \hat{x} is fixed, this can be reformulated as:

$$\begin{aligned} & \operatorname{argmin} \lambda^T z + \frac{1}{2} \rho \|z - A \hat{x}\|_2^2 \\ & \text{subject to: } l \leq z \leq u \end{aligned}$$

...And finally separated in to n problems (one per variable) in the form:

$$\operatorname{argmin}_{z_j} \left\{ \lambda_j z_j + \frac{1}{2} \rho (z_j - A_j \hat{x})^2 \mid l \leq z_j \leq u \right\}$$

- These are all very easy to solve

Some Considerations

We used the ADMM to break QP into a sequence of simpler problems

The method can be used in other clever ways:

- Optimization with non-differentiable regularizers
- Parallel training, by splitting examples into multiple problems
- ...And using constraints to reach a consensus

The ADMM is best used for convex problems

- Classical results are for convex problems only
- There are some (local) results for non-convex problems (e.g. [this one](#))
- ...But in practice it's less reliable

About the convergence pace

- It's very fast in the first iterations, but much slower later
- You can get high-quality solutions early, but reaching the optimum takes long
- All in all, it's best to use the ADMM as an approximate method