Convex Optimization Solver

Yutong Zhou

December 24, 2024

1 Introduction

We are always making decisions. Whether to go to sleep now or 15 minutes later. Whether to invest some capital in this asset or in that asset. Whether the robot should move in this way or that way. Decision making is everywhere. Some problems are simple enough such that we can eyespot the best choice intuitively with minimal reasoning. However, many problems are too complex. There are too many factors to consider and we are concerned if the choice we made is optimal. Sometimes not selecting the best choice will bring an undesirable huge cost. Thus, we need math to help us make decisions efficiently and systematically, guaranteeing that the decision being made is the best we could achieve.

We need to first formulate the problem in mathematical language. Usually, we abstract out the key factors that contribute to the problem and then quantify them using real numbers. Then we use a bunch of inequalities and equalities to describe the limitations and relationships among these factors, which are called inequality and equality constraints. Finally, we design a reward function, or objective function, to quantify the quality of each decision. In this way, we project the problem into a high-dimensional real vector space, with each dimension corresponding to one of the factors concerned and each vector in the space corresponding to one possible decision. Decisions that satisfy all specified constraints are feasible, while those that violate any of the constraints are infeasible.

By making the best decision we are essentially looking into the feasible domain to find the vector that optimize the objective function. Whether to minimize or maximize the objective function depends on how we define it. For example, if the objective function represents the cost of each decision, then we are trying to minimize it. On the other hand, if the objective function represents the reward of every decision, then we are trying to maximize it.

As a result, in order to make decisions we convert decision making problems into mathematical optimization problems. One class of these optimization problems is called convex optimization problem, where the objective function and constraint functions are all convex. Such a nice property will make the problem easier to solve. Convex optimization is a deep field with strong mathematical background. If one converts the problem into a convex optimization problem, then the problem is basically solved. This article focuses on solving a subclass of convex optimization problem called linear program, where the objective function and constraint functions are all linear, by designing and implementing a numerical linear program solver that can be applied to arbitrary dimension. Convex optimization is a generalization of linear program. Thus, we can extend to solving an arbitrary convex problem with little effort if we had a way to solve linear program. Also, this article is assuming we are minimizing the objective function.

2 Application

Before diving into detail how to implement the solver, I would like to list out some of the applications of optimization. Note that often the case the transformed optimization problem is non-convex, but we could approximate it using a convex optimization problem or use similar method to find a local optimal solution.

- In Machine Learning, we train the model by converting the problem into an optimization problem in the space of parameters. We optimize the loss function and the resulting vector gives the optimal parameters for the model.
- Motion planning and trajectory generation are fundamental tasks in autonomous robotics. One way to generate a trajectory is by converting the problem into an optimization problem, where the problem lies in the space of the robot's state and the objective function is time or energy or whatever of our interest at our discretion.
- In portfolio optimization, we are trying to decide how to invest our money in order to gain the best return with as little risk as possible. In this case we are optimizing in the portfolio allocation space. The objective function would be the risk or variance of the return.

3 Implementation

In this section, we will dive into the process of designing and implementing a a linear program solver. A few trials and failures in the process will also be presented. The final solver should be able to be applied to arbitrary dimensions, considering both equality constraints and inequality constraints. The limitation of the final solver would be discussed in the Result section.

3.1 Initialization

In Homework 2 Implementation 3, we explored the implementation of a linear program solver using the Barrier Method. It is a powerful solver in that it is fast and can be applied to any convex problem. Thanks to the virtue of the Barrier Method, we have a clean and beautiful duality gap that control the error of the solution and thus provide us with a stopping criteria. In the Appendix I will show you my previous work deriving this beautiful duality gap. But it also has a few drawbacks that need improvement:

- It does not consider equality constraints.
- It requires the initial guess to be feasible.
- It only works in 2D.

Despite these limitations, this naive solver is a good start, from which we could resume and extend it to our desired goal. In the next few subsections I will fix these limitations one by one.

3.2 Equality Constraints

In this subsection we would assume the linear program is in 2D and the initial guess satisfies all inequality constraints. The main focus of this section is how to mold equality constraint

into the problem. Let's play with a toy example that is easy to visualize and then generalize it. Consider the linear program:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Gx \leq h, \\ & Ax = b \end{aligned} \tag{1}$$

where $c = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $G = \begin{bmatrix} 0.7071 & 0.7071 \\ -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ -0.7071 & -0.7071 \end{bmatrix}$, $h = \begin{bmatrix} 1.5 \\ 1.5 \\ 1 \\ 1 \end{bmatrix}$, $A = \begin{bmatrix} 1 & -2 \end{bmatrix}$, $b = \begin{bmatrix} 0.7 \end{bmatrix}$.
Note that in 2D we could have at most one inequality constraint. More than one inequality constraint will make the problem trivial.

