Augmented Lagrangian Method for Equality and Inequality Constrained Convex Quadratic Programs

Govind Chari

May 9, 2022

1 Introduction

Convex quadratic programs have wide applications from SpaceX's Falcon 9 landing algorithm [1,2], model predictive control for robots such as Boston Dynamic's Atlas Robot [3] and MIT's Cheetah robot [4], and portfolio optimization in finance. Thus it is important to have high speed solvers for quadratic programs that allow for real-time optimization. One such method that is appealing due to its ability to warm-start is the augmented Lagrangian method. An extension of this method, ADMM, is implemented in the commercial solver OSQP which is one of the fastest QP solvers available [5].

2 Augmented Lagrangian Theory

The augmented Lagrangian method was first introduced in 1969 by Magnus Hestenes [6]. In this original paper, only equality constraints were considered. Since then, the method has been extended to equality and inequality constraints. The augmented Lagrangian method is a primal-dual method, since estimates of the dual variables are updated after each iteration according to how much each constraint is being violated by.

The method that I implemented was from [7]. The idea of the augmented Lagrangian method is to transform a constrained optimization problem into an unconstrained problem by augmenting the Lagrangian with a quadratic penalty. After each unconstrained minimization of the augmented Lagrangian, the dual variables are updated and as the dual variables converge to the true dual solution, the argument that minimizes the augmented Lagrangian becomes the primal solution.

Our standard quadratic program can be written as

$$\begin{array}{ll} \min\limits_{x} & \frac{1}{2}x^TQx + q^Tx \\ \text{s.t.} & Ax = b \\ & Cx \preceq d \end{array}$$

Where $\preceq$ indicates element-wise inequalities, and $Q > 0$. We can then write the augmented Lagrangian as

$$\mathcal{L}_\rho(x, \tilde{\lambda}, \tilde{\mu}, \rho) = \frac{1}{2}x^TQx + q^Tx + \tilde{\lambda}^T(Ax - b) + \tilde{\mu}^T(Cx - d) + \frac{\rho}{2}(Ax - b)^T(Ax - b) + \frac{1}{2}(Cx - d)^T I_\rho (Cx - d)$$

Where I_ρ is a diagonal matrix with $I_\rho(i,i) = 0$ if $(Cx - d)_i < 0$ and $\mu_i = 0$. Otherwise $I_\rho(i,i) = \rho$. This structure of I_ρ is to ensure that a penalty is only applied when the inequality constraint is violated. The vectors $\tilde{\lambda}$ and $\tilde{\mu}$ are the estimated dual variables and ρ is the penalty term whose value dictates how strongly constraint violations are penalized. As $\rho \to \infty$ the minimizer x of the augmented Lagrangian is an exact solution to our QP. However, as ρ becomes very large, the Hessian of $\mathcal{L}_\rho$ becomes ill-conditioned. Luckily, if we are updating the dual variables after each iteration, we do not need make ρ very large for strict constraint satisfaction since the penalty is "offloaded" onto the dual variables.

The first four terms are the standard Lagrangian for quadratic programs with the estimated dual variables and the last two terms are quadratic penalty terms to penalize constraint violations.

To get some insight on how to update our dual variables, we will consider the stationarity condition of the Lagrangian and Augmented Lagrangian of our QP.

$$\nabla_x \mathcal{L}(x, \lambda, \mu) = Qx + q + A^T\lambda + C^T\mu = 0$$

$$\nabla_x \mathcal{L}_\rho(x, \tilde{\lambda}, \tilde{\mu}, \rho) = Qx + q + A^T(\tilde{\lambda} + \rho(Ax - b)) + C^T(\tilde{\mu} + I_\rho(Cx - d)) = 0$$

We can see that the terms $\tilde{\lambda} + \rho(Ax - b)$ and $\tilde{\mu} + I_\rho(Cx - d)$ act as our dual variables λ and μ and as we do more and more iterations, $\tilde{\lambda}$ and $\tilde{\mu}$ will converge to λ and μ respectively, and $Ax - b$ and $I_\rho(Cx - d)$ will converge to zero meaning we will satisfy our constraints. The general algorithm for this can be written as follows

