Implementing an Algorithm for Detecting Redundant Linear Constraints in Semidefinite Programming

Michael Zimmermann

Concordia University

Saint Paul, MN 55104

July 21, 2006

Abstract

We study the system consisting of a linear matrix inequality (LMI) constraint and linear constraints of the form:

$$A(x) := \ A_0 + \sum_{i=1}^{n} x_i A_i \succeq 0,$$
$$b_j + a_j^T x \geq 0 \ (j = 1, 2, \ldots, q)$$

where A_i are $m \times m$ symmetric matrices, a_j and $x \in \mathbb{R}^n$, and $b_j \in \mathbb{R}$. $A(x) \succeq 0$ means that $A(x)$ is positive semidefinite. A constraint in the above system is *redundant* if eliminating it from the system does not change the feasible region. Otherwise it is referred to as *necessary*. There are computational and cognitive benefits to removing redundant constraints in semidefinite programming. In a recent paper, Jibrin and Daniel developed ideas for identifying redundant linear constraints in the given system. We improve and implement these ideas as an algorithm. We test the algorithm on various examples.

Acknowledgments

This research was funded by the National Science Foundation REU program.

Grant awarded to:

Dr. Terence Blows

Department of Mathematics and Statistics

Northern Arizona University, Flagstaff, AZ 86001.

Research performed under the advisement of:

Dr. Shafiu Jibrin

Department of Mathematics and Statistics

Northern Arizona University, Flagstaff, AZ 86001.

1 Introduction

The branch of operations research known as semidefinite programming is relatively new as in-depth study of it has gone on for only about a decade. The field of semidefinite programming deals with optimization problems over linear matrix inequality constraints. Linear programming (LP) and quadratically constrained quadratic programming (QCQP) are both special cases of SDP. We consider a semidefinite program of the following form:

$$SDP: \quad \min \quad c^T x \tag{1.1}$$

$$\text{s.t.} \quad A(x) := \; A_0 + \sum_{i=1}^{n} x_i A_i \succeq 0 \tag{1.2}$$

$$b_j + a_j^T x \geq 0 \;\; (j = 1, 2, \ldots, q) \tag{1.3}$$

where A_i are $m \times m$ symmetric matrices, a_j and $x \in \mathbb{R}^n$, and $b_j \in \mathbb{R}$. The above SDP is referred to as the *primal*. The *dual* of the SDP is given by:

$$\begin{aligned} DSDP: \; \max \quad & -A_0 \bullet Z - \sum_{j=1}^{q} b_j y_j \\ \text{s.t.} \quad & A_i \bullet Z + \sum_{j=1}^{q} (a_j)_i y_j = c_i \quad (i = 1, 2, \ldots, n) \\ & Z \succeq 0 \\ & u_j \geq 0 \quad (j = 1, 2, \ldots, q) \end{aligned}$$

where the variables are symmetric $m \times m$ matrix Z and $y_j \in \mathbb{R}$.

The basic study of SDP began in the 1940s. Interest in this branch of optimization has grown astoundingly since the 1990s. This interest was spurred on predominantly by discoveries of new applications for SDP [14]. Some of these many applications appear in Graph-Partitioning, closest correlation matrix problems, Ricatti equations, min-max eigenvalue problems, matrix norm minimization, eigenvalue localizations [13]. It is stated in [11] that "[SDP] has applications in computational geometry, experimental design, information and communication theory,...". It is an important numerical tool for analysis and synthesis in systems in control theory and combinatorial optimization. SDP can even be applied to circuit partitioning [2]. A fantastic overview

of the theory, application, and solving of LMI problems is given in [10]. Various methods exist for solving SDP problems e.g. [12]. One very interesting approach, taken by [7], uses bundle methods to simplify an SDP down to an LP by using linear constraints to relax the SDP constraints.

