An Algorithm for Solving the Convex Feasibility Problem With Linear Matrix Inequality Constraints and an Implementation for Second-Order Cones

Bryan Karlovitz

July 19, 2012

West Chester University of Pennsylvania
Department of Mathematics
25 University Avenue
West Chester, PA 19383

Abstract

Rami et al (see [4]) give an algorithm for solving the convex feasibility problem with a single linear matrix inequality constraint. We extend this to a system of linear matrix inequalities while giving an implementation that exploits the structure of the problem. We investigate the effect of the value used for eigenvalue replacement and give numerical evidence indicating that the choices of 0 and 1 used in [4] are not the best. A modification of this algorithm is proposed and we give an implementation for the special case of second-order cone constraints. Finally, we present the results of numerical experiments which indicate that our methods are effective.

1 Introduction

A semidefinite programming problem (SDP) is a type of convex optimization problem where we minimize a linear objective function of the variable $x \in \mathbb{R}^n$ subject to a set of linear matrix inequalities (LMIs):

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & A^{(j)}(x) \succeq 0, \ (j = 1, ..., q) \end{array} \tag{1}$$

where

$$A^{(j)}(x) \equiv A_0^{(j)} + \sum_{i=1}^{n} x_i A_i^{(j)}.$$

$A_0^{(j)}, ..., A_n^{(j)} \in \mathbb{R}^{m_j \times m_j}$ are symmetric matrices and $c \in \mathbb{R}^n$. $A^{(j)}(x) \succeq 0$ means that the matrix $A^{(j)}(x)$ is positive semidefinite; that is, $A^{(j)}(x)$ has nonnegative eigenvalues. The notation $A^{(j)}(x) \succ 0$ means that $A^{(j)}(x)$ is positive definite, or that $A^{(j)}(x)$ has positive eigenvalues. There are other equivalent definitions of positive semidefinite and positive definite matrices (see, e.g., [6]).

Semidefinite programming generalizes many types of optimization problems. For example, linear, quadratic, and second-order cone programming problems can all be expressed as SDPs. Semidefinite programming has also been used in combinatorial optimization, control theory, and engineering applications. See [7] for details.

This paper concerns finding a feasible point $x \in \mathbb{R}^n$ for the problem (1); that is, we want to find a point that satisfies all q constraints. We will assume throughout that the interior of the feasible region is nonempty. This is an example of what is called a convex feasibility problem.

Let $C_1, ..., C_r$ be closed convex sets in a Hilbert Space $\mathcal{H}$ such that

$$C_1 \cap ... \cap C_r \neq \varnothing.$$

The general convex feasibility problem is to find a point in the intersection. Many algorithms for solving convex optimization problems, including SDPs, begin with the assumption that a feasible point is known. So the convex feasibility problem is an important part of those algorithms. Furthermore, the convex feasibility problem appears in numerous engineering applications. See, e.g., [2].

The alternating projections method is a simple algorithm for solving the convex feasibility problem. It was first studied by Von Neumann in 1950. Many generalizations and modifications of the idea have been developed (see [1]).

To find a point in the intersection of the sets $C_1, ..., C_r$ given above, the basic idea of the alternating projections method is to compute the sequence $\{x_k\}$ defined by

$$x_{k+1} = \mathcal{P}_{\mathcal{C}_{\phi(k)}}(x_k), \text{ where } \phi(k) = (k \bmod \mathrm{r}) + 1,$$

where $\mathcal{P}_{\mathcal{C}_{\phi(k)}}$ is the projection onto the set $\mathcal{C}_{\phi(k)}$.

This approach often needs an infinite number of iterations to converge. The paper [4] develops a version of the alternating projections method that converges in a finite number of steps when the sets $C_1, ..., C_q$ are closed convex cones.

We will begin this paper by briefly explaining the algorithm proposed by Rami et al. This is developed in [4] for a single LMI constraint. We extend the algorithm to handle problems with q LMI constraints by using the fact that a system of LMIs can be written as a single LMI in block-diagonal form. Our implementation of this version exploits the structure of the problem. Then we investigate the effect of the value δ used for eigenvalue replacement in the algorithm on the number of iterations required. We show numerical evidence that indicates the choices of 0 and 1 used in [4] are not necessarily the best. Next we present a new modification of the algorithm where an eigenvalue shift is used in the spectral decomposition step of the algorithm. We also tried a variation of the eigenvalue shift that uses Cholesky factorization. However this method was numerically unstable. Then we look at an implementation for the special case of second-order cone constraints. Finally, we give the results of our numerical experiments, which suggest that our methods are effective.

2 The Algorithm Applied to a Single LMI Constraint

2.1 Preliminary Theoretical Results

Notation. $\mathcal{S}_m$ denotes the set of real symmetric matrices of size n. $\mathcal{S}_m^+$ denotes the set of real symmetric positive semidefinite matrices of size m. $\bar{C}$ denotes the closure of the set C. If $A_1, ..., A_n$ are square matrices, not necessarily all the same size, then we use $\mathbf{diag}(A_1, ..., A_n)$ to denote the block diagonal matrix with A_i as its i^{th} block. $A \bullet B$ is the dot product of the matrices A and B. $\lambda_{min}(A)$ denotes the smallest eigenvalue of the matrix A.

Theorem 1 below is the well known Hilbert Projection Theorem. See [5] for a proof.

Theorem 1. *Let C be a nonempty closed convex subset of a real Hilbert space $\mathcal{H}$. Then for any $x \in \mathcal{H}$ there exists a unique $\hat{x} \in C$ such that*

$$||x - \hat{x}|| \leq ||x - y||, \ \forall y \in C. \tag{2}$$

Moreover, $\hat{x} \in C$ satisfies (1) if and only if

$$\langle x - \hat{x}, y - x \rangle \leq 0, \ \forall y \in C.$$

We will write $\hat{x} = \mathcal{P}_C(x)$.

Definition 1. The Gramain matrix of a set of vectors $v_1, ..., v_n$ in an inner product space is defined by $G_{ij} = \langle v_i, v_j \rangle$.

Fact 1. *The set of vectors $v_1, ..., v_n$ is linearly independent if and only if G is invertible.*

The following three lemmas are given in [4].

Lemma 1. *Let $\mathcal{L}$ be a finite dimensional linear subspace in a Hilbert space $\mathcal{H}$ and let $x_1, ..., x_n$ be a basis for $\mathcal{L}$. Then the Gramian matrix $G = (\langle x_i, x_j \rangle)_{i,j \leq n}$ is invertible and the metric projection onto $\mathcal{L}$ is given by*

$$\mathcal{P}_{\mathcal{L}}(x) = \sum_{i=1}^{n} \alpha_i x_i,$$

where

$$(\alpha_1, ..., \alpha_n)^T = G^{-1}(\langle x, x_1 \rangle, ..., \langle x, x_n \rangle)^T.$$

Proof. A calculation shows that this satisfies (1) in Theorem 1. □

Lemma 2. *The metric projection of any $M \in S_m$ onto S_m^+ is computed as follows. Let $M = VDV^T$ be the eigenvalue-eigenvector decomposition where $D = \mathbf{diag}(d_1, ..., d_m)$. Define $\bar{D} = \mathbf{diag}(\bar{d_1}, ..., \bar{d_m})$ with*

$$\begin{array}{ll} \bar{d_i} = d_i & if\ d_i \geq 0, \\ \bar{d_i} = 0 & if\ d_i < 0. \end{array}$$

Then $\mathcal{P}_{S_m^+}(M) = V\bar{D}V^T$.

Proof. A calculation shows that this satisfies (1) in Theorem 1. □

Lemma 3. *Let C be a closed convex subset of a real Hilbert space $\mathcal{H}$ and let $x \in \mathcal{H}$. Then the projection onto $C + x$ is given by*

$$\mathcal{P}_{C+x}(y) = \mathcal{P}_C(y - x) + x,\ \forall y \in \mathcal{H}.$$

Proof. By Theorem 1 we have

$$\langle \mathcal{P}_C(y - x) - (y - x), \mathcal{P}_C(y - x) - h \rangle \leq 0,\ \forall h \in C.$$

This implies

$$\langle (\mathcal{P}_C(y - x) + x) - y, (\mathcal{P}_C(y - x) + x) - (x + h) \rangle \leq 0,\ \forall h \in C.$$

$\mathcal{P}_{C+x}(y) = \mathcal{P}_C(y - x) + x$ follows. □

Remark 1. $\mathcal{S}_m$ is a finite-dimensional Hilbert space with inner product $\langle X, Y \rangle \equiv Tr(X, Y) = X \bullet Y$. $\overset{\circ}{\mathcal{S}_m^+}$ is the open cone of positive definite matrices.

