# 6326 Final Report

Zach Rosenof (zer2), Mitch Perry (mop25), Mike Sosa (mcs348)

May 10th, 2017

## 1 Introduction

In 2015, James Renegar published a framework for transforming convex conic optimization problems into simpler convex formulations to solve with subgradient methods. This framework could be useful for solving certain problems more easily, particularly high-dimensional ones which traditional conic solvers struggle with. As of yet, it has not been fully implemented in any coding language.

We have implemented the method for the specific case of semi-definite programs (SDP's). Our code is not optimized to the level of commercial solvers, but is reasonably fast and accurate. We also used our program to investigate some properties of Renegar's method which are not explored in his paper. The method calls for the choice of some number z that is dependent on the input of the SDP. Renegar's paper gives a condition that this value z must meet, but besides this the choice of z is arbitrary. We observed how different strategies for choosing z affected the performance of our solver in an attempt to posit a rule of thumb for the best choice of z.

## 2 General Theory Background

We are concerned with general conic programs, of the form

$$\min\ c \cdot x$$

$$\text{s.t. } x \in \text{Affine}$$

$$x \in K$$

Where $\cdot$ is some inner product for the relevant space - generally the dot product for vectors and trace for matrices. We first define a distinguished direction $e$, which can be any feasible solution to the problem above.

With this distinguished direction $e$, we define the function $\lambda_{\min}(x)$:

$$\lambda_{\min}(x) := \inf\{\lambda : x - \lambda e \notin K\}$$

The intuition for this function is straightforward. From a particular x, we add or subtract the distinguished direction until we get to the border of the cone. The number of times we must add or subtract the distinguished direction is the value of $\lambda_{\min}(x)$ (it is negative if we are adding, positive if we subtract).

Next, we find some z such that $z < c \cdot e$. With this z, we define the affine space $\text{Affine}_z = \{x : \ x \in \text{Affine}, \ c \cdot x = z\}$. Now that we have $\lambda_{\min}(x)$ and $\text{Affine}_\text{Z}$, we can write a new problem.

$$\max \lambda_{\min}(x)$$

$$\text{s.t. } x \in \text{Affine}_z$$

Renegar shows that solving this problem is equivalent to solving the original [1], and provides a way to convert between a solution of one to the solution of the other. He also shows that, unlike the original conic program, this formulation can be solved using supgradient methods. Our program uses this framework to solve SDPs.

# 3 Theory Applied to Semi-Definite Programs

## 3.1 Case When Identity is Feasible

In the special case when the identity matrix ($I$) is a feasible solution to the given SDP, Renegar's method gives a particularly simple transformed problem. We choose $I$ as the distinguished direction, and with this choice the function $\lambda_{\min}(x)$ becomes the minimum eigenvalue of $x$ [1]. This is good, because the minimum eigenvalue function is well-understood and has a relatively simple expression for its supgradient. Then all we have to do is choose some scalar z less than $\text{tr}(c \cdot e) = \text{tr}(c)$, and we have a program which supgradient methods can easily be applied to.

## 3.2 Converting Semi-Definite Programs

The identity matrix is only feasible in the case when it happens to be within the given affine space, which is not guaranteed. So we can't apply the minimum eigenvalue approach to every SDP. Fortunately, there is a way to convert a general SDP to one for which $I$ is feasible. Consider an SDP of the form

$$\begin{aligned} &\min C \cdot X \\ &\text{s.t. } A_1 \cdot X = b_1 \\ &A_2 \cdot X = b_2 \\ &\vdots \\ &A_m \cdot X = b_m \\ &X \in K \end{aligned} \tag{P}$$

Where $K$ is the cone of positive semidefinite matrices and $C$, $A_1$, $A_2$, ..., $A_m$ are symmetric. We would like to find a corresponding SDP whose solution set has a 1:1 mapping to the problem given above and whose set of optimal solutions has a 1:1 mapping to the problem above. The new SDP should have the identity matrix as a feasible solution as well.