3.2.1 An intuition that does not work: Convert the equality into inequality

Currently we do not have any tool to confine the solution on the equality constraints. However, we do have the tool, the Barrier Method, to restrict the solution inside the inequality constraints. What if we could convert the equality constraints into inequality constraints, then it seems that we do not need to invent any new tools. The conversion is actually easy:

$$Ax = b \leftrightarrow Ax \leq b \wedge -Ax \leq -b \tag{2}$$

This conversion is exciting at first glance, but the nature of the Barrier Method will prevent us from going further. First of all, this conversion will almost certainly make the initial guess infeasible, because the likelihood of making an initial guess that satisfies the equality constraint is zero. As long as the initial guess is not lying on the equality constraint, one of the transformed inequality constraints must be violated. Even when the initial guess is lying on the equality constraint, and as a result after conversion every inequality constraints are satisfied and thus the initial guess is feasible, the solver will still collapse. Lying on any inequality constraints will lead to the logarithm inside the Barrier Method evaluate at $x = 0$, which is undefined. As a result, the Barrier Method does not allow the point to lie on any inequality bound. We need some other way to enforce the equality constraint.

3.2.2 Quadratic Augmentation

Take a look at the equality constraint in (1) and move the right hand side to the left, we get:

$$Ax - b = 0 \tag{3}$$

The intuition is when $Ax - b = 0$, we are satisfied. When $Ax - b \neq 0$, we are unhappy and have a strong incentive to make $||Ax - b||$ as small as possible. If we become a bit flexible and allow $||Ax - b|| < \epsilon$ for some small $\epsilon > 0$, then we can transform the problem and get rid of the equality constraint by augmenting the objective function with a quadratic penalty term:

$$\begin{aligned} \text{minimize} \quad & c^T x + k_{cons}(Ax - b)^T(Ax - b) \\ \text{subject to} \quad & Gx \leq h, \end{aligned} \tag{4}$$

where k_{cons} is some large real number that determine the severity of the penalty. A larger k_{cons} will result in a stronger enforcement of the equality constraint.
Now that we have modified the objective function, we need to recompute it's gradient and hessian:

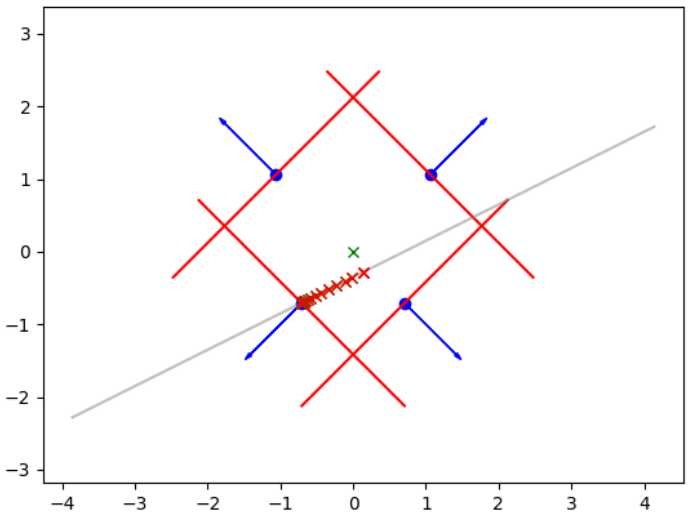

Figure 1: The solving process of the solver. Later we will call this process Phase2. The grey line represents the equality constraint. Note that the solver enforce the equality constraint pretty well during the solving process due to a relative large k_{cons}.

Let $f(x) = c^T x + k_{cons}(Ax - b)^T(Ax - b)$, then:

$$\nabla f = c + 2k_{cons}(Ax - b)^T A \tag{5}$$

$$\nabla^2 f = 2k_{cons}A^T A \tag{6}$$

Finally, we implement (5) and (6) into the code. The way I implement the code is in another form but equivalent. Note that when we derive (5) and (6), we are not making any assumptions on the dimensionality of the problem, thus it could be generalized to any dimension without further effort.

Let's play with the toy example (1) to visualize and verify the solving process. Set the initial guess $x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and the penalty constant $k_{cons} = 1000$. The visualized solving process is illustrated in Figure 1. As we can see the solver enforce the equality constraint pretty well during the solving process due to a relative large k_{cons}.

3.3 Phase1-Phase2

We just developed a decent way to handle the equality constraint by molding the equality constraint into the objective function, where the problem is transformed into an optimization problem that only contains inequality constraints. In this subsection we will assume the problem has already been transformed into (4). Under this assumption, when we talk about feasibility, we only care about those inequality constraints.