2.2 The Algorithm

We consider the LMI feasibility problem

$$\begin{gathered}\text{Find } x \in \mathbb{R}^n \text{such that} \\ F(x) \equiv F_0 + \sum_{i=1}^{n} x_i F_i \succeq 0,\end{gathered} \tag{3}$$

where the F_i's are linearly independent matrices in $\mathcal{S}_m$.

For this paper we assume that the feasible region defined by the LMI constraint has a nonempty interior.

In [4], it is shown that this algorithm has asymptotic convergence. It is also shown that this algorithm will converge in a finite number of steps when $F_0 = 0$.

The Gramian matrix G associated with $F_1, ..., F_n$ is given by

$$G = (F_i \bullet F_j)_{i,j \leq n}.$$

The affine space $\mathcal{A}$ associated with (2) is

$$\mathcal{A} = \{F(x) | x \in \mathbb{R}^n\}.$$

By Lemmas 1 and 3, we have that the projection $\mathcal{P}_{\mathcal{A}}$ of $\mathcal{S}_m$ onto $\mathcal{A}$ is computed as

$$\mathcal{P}_{\mathcal{A}}(S) = F_0 + \sum_{i=1}^{n} \alpha_i F_i,$$

for $S \in \mathcal{S}_m$ where

$$(\alpha_1, ..., \alpha_n)^T = G^{-1}[((S - F_0) \bullet F_1), ..., ((S - F_0) \bullet F_n)]^T. \tag{4}$$

The affine space $\mathcal{A}$ is the set of matrices that are equal to $F(x)$ for some $x \in \mathbb{R}^n$. We want to find an x that makes $F(x)$ positive semidefinite. We can do this by finding a matrix $S \in \mathcal{A}$ that is positive semidefinite and then computing the vector α associated with $\mathcal{P}_{\mathcal{A}}(S)$. Note that $\mathcal{P}_{\mathcal{A}}(S) = S$ when $S \in \mathcal{A}$. Since $\mathcal{P}_{\mathcal{A}}(S) = F(\alpha)$ we see that α is a solution to (2) when S is positive semidefinite. This is the idea behind the *Finite Steps Algorithm*. See Algorithm 1 for the pseudocode.

S_k denotes the k^{th} iteration of the algorithm. S_1 is chosen to be any real symmetric matrix with same size as $F(x)$. There are two steps that we perform depending on whether k is even or odd.

Step1 : If k is odd, let $S_{k+1} = \mathcal{P}_{\mathcal{A}}(S_k)$. This gives $S_{k+1} \in \mathcal{A}$; we need to check if it is positive semidefinite. Do this by looking at the eigenvalues. If $\lambda_{min}(S_{k+1}) \geq 0$, then the α associated with S_{k+1} is a solution. Otherwise continue to Step 2.

Step2 : If k is even, then let VDV^T, with $D = \mathbf{diag}(d_1, ..., d_n)$, be the spectral decomposition of S_k. Define $\bar{D} = \mathbf{diag}(\bar{d_1}, ..., \bar{d_n})$ where

$$\bar{d}_i = d_i \qquad if\ d_i \geq 0,$$
$$\bar{d}_i = 0 \qquad if\ d_i < 0.$$

We note here that the algorithm is also presented using 1 instead of 0 in the above eigenvalue replacement. Let $S_{k+1} = V\bar{D}V^T$. Note that when we use 0 this is the projection given in Lemma 2. Now we know that S_{k+1} is positive semidefinite; we need to check if it is in $\mathcal{A}$. Do this by looking at the projection onto $\mathcal{A}$. If $\mathcal{P}_{\mathcal{A}}(S_{k+1}) = S_{k+1}$, then $S_{k+1} \in \mathcal{A}$ and the associated α is a solution. Otherwise return to Step 1. See Algorithm 1 for the pseudocode.

This is a version of the alternating projections method where the sets we are projecting onto are $\mathcal{C}_1 = \mathcal{A}$ and $\mathcal{C}_2 = \mathcal{S}_m^+$. We want to find a matrix that is simultaneously in both of these sets. Each projection steps gives us a matrix that is in either $\mathcal{A}$ or $\mathcal{S}_m^+$ and we must check to see if it is in the other set.

Algorithm 1 The algorithm for a single LMI constraint

Initialization: Choose any $S \in \mathcal{S}_m$ and let $S_1 = S$.
Set $k = 1$, $tol = 0.001$, $maxIter = 500$, and $stop = 0$.
Suppose that S_k is the current iteration.
while $stop = 0$ and $k < maxIter$ **do**
  if k is odd **then**
    $S_{k+1} = \mathcal{P}_{\mathcal{A}}(S_k)$
    if $\lambda_{min}(S_{k+1}) \geq 0$ **then**
      $stop = 1$
      return α
    end if
  else if k is even **then**
    Compute $S_k = V\mathbf{diag}(d_1, ..., d_n)V^T$.
    $S_{k+1} = V\mathbf{diag}(\bar{d}_1, ..., \bar{d}_n)V^T$, where
     $\bar{d}_i = d_i$ if $d_i \geq 0$ (resp. 1),
     $\bar{d}_i = 0$ (resp. 1) if $d_i < 0$ (resp. 1).
    if $\mathcal{P}_{\mathcal{A}}(S_{k+1}) = S_{k+1}$ **then**
      $stop = 1$
      return α
    end if
  end if
  $k \leftarrow k + 1$
end while

3 Implementation for a System of LMIs

In this section we give an implementation of the above algorithm for LMI feasibility problems having q constraints. We look at how the structure of the iterations in the algorithm can be used to make the computations efficient.

3.1 Computations with Block-Diagonal Structure

We consider the problem

$$\begin{array}{c} \text{Find } x \in \mathbb{R}^n \text{such that} \\ A^{(j)}(x) \equiv A_0^{(j)} + \sum_{i=1}^n x_i A_i^{(j)} \succeq 0, \ (j = 1, ..., q). \end{array} \tag{5}$$

$A_0^{(j)}$ and each $A_i^{(j)}$, $j = 1, ..., q$, are symmetric matrices in $\mathbb{R}^{m_j \times m_j}$. The q LMI constraints in (4) are equivalent to the single LMI constraint

$$F(x) \equiv F_0 + \sum_{i=1}^n x_i F_i \succeq 0, \tag{6}$$

where $F_0 = \mathbf{diag}(A_0^{(1)}, ..., A_0^{(q)})$ and $F_i = \mathbf{diag}(A_i^{(1)}, ..., A_i^{(q)})$.

This shows we can use the *Finite Steps Algorithm* to solve problem (4) provided that we write the q LMI constraints as an equivalent single LMI constraint in block diagonal form. However, the block diagonal matrices become more sparse as the number of constraints goes up. Using the entire matrices makes certain calculations, like computing the Gramian matrix G or testing the eigenvalues of an iteration S_k, slower than they need to be. We will now describe how the block-diagonal structure of the matrices allows us to improve the efficiency of the computations.

In order to use the algorithm we need the Gramian matrix G associated with $F_1, ..., F_n$. $G_{ik} = F_i \bullet F_k$, so if F_i and F_k have lots of off-diagonal entries with the value 0, then the dot product computation will involve lots of multiplications and additions by 0. The following fact lets us avoid this.