Let's look at some feasible solution $E$ to the given SDP. Consider the eigenvalue decomposition of $E$. Here $E = Q\Lambda Q^T$, where $Q$ orthogonal and $\Lambda$ is the matrix containing the eigenvalues of $E$

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \dots & \vdots \\ 0 & \dots & \dots & \dots & \lambda_n \end{bmatrix}$$

Since $E \succ 0$, we know that it has a unique square root, denoted $E^{1/2}$, and this square root is

$$E^{1/2} = Q\Lambda^{1/2}Q^T = Q \begin{bmatrix} \sqrt{\lambda_1} & 0 & 0 & \dots & 0 \\ 0 & \sqrt{\lambda_2} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \dots & \vdots \\ 0 & \dots & \dots & \dots & \sqrt{\lambda_n} \end{bmatrix} Q^T$$

We see that we can use $E^{1/2}$ to formulate a new SDP that's equivalent to the original problem. Let us replace $C$ with $C' = E^{\frac{1}{2}}CE^{\frac{1}{2}}$, $A_i$ with $A_i' = E^{\frac{1}{2}}A_iE^{\frac{1}{2}}$ and $X$ with $X' = E^{-\frac{1}{2}}XE^{-\frac{1}{2}}$.

We hope to show that this modified problem has the same feasible and optimal sets as the original problem, and has the identity as a distinguished direction. First, We will show that the identity will be a distinguished direction. Note that

$$\text{tr}(ABC) = \text{tr}(CAB) = \text{tr}(BCA)$$

So we can say

$$\text{tr}((E^{1/2}A_iE^{1/2})I)$$

$$= \text{tr}(E^{1/2} A_i E^{1/2})$$
$$= \text{tr}(A_i E^{1/2} E^{1/2})$$
$$= \text{tr}(A_i E)$$

$E$ is a feasible solution to the original problem, so this is equal to $b_i$. Putting this together,

$$\text{tr}((E^{1/2} A_i E^{1/2}) I) = b_i$$

The identity is clearly positive semidefinite, and so we can use it as a distinguished direction for the modified problem.

Now let's check to see if the feasible region for the modified problem is the same as that of the original.

$$\text{tr}(A^{'} X^{'}) = \text{tr}((E^{\frac{1}{2}} A_i E^{\frac{1}{2}})(E^{-\frac{1}{2}} X E^{-\frac{1}{2}}))$$
$$= \text{tr}((E^{\frac{1}{2}} A_i X E^{-\frac{1}{2}})$$
$$= \text{tr}(E^{-\frac{1}{2}} E^{\frac{1}{2}} A_i X)$$
$$= \text{tr}(A_i X) = b$$

So if $X$ is feasible for the original problem, then $X^{'}$ is feasible for the modified problem, and vice versa. Now all that remains is to show an equivalence between the objective functions, which proceeds similarly.

$$\text{tr}(C^{'} X^{'}) = \text{tr}((E^{\frac{1}{2}} C E^{\frac{1}{2}})(E^{-\frac{1}{2}} X E^{-\frac{1}{2}}))$$
$$= \text{tr}((E^{\frac{1}{2}} C X E^{-\frac{1}{2}})$$
$$= \text{tr}(E^{-\frac{1}{2}} E^{\frac{1}{2}} C X)$$
$$= \text{tr}(C X)$$

Through this, we can see that a solution to the modified problem with $X^{'}$, $A_i^{'}$, and $C^{'}$ will map to a solution to the original problem. The objective value will be the same, and to recover the solution $X$ to the original problem, we simply use the formula

$$X = E^{\frac{1}{2}} X^{'} E^{\frac{1}{2}}$$

So if we have an SDP for which $I$ is not feasible, we can convert it to an SDP for which it is, solve it using the relatively simple minimum eigenvalue approach, then convert it back to obtain a solution to the original problem. This is how our code handles SDPs for which $I$ is infeasible.

