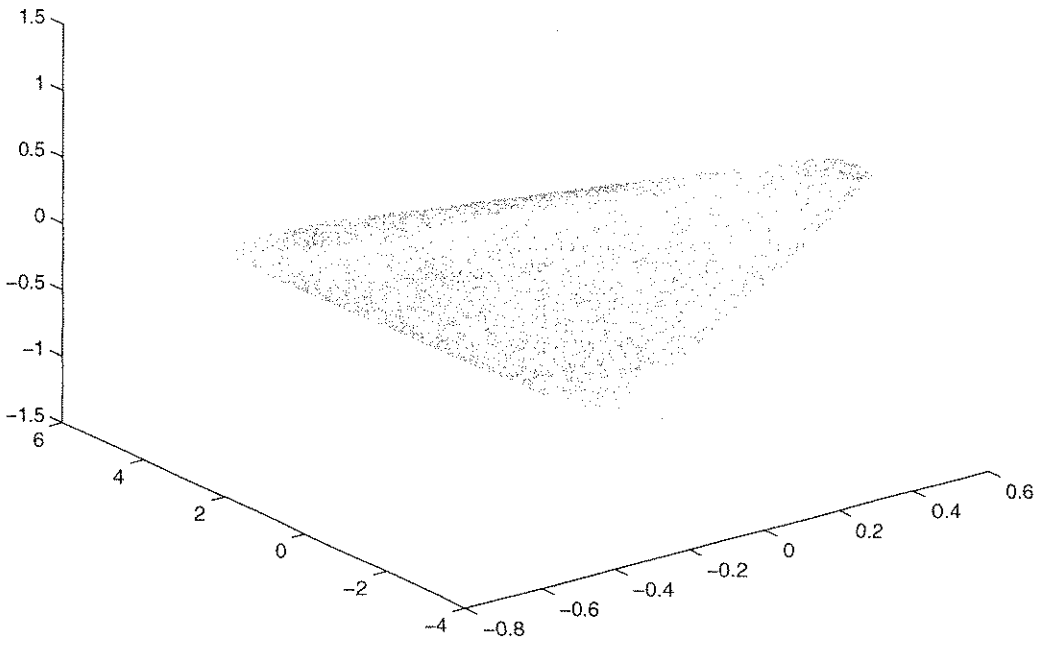
Improving the Semidefinite Coordinate Direction (SCD) Method for the Detection of Necessary Linear Matrix Inequalities

Steve Formaneck
Department of Mathematics and Statistics
Northern Arizona University, Flagstaff, Arizona 86011, U.S.A.

July 25, 2001



Abstract

Reducing the size of semidefinite programming problems by finding necessary linear matrix inequalities is a valuable tool in the field of operations research. There are three variations of the semidefinite stand-and-hit (SSH) method for detecting necessary linear matrix inequalities. One such method is the semidefinite coordinate direction (SCD) method. In this project, I improve the SCD method.

I will show ways in which to detect and handle an unbounded feasible region for a set of LMIs within the SCD algorithm. Further, I will investigate alterations to the SCD method to improve computation time, and thus create a more efficient algorithm. MATLAB examples are used to test, illustrate, and emphasize the ideas expressed.

Contents

1 Introduction

1.1 Background

A *semidefinite program (SDP)* is defined as follows:

$$\begin{aligned} &\min c^T x \\ &s.t. A^{(j)}(x) := A_0^{(j)} + \textstyle\sum_{i=1}^{n} x_i A_i^{(j)} \succeq 0, \qquad\qquad j = 1, 2, ..., q \end{aligned}$$

where $A_i^{(j)}, i = 0, 1, ..., n$ are $m_j \times m_j$ symmetric matrices and $x, c \in \Re^n$ and q represents the number of constraints. The following is an example of a system of 5 LMI constraints:

LMI Example 1:

$$\begin{aligned} A^{(1)}(x) &= \begin{bmatrix} 10 & -1 \\ -1 & 5 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0 \\ A^{(2)}(x) &= \begin{bmatrix} 10 & 0 \\ 0 & 3 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0 \\ A^{(3)}(x) &= \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix} + x_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0 \\ A^{(4)}(x) &= \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0 \\ A^{(5)}(x) &= \begin{bmatrix} 4.6 & 0 \\ 0 & 1 \end{bmatrix} + x_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \succeq 0. \end{aligned}$$

There are many reasons why semidefinite programming is important. For one, positive semidefinite, as well as positive definite, constraints arise in many important applications. There are many applications in engineering, statistics and combinatorial optimization. Such applications include finding optimal travel routes for UPS deliveries and bin packing for airlines. Second, many convex optimization problems, such as linear programming, can be cast as semidefinite programs. Thus, semidefinite programming offers a unified way to derive algorithms and properties for a wider amount of convex optimization problems than can be covered in linear programming alone. A third, and most important reason to study semidefinite programming is that they can be solved very efficiently, both in theory as well as in practice [6].