Fact 2. *Let $G^{(j)}$ be the Gramian matrix associated with $A^{(j)}(x)$ in (4) and let G be the Gramian matrix associated with $F(x)$ in (4). Then*

$$G = \sum_{j=1}^q G^{(j)}.$$

Proof. $G_{ik} = F_i \bullet F_k$. F_i and F_k are block diagonal matrices whose corresponding blocks are the same size, so

$$\begin{aligned} F_i \bullet F_k &= \sum_{j=1}^q A_i^{(j)} \bullet A_k^{(j)} \\ &= \sum_{j=1}^q G_{ik}^{(j)}. \end{aligned}$$

Then

$$G_{ik} = \sum_{j=1}^q G_{ik}^{(j)}.$$

Hence

$$G = \sum_{j=1}^{q} G^{(j)}.$$

□

To compute G, it is more efficient to compute each $G^{(j)}$ and sum the results than it is to compute G directly.

Remark 2. The eigenvalues and eigenvectors of a block-diagonal matrix are the combined eigenvalues and eigenvectors of the individual blocks.

Fact 3. *Each iteration S_k of the algorithm, with the possible exception of S_1, has the same block-diagonal structure as $F(x)$.*

Proof. Since S_k comes from one of two possible calculations, there are two cases to consider. First, if k is even, then

$$\begin{aligned} S_k &= \mathcal{P}_{\mathcal{A}}(S_{k-1}) \\ &= F_0 + \sum_{i=1}^{n} \alpha_i F_i \quad . \end{aligned}$$

Since F_0 and each F_i have the same block diagonal structure as $F(x)$, we see that S_k has the block diagonal strucure.

Now, if k is odd, $k \neq 1$, then S_k is computed by first finding the spectral decomposition $S_{k-1} = VDV^T$ and then modifying the entries of D in some way. Since k is odd, we know that $k-1$ is even and so S_{k-1} has the block-diagonal structure. We claim that it is equivalent to compute S_k by performing the spectral decomposition and eigenvalue modification on each block of S_{k-1}. Using Remark 2, we see that computing the spectral decompositions on the blocks of S_{k-1} gives the same set of eigenvalues and eigenvectors as computing the spectral decomposition of S_{k-1}. In other words we get the same spectral decomposition. Hence, the two methods are equivalent. □

We can initialize S_1 as an aribitrary block-diagonal matrix so that every iteration has the same block-diagonal structure.

The significance of Fact 3 is that it shows us every computation in the algorithm is done with block-diagonal matrices. Since the only parts of these matrices that matter in the computations are the blocks themselves, we can simply store the blocks instead of the whole matrices. This saves memory space and computation time.

The other calculation that uses lots of block-diagonal matrices and where we can save computation time is in calculating the vector

$$[((S - F_0) \bullet F_1), ..., ((S - F_0) \bullet F_n)]^T$$

used to find α. See Lemmas 1 and 3.

Since S has the same block-diagonal structure as F_0 and each F_i, this vector is found using that

$$(S - F_0) \bullet F_i \quad = \quad \sum_{j=1}^{q} (S^{(j)} - A_0^{(j)}) \bullet A_i^{(j)}.$$

As in our method for computing the Gramian matrix, this lets us avoid all of the off-diagonal 0 entries.

We also use that each $S^{(i)}$ and each $A^{(i)}$ is symmetric in our computations. We can save computation time by only computing the diagonal and above-diagonal entries.

Algorithm 2 Extension of Finite the algorithm for a system of LMIs

Initialization: For $j = 1, ..., q$ choose symmetric $S^{(j)} \in \mathbb{R}^{m_j}$
These are the blocks of the block-diagonal matrix S_1.
Set $k = 1$, $tol = 0.001$, $maxIter = 500$, and $stop = 0$.
Suppose that S_k is the current iteration.
while $stop = 0$ and $k < maxIter$ **do**
  if k is odd **then**
    $S_{k+1} = \mathcal{P}_{\mathcal{A}}(S_k)$
    if $\lambda_{min}(S_{k+1}) \geq 0$ **then**
      $stop = 1$
      return α
    end if
  else if k is even **then**
    Compute $S_k = V\mathbf{diag}(d_1, ..., d_n)V^T$.
    $S_{k+1} = V\mathbf{diag}(\bar{d}_1, ..., \bar{d}_n)V^T$, where
     $\bar{d}_i = d_i \quad$ if $d_i \geq 0$,
     $\bar{d}_i = 0 \quad$ if $d_i < 0$.
    if $\mathcal{P}_{\mathcal{A}}(S_{k+1}) = S_{k+1}$ **then**
      $stop = 1$
      return α
    end if
  end if
  $k \leftarrow k + 1$
end while

3.2 Methods For Modifying the Eigenvalues

3.2.1 Eigenvalue Replacement

Step 2 in the algorithm requires us to modify the eigenvalues of the current iteration. The approach in [4] is to do the spectral decmposition of the iteration S_k and either replace all eigenvalues that are less than 0 with the value 0 and

leave all other eigenvalues alone or replace all eigenvalues that are less than 1 with the value 1 and leave all other eigenvalues alone. More generally, we can choose a value δ and replace all eigenvalues that are less than δ with the value δ and leave all other eigenvalues alone. This approach will be refered to as *eigenvalue replacement*. [4] discusses *eigenvalue replacement* with $\delta = 0$ and $\delta = 1$.

Our numerical experiments suggest that the choice of δ has an important effect on the number of iterations that the algorithm takes to find a feasible point. We also see that the best choice of δ, in terms of the number of iterations, depends on the problem and that it can be a value other than 0 or 1. See Section 5 for plots of δ versus the number of iterations required by the algorithm.

3.2.2 Eigenvalue Shift

Another approach to modifying the eigenvalues in Step 2 of the algorithm is to do the spectral decomposition of the iteration S_k and add the same number to all eigenvalues.

For this approach we choose a value of δ as in the *eigenvalue replacement*. Then we let $\tau = max(0, \delta - \lambda_{min}(S_k))$. Then if $S_k = VDV^T$ is the spectral decomposition of S_k we have

$$S_{k+1} = V(D + \tau I)V^T.$$

This approach will be refered to as *eigenvalue shift*.

3.2.3 Cholesky Decomposition Eigenvalue Shift

In this section we want to mention a method of eigenvalue modification that did not work. This method is discussed in [3]. This approach is a version of the *eigenvalue shift* that uses the Cholesky decompostion rather than the spectral decomposition. Algorithm 3 shows the pseudocode. [3] mentioned that this algorithm can be numerically unstable in the sense that the entries in L can become arbitrarily large. This is exactly the problem that we had. [3] also discusses a a modification of this method for avoiding this problem. But we did not have the time to implement it.

3.3 Illustration of the Algorithm

In this section we want to give an illustration of how the algorithm works. We will do this with the following example.

Example 1. Find a point $x \in \mathbb{R}^2$ that satisfies

$$A^{(1)}(x) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \succeq 0$$

$$A^{(2)}(x) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} + x_1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + x_2 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \succeq 0$$

Algorithm 3 Eigenvalue Shift with Cholesky Decomposition

Choose $\beta > 0$
if $min_i(a_{ii}) > 0$ **then**
  $\tau_0 = 0$
else
  $\tau_0 = \beta - min_i(a_{ii})$
end if
for $k = 0, 1, 2, ...$ **do**
  Attempt the Cholesky factorization to obtain $LL^T = A + \tau_k I$
  if $t$the factorization is successful **then**
    stop
    return L
  else
    $\tau_{k+1} = max(10\tau_k, \beta)$
  end if
end for

We will of course solve this using the *Finite Steps Algorithm* and we want some sense of how the iterations converge to a feasible point. But the iterations of the algorithm are matrices so we cannot plot them. However, at every other step in the algorithm we compute the vector α, which we can plot. Seeing these α's gives us a graphical sense of what the *Finite Steps Algorithm* is doing.

Figures 1 and 2 were generated using eigenvalue replacement with $\delta = 0$. The behavior of the algorithm shown here is typical of what we observed in using replacement with $\delta = 0$. In particular, the algorithm starts out by making large jumps towards the feasible region. Then as the points get closer to the boundary the algorithm makes smaller jumps and we end up doing a lot of calculations with points that are almost feasible. Also notice that the feasible point found here is on the boundary. Figure 3 was generated using eigenvalue replacement with $\delta = 1$. Notice that we do not get the slowing down near the boundary that occurred with $\delta = 0$. Figure 4 was generated using spectral eigenvalue shift with $\delta = 0$. In general, we observed that eigenvalue shift with $\delta = 0$ does not have the slowing down near the boundary that we find using eigenvalue replacement with $\delta = 0$. Figure 5 was generated using spectral eigenvalue shift with $\delta = 1$.