Now Let's shift gear and focus on how to deal with the infeasible initial guess. When the structure of the problem is simple and there are few inequality constraints, we could tell a feasible initial guess manually. However, when it comes to a higher dimensional problem or the number of inequality constraints increases, coming up with a feasible initial guess becomes

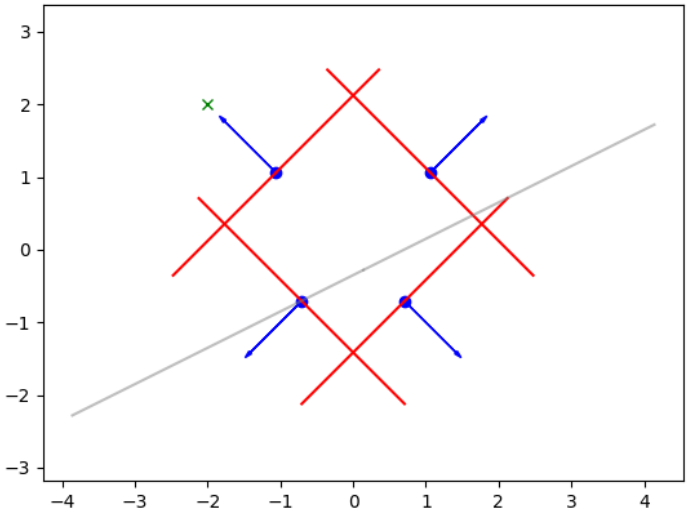

Figure 2: An infeasible point (Drew in green). Only the inequality constraint on the top left corner is violated, while the other inequality constraints are satisfied

a pain. As a result, we need a systematic way to generate a feasible initial guess. I call the process of finding a feasible initial guess Phase1. After having found a feasible initial guess, the upcoming solving process is called Phase2. Phase2 is just a renaming of the solving process and is already done in the previous section. The main focus of this section is Phase1.

3.3.1 Why an infeasible initial guess is not solvable for the solver?

Again it's due to the nature of the underlying logarithm inside the Barrier Method. An infeasible point will make the logarithm be evaluated at a negative real value, which would raise undefined behavior.

3.3.2 Phase1

An infeasible point lies outside the convex hull enclosed by the hyperplanes defined by the inequality constraints. All we care about during Phase1 is to push the point into the convex hull, i.e. the feasible domain. As a result, we can ignore the objective function tentatively.

Observe that an infeasible point is not entirely infeasible: It will at least satisfy some of the inequality constraints, while violating others. The desired consequence of the push should make the point go through those violated hyperplanes, while remain satisfying those hyperplanes that were already satisfied. For those hyperplanes (or inequality constraints) that are already satisfied, we could reuse the logarithm barrier to prevent the point from escaping. While for those hyperplanes that are violated, I designed a mechanism called linear pusher.

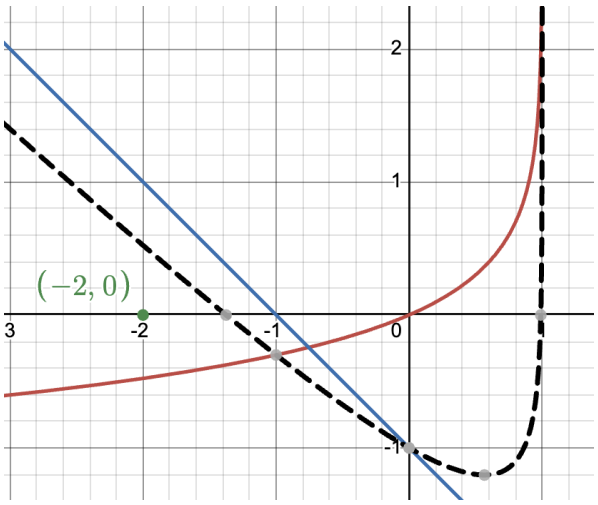

Figure 3: Visualization of the barrier and the linear pusher. The feasible domain is $[-1, 1]$. The blue line is the linear pusher. The red curve is the barrier. The black dashed line is $-\nabla f(x)$, which is the summation of the blue and the red. The green point is the infeasible initial guess. The blue line is providing a slope such that the green point will slide to the right until reaching the equilibrium (the critical point of the black dashed line), which is within the feasible domain.

Logarithm Barrier and Linear Pusher

Let's play with a toy example to illustrate:

$$\begin{aligned} &\text{minimize} \quad 0 \\ &\text{subject to} \quad x \leq 1, \\ &\qquad\qquad -x \leq 1 \end{aligned} \tag{7}$$

Note that the objective function in this toy example is set to 0, which we make it trivial deliberately. Assume we have an infeasible initial guess $x = -2$. In this case, the first inequality constraint is satisfied while the second inequality constraint is violated.