The solution set of a system of LMIs is called the *feasible region*. An LMI constraint is defined as *necessary* if removing the constraint causes the feasible region to change. A *redundant* LMI constraint is a constraint that doesn't change the system's feasible region when removed. In the the example system, the following feasible region is indicated by figure 1.

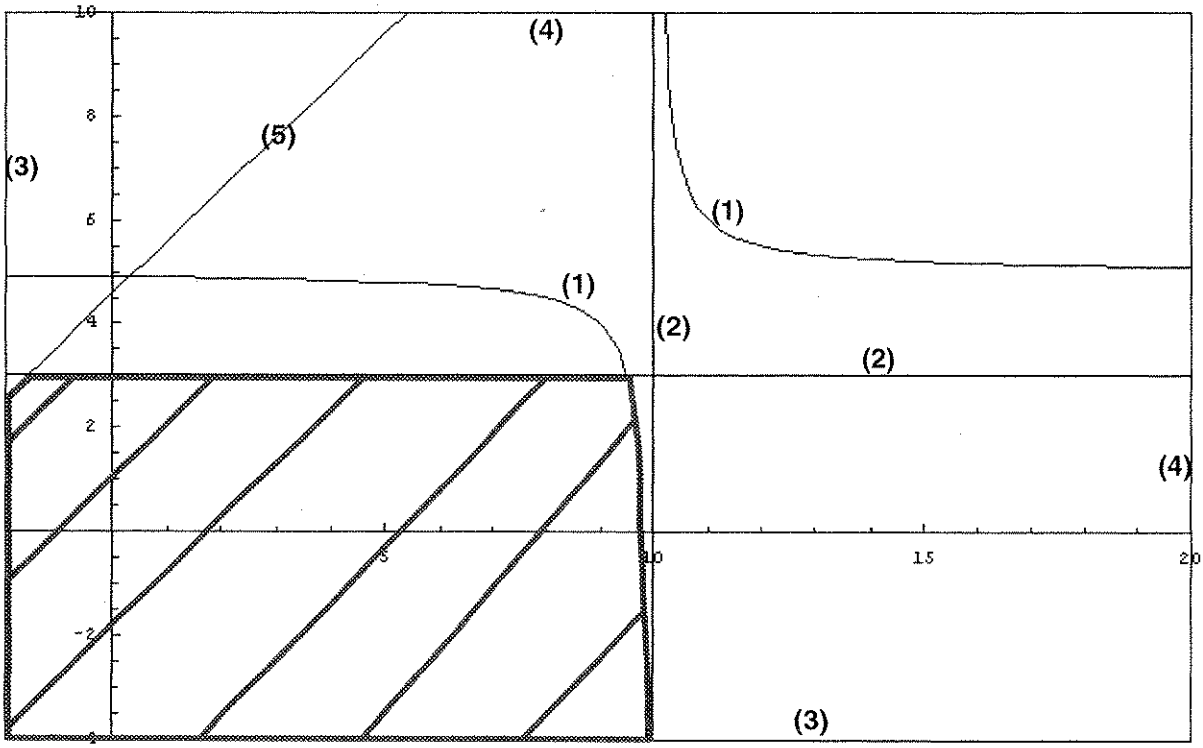


Figure 1: LMI Example 1.

Figure 1 shows that constraints 1,2,3 and 5 are necessary, whereas constraint 4 is redundant since removing it would not alter the size of the feasible region.

Another LMI example that is used in the paper for testing purposes is shown below:

LMI Example 2:

$$
\begin{aligned}
A^{(1)}(x) &= \begin{bmatrix} 5 & -1 \\ -1 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0 \\
A^{(2)}(x) &= \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \succeq 0 \\
A^{(3)}(x) &= \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0 \\
A^{(4)}(x) &= \begin{bmatrix} 3.8 & 0 \\ 0 & 3.8 \end{bmatrix} + x_1 \begin{bmatrix} -0.4 & 0 \\ 0 & -0.4 \end{bmatrix} + x_2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0 \\
A^{(5)}(x) &= \begin{bmatrix} 2.6 & 0 \\ 0 & 2.6 \end{bmatrix} + x_1 \begin{bmatrix} 0.8 & 0 \\ 0 & 0.8 \end{bmatrix} + x_2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \succeq 0.
\end{aligned}
$$

For this system, the feasible region is as follows:

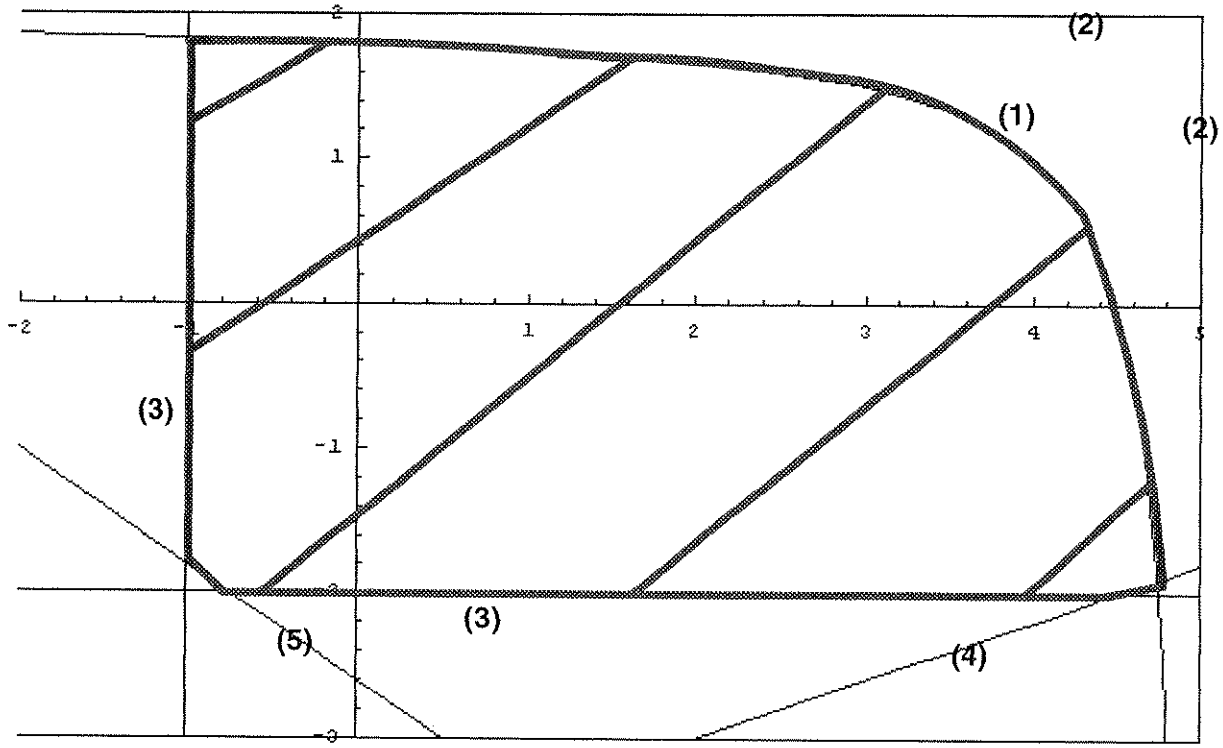


Figure 2: LMI Example 2.

For Example 2, constraints 1,3,4, and 5 are necessary constraints, while constraint 2 is redundant. Knowing necessary constraints is very helpful in reducing computation time in solving SDPs. This is especially true for a large system containing several constraints and variables, as well as large matrices. For these systems, knowing all the necessary constraints allows for removal of the redundant constraints which are not required in the SDP solution process. With fewer constraints in the system, solving the system is computationally less expensive. For more on redundancy, refer to [1].

The problem of deciding whether or not a constraint is redundant is an NP-complete problem [3]. In the three varying methods, finding a maximum generalized eigenvalue, λ_{max} and a minimum eigenvalue, λ_{min} is critical in detecting the necessary constraints of a system. There are a number of methods to choose from to find these values. One of these algorithms uses cholesky decomposition to aid in the process of finding these values. Its complexity is $O(m_j^3)$ with specifically $m_j^3/6+O(m_j^2)$ multiplications and $m_j^3/6 + O(m_j^2)$ additions [5]. The algorithm is as follows:

Let A be a symmetric and positive definite matrix. Cholesky decomposition breaks down the matrix into the form $A = LL^T$, where L is a lower triangular matrix. To derive A,we simply equate coefficients on both sides of the equation to obtain:

$a_{11} = l_{11}^2 \longrightarrow l_{11} = \sqrt{a_{11}}$
$a_{21} = l_{21}l_{11} \longrightarrow l_{21} = a_{21}/l_{11}, ...l_{n1} = a_{n1}/l_{11}$
$a_{22} = l_{21}^2 + l_{22}^2 \longrightarrow l_{22} = \sqrt{(a_{22} - l_{21}^2)}$

$a_{32} = l_{31}l_{21} + l_{32}l_{22} \longrightarrow l_{32} = (a_{32} - l_{31}l_{21})/l_{22}, etc.$

So,in general for $i = 1, ..., n$ and $j = 1 + 1, ..., n$:

$l_{ii} = \sqrt{(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2)}$
$l_{ji} = a_{ji} - \sum_{k=1}^{i-1} l_{jk}l_{ik})/l_{ii}$

And since A is symmetric and positive definite, the expression under the square root is guaranteed to always be positive, and l_{ij} are always real. In addition, the matrix L is positive definite.