4 The Algorithm Applied to Second-Order Cone Constraint

Consider the second-order cone (SOC) feasibility problem

$$\begin{array}{c} \text{Find } x \in \mathbb{R}^n \text{ such that} \\ ||Ax + b|| \leq c^T x + d, \end{array} \tag{7}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $d \in \mathbb{R}$.

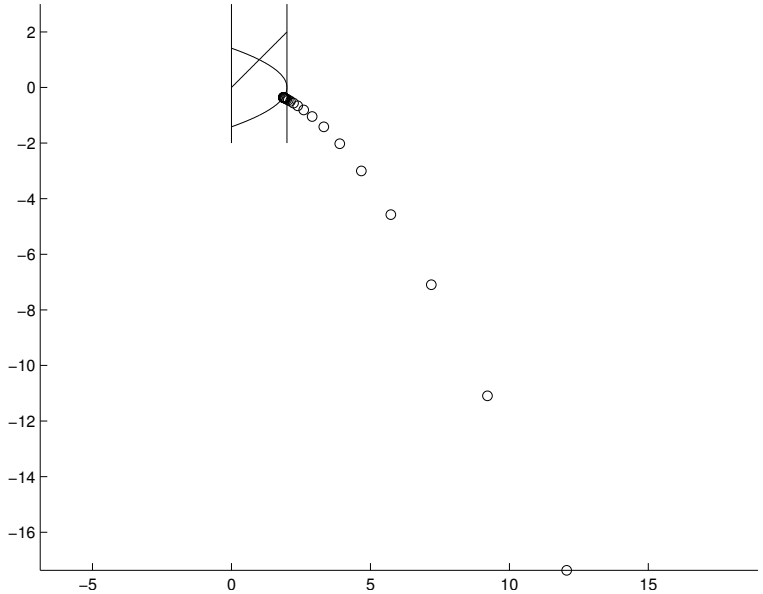

Figure 1: This plot shows the progess of the α's in the algorithm on Example 1 using eigenvalue replacement with $\delta = 0$.

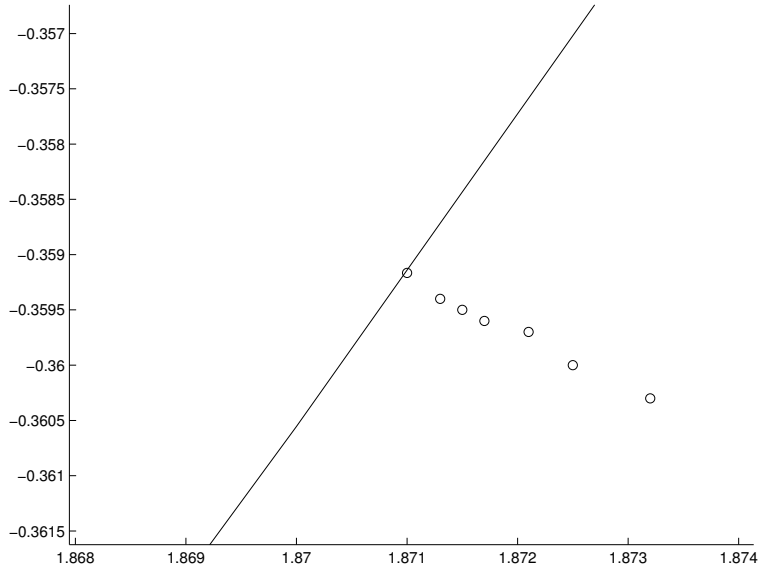

Figure 2: This plot shows a zoomed-in view of the last 7 α's in the algorithm on Example 1 using $\delta = 0$.

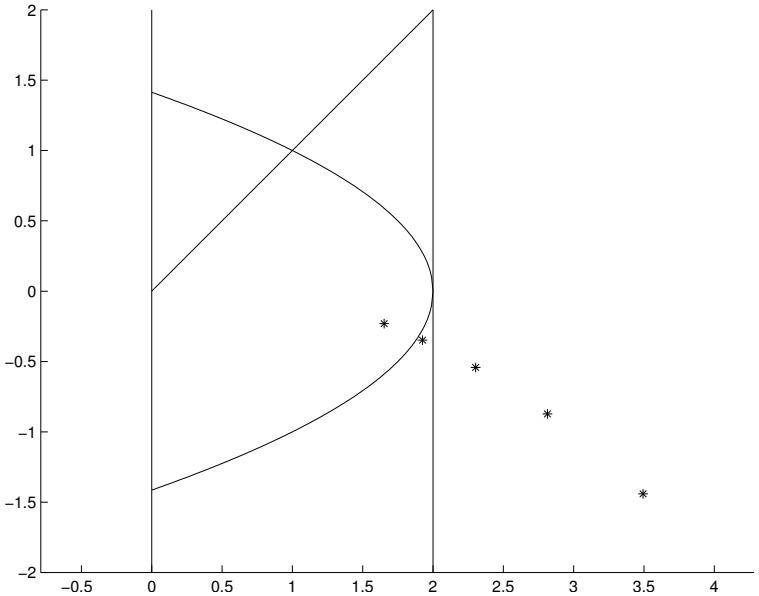

Figure 3: This plot shows the progress of the α's in the algorithm on Example 1 using $\delta = 1$.

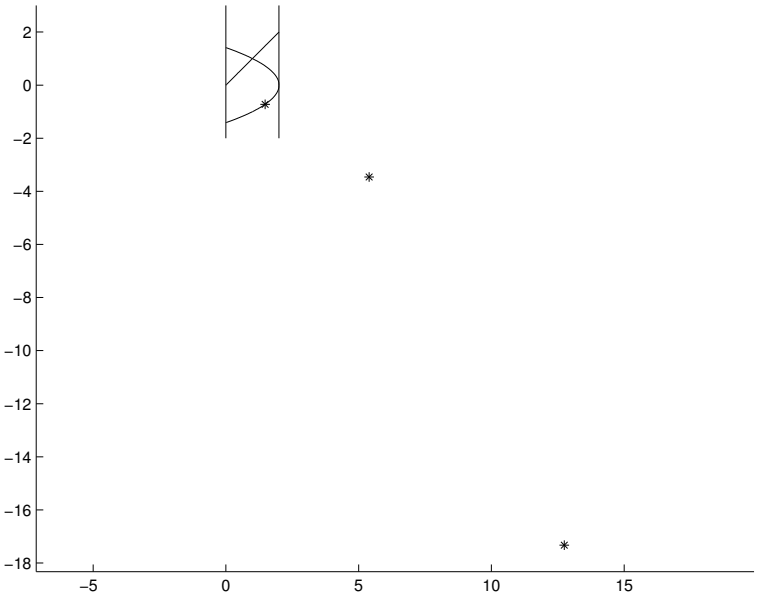

Figure 4: This plot shows the progress of the α's in the algorithm using eigenvalue shift with $\delta = 0$.

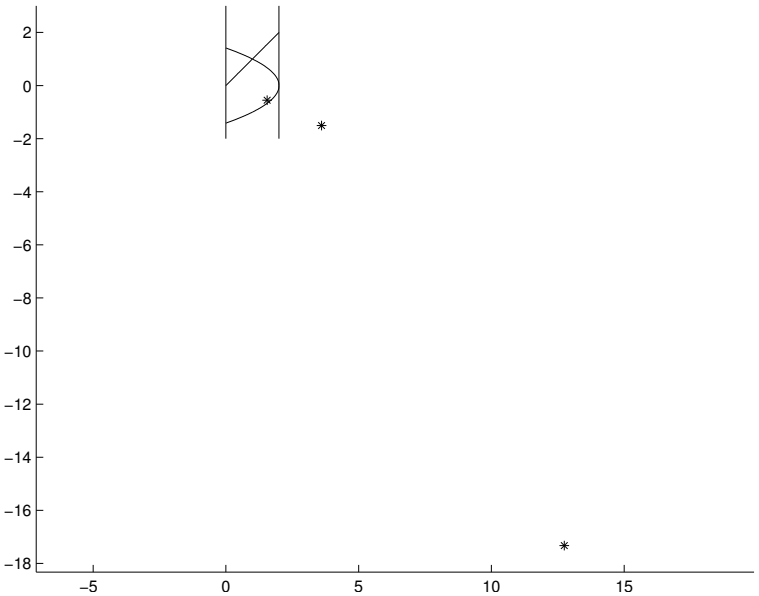

Figure 5: This plot shows the progress of the α's in the algorithm using eigenvalue shift with $\delta = 1$.