Recall when we were implementing the Barrier Method, we were essentially transforming a constrained optimization problem into an unconstrained optimization problem. Here we will adopt the same idea, by molding inequality constraints into the objective function.
For the first inequality constraint that is satisfied, we will construct a barrier as used to. This barrier serve as a wall to prevent the point from escaping:

$$x \leq 1 \rightarrow x - 1 \leq 0 \rightarrow f(x) = x - 1 \rightarrow b(x) = -log(-f(x)) = -log(1 - x) \tag{8}$$

For the second inequality constraint that is violated, we will construct a linear pusher as follow. The linear pusher will provide a gradient that serve as a incentive for the point to move toward the feasible domain:

$$-x \leq 1 \rightarrow -x - 1 \leq 0 \rightarrow p(x) = -x - 1 \tag{9}$$

Adding $b(x)$ in (8) and $p(x)$ in (9) to the objective function and transform (7) into unconstrained one:

$$\text{minimize} \quad -log(1 - x) - x - 1 \tag{10}$$

Let $f(x) = -log(1-x) - x - 1$, then

$$-\nabla f = -\frac{1}{x-1} + 1 \tag{11}$$

Because we are minimizing the objective function, $-\nabla f$ will point to the direction where $f(x)$ is decreasing the fastest. Note that $f(x) = b(x) + p(x)$. By decreasing $f(x)$ we are more or less decreasing $p(x)$. Thus we are trying to push the point towards the direction such that $p(x) \leq 0$, approaching our intention to satisfy the violated inequality constraint.

We can interpret this $-\nabla f$ as the force imposed on the point. The first term $-\frac{1}{x-1}$ is the force imposed by the barrier. It's obvious that this force is preventing the point from passing the boundary x = 1. The second term +1 is the force imposed by the linear pusher, it is a constant force that will keep pushing the point towards the feasible domain. The visualization of this process is illustrated in Figure 3. The reason I name it linear pusher is because the force constructed this way is always constant.

Now we could generalize this toy example. Consider

$$\begin{aligned} &\text{minimize} \quad 0 \\ &\text{subject to} \quad Gx \leq h, \end{aligned} \tag{12}$$

And an infeasible initial guess x_0. Still we are ignoring the objective function, only considering the inequality constraints. Define two sets of inequality constraints: A and B, where A is the set of those inequality constraints that are satisfied by x_0, while B is the set of those inequality constraints that are not satisfied.

For those inequality constraints in set A, we construct barriers. For those inequality constraints in set B, we construct linear pushers. Thus, we can transform the constrained optimization problem (12) into the unconstrained optimization problem:

$$\text{minimize} \quad \sum_{A} b(x) + \sum_{B} p(x) \tag{13}$$

Note that the initial infeasible point become feasible after transformation.
For implementation detail: We first need a function to tell which inequality constraint is violated while which is satisfied. Then we need a for loop to iterate through each inequality constraint and construct either linear pusher or barrier correspondingly and append them to the objective function. Then we minimize the objective function using the Barrier Method. The solution should be a point that is feasible to the original problem, which we can pass to Phase2 as a feasible initial guess.

An example of Phase1 process is illustrated in Figure 4. As I tried more experiments, it turned out Phase1 with linear pusher does not always success. In the Results section I will discuss the limitation of the linear pusher and potential method to mitigate.

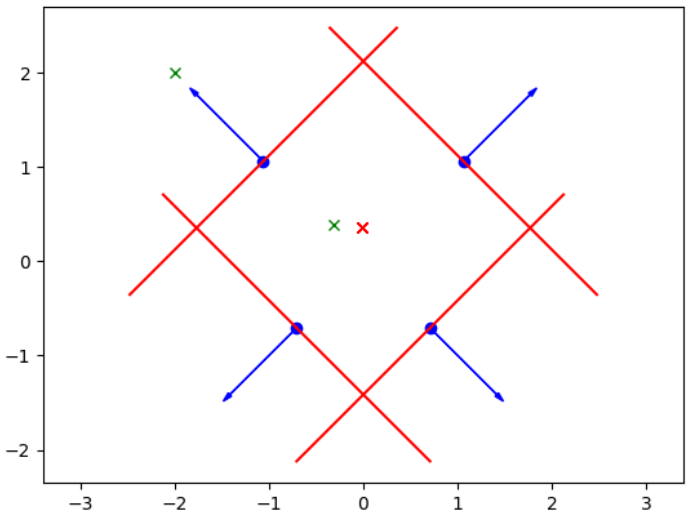

Figure 4: The green point on the top left corner outside the boundary is the initial guess. Only the top left inequality constraint is violated. The final solution of Phase1 is the red point. As it evolve the linear pusher is pushing the point towards the feasible domain while the barriers are preventing the point from escaping.