# 4 Implementation and Testing

We implemented our solver in Python, making extensive use of the Numpy package. We also use the CVXPY package to find an initial feasible solution to start the algorithm with. The solver requires the user to input their problem in a particular way. Namely, the user must formulate their problem in the same form as (P). The objective matrix $C$ must be given as a symmetric numpy matrix, the constraint matrices $A_1$, $A_2$,..., $A_m$ must be given as a list of symmetric numpy matrices, and $b_1$, $b_2$,..., $b_m$ must be given as an (m x 1)-dimensional numpy matrix. The following code illustrates how to solve an SDP with the solver.

```
import numpy
from GeneralCaseTolerance import *
A = [numpy.matrix([[1, 0, 1], [0, 3, 7], [1, 7, 5]]),
     numpy.matrix([[0, 2, 8], [2, 6, 0], [8, 0, 4]])]
c = numpy.matrix([[1, 2, 3], [2, 9, 0], [3, 0, 7]])
b = numpy.matrix([[6], [15]])
eps = 0.1
tol = 1e-2
max_iterations = 1000
opt_sol, opt_sol_val, iterations = RenegarSDPv2(A, b, c, eps, tol, max_iterations)
```

According to CVXPY, the optimal solution for this problem has a value of 9.77, while the solution returned by our solver has a value of 9.81. One should also note that the solver also returns how many iterations it took to converge on a solution.

A discussion of the meaning of the `tol` and `max_iterations` inputs seen in the example above can be found in the Roadblocks section. The input `eps` is the desired relative error between the value of the actual optimal solution and the value of the solution returned by the solver and is used in the supgradient algorithm we used for the solver [1].

Finally, we wrote a handful of testing scripts. These scripts test the core functions used in the solver; they also run the solver on a number of examples and compare the output to the answer given by CVXPY. To make sure our program was accurate for a wide variety of problems, we created an SDP example problem generator that we used to generate random input for the solver. This generator also compares the output of our solver with what's given by CVXPY.

# 5 Roadblocks

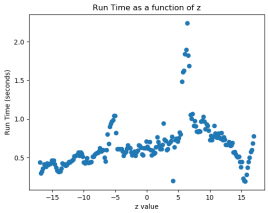
One issue we encountered was that it is difficult to detect when a problem is unbounded. With our original approach, we simply returned the value found after a certain number of iterations. But this gives no guarantee that a problem is actually bounded, which could be problematic for users. So we wanted to

tweak our approach to warn users when a problem was potentially unbounded. Rather than run a fixed number of iterations of the supgradient method, we altered our solver so that it stops after either 1) the iterations begin to converge or 2) a maximum number of iterations are run. We leave it up to the user to input a tolerance and a maximum number of iterations. While our solver still can't detect if an SDP is unbounded, it can issue a warning to the user if the maximum number of iterations is reached.

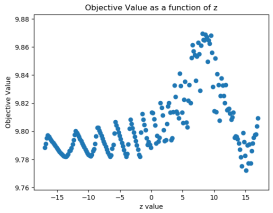
We ran into a second problem while changing our solver to be more aware of unboundedness, namely, we weren't sure how to check that the iterations were converging on a solution. It was difficult to pinpoint exactly the conditions under which the values were converging, to differentiate it from when the values were decreasing continuously. At first, we naively assumed that if the objective function values of successive iterations were very close together, then the algorithm must be converging on a solution and we can stop. However, we found that even after the algorithm converges on a solution, the objective function values of successive iterations will oscillate above and below a fixed value. With this in mind, we updated our solver to terminate when the recent average (e.g. an average over the last ten iterations) of the objective function values begins to converge. This resolution is not perfect, but works reasonably well on test cases.