Identification of redundant LMI constraints in semidefinte programming is an *NP-complete* problem, in general [6]. Thus, we focus on the identification of linear constraints. There has been a great deal of work done in the case of linear programing to identify redundant constraints [1]. To generalize these techniques to SDP is not always possible, but has been successful in a good deal of cases. In [6] probabilistic techniques are given that declare LMI constraints as redundant by identifying as many necessary LMI constraints as they can and then saying every other constraint is redundant. However, the chance still remains that a necessary constraint could be declared redundant by these methods. Another unique approach to identifying redundancy comes to us in [4] where they "study the detection of redundant linear constraints using a lexicographic solved form." This allowed the authors to identify linear constraints without the computational weight of an optimization problem.

We say the primal is strictly feasible if there is an x_0 such that $A^{(j)}(x_0) \succ 0$ (positive definite) for all $j = 1, 2, \ldots, q$. Similarly, the dual is strictly feasible if it has a feasible solution Z_0 that satisfies $(Z_0)_j \succ 0$ for all $j = 1, 2, \ldots, q$. If either the primal or the dual are strictly feasible, the primal optimal solution x^* and dual optimal solution Z^* are related to each other via the three Karush-Kuhn-Tucker (KKT) conditions. These conditions, as described in ([5], [12]), are:

1. $A(x^*) \succeq 0$, $b_j + a_j^T x^* \geq 0 \quad (j = 1, 2, \ldots, q)$
   called *primal feasibility*
2. $A_i \bullet Z^* + \sum_{j=1}^{q} (a_j)_i y_j^* = (a_k)_i \quad (i = 1, 2, \ldots, n)$
   called *dual feasibility*
3. $A(x^*)Z^* = 0$, $(b_j + a_j^T x^*) y_j^* = 0 \quad (j = 1, 2, \ldots, q)$
   called *complimentary slackness*

There are benefits to identifying redundant or necessary constraints and thereby reducing the total number of constraints in an SDP. These benefits include reduced computation time and gaining a more accurate and comprehensive understanding of your model. We are concerned with

determining all redundant linear constraints from the system:

$$A(x) \succeq 0 \tag{1.4}$$

$$b_j + a_j^T x \geq 0 \quad (j = 1, 2, \ldots, q) \tag{1.5}$$

Define the feasible regions:

$$\mathcal{R} = \{x \in \mathbb{R}^n : A(x) \succeq 0,\ b_j + a_j^T x \geq 0,\ j = 1, 2, \ldots, q\} \tag{1.6}$$

$$\mathcal{R}_k = \{x \in \mathbb{R}^n : A(x) \succeq 0,\ b_j + a_j^T x \geq 0,\ j = 1, 2, \ldots, k-1, k+1, \ldots, q\} \tag{1.7}$$

The k^{th} linear constraint is said to be redundant with respect to $\mathcal{R}$ if $\mathcal{R} = \mathcal{R}_k$ (see Figure 1). Otherwise it is necessary (the feasible region changes with its removal). A more formal interpretation is given in [5]. In Section 2 these concepts are further explored.

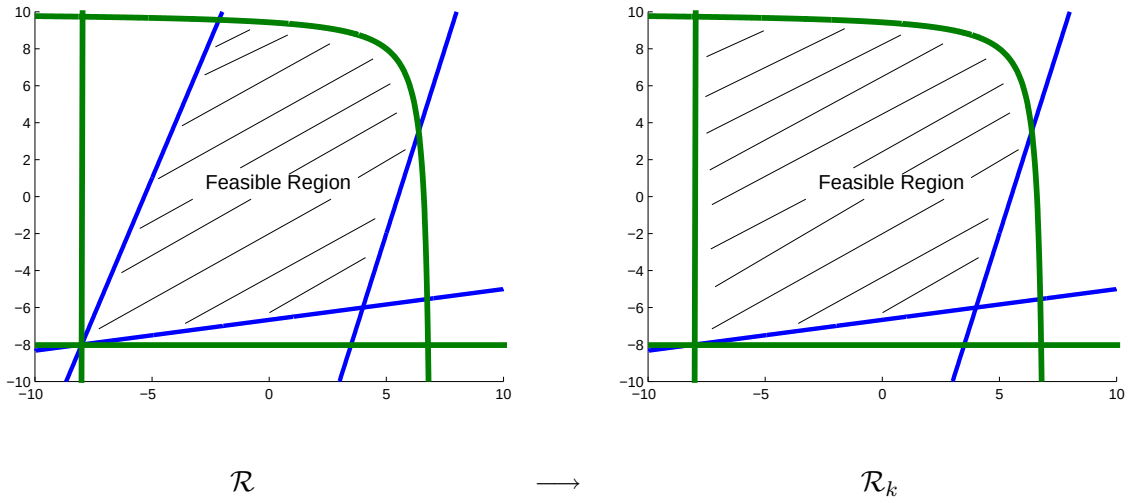

Figure 1:

We wish to identify all redundant linear constraints in system (1.1)-(1.3) with respect to the original problem set. We assume that there is a relatively large amount of linear constraints. The theory behind this work is portrayed in [5] but extended here and implemented in a software package. We solve the SDP without the k^{th} linear constraint (SDP_k) and evaluate all other linear constraints that have not been identified at the optimal point of this system. If the constraint has a zero value at the optimal point we say that the constraint is binding; that it intersects the

boundary of the feasible region at the optimal point. If the LMI is binding we look to the multiplicity of $\lambda = 0$ to judge its smoothness. If the multiplicity of the zero eigenvalue is strictly greater than one the LMI is non-smooth. The gradients of all binding linear constraints and the LMI constraint (if binding and smooth) are calculated and stored as a matrix. This matrix is then tested for linear dependence of its columns. If a column is found to be a linear combination of the others its corresponding constraint is deemed redundant. If no such combination exists the constraint is necessary. A different approach is required for identifying linear dependence in cases where the LMI is binding but not smooth. This process is repeated until all $k = (1, 2, \ldots, q)$ have either been tested or identified by other constraints. Thus we will always require $< q$ iterations to identify all constraints if at least one unidentified linear constraint is binding at an optimal point of some SDP_k. A more through explanation of the algorithm is discussed in Section 3 with further implementation in Section 4.

Section 5 gives some graphical examples of this program's abilities. We also compared the computation time required to find all redundant linear constraints by solving all q SDP_k's with the time taken by our algorithm to accomplish the same task.

2 Methodology

In this section, we describe the theory and methods of our algorithm for determining redundant linear constraints in the system (1.4)-(1.5).

To determine if the k^{th} linear constraint in the system (1.4)-(1.5) is redundant we solve SDP_k given by:

$$SDP_k: \quad \min \ \ a_k^T x \tag{2.1}$$

$$\text{s.t.} \ \ x \in \ \mathcal{R}_k \tag{2.2}$$

where $\mathcal{R}_k$ is the feasible region without the k^{th} linear constraint as defined by (1.7) and displayed in Figure 1. We assume throughout that each SDP_k and its dual are strictly feasible.

It has been shown that the linear constraint $b_k + a_k^T x \geq 0$ is redundant with respect to $\mathcal{R}$ iff

SPD_k has an optimal solution x_k^* in $\mathcal{R}_k$ that satisfies $b_k + a_k^T x_k^* \geq 0$. Utilizing this, one can identify all redundant linear constraints by solving SDP_k for all $k = 1, 2, \ldots, q$.

It has also been shown that special shortcuts can be used to identify all redundant linear constraints without having to solve all q SDP_k's. These shortcuts are described in the following.

Let $\mathcal{R}^A$ be the feasible region as described by the LMI constraint alone. Condider the orthogonal decomposition of $A(x^*)$:

$$A(x^*) = [Q_1 Q_2] diag(0, \ldots, 0, \lambda_1, \lambda_2, \ldots, \lambda_r)[Q_1 Q_2]^T. \tag{2.3}$$

where Q_1 is a matrix comprised of the eigenvectors of $A(x^*)$ which correspond to eigenvalues of $\lambda = 0$.

If x^* is a boundry point of $\mathcal{R}$ where $\mathcal{R}^A$ is smooth, then the gradient y of $A(x^*) \succeq 0$ at x^* is given by

$$y_i = Q_1^T A_i Q_1 \ (i = 1, 2, \ldots, n) \tag{2.4}$$

Let x_k^* be the optimal point of SDP_k and let I_k be the index set of the linear constraints binding at x_k^*.