In general, the inequality $||u|| \leq t$, for $u \in \mathbb{R}^n$ and $t \in \mathbb{R}$, can be written in LMI form as

$$\begin{bmatrix} tI & u \\ u^T & t \end{bmatrix} \succeq 0.$$

So the SOC inequality (7) is equivalent to

$$\begin{bmatrix} (c^T x + d)I & Ax + b \\ (Ax + b)^T & c^T x + d \end{bmatrix} \succeq 0. \tag{8}$$

Let $A = [a_1 ... a_n]$, where a_i is the i^{th} column of A. Then we can write (7) in the form of (2) as

$$\begin{bmatrix} dI & b \\ b^T & d \end{bmatrix} + \sum_{i=1}^{n} x_i \begin{bmatrix} c_i I & a_i \\ a_i^T & c_i \end{bmatrix} \succeq 0, \tag{9}$$

Hence (7) is equivalent to

$$F(x) \equiv F_0 + \sum_{i=1}^{n} x_i F_i \succeq 0,$$

with

$$F_0 = \begin{bmatrix} dI & b \\ b^T & d \end{bmatrix},$$

and

$$F_i = \begin{bmatrix} c_i I & a_i \\ a_i^T & c_i \end{bmatrix}.$$

This shows we can use the algorithm to solve problem (7).

4.1 Implementation for a Second-Order Cone Constraint

The following lemma gives us an efficient way to compute the Gramian matrix when we have second-order cone structure. Table 7 shows that using this calulation is effective for saving time.

Lemma 4. *The Gramian matrix $G = [G_{ij}]$ associated with (8) is given by $G_{ij} = (m+1)c_i c_j + 2a_i^T a_j$.*

Proof. By the definition of the Gramian we have $G_{ij} = F_i \bullet F_j$. Then

$$\begin{aligned} F_i \bullet F_j &= \begin{bmatrix} c_i I & a_i \\ a_i^T & c_i \end{bmatrix} \bullet \begin{bmatrix} c_j I & a_j \\ a_j^T & c_j \end{bmatrix}. \\ &= (m+1)c_i c_j + 2a_i^T a_j \end{aligned}$$

□

In order to run the *Finite Steps Algorithm* we must have that the Gramian matrix G associated with the constraint is invertible. The following theorem give a sufficient condition that problem (6) satisfied this.

Theorem 2. *Let*

$$||Ax + b|| \leq c^T x + d$$

be a second-order cone inequality and let

$$\begin{bmatrix} dI & b \\ b^T & d \end{bmatrix} + \sum_{i=1}^{n} x_i \begin{bmatrix} c_i I & a_i \\ a_i^T & c_i \end{bmatrix} \succeq 0$$

be the equivalent LMI. Then the Gramian matrix G associated with the LMI is invertable if the columns of the matrix A are independent.

Proof. By Fact 1, G is invertable if the matrices

$$\begin{bmatrix} c_i I & a_i \\ a_i^T & c_i \end{bmatrix}, \; i = 1, ..., n \tag{10}$$

are linearly independent. The equation

$$\beta_1 \begin{bmatrix} c_1 I & a_1 \\ a_1^T & c_1 \end{bmatrix} + ... + \beta_n \begin{bmatrix} c_n I & a_n \\ a_n^T & c_n \end{bmatrix} = 0, \tag{11}$$

for $\beta \in \mathbb{R}^n$ is equivalent to the two sums

$$\sum_{i=1}^{n} \beta_i c_i = 0, \tag{12}$$

and

$$\sum_{i=1}^{n} \beta_i a_i = 0. \tag{13}$$

Suppose that the columns of A are linearly independent. Then the only vector that satisfies (13) is $\beta = 0$. So the only vector that satisfies (11) is $\beta = 0$. □

Theorem 3 shows that the columns of A being independent is a sufficient condition for G to be invertable. But it is not necessary as the follwing example shows.

Example 2. Consider the second-order cone

$$||Ax + b|| \leq c^T x + d$$

with

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, d = 0.$$

This is written in LMI form as

$$x_1 \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix} + x_2 \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 2 & 4 & 1 \end{bmatrix} \succeq 0. \tag{14}$$

Clearly the columns of A are not independent. But we claim that the matrices in (14) are independent. For $\beta \in \mathbb{R}^2$, the equation

$$\beta_1 \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix} + \beta_2 \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 2 & 4 & 1 \end{bmatrix} = 0$$

is equivalent to the two sums

$$\beta_1 + \beta_2 = 0 \tag{15}$$

and

$$\beta_1 \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \beta_2 \begin{bmatrix} 2 \\ 4 \end{bmatrix} = 0. \tag{16}$$

If β satisfies (16) the we must have $\beta = \begin{bmatrix} 2s \\ -s \end{bmatrix}$ for some s. Then (15) says $2s - s = s = 0$.

Now we will give an illustration of the *Finite Steps Algorithm* with second-order cone constraints. We will use the following example.

Example 3. Consider the system of second-order cones

$$\begin{array}{rcl} c_1^T x + d_1 - ||A_1 x + b_1|| & \geq & 0 \\ c_2^T x + d_2 - ||A_2 x + b_2|| & \geq & 0 \\ c_3^T x + d_3 - ||A_3 x + b_3|| & \geq & 0 \end{array}$$

where

$$A_1 = \begin{bmatrix} 6 & -7 \\ 2 & 3 \end{bmatrix}, b_1 = \begin{bmatrix} -7 \\ 1 \end{bmatrix}, c_1 = \begin{bmatrix} 5 \\ 8 \end{bmatrix}, d_1 = 8$$

$$A_2 = \begin{bmatrix} -4 & 3 \\ 3 & -8 \end{bmatrix}, b_2 = \begin{bmatrix} -2 \\ -3 \end{bmatrix}, c_2 = \begin{bmatrix} 4 \\ -8 \end{bmatrix}, d_1 = 8$$

$$A_3 = \begin{bmatrix} -2 & -2 \\ 2 & -4 \end{bmatrix}, b_3 = \begin{bmatrix} -9 \\ 10 \end{bmatrix}, c_3 = \begin{bmatrix} 3 \\ -4 \end{bmatrix}, d_3 = 6$$

Then $x \in \mathbb{R}^2$ and this can be written as the following system of LMIs.

$$\begin{bmatrix} 8 & 0 & -7 \\ 0 & 8 & 1 \\ -7 & 1 & 8 \end{bmatrix} + x_1 \begin{bmatrix} 5 & 0 & 6 \\ 0 & 5 & 2 \\ 6 & 2 & 5 \end{bmatrix} + x_2 \begin{bmatrix} 8 & 0 & -7 \\ 0 & 8 & 3 \\ -7 & 3 & 8 \end{bmatrix} \succeq 0$$

$$\begin{bmatrix} 8 & 0 & -2 \\ 0 & 8 & -3 \\ -2 & -3 & 8 \end{bmatrix} + x_1 \begin{bmatrix} 4 & 0 & -4 \\ 0 & 4 & 3 \\ -4 & 3 & 4 \end{bmatrix} + x_2 \begin{bmatrix} 8 & 0 & 3 \\ 0 & 8 & -8 \\ 3 & -8 & 8 \end{bmatrix} \succeq 0$$

$$\begin{bmatrix} 6 & 0 & -9 \\ 0 & 6 & -10 \\ -9 & -10 & 6 \end{bmatrix} + x_1 \begin{bmatrix} 3 & 0 & -2 \\ 0 & 3 & 2 \\ -2 & 2 & 3 \end{bmatrix} + x_2 \begin{bmatrix} -4 & 0 & -2 \\ 0 & -4 & -4 \\ -2 & -4 & -4 \end{bmatrix} \succeq 0$$

Figure 6 shows the progress of the α's for the algorithm using eigenvalue replacement with $\delta = 20$. We have labeled the points in the order that they were found by the algorithm. Notice that the feasible region was jumped over twice before a feasible point was found.