Algorithm 1 Augmented Lagrangian Solver

```
Initialize:
    $x = 0,\ \tilde{\lambda} = 0,\ \tilde{\mu} = 0,\ \rho = 1$
while Not Converged do
    $x^{k+1} \leftarrow \underset{x}{\operatorname{argmin}}\ \mathcal{L}_\rho(x, \tilde{\lambda}^k, \tilde{\mu}^k, \rho)$
    $\tilde{\lambda}^{k+1} \leftarrow \tilde{\lambda}^k + \rho(Ax^{k+1} - b)$
    $\tilde{\mu}^{k+1} \leftarrow \tilde{\mu}^k + I_\rho(Cx^{k+1} - d)$
    Update $I_\rho$
    Increase $\rho$ (Optional)
    Check Convergence
end while
```

3 Implementation

My implementation of this algorithm can be found here. My naive implementation can be found in extras/qp.jl. This implementation of the algorithm is a very natural implementation. However, it is slow due to a large number of memory allocations. In Julia, the result of the standard matrix multiplication $b = Ax$ allocates memory for the product of Ax then copies the value into b. I also wrote a faster implementation with fewer memory allocations which is the the src/ directory. Memory allocations are avoided in the *solve*! method allocating a workspace at the time of problem instantiation and making use of broadcasting and the in-place matrix multiplication method *mul*!. The only memory allocation occurs in the backslash operation when using Newton's method to mimimize the augmented Lagrangian.

Additionally, in my implementation, I increased the penalty ρ by a factor of ten every ten iterations.

Additionally there are a set of unit tests in the test/ directory that ensure that the results from my implementation are accurate by comparing to the results of the solver OSQP. The main test case is a portfolio optimization problem of the following form

$$\begin{aligned} \min_x \quad & \tfrac{1}{2}x^T Q x + q^T x \\ \text{s.t.} \quad & \mathbf{1}^T x = 1 \\ & x \geq 0 \end{aligned}$$

where x is a vector that represents the proportion of money allocated to each asset, Q is the return covariance of the assets, and q contains is the mean return of each asset, and the constraints state that all money must be allocated and you cannot short any asset.

4 Future Work

After submitting this project, I plan to spend more time increasing the performance of the solver. I think the main place I can gain performance is by using an in-place factor-solve routine when solving the Newton system rather than using backslash which allocates memory.

5 References

1. L. Blackmore, Autonomous precision landing of space rockets, The Bridge on Frontiers of Engineering, 4 (2016), pp. 15–20.
2. Mattingley, J., Boyd, S. CVXGEN: a code generator for embedded convex optimization. Optim Eng 13, 1–27 (2012). https://doi.org/10.1007/s11081-011-9176-9
3. S. Feng, E. Whitman, X. Xinjilefu and C. G. Atkeson, "Optimization based full body control for the atlas robot," 2014 IEEE-RAS International Conference on Humanoid Robots, 2014, pp. 120-127, doi: 10.1109/HUMANOIDS.2014.7041347.
4. Carlo, Jared Wensing, Patrick Katz, Benjamin Bledt, Gerardo Kim, Sangbae. (2018). Dynamic Locomotion in the MIT Cheetah 3 Through Convex Model-Predictive Control. 1-9. 10.1109/IROS.2018.8594448.
5. Stellato, Bartolomeo, et al. "OSQP: An operator splitting solver for quadratic programs." Mathematical Programming Computation 12.4 (2020): 637-672.
6. Hestenes, Magnus R.. "Multiplier and gradient methods." Journal of Optimization Theory and Applications 4 (1969): 303-320.
7. Tracy, Kevin. "Augmented Lagrangian Tutorial" 16-745: Optimal Control, Carnegie Mellon University, 2 Feb. 2022. Class Handout.