1.2 Semidefinite Stand-and-Hit (SSH) Method

The SSH algorithm, a Monte Carlo algorithm for detecting necessary constraints, is constructed in the following way, assuming that the feasible region, $\Re$, is a completely bounded region [3]. First, fix a *standing point* x_0 in the interior of $\Re$. A random direction s (chosen uniformly from the surface of the unit hypersphere, $S^{n-1} = \{x\varepsilon\Re^n : x_1^2 + x_2^2 + ... + x_n^2 = 1\}$) and x_0 define two line segments $\{x_0 + \sigma s | \sigma_+ \geq 0\}$ and $\{x_0 - \sigma_- s | \sigma \geq 0\}$. These two points intersect with the boundary of the feasible region, $\Re$, at places called *hit points*. These hit points identify one or two necessary constraints to the system. The distances from hit point to boundary are defined by σ_+ and σ_- respectively. The distances I am referring to can be seen in Figure 3 below.
Given the standing point x_0 and direction vector s form the symmetric matrix:

$$B_j(s, x_0) = -A^j(x_0)^{-1/2}(\sum_{i=1}^{n} s_i A_i^{(j)})A^j(x_0)^{-1/2}$$

The SSH algorithm involves calculating the maximum and minimum eigenvalues of $B_j(s, x_0)$ and finding $\sigma_+^{(j)}, \sigma_-^{(j)}$ for $1 \leq j \leq q$. σ in this case is defined as $1/\lambda$ and $\sigma_+^{(j)}$ is the largest value of σ found for constraint j, while $\sigma_-^{(j)}$ is the smallest value of σ found for constraint j. Next, calculate $\sigma_- = min\{\sigma_-^{(j)} | 1 \leq j \leq q\}$ and $\sigma_+ = min\{\sigma_+^{(j)} | 1 \leq j \leq q\}$. Now, for $1 \leq j \leq q$, if $\sigma_+^{(k)} = \sigma_+$ or $\sigma_-^{(k)} = \sigma_-$ and $k \notin \Gamma$, set $\Gamma = \Gamma \bigcup k$. Choose u from $\bigcup(0, 1)$ and set $x_0 = x_0 + (u(\sigma_+ + \sigma_-) - \sigma_-)s$ [3].

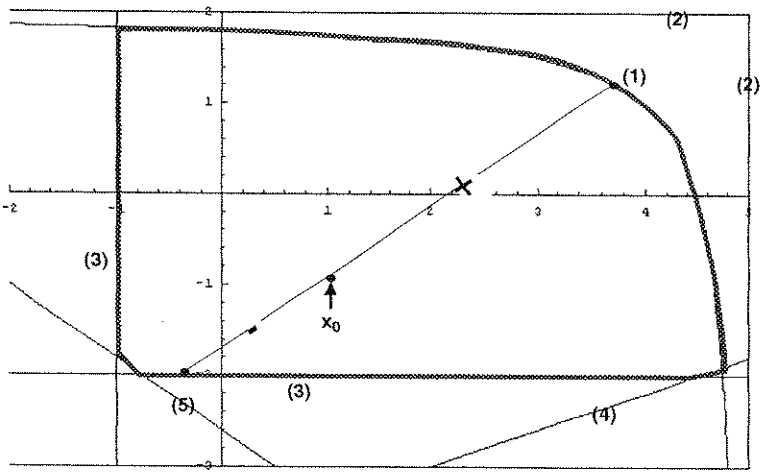


Figure 3: Example 2 showing distances from standing point to hitting points on boundaries.

For each new iteration of the algorithm, a new direction vector is created by choosing the new vector from a uniform distribution as described earlier. The same standing point is used in each iteration throughout the algorithm and the necessary constraints for each successive vector are detected. This process continues until a stopping rule is reached. For example, the algorithm may be told to stop after 1,000 iterations or after finding 5 necessary constraints.

1.3 Variations of the SSH Method

The semidefinite hypersphere direction (SHD) method is similar the the SSH method with one notable exception: a new standing point, x_i, is randomly chosen from the feasible region of the system each successive iteration. This new point is uniformly chosen along the line segment joining the two hit points of the previous iteration. In each iteration new hit points are calculated using the new standing point and direction vector.

The SCD algorithm, a Monte Carlo algorithm for detecting necessary constraints, is constructed in the following way, assuming that the feasible region, $\Re$, is a completely bounded region. First, fix a *standing point* x_0 in the interior of $\Re$. A random direction s (chosen from $2n$ directions e_i parallel to the coordinate axis) and x_0 define two line segments $\{x_0 + \sigma s | \sigma \geq 0\}$ and $\{x_0 - \sigma s | \sigma \geq 0\}$. These two points intersect with the boundary of the feasible region, $\Re$, at *hit points* which identify at least one necessary constraint in the system.