3.3.3 Combining Phase1 and Phase2

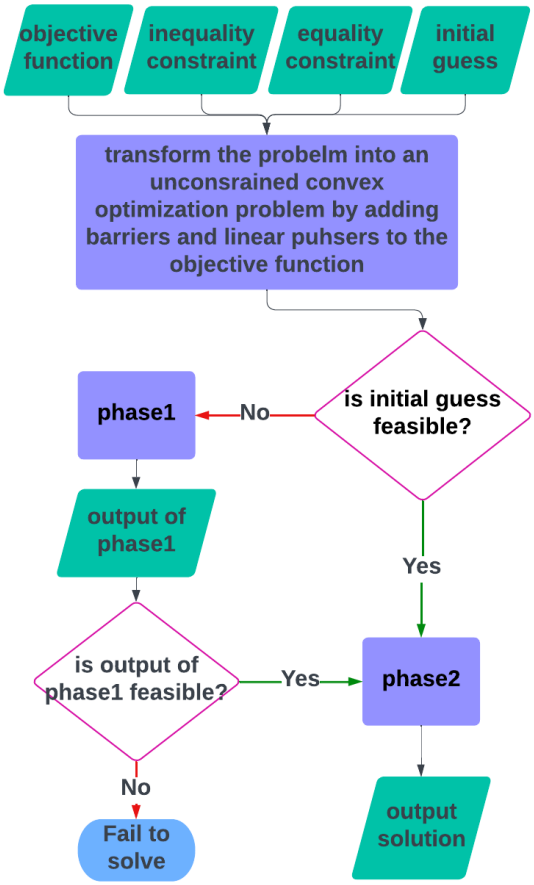

Figure 5: Block diagram of the full solve process combining Phase1 and Phase2.

So far we have successfully implemented Phase1 and Phase2 individually. Now we need to combine these two steps to finish the solver. It's a bit more than simply concatenating two parts of code, recall Phase1 is not always successful. Moreover, if the initial guess happen to be feasible, then we don't need Phase1 at all. Thus, before and after Phase1, we need to check if current initial guess is feasible or not, then we decide whether to keep going or raise an alert and terminate the solving process.

4 Results

In this section I will first showcase two examples of the full solve process. One 2D with visualization and one 6D with validation from an existing solver. Then I will discuss the limitation of the linear pusher in Phase1 and potential methods to mitigate.

In all examples the convex optimization problem will be the linear program:

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Gx \leq h, \\ & && Ax = b \end{aligned} \tag{14}$$

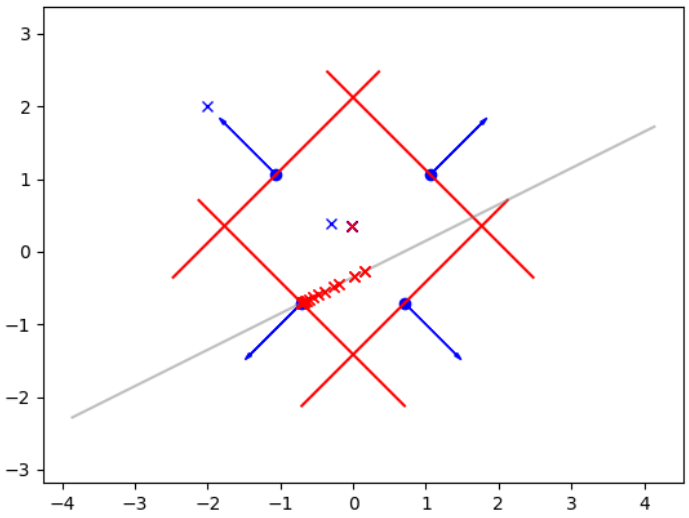

Figure 6: Example 1. The blue point in the top left corner is the infeasible initial guess. The blue points are the trajectory during Phase1. The red points are the trajectory during Phase2.

4.1 Example 1

Let $c = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $G = \begin{bmatrix} 0.7071 & 0.7071 \\ -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ -0.7071 & -0.7071 \end{bmatrix}$, $h = \begin{bmatrix} 1.5 \\ 1.5 \\ 1 \\ 1 \end{bmatrix}$, $A = \begin{bmatrix} 1 & -2 \end{bmatrix}$, $b = \begin{bmatrix} 0.7 \end{bmatrix}$.

With infeasible initial guess $x_0 = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$.

The solution by our solver is $\begin{bmatrix} -0.708533 \\ -0.704350 \end{bmatrix}$. The solution by the CVX solver is $\begin{bmatrix} -0.70948475 \\ -0.70474237 \end{bmatrix}$.

The visualization of the solving process is illustrated in Figure 6.