Finally, we also found that the value z used to define the set $\text{Affine}_z$ can have a large impact on the convergence of and final value returned by the solver. We did an experimental analysis of how the choice of z affects the convergence speed and accuracy of the solver on two example problems in an attempt to see if there's a "best" value for z. Both of the examples have the identity matrix as a feasible solution, and both have the objective function value of the identity matrix equal to 17. However, they differ in their constraints, and we can see from the plots below that there's no clear choice of z that works for both problems (in fact, one of the better choices for z for the first example actually leads to our solver failing to converge on the second example). Plots for the run time, number of iterations, and objective function value for different choices of z for the two examples can be seen below. The true objective function value for the first example is 9.76 and the true objective function value for the second example is 5.4.

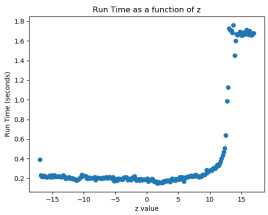
Example #1 Plots

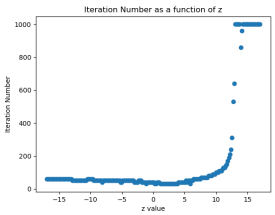


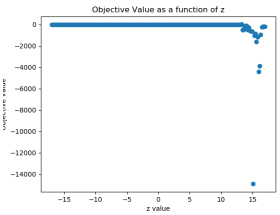





Example #2 Plots




With some playing around, we found a choice of z $(\text{tr}(c * e) - .9 * |\text{tr}(c * e)|)$ that allows our solver to converge and be accurate, although we can't be sure how optimal this method of choosing z is.

# 6 Limitations and Future Work

There are a number of ways in which our solver could be improved.

1. Performance optimizations. We were not particularly concerned with optimal performance, so there are likely many ways we could optimize our code for speed.

2. Input flexibility. Right now, our solver only accepts a very specific formulation of an SDP. However, there are many equivalent formulations which are valid SDPs but will not be understood by our solver.

3. Broadening the scope of the solver. The solver would be much more useful if it could solve conic optimization problems other than just SDPs. However, this is a difficult goal to accomplish since the reformulation of a problem is highly dependent on the cone $K$ for that particular type of problem. Independent work on other cones, particularly when the cone $K$ is the positive orthant, is already underway (see Sergio's project in the ORIE6326/DDATP repository).

4. Checking for unboundedness. Our solver simply gives a warning that the problem may be unbounded if it reaches the maximum number of iterations given by the user. It would be useful to know for sure if a problem is unbounded or not.

5. An analysis of the effect of different `eps` values. Currently the convergence and accuracy of the solver can depend on the desired relative accuracy. Further exploration of the effects of the chosen relative accuracy could potentially allow us to provide a "recommended" value for `eps`.

6. More z-testing. We only looked at the effect of the choice of z for a few SDPs, where z is the number that specifies the set $\text{Affine}_z$ in the modified problem. Further analysis might give us a better idea of how to best choose the value z.

7. Finding a distinguished direction. Our solver uses CVXPY to find a feasible solution (i.e. distinguished direction) $e$. Finding a way to directly compute an $e$ for any given problem would improve the speed, and possibly accuracy, of the solver.

# 7 Conclusion

We have successfully implemented an SDP solver using the method published by James Renegar [1]. In its current state, it is not useful as a practical solver, because it is significantly slower than optimized solvers such as CVXPY. In the future, we hope to perform optimizations and expand the scope of the program to solve more kinds of conic problems. With this work, the program could become a useful tool for optimizers.

We also used our program to investigate the effect of the z-value on the convergence rate. Our results suggest that there is no obvious, one-size-fits-all choice for z that leads to fast convergence. However, more work in this area could illuminate some patterns.

Lastly, we'd like to thank Professor Renegar himself for hashing out ideas with us.

# References

[1] J. Renegar. A FrameWork For Applying Subgradient Methods To Conic Optimization Problems (version 2)*. *arXiv preprint arXiv:1503.02611v2 [math.OC]*, 16 June 2015