The symmetric matrix given by the standing point x_0 and direction vector s is given by the equation:

$$B_j(s, x_0) = -A^j(x_0)^{-1/2}(\sum_{i=1}^{n} s_i A_i^{(j)})A^j(x_0)^{-1/2} = -A^j(x_0)^{-1/2}A_i^{(j)}A^j(x_0)^{-1/2}$$

2 Using Cholesky Decomposition in SCD Hitting Step

The idea used behind the previous equation is Schur Decomposition. This is where $A = A^{1/2} * A^{1/2}$. Another way to break down A is with Cholesky decomposition as described earlier. The Cholesky Method results in changing the formula to the following:

$$B_j^*(s, x_0) = -L^{-1}(\sum_{i=1}^{n} s_i A_i^{(j)})(L^{-1})^T \text{ where } 1 \leq j \leq q.$$

Here we assign the smallest and largest eigenvalues of a symmetric matrix B^* by $\lambda_{min}(B^*)$ and $\lambda_{max}(B^*)$ respectively.

Theorem 2.1 Assume the ray $\{x_0 + \sigma s | \sigma \geq 0\}(\{x_0 - \sigma s | \sigma \geq 0\})$ intersects the boundary of $A^{(j)}(x) \geq 0$ at $x_0 + \sigma_+ s(x_0 - \sigma_- s)$. Then

$$\sigma_+^{(j)} = 1/\lambda_{max}^+(B_j^*(s, x_0)) \quad (1)$$

$$\sigma_-^{(j)} = -1/\lambda_{min}^-(B_j^*(s, x_0)) \quad (2)$$

Proof :
We show when a hitting point is on the boundary of the feasible region, that it must have at least one eigenvalue of zero, but when the point's in the interior, the value is of all eigenvalues are greater

than zero.

$\sigma_+^{(j)} = max\{\sigma | \sigma > 0, A^{(j)}(x_0 + \sigma s) \geq 0\}$

$= min\{\mu | \mu > 0, det[A^{(j)}(x_0 + \mu s)] = 0\}$

When $\mu > 0$

$det[A^{(j)}(x_0 + \mu s)] = 0$

$\iff det[A^{(j)}(x_0) + \mu \Sigma_{i=1}^n A_i A_i^{(j)}] = 0$

$\iff det[(1/\mu)I + L^{-1}(\Sigma_{i=1}^n A_i A_i^{(j)})(L^{-1})^T] = 0$

$\iff det[(1/\mu)I - B_j^*(s, x_0)] = 0$

$\iff 1/\mu$ is an eigenvalue of $B_j^*(s, x_0)$.

Thus : $\sigma_+^{(j)} = min\{\mu | \mu > 0,\ 1/\mu$ is an eigenvalue of $B_j^*(s, x_0)\}$

$= min\{1/\lambda | \lambda > 0,\ \lambda$ is an eigenvalue of $B_j^*(s, x_0)\}$

$= 1/\lambda_{max}^+(B_j^*(s, x_0))$

A similar proof can be shown for (2).

For each new iteration of the algorithm, a new standing point is created by uniformly picking the new point from the line segment created by the two hit points from the last iteration, just as in the SHD algorithm. The major computational problem with changing the standing point every iteration is that $A^j(x_0)^{-T}$ must be recomputed for every such iteration.

Comparing speeds between Cholesky decomposition and Schur decomposition yielded the results with variations in matrix size, number of variables, and number of constraints described in Table 1.

As shown from the table, as the time to run the methods increases due to larger problems, Cholesky factorization seems to be faster and thus less computational.

Although cholesky decomposition is faster and tends to be computationally less expensive than schur decomposition, there is one factor that goes against the cholesky method, that is accuracy when $A^{(j)}(x_0)$ is ill-conditioned. It follows that $B_j(s, x_0)$ is ill-conditioned when $A^{(j)}(x_0)$ is ill-conditioned [4]. When $A^{(j)}(x_0)$ is ill-conditioned, using schur decomposition can negate this problem and calculate eigenvalues quite accurately due to how $A(x_0) = A(x_0)^{1/2}A(x_0)^{1/2}$ is formed. The way cholesky decomposition is constructed can lead to roundoff errors for L and may result in less accurately computed eigenvalues. This occurs mostly for the smaller (in magnitude) eigenvalues from a given matrix [7].

	LMI		*time(seconds)*	
q	n	m	cholesky	schur
5	2	2	6.3290	11.2760
11	11	11	55.7400	139.1900
20	20	20	418.8220	814.3310
15	15	12	61.1580	198.7260
10	20	15	94.3560	230.77720
30	4	5	32.5570	122.3060
40	5	5	40.3690	99.8640
50	2	2	3.1240	3.0840
5	20	3	.1310	.0800
10	30	5	4.3560	1.1410

Table 1: Time Comparison to Run SCD Method using Cholesky and Schur Decompositions (1,000 iterations)

Since the only eigenvalues that are of interest when performing the SCD algorithm are the largest (in magnitude) positive and negative eigenvalues, this is not really a serious problem. The situations in which this is more likely a factor is for a point near the boundary. If you recall, the distances to the boundaries increase in length as the eigenvalues become smaller, from the factor $1/\lambda$. Thus, when a standing point is in something like an enormous ellipse of 1 constraint and near the boundary of that constraint, the eigenvalue computed to the furthest boundary will be extremely small, resulting in a possibility of great round-off errors.