4.2 Example 2

Let $c = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 0 \\ 1 \\ -2 \end{bmatrix}$, $G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$, $h = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{bmatrix}$, $A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & 1 & 1 \end{bmatrix}$,

$b = \begin{bmatrix} 0 \\ 0.2 \end{bmatrix}$. With infeasible initial guess $x_0 = \begin{bmatrix} 4 \\ 4 \\ 4 \\ 4 \\ 4 \\ 4 \end{bmatrix}$.

The solution by our solver is $\begin{bmatrix} -1.9994288 \\ -0.10024999 \\ -1.99971439 \\ 1.99942883 \\ 0.09965476 \\ 1.99980959 \end{bmatrix}$. The solution by the CVX solver is $\begin{bmatrix} -2. \\ -0.1 \\ -2. \\ 2. \\ 0.1 \\ 2. \end{bmatrix}$.

4.3 Limitation of the linear pusher

Consider the case: $c = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $G = \begin{bmatrix} 0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ -1 & 0 \end{bmatrix}$, $h = \begin{bmatrix} 1.5 \\ 1 \\ -1 \end{bmatrix}$, $A = \begin{bmatrix} 2 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 \end{bmatrix}$.

With infeasible initial guess $x_0 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$.

It looks like a simple and ordinary linear program, but unfortunately our solver will fail. If we take a look at the visualization of the Phase1 process in Figure 7, we could spot where the problem lies: During Phase1 the solver is converging outside the inequality constraints! As a result Phase1 would fail to provide a feasible point for Phase2 to start with.

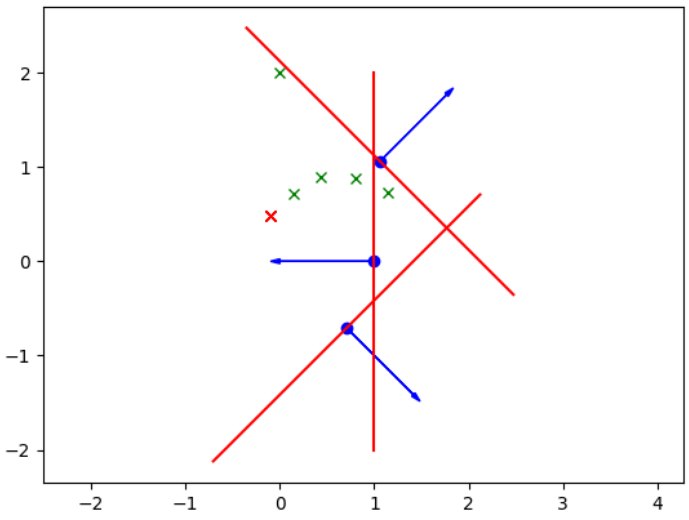

Figure 7: The green point on the top left corner is the infeasible initial guess. The red point is the solution given by Phase1. The solver is converging outside the inequality constraints! As a result Phase1 would fail to provide a feasible point for Phase2 to start with.

To make it clear, the solver is implemented correctly. It is actually generating the solution as required: the minimum point of the objective function. The solution by our solver matches with wolframAlpha, see Figure 8. In both case the minimum point lies outside the inequality constraints.

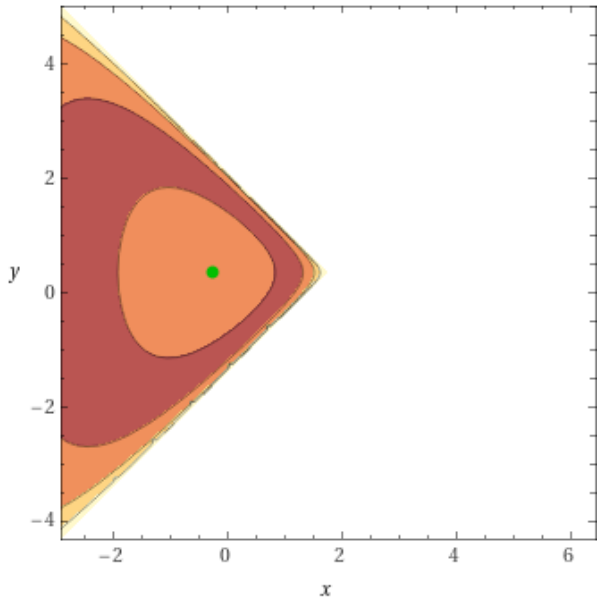

Figure 8: The contour map of the objective function during Phase1 by WolframAlpha. The green point, which is the solution, lies to the left of the inequality constraints $x = 1$, which is consistant with our solver.