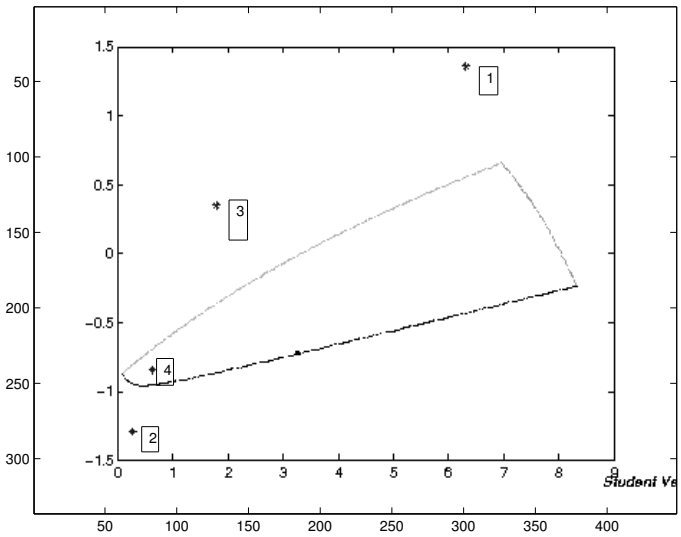

Figure 6:
This plot shows the progress of the α's for the algorithm for the problem in Example 3 using eigenvalue shift with $\delta = 20$.

5 Numerical Experiments

In our numerical experiments we seek to test the effectiveness of our various modifications of the algorithm. We are interested in the number of iterations and the time that the algorithm takes to run. For these experiements we set the maximum number of iterations to be 500. For the general LMI problem we have 25 randomly generated test problems for which we chose *q, m, and n.* For each of these problems, 0 is a feasible point. See Table 1.

5.1 Choice of δ Versus Number of Iteration

Here we give the results of our numerical experiments which investigate how the choice of δ effects the number of iterations required by the algorithm. We look at the version of the algorithm using eigenvalue replacement with spectral decomposition and eigenvalue shift with spectral decomposition. For Problem 1 and Problem 2 we ran the algorithm with δ ranging from 0 to 100 in steps of 0.1. The maximum number of iterations was set to 500.

Figure 6 shows the plot of δ versus the number of iterations for LMI Problem 1 using eigenvalue replacement. Notice that $\delta = 0$ required 60 iterations and $\delta = 1$ required 21 iterations. The least number of iterations required was 15 and that obtained with all choices of δ that we tested in the interval $[2.7, 3.4]$.

LMI Problem	m	n	q
1	2	2	2
2	1	2	3
3	14	3	2
4	9	5	15
5	8	13	15
6	11	6	12
7	5	2	7
8	3	7	8
9	10	5	11
10	6	15	9
11	14	11	8
12	13	3	4
13	3	14	6
14	13	6	9
15	9	6	12
16	15	12	2
17	12	3	6
18	11	10	4
19	4	4	9
20	2	7	7
21	6	5	5
22	4	3	13
23	9	13	3
24	2	4	7
25	3	3	3

Table 1: Test problems with dimension and number of constraints

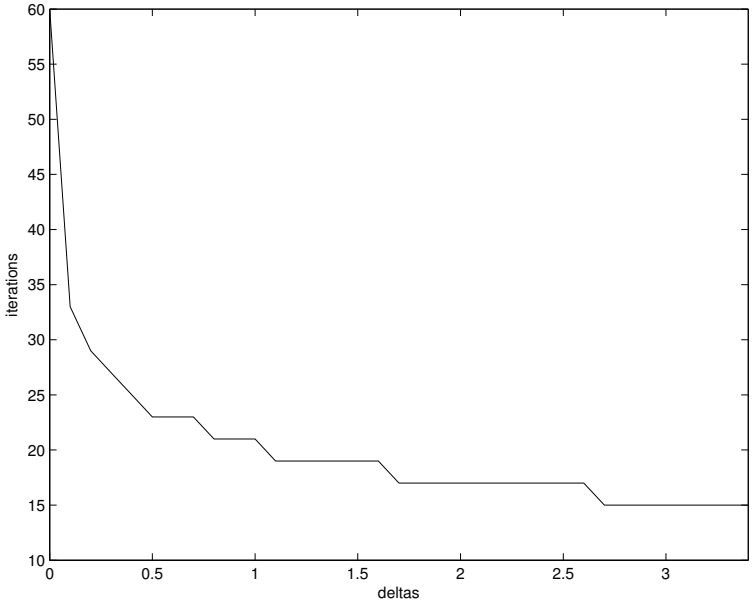

Figure 7:
This plot was generated for LMI Problem 1 using eigenvalue replacement.

All choices of δ that we tested in the interval $[3.4, 100]$ caused the algorithm to reach the maximum number of iterations allowed without finding a feasible point.

Figure 7 also shows a plot of δ versus the number of iterations for LMI Problem 1 using eigenvalue shift. In this problem, $\delta = 0$ and $\delta = 1$ both required 7 iterations. The least number of iterations required was 3 and that was obtained with all choices of δ that we tested in the interval $[39.9, 100]$.

Figure 8 shows the plot of δ versus the number of iterations for LMI Problem 3 using eigenvalue replacement. $\delta = 0$ required 350 iterations and $\delta = 1$ required 31 iterations. The least number of iterations required was 5 and that was obtained with all choices of δ that we tested in the interval $[8.4, 21.1]$. All choices of δ that were tested in the interval $[21.1, 100]$ caused the algorithm to reach the maximum number of iterations allowed without finding a feasible point.

Figure 9 shows the plot of δ versus the number of iterations for LMI Problem 3 using eigenvalue shift. $\delta = 0$ and $\delta = 1$ both required 5 iterations. This was the least number of iterations required and it was obtained with all δ's tested in the interval $[0, 16.3]$.

5.2 Comparison of Methods for Eigenvalue Modification

Table 2 compares the two methods of eigenvalue modification with $\delta = 0$.

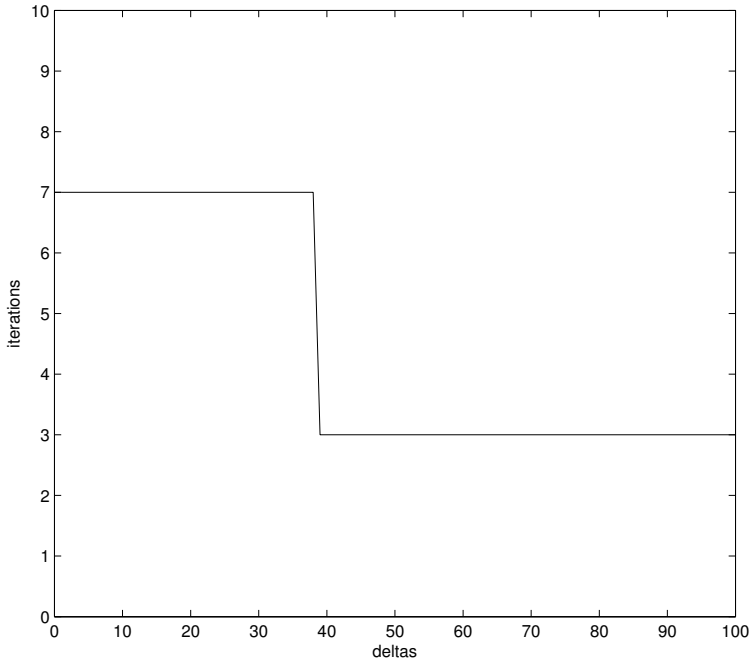

Figure 8:
This plot was generated for LMI Problem 1 using eigenvalue shift.