Lemma 2.1. [5] *At a smooth boundary point, if the gradients of all the constraints binding at x_k^* are linearly independent, then such constraints are necessary.*

Lemma 2.2. [5] *Supoose the LMI is not binding at x_k^* or it is binding and smooth at x_k^* and $t \in I_k$. Then t is redundant iff the following system is feasible:*

$$\alpha Q_1^T A_i Q_1 + \sum_{j \in I_k \setminus \{t\}} u_j (a_j)_i = (a_t)_i \quad (i = 1, 2, \ldots, n) \tag{2.5}$$

$$\alpha, u_j \geq 0 \quad \text{for } j \in I_k \setminus \{t\} \tag{2.6}$$

where $\alpha = 0$ if the LMI is not binding at x_k^* and $\alpha \geq 0$ if the LMI is binding and smooth at x_k^*.

The problem in the above lemma is an LP feasibility problem with n equations and $|I_k|$ variables. For numerical reasons SeDuMi requires that $|I_k| > n$ [9]. We think that $|I_k| > n$ also guarantees strict feasiblity of the linear system. If $|I_k| \leq n$ we use Theorem 2.1 to check for feasibility in system (2.5)-(2.6)

Consider the following problem,

$$\begin{aligned} \min \quad & x_1 + x_2 + \cdots + x_n \\ \text{s.t.} \quad & \alpha Q_1^T A_i Q_1 + \sum_{j \in I_k \setminus \{t\}} u_j (a_j)_i + x_i = (a_t)_i \quad (i = 1, 2, \ldots, n) \\ & \alpha, u_j \geq 0 \quad \text{for } j \in I_k \setminus \{t\} \\ & x_i \geq 0 \quad (i = 1, 2, \ldots, n) \end{aligned}$$

where $\alpha = 0$ if the LMI is not binding at x_k^*. This is an SDP with n equations and $|I_k| + n$ variables.

Theorem 2.1. *System* (2.5)-(2.6) *is feasible iff the above SDP has an optimal solution* (x_i^*, α^*, u_j^*) *such that* $x_i^* = 0 \ \forall \ i = 1, 2, \ldots, n$.

Proof: Suppose (2.5)-(2.6) has a feasible point (α^*, u_j^*). Let $x_i^* = 0 \ \forall \ i$. Then (x_i^*, α^*, u_j^*) is an optimal solution of the above SDP. Conversely, suppose $\exists \ (x_i^*, \alpha^*, u_j^*)$ that is an optimal solution of the above SDP and $x_i^* = 0 \ \forall \ i$. Then (α^*, u_j^*) is a feasible point to the linear system (2.5)-(2.6). $\square$

Lemma 2.3. [5] *Assume the LMI is binding and* $\mathcal{R}^A$ *is not smooth at* x_k^*. *Constraint* t *is redundant iff the following system is feasible:*

$$(Q_1^T A_i Q_1) \bullet (Q_1^T Z Q_1) + \sum_{j \in I_k \setminus \{t\}} u_j (a_j)_i = (a_t)_i \quad (i = 1, 2, \ldots, n) \tag{2.7}$$

$$Z \succeq 0 \tag{2.8}$$

$$u_j \geq 0 \quad \text{for } j \in I_k \setminus \{t\} \tag{2.9}$$

This can be used to identify other constriants as redundant if there are other linear constraints binding at x_k^* and there exists at least one dependent gradient lying within the cone of gradients.

Theorem 2.2. *Assume the LMI is binding and* $\mathcal{R}^A$ *is not smooth at* x_k^*. *Then the linear constraint* t *is redundant iff the following system is feasible:*

$$(Q_1^T A_i Q_1) \bullet W + \sum_{j \in I_k \setminus \{t\}} u_j (a_j)_i = (a_t)_i \quad (i = 1, 2, \ldots, n) \tag{2.10}$$

$$W \succeq 0 \tag{2.11}$$

$$u_j \geq 0 \quad j \in I_k \setminus \{t\} \tag{2.12}$$

Note the change of variables from system (2.7)-(2.9) from Z to W. Computation has been reduced in that W has smaller size. Instead of dimension $m \times m$, W is an $r \times r$ symmetric matrix where r is the multiplicity of $\lambda = 0$ in the decomposition (2.3).