Figure 5 shows standing points that were chosen both near the boundary and closer to the center of the feasible region. Using the points from the figure as standing points, one iteration of the SCD method ran using only the horizontal vectors for the choice of direction, for each individual point. The results were that both methods found the exact hitting points on the boundaries, to 4 decimal places (only 4 decimal places were computed).

Analyzing further, the standing point (0,7.7999) was chosen and the eigenvalue found to constraint 1 was 9999.9999991742 using cholesky decomposition. Schur decomposition was extremely close to that of cholesky with a value of 9999.9999991795.

Another point, (9.8745,-2.99) was chosen and tested. The results were quite similar, with cholesky resulting in an eigenvalue of 2910.74681238068 compared to schur with 2910.74681238101. These differences are not going to come close to throw off which constraints are detected for this problem. Yet, in order to test ill-conditioned matrices with cholesky factorization further, it would be best to use larger sized matrices with both very large eigenvalues and very small eigenvalues and more rigorously test these situations.

Since the SCD algorithm is not concerned with precise calculations for distances to boundaries, but just the detection of the correct, necessary constraints, cholesky decomposition does an adequate job and at a faster rate. The only places that it has a slight chance at failing is in extremely large

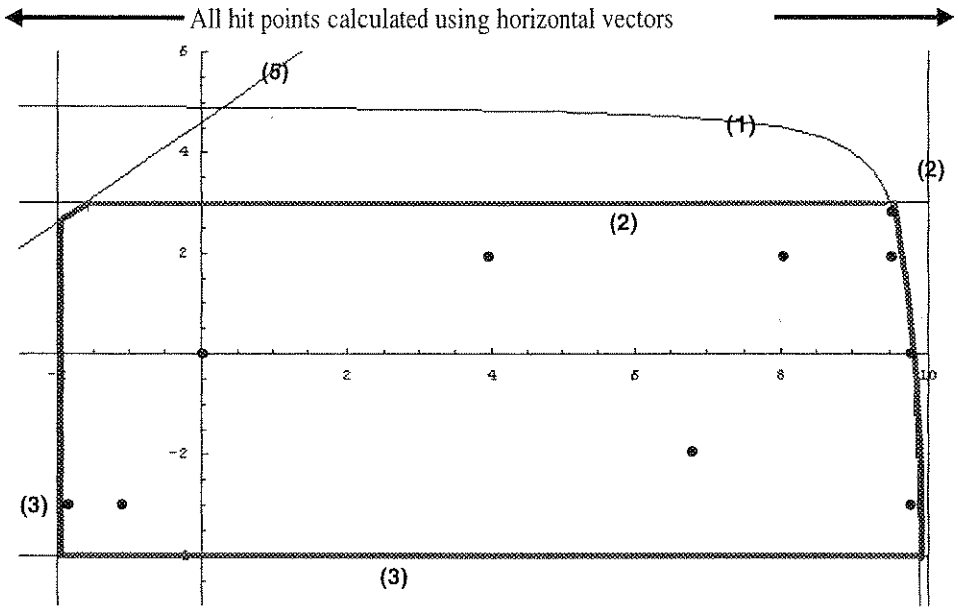


Figure 4: Testing Points for Accuracy of Schur and Cholesky Decompositions.

feasible regions, which tend not to occur frequently in most SDP problems.

3 Finding The Boundary of a Feasible Region Using SCD

As with the SSH and SHD algorithms, the search vector is chosen through a uniform distribution of $2n$ coordinate vectors, while the standing point, x_0, is chosen uniformly between the two hit points. The figure below suggests that the hitting points are asymptotically uniform on the boundary of the feasible region. Figure 5 shows the distribution of hit points for a case where $q = 10$, $n = 2$,$m = 3$, and 100 iterations (200 total hit points).

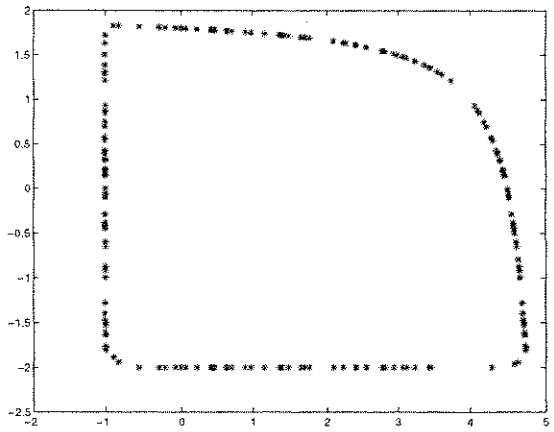


Figure 5: 2D boundary plot of hit points for Example 2

When running the SCD algorithm on the system of LMIs for 10,000 iterations, the boundary is well bolded with points, as seen in figure 6. This shows the boundary of the feasible region is well detected.