In this special case we can resolve this issue. Observe that in Figure 7, during the iteration

of the Barrier Method, there is one green point that successfully enters the feasible domain. Thus, we can design a dynamic flag that checks, during each iteration of the Barrier Method, if the current point has satisfied any inequality constraints that were violated previously. If yes, then we transform the corresponding linear pushers into barriers to prevent the point from escaping those newly satisfied inequality constraints again. After implementing this idea into the solver, we can resolve this special case, see Figure 9

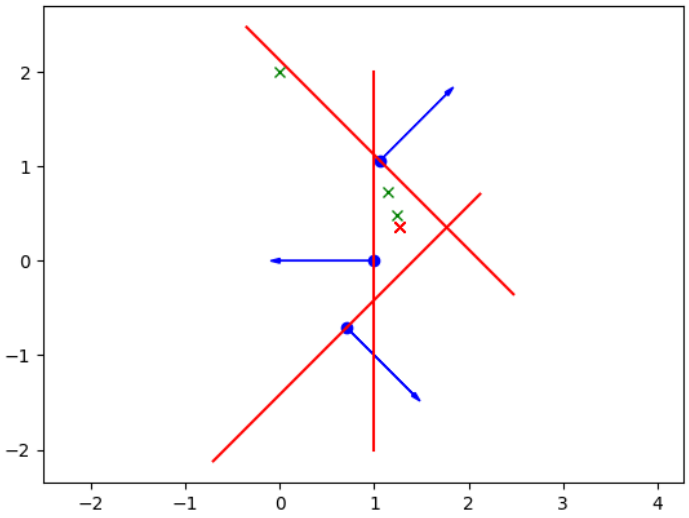

Figure 9: Phase1 with dynamic flag. The green point on the top left corner is the infeasible initial guess. The red point is the solution of Phase1. Once the point goes to the right of the vertical inequality constraint, instead of keeping the linear puhser, we use a barrier instead to prevent the point from escaping again.

As a result, for the full solve process of this problem:
The solution by our solver is $\begin{bmatrix} 1.1370231 \\ -0.27454333 \end{bmatrix}$. The solution by the CVX solver is $\begin{bmatrix} 1.13807571 \\ -0.27615142 \end{bmatrix}$.
The visualization of the solving process is illustrated in Figure 10.

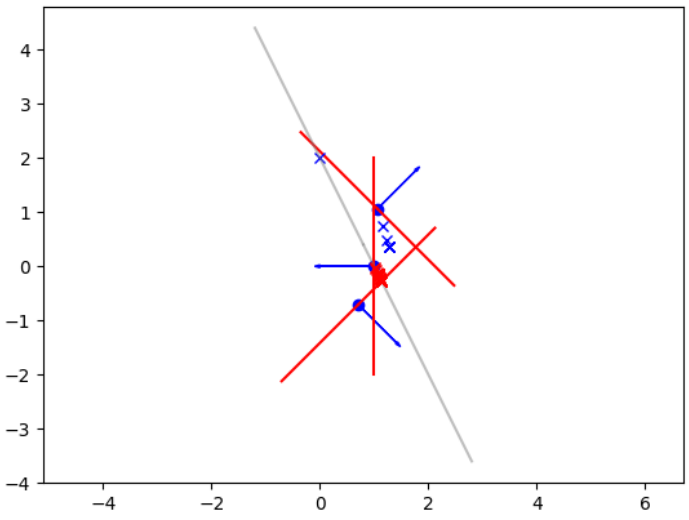

Figure 10: Full solve process after implementing the dynamic flag. The blue point in the top left corner is the infeasible initial guess. The blue points are the trajectory during Phase1. The red points are the trajectory during Phase2.

However, the dynamic flag only mitigates this issue. If we set the vertical inequality constraint to be $x \geq 1.5$, then in all iterations the point never enter the feasible domain and the dynamic flag can't do anything at all.

The fundamental reason is: Sometimes the linear pusher we construct is not strong enough. It merely exerts a constant force to the point. Especially when the area/volume enclosed by the inequality constraints is too small and the effect of the barrier could not be ignored, the linear pusher will fail to provide a sufficient force to offset the force from the barriers.

Instead of using a linear pusher, we could use an exponential pusher. Recall the way we construct the linear pusher in (9). Similarly, we could construct an exponential pusher:

$$-x \leq 1 \rightarrow -x - 1 \leq 0 \rightarrow p(x) = e^{k_p(-x-1)} \tag{15}$$

Where $k_p \propto$ the area/volume enclosed by the inequality constraints.

The exponential pusher would provide a much stronger force than the linear pusher and it could be anticipated that it is going to converge faster during Phase1 process. However, implementing this exponential pusher will require another function to calculate the area/volume enclosed by the inequality constraints. Also, we need to fine tune the parameter k_p.