Proof: By Lemma 2.3, it suffices to show that system (2.7)-(2.9) is feasible iff the above system is feasible. Suppose Z is a feasible point of system (2.7)-(2.9). Then,

$$Z \succeq 0 \implies Q^T Z Q = [Q_1 Q_2]^T Z [Q_1 Q_2] = \begin{bmatrix} Q_1^T Z Q_1 & Q_1^T Z Q_2 \\ Q_2^T Z Q_1 & Q_2^T Z Q_2 \end{bmatrix} \succeq 0$$

Since $Q^T Z Q \succeq 0$, all minors of $Q_1^T Z Q_1$ must be nonnegative. So, $Q_1^T Z Q_1 \succeq 0$. We take $W = Q_1^T Z Q_1$. Conversely, suppose W is a feasible point of (2.10)-(2.12). Let $Z = Q_1 W Q_1^T$. Then,

$$Z \succeq 0 \implies Q_1^T Z Q_1 = Q_1^T Q_1 W Q_1^T Q_1 = W \succeq 0$$

So, Z is a feasible point of (2.7)-(2.9) $\square$.

Now we would like to check the feasibility of (2.10)-(2.12) This is an SDP feasibility problem with n equations and $|I_k| - 1 + \frac{r(r+1)}{2}$ variables. For numerical reasons, SeDuMi requires that $(|I_k| - 1 + r^2) > n$ [9]. We think that $(|I_k| - 1 + r^2) > n$ also guarantees strict feasiblity of the linear system.

Note that there are in fact $\frac{r(r+1)}{2}$ variables contributed from the LMI, but SeDuMi does not assume (at least for input purposes) that the A_i are symmetric. Therefore it requires the input of r^2 variables to build the entire matrix. If $(|I_k| - 1 + r^2) \leq n$ we use the following result to check feasibility of (2.10)-(2.12).

Consider the following SDP problem:

$$\begin{aligned} \min \quad & x_1 + x_2 + \cdots + x_n \\ \text{s.t.} \quad & (Q_1^T A_i Q_1) \bullet W + \sum_{j \in I_k \setminus \{t\}} u_j (a_j)_i + x_i = (a_t)_i \quad (i = 1, 2, \ldots, n) \\ & W \succeq 0 \\ & u_j \geq 0 \quad j \in I_k \setminus \{t\} \\ & x_i \geq 0 \quad (i = 1, 2, \ldots, n) \end{aligned}$$

This is an SDP with $|I_k| - 1 + \frac{r(r+1)}{2} + n$ variables.

Theorem 2.3. *System* (2.10)-(2.12) *is feasible iff the above SDP has an optimal solution* (x_i^*, u_j^*, W^*) *such that* $x_i^* = 0 \ \forall \ i = 1, 2, \ldots, n$.

Proof: Similar to the proof of Theorem 2.1. □

3 Explanation

This section is dedicated to explaining in further detail the implementation of our program. The main steps follow. The MATLAB add-on SeDuMi (Self-Dual Minimization) [8] toolbox accepts input of the form:

$$\begin{aligned} SDP: \quad & \max \ b^T y \\ & \text{s.t.} \ \ A_0 - \sum_{i=1}^{n} y_i A_i \succeq 0 \end{aligned}$$

-or-

$$\begin{aligned} DSDP: \ & \min \ Tr(A_o x) \\ & \text{s.t.} \ \ Tr(A_i X) = b_i \ \ \text{for} \ (i = 1, 2, \ldots, n) \\ & \quad X \succeq 0 \end{aligned}$$

To accomplish this we feed our constraints into SeDuMi in the following way.