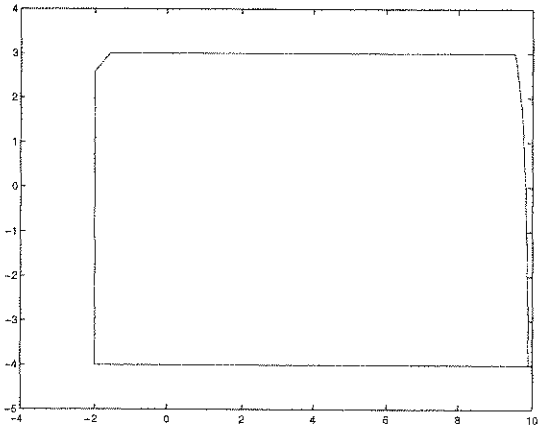


Figure 6: 2D boundary plot of hit points for Example 1

4 Improving Vector Selection for SCD Method

When choosing a direction vector for finding hit points for the SCD method, choosing the same direction multiple times back-to-back can lead to a waste of iterations. This is due to the fact that the same hit points will be found if the same direction is chosen two or more times in a row. The solution is simple to this problem: after the first iteration, limit the choice of direction for each successive iteration to all possible directions except the previous chosen direction.

LMI	*time(seconds)*		*iterations*	
n	new	old	new	old
2	4.8970	19.6580	47	189
2	13.5990	13.0790	137	132
3	12.0070	16.3540	108	147
3	9.8240	37.4440	93	355
4	128.0240	219.4660	1012	1862
5	50.9140	94.606	399	742
6	825.6670	1.0909 e+003	5829	7454

Table 2: Time and Iteration Comparisons between new and old Vector Selection Methods (with q=50, m=3)

As shown from Table 2, forcing a change in the direction vectors for each iteration improves the speed and the number of iterations it takes to find all necessary constraints of an SDP problem. It is also reasonable to conclude that as the value of n nears 2, the new direction method becomes more and more useful since the probability of choosing the same vector as last iteration increases as n becomes smaller.

5 Detecting and Handling Unbounded Situations in SCD

Unbounded feasible regions can cause difficulties for nonlinear interval analysis of LMIs. An iteration of SCD could result in the detection of an infinite amount of feasible points for a specific direction, so that the variable ranges cannot be tightened and a hitting point is impossible to find.

One approach for solving this problem is that of using a *nucleus box* [2]. The idea behind this method is to create a small box around the center of the origin and increase the box's size when unboundedness is found for that specific box. Then only the segments of the box that were detected as unbounded are first increased by +1. As unboundedness keeps occurring for that segment, the scaling of the segment's distance is increased by powers of $10(10^1, 10^2, ..., 10^5)$. This approach is reasonable, yet knowing where to set the bounds on expanding the box is very sketchy. Also, the nucleus box method adds more complexity by adding up to four more constraints to the system.

Another solution to the problem, and the one used in this research, involves looking at the eigenvalues of B_j. If all the eigenvalues of the calculated B_j are strictly negative, then the feasible region in the positive vector direction is unbounded. Conversely, if all eigenvalues found are positive, then the negative direction vector is proven to be unbounded.

Once unboundedness is found, then the SCD method should be able to continue searching for constraints until the halting point has been reached. There are several ways to approach this problem. The 3 methods approached and implemented are as follows:

1. Method implements change of direction for each iteration as described previously. Thus, when unboundedness is detected, a new search direction is chosen using the same point as in the last iteration.

2. Method does not implement the change of direction for each iteration. When unboundedness is detected, method picks a new standing point from a uniform distribution on the line connecting the standing point and the one hitting point detected, both from the last iteration.

3. Method implements change of direction for each iteration as described previously. When unboundedness is detected, method picks a new standing point from a uniform distribution on the line connecting the standing point and the one hitting point detected, both from the last iteration.

Running 10 different unbounded LMIs through the 3 methods above yields the the results for speed shown in table 3. All the unbounded cases were created by removing selected constraints from the

2 examples in the paper.