5 Appendix: Deriving the Duality Gap

In this section I will go through the process of deriving the duality gap of the Barrier Method.

the Lagrangian for the centering problem show be

$L(\vec{x}, \vec{v}) = f(\vec{x}) + (A\vec{x} - \vec{b})^T \vec{v}$, $\vec{v} \in \mathbb{R}^p$, p is the number of equality contraints

note that the inequality contraints are implicitly included in $f(x)$

define $\vec{x}^*(t)$ as the primal solution to the centering problem

assume $\vec{x}^*(t)$ is strictly feasible,

according to KKT condition.

there exist a $\hat{v} \in \mathbb{R}^p$, p is the number of equality contraints

such that:

$$\nabla_{\vec{x}} L(\vec{x}^*(t), \hat{v}) = \nabla_{\vec{x}} L(\vec{x}, \hat{v}) \Big|_{x = x^*(t)}$$

$$= \nabla_{\vec{x}} (f(\vec{x}) + (A\vec{x} - \vec{b})^T \hat{v}) \Big|_{\vec{x} = \vec{x}^*(t)}$$

$$= \nabla_{\vec{x}} f(\vec{x}) \Big|_{x = \vec{x}^*(t)} + \nabla_{\vec{x}} \left[(A\vec{x} - \vec{b})^T \cdot \hat{v} \right] \Big|_{x = \vec{x}^*(t)}$$

$$= \nabla_x f(x^*(t)) + A^T \hat{v} \quad \cdots\cdots \textcircled{*}$$

$$\nabla_x f(x^*(t)) = \nabla_x \left[t f_0(x) - \sum_{i=1}^{m} \log(-f_i(x)) \right] \Big|_{x = x^*(t)}$$

$$= t \nabla_x f_0(x) + \sum_{i=1}^{m} \frac{1}{-f_i(x)} \nabla_x f_i(x) \Big|_{x = x^*(t)}$$

so ⑧ : $t \nabla_{\vec{x}} f_0(\vec{x^*(t)}) + \sum_{i=1}^{m} \frac{1}{-f_i(\vec{x^*(t)})} \cdot \nabla_{\vec{x}} f_i(\vec{x^*(t)}) + A^T \hat{v} = 0$

$\rightarrow \nabla_{\vec{x}} f_0(\vec{x^*(t)}) + \sum_{i=1}^{m} \frac{1}{-t \cdot f_i(\vec{x^*(t)})} \cdot \nabla_{\vec{x}} f_i(\vec{x^*(t)}) + A^T \cdot \frac{\hat{v}}{t} = 0$ ⑧⑧

define $\lambda_i^*(t) = \frac{1}{-t \cdot f_i(\vec{x^*(t)})}$, $\vec{v}^*(t) = \frac{\hat{v}}{t}$

because $\vec{x^*(t)}$ is strictly feasable, we have $f_i(\vec{x^*(t)}) < 0$

$\rightarrow \lambda_i^*(t) > 0$

go back to the original convext optimization problem

we construct the Lagrangian

$L(\vec{x}, \vec{\lambda}, \vec{v}) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(\vec{x}) + \vec{v}^T \cdot (A\vec{x} - \vec{b})$

not that $\vec{x}^*(t)$ minimize $L(\vec{x}, \vec{\lambda}, \vec{v})$ when

$\vec{\lambda} = \vec{\lambda}^*(t)$ and $\vec{v} = \vec{v}^*(t)$, from ⑧⑧

this menus $g(\vec{\lambda}^*(t), \vec{v}^*(t)) = \inf_x L(\vec{x}, \vec{\lambda}^*, \vec{v}^*)$

is finite and $(\vec{\lambda^*(t)}, \vec{v}^*(t))$ is dual feasible

so $g(\vec{\lambda}^*(t), \vec{v}^*(t)) = L(\vec{x}^*(t), \vec{\lambda}^*(t), \vec{v}^*(t))$

$= f_0(\vec{x}^*(t)) + \sum_{i=1}^{m} \lambda_i^*(t) f_i(\vec{x}^*(t)) + \vec{v}^*(t)^T (A\vec{x}^*(t) - \vec{b})$

$$= f_0(\bar{x}^*(t+1)) + \sum_{i=1}^{m} -\frac{f_i(\bar{x}^*(t))}{t\, f_i(\bar{x}^*(t+1))} + 0$$

$$= f_0(\bar{x}^*(t+1)) - \frac{m}{t}$$

so the duality gap for $\bar{x}^*(t)$, $\bar{\lambda}^*(t)$, $\bar{v}^*(t)$ should be

$$f_0(\bar{x}^*(t)) - g(\bar{\lambda}^*(t), \bar{v}^*(t)) = \frac{m}{t}$$

where m is number of inequality functions and t is the parameter "optimization force increment"