$$\begin{aligned} SDP: \ & \max \ -b^T y \\ & \text{s.t.} \quad c - A^T y \succeq 0 \end{aligned}$$

where

$$c = vec(A_0)$$

$$A^T = [vec(A_1), \ldots, vec(A_n)]$$

and

$$vec\left(\begin{bmatrix} a_{1,1} & c_{1,2} \\ b_{2,1} & d_{2,2} \end{bmatrix}\right) = \begin{bmatrix} a_{1,1} \\ b_{2,1} \\ c_{3,1} \\ d_{4,1} \end{bmatrix}$$

Our constraints, now in the proper form, are evaluated by SeDuMi. Note that SeDuMi automatically maximizes the primal objective function. Thus, we take the negative of our objective function, $-b^T y$, to minimize. We evaluate $b_k + a_k^* x = p$. If $p < 0$ then the feasible region has changed with its removal and we identify the k^{th} constraint as necessary.

If the k^{th} constraint is redundant we use the optimal solution x_k^* to system (2.1)-(2.2) given by SeDuMi to identify all linear constraints that are binding at x_k^*. Every linear constraint, except the k^{th}, that has not yet been identified is evaluated at this optimal solution. If it has a near zero value (absolute value $< 1e^{-4}$ is used as a tolerance for numerical error) we say that the constraint is binding. The indices of the binding linear constraints are captured as a row vector named I_k.

The LMI constraint, $A(x_k^*)$, is now evaluated. We are concerned with the eigenvalues of $A(x_k^*)$. If there are any $\lambda = 0$ the LMI is binding at x_k^*. If $\lambda = 0$ has multiplicity equal to one then the LMI is smooth at x_k^*. If the multiplicity is strictly greater than one the LMI is not smooth. This is true because within the feasible region $A \succ 0$, outside $A \prec 0$, but on the boundary $A \succeq 0$. Thus, if the LMI is binding it must have at least one zero eigenvalue. Each $\lambda = 0$ will satisfy one of the minors of the LMI. So, if there are two or more $\lambda = 0$, there is an intersection of functions corresponding to those minors and the LMI is not smooth.

Knowing all constraints binding at x_k^* the gradient of each constraint is calculated. For the linear constraints we simply capture each a_j from (1). The gradient of the LMI is only calculated if it is smooth and binding at x_k^*. Using the eigenvalues, we gather the eigenvectors of $A(x_k^*)$ and place all such vectors corresponding to $\lambda = 0$ in a matrix Q_1 and employ (2.4). All gradient vectors are rounded to the nearest hundredth and organized into a matrix G.

From hence forth we will mean *positive* linear dependence (independence) whenever linear dependence (independence) is stated as we are concerned with negative gradients lying within the normal cone, not the whole of $\mathbb{R}^n$. Define c as the number of columns in G, that is the number of binding constraints with calculated gradient at x_k^*. If $c > n$, the vectors must be linearly depen-

dent. When $c \leq n$ we use the diagonal of rref(G) for a quick verdict. Since all values along the main diagonal of the resultant matrix are zero (corresponding to a linearly dependent vector) or one (corresponding to a linearly independent vector), we can easily judge dependence. If there is at lease one zero along the diagonal there is a linearly dependent vector(s); otherwise the vector(s) are linearly independent.

We seek to know which of the gradients binding at x_k^*, corresponding to linearly dependent linear constraints, can be written as a linear combination of the others and are therefore redundant. To accomplish this when the LMI is binding and smooth, or not binding at all, we employ Theorem 2.1. First, we gather all linear and LMI gradients into a matrix G. We exclude the i^{th} column of G and make a new matrix A comprised of all vectors but the i^{th}. This i^{th} vector is called b. We make a vector c with length equal to the number of columns in A. Note that c is the objective function, here zero valued. In the case where there are more elements in c than in b, these three variables are fed into SeDuMi to satisfy the following primal form:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x_i \geq 0 \ for \ i = 1, 2, \ldots, n \end{aligned}$$

If there is a feasible region associated with this input then b is a linear combination of the other binding gradients. The index of the linear constraint corresponding to gradient vector i is labeled as redundant. If there is no feasible region (no possible linear combination) we say the constraint is necessary and label its index as such.

In the case where there are equal elements in c and b, or more elements in b, we must augment A and c with free variables. This is simply to satisfy the input requirements of SeDuMi. A is appended with