	LMI	*Time For* 10,000 *iterations*(*seconds*)		
Example	*Constraints*	*Method* 1	*Method* 2	*Method* 3
2	3,4,5	48.1890	48.6400	48.9900
2	1,2,4	48.1690	48.5400	49.0810
2	3,5	32.5870	33.5480	34.0890
2	1,2	32.7980	32.5470	33.0370
2	4,5	33.1480	33.2970	33.8380
1	1,2,4,5	63.3010	62.6700	64.8130
1	3,5	33.0580	33.3180	34.0490
1	1,2,5	48.2300	48.4800	49.4410
1	1,2,4	48.7900	47.7290	48.1990
1	2,4,5	47.8690	47.5980	48.2190

Table 3: Testing Iteration Speed of Various Unbounded LMI detection and Handling Methods

As shown in Table 3, Method 1 and Method 2 are very similar in speed. Method 3 is slower than both Method 1 and Method 2. It is worthy to note that there are only $2n$ directions to pick vectors from. This leads to the change of direction method to be very useful in not repeating as many hit points. If tested on an SDP with a larger value for n, the change of direction would not be as useful.

On the same 10 examples tests were conducted to find how quick and in how many iterations each algorithm could find all necessary constraints. The results are listed in Table 4:

	LMI	*Finding Redundant Constraints*(*seconds*(*iterations*))		
Example	*Constraints*	*Method* 1	*Method* 2	*Method* 3
2	3,4,5	0.1800(37)	0.0600(10)	0.0200(4)
2	1,2,4	0.0710(15)	0.0400(7)	0.0500(9)
2	3,5	Stuck	0.0200(3)	0.0200(4)
2	1,2	Stuck	0.0200(4)	0.0200(3)
2	4,5	0.0100(2)	0.0100(1)	0.0100(1)
1	1,2,4,5	0.0900(14)	0.0700(10)	0.1000(14)
1	3,5	0.0100(1)	0.0200(3)	0.0100(1)
1	1,2,5	0.0100(2)	0.0100(2)	0.0100(2)
1	1,2,4	Stuck	.0400(8)	0.1200(24)
1	2,4,5	0.0100(2)	0.0100(1)	0.0200(2)

Table 4: Testing Speed and Ability to Find Constraints for Various Unbounded LMI detection and Handling Methods

As seen in the table, Method 2 seems to be slightly faster and runs through less iterations than both Methods 1 and 3. Method 1 has a tendency to become stuck in it's iterations due to the fact that whenever it detects unboundedness, it just changes its direction. One such example is where the algorithm becomes caught on one line that doesn't detect unboundedness, while changing direction anywhere on that line leads to unbounded cases. If there exists a constraint that is unable to be reached from branching off of that one line, then the algorithm can easily continue forever without finding all of the necessary constraints. This phenomenon can also occur in Methods 2 and 3, but

due to their adaption method of finding a new standing point under unbounded situations, the occurrence is less likely.

6 Conclusions and Further Research

I have described how to improve the Semidefinite Coordinate Direction method for finding redundant constraints. In addition, I found that cholesky factorization can replace schur decomposition with virtually no affect on finding the necessary constraints of SDPs.

I have shown that reducing redundancy in choosing vector directions for the SCD method can improve the execution time for finding necessary constraints.

Finally, I developed 3 methods for detecting and handling unbounded SDP problems, which don't occur often, but nonetheless are useful when and if the situation is met.

There are many possibilities for further research to be performed on the subject presented in this paper. Comparing accuracy between schur and cholesky decompositions could be more deeply analyzed using more extreme cases and with larger matrices in order to obtain both large and small eigenvalues. For the unbounded algorithms given in the paper, a diverse group of unbounded examples with more LMIs involved would aid in testing how well each algorithm works. In addition, testing the new and faster algorithms developed in this paper in comparison to the SSH algorithm would be helpful in the overall analysis of the SCD algorithm.

7 Acknowledgements

This research was supported by the National Science Foundation for the Research Experience for Undergraduates (REU) program at Northern Arizona University. I would like to thank Dr. Shafiu Jibrin for his help in writing this paper.

References

[1] R. J. Caron, A. Boneh, S. Boneh, "Redundancy", *Sensitivity Analysis and Parametric Programming*, Kluwer Academic, 1996

[2] J. W. Chinneck, "Discovering the Characteristics of Mathematical Programs via Sampling", Carleton University, Ottowa, Ontario, June 2000

[3] S. Jibrin, I. S. Pressman, "Monte Carlo Algorithms for the Detection of Necessary Linear Matrix Inequality Constraints" to appear in: *International Journal of Nonlinear Sciences and Numerical Simulation*, vol. 2, no. 2, 2001, pg. 139-154

[4] G. H. Golub, C. F. Loan, *Matrix Computations*, Ed. 3, John Hopkins Press, Baltimore and London, 1996, pg. 461-467

[5] R. Kress, *Numerical Analysis*, Springer-Verlang New York Inc., 1998, pg. 19

[6] L. Vendenberghe, S. Boyd, "Semidefinite Programming", *Siam Review*, March 1996

[7] J. H. Wilkinson, *Algebraic Eigenvalue Problem*, Oxford University Press, 1965