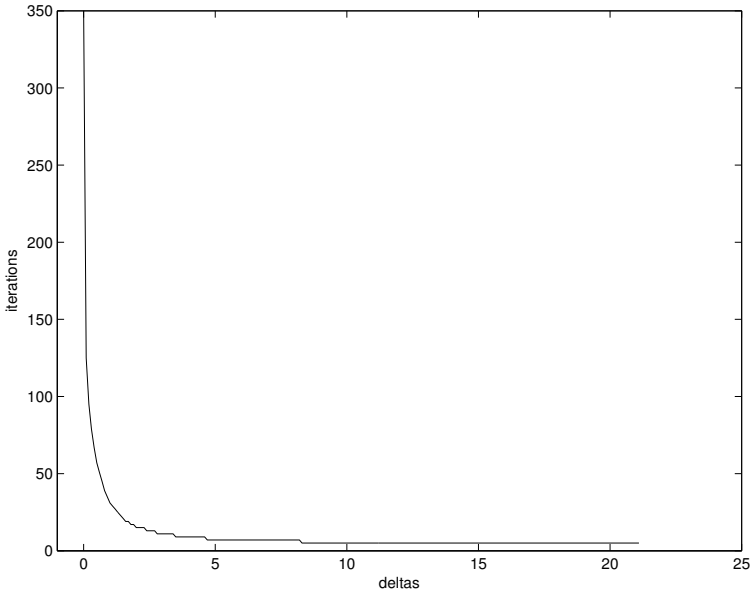

Figure 9:
This plot was generated for LMI Problem 3 using eigenvalue replacement.

	Eigenvalue Replacement		Eigenvalue Shift (Spectral)	
LMI Problem	Iterations	Time (sec)	Iterations	Time (sec)
1	60	0.0750	7	0.1261
2	8	0.0517	3	0.0112
3	350	0.2860	5	0.0123
4	NaN	NaN	5	0.0494
5	NaN	NaN	NaN	NaN
6	NaN	NaN	3	0.0635
7	NaN	NaN	3	0.0188
8	178	0.5015	NaN	NaN
9	NaN	NaN	5	0.0926
10	358	1.6094	NaN	NaN
11	NaN	NaN	5	0.1077
12	NaN	NaN	3	0.0141
13	102	0.3638	9	0.1374
14	NaN	NaN	5	0.0677
15	NaN	NaN	5	0.0965
16	5	0.0910	3	0.0194
17	NaN	NaN	3	0.0319
18	NaN	NaN	3	0.0265
19	NaN	NaN	NaN	NaN
20	54	0.1932	NaN	NaN
21	NaN	NaN	3	0.0235
22	246	0.6193	5	0.0695
23	13	0.1084	5	0.0391
24	92	0.1958	9	0.0965
25	94	0.1201	NaN	NaN

Table 2: This table was computed with $\delta = 0$.

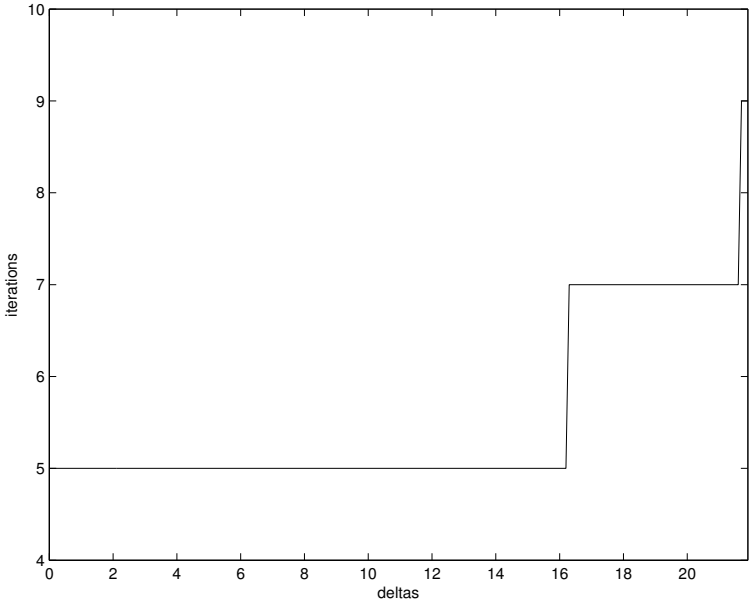

Figure 10:
This plot was generated for Problem 3 using eigenvalue shift.

Table 3 compares the two methods of eigenvalue modification using $\delta = 1$.

Table 4 compares eigenvalue replacement with $\delta = 0, 1,$ and 10.

Table 5 compares spectral decomposition eigenvalue shift with $\delta = 0, 1,$ and 10.

5.3 Second-Order Cone Implementation Results

In this section we will compare the performance of the SOC-specific implementation to the general implementation. We generated 25 second-order cone problems and expressed them as LMIs. For each problem, 0 is a feasible point.

Table 7 compares the results of the regular LMI implementation and the SOC implementation. Both use eigenvalue shift with $\delta = 25$.

Table 8 shows the results of using $\delta = 0$ and $\delta = 1$ in the SOC implementation.

6 Conclusions

The first thing we notice is that the choice of δ has an important effect on the performance of the algorithm with eigenvalue replacement. With $\delta = 0$, the algorithm failed to find a feasible point in the maximum number of iterations for 13 out of the 25 LMI test problems. With $\delta = 1$, a feasible point was found in every case. The algorithm also performed better in terms of time and number of

	Eigenvalue Replacement		Eigenvalue Shift	
Problem	Iterations	Time (sec)	Iterations	Time (sec)
1	21	0.0816	7	0.0286
2	3	0.0046	3	0.0029
3	31	0.0519	5	0.0122
4	55	0.4573	5	0.0720
5	155	1.0930	NaN	NaN
6	35	0.3400	3	0.0756
7	71	0.1787	5	0.0205
8	35	0.1407	NaN	NaN
9	45	0.1781	5	0.0565
10	41	0.2795	NaN	NaN
11	71	0.3520	5	0.1260
12	31	0.0976	3	0.0147
13	13	0.0939	7	0.1179
14	35	0.2995	5	0.0583
15	77	0.5542	5	0.0859
16	5	0.1047	3	0.0240
17	55	0.1192	3	0.0523
18	45	0.1648	3	0.0314
19	75	0.2713	NaN	NaN
20	9	0.0729	7	0.1110
21	181	0.2877	3	0.0247
22	35	0.1420	5	0.0621
23	9	0.1068	5	0.0378
24	17	0.0817	7	0.0606
25	11	0.0445	NaN	NaN

Table 3: This table was computed with $\delta = 1$.

	$\delta = 0$		$\delta = 1$		$\delta = 10$	
LMI Problem	Iterations	Time (sec)	Iterations	Time (sec)	Iterations	Time (sec)
1	60	0.0750	21	0.0616	NaN	NaN
2	8	0.0517	3	0.0046	3	0.0017
3	350	0.2860	31	0.0519	5	0.0037
4	NaN	NaN	55	0.4573	5	0.0497
5	NaN	NaN	155	1.0930	NaN	NaN
6	NaN	NaN	35	0.3400	5	0.0372
7	NaN	NaN	71	0.1787	5	0.0077
8	178	0.5015	35	0.1407	13	0.0429
9	NaN	NaN	45	0.1781	7	0.0298
10	358	1.6094	41	0.2795	9	0.0896
11	NaN	NaN	71	0.3520	7	0.0521
12	NaN	NaN	31	0.0976	7	0.0098
13	102	0.3638	13	0.0939	7	0.0473
14	NaN	NaN	35	0.2995	5	0.0319
15	NaN	NaN	77	0.5542	7	0.0456
16	5	0.0910	5	0.1047	3	0.0112
17	NaN	NaN	55	0.1192	7	0.0125
18	NaN	NaN	45	0.1648	7	0.0237
19	NaN	NaN	75	0.2713	NaN	NaN
20	54	0.1932	9	0.0729	5	0.0178
21	NaN	NaN	181	0.2877	NaN	NaN
22	246	0.6193	35	0.1420	7	0.0195
23	13	0.1084	9	0.1068	7	0.0265
24	92	0.1958	17	0.0817	9	0.0164
25	94	0.1201	11	0.0445	5	0.0042

Table 4: Comparison of eigenvalue replacement with $\delta = 0, 1,$ and 10.

	$\delta = 0$		$\delta = 1$		$\delta = 10$	
LMI Problem	Iterations	Time (sec)	Iterations	Time (sec)	Iterations	Time (sec)
1	7	0.1261	7	0.0286	8	0.0808
2	3	0.0112	3	0.0029	3	0.0017
3	5	0.0123	5	0.0122	5	0.0038
4	5	0.0494	5	0.0720	4	0.0361
5	NaN	NaN	NaN	NaN	NaN	NaN
6	3	0.0635	3	0.0756	4	0.0368
7	3	0.0188	5	0.0205	5	0.0076
8	NaN	NaN	NaN	NaN	5	0.0209
9	5	0.0926	5	0.0565	5	0.0279
10	NaN	NaN	NaN	NaN	6	0.0681
11	5	0.1077	5	0.1260	5	0.0522
12	3	0.0141	3	0.0147	5	0.0065
13	9	0.1374	7	0.1179	8	0.0548
14	5	0.0677	5	0.0583	4	0.0321
15	5	0.0965	5	0.0859	4	0.0330
16	3	0.0194	3	0.0240	3	0.0113
17	3	0.0319	3	0.0523	6	0.0203
18	3	0.0265	3	0.0314	3	0.0141
19	NaN	NaN	NaN	NaN	NaN	NaN
20	NaN	NaN	7	0.1110	7	0.0227
21	3	0.0235	3	0.0247	2	0.0068
22	5	0.0695	5	0.0621	4	0.0157
23	5	0.0391	5	0.0378	5	0.0194
24	9	0.0965	7	0.0606	6	0.0138
25	NaN	NaN	NaN	NaN	4	0.0041

Table 5: Comparison of spectral eigenvalue shift with $\delta = 0, 1,$ and 10

SOC Problem	m	n	q
1	2	2	2
2	3	2	3
3	14	3	2
4	9	5	15
5	8	13	15
6	11	6	12
7	5	2	7
8	3	7	8
9	10	5	11
10	6	15	9
11	14	11	8
12	13	3	4
13	3	14	6
14	13	6	9
15	9	6	12
16	15	12	2
17	12	3	6
18	11	10	4
19	4	4	9
20	2	7	7
21	6	5	5
22	4	3	13
23	9	13	3
24	2	4	7
25	3	3	3

Table 6: SOC Test Problems

	LMI Implementation		SOC Implementation	
SOC Problem	Iterations	Time (sec)	Iterations	Time (sec)
1	3	0.0350	3	0.0317
2	19	0.0267	19	0.0150
3	233	0.1227	233	0.1137
4	229	1.2274	229	1.2222
5	221	1.8352	221	1.8239
6	NaN	NaN	NaN	NaN
7	3	0.0041	3	0.0032
8	5	0.0196	5	0.0154
9	403	1.3059	403	1.2163
10	5	0.0643	5	0.0447
11	89	0.3951	89	0.3821
12	99	0.1428	99	0.1332
13	3	0.0247	3	0.0148
14	NaN	NaN	NaN	NaN
15	NaN	NaN	NaN	NaN
16	7	0.0171	7	0.0140
17	143	0.2115	143	0.2127
18	115	0.2865	115	0.2790
19	447	0.9184	447	0.8900
20	5	0.0169	5	0.0149
21	285	0.4079	285	0.4042
22	67	0.1661	67	0.1649
23	5	0.0188	5	0.0139
24	3	0.0062	3	0.0056
25	3	0.0025	3	0.0016

Table 7: Comparison of LMI implementation and SOC implementation with $\delta = 25$

	$\delta = 0$		$\delta = 1$	
SOC Problem	Iterations	Time (sec)	Iterations	Time (sec)
1	NaN	NaN	NaN	NaN
2	NaN	NaN	5	0.1119
3	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN
7	NaN	NaN	3	0.0038
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN
10	NaN	NaN	NaN	NaN
11	NaN	NaN	NaN	NaN
12	NaN	NaN	NaN	NaN
13	4	0.0278	4	0.0221
14	NaN	NaN	NaN	NaN
15	NaN	NaN	NaN	NaN
16	7	0.0137	3	0.0073
17	NaN	NaN	NaN	NaN
18	NaN	NaN	NaN	NaN
19	NaN	NaN	NaN	NaN
20	7	0.0205	7	0.0207
21	NaN	NaN	NaN	NaN
22	NaN	NaN	NaN	NaN
23	NaN	NaN	NaN	NaN
24	NaN	NaN	NaN	NaN
25	3	0.0024	7	0.0026

Table 8: SOC implementation using eigenvalue shift with $\delta = 0$ and $\delta = 1$.

iterations with $\delta = 1$. In 11 of the 12 problems for which both cases converged, $\delta = 1$ found a feasible point in less iterations. In the other problem the number of iterations was the same. In each of the 11 problem where $\delta = 1$ required less iterations, it also required less time. But in the problems where the number of iterations was the same, $\delta = 0$ required less time.

We also tested the performance of the algorithm using eigenvalue replacement with $\delta = 10$. Here a feasible point was found in 21 out of the 25 test problems. In 20 of these, the algorithm required less iterations than $\delta = 0$ and $\delta = 1$. In the other problem $\delta = 0$ did not find a feasible point and $\delta = 1$ and $\delta = 10$ both needed 3 iterations. In all 20 problems $\delta = 10$ used less time then either $\delta = 0$ or $\delta = 1$.

The choice of δ also had an effect in eigenvalue shift. We tested the algorithm with $\delta = 0$, $\delta = 1$, and $\delta = 10$. $\delta = 0$ found a feasible point in 19 of the 25 problems, $\delta = 1$ found a feasible point in 20 of the 25 problems, and $\delta = 10$ found a feasible point in 23 of the 25 problems. For the two problems in which $\delta = 10$ did not find a feasible point, both $\delta = 0$ and $\delta = 1$ also did not find a feasible point.

It is interesting that the number of iterations required by the algorithm is similar for these three choices of δ. The time required by the algorithm is similar with $\delta = 0$ and $\delta = 1$. But $\delta = 10$ needed less time than either $\delta = 0$ or $\delta = 1$ in each of the problems for which it found a feasible point except for LMI Problem 1.

Our data gives evidence that the algorithm performs better with eigenvalue shift than with eigenvalue replacement. With $\delta = 0$, eigenvalue replacement found a feasible point in 12 out of the 25 test problems while eigenvalue shift found a feasible point in 19 of the problems. There were 8 problems for which replacement and shift both found a feasible point. Eigenvalue shift had less iterations in all 8 of these and less time in 7 of the 8.

With $\delta = 1$, eigenvalue replacement found a feasible point in all 25 test problems and eigenvalue shift found a feasible point in 20 of the test problems. In the 20 problems where both methods found a feasible point, eigenvalue shift had less iterations in 19 and the methods had the same number of iterations in the other problem. Eigenvalue shift used less time in 19 of the 20 problems.

7 Future Work and Acknowledgments

The next step would like to compare this algorithm to other algorithms for solving feasibility problems that a used in practice. How does our method compare to methods that are currently used? It would also be interesting to modify the algorithm to handle additional constraints of the form

$$A_i \bullet X = b_i.$$

This research was conducted at Northern Arizona University under the NSF-supported REU program. I would also like to thank Dr. Shafiu Jibrin for mentoring the project.

References

[1] Heinz H. Bauschke and Jonathan M. Borwein. On projection algorithms for solving convex feasibility problems. *SIAM Rev.*, 38(3):367–426, 1996.

[2] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. Applications of second-order cone programming. *Linear Algebra Appl.*, 284(1-3):193–228, 1998. ILAS Symposium on Fast Algorithms for Control, Signals and Image Processing (Winnipeg, MB, 1997).

[3] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer Series in Operations Research. Springer-Verlag, New York, 1999.

[4] M. Ait Rami, U. Helmke, and J. B. Moore. A finite steps algorithm for solving convex feasibility problems. *J. Global Optim.*, 38(1):143–160, 2007.

[5] Walter Rudin. *Real and complex analysis*. McGraw-Hill Book Co., New York, third edition, 1987.

[6] Gilbert Strang. *Linear algebra and its applications*. Academic Press [Harcourt Brace Jovanovich Publishers], New York, second edition, 1980.

[7] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Rev.*, 38(1):49–95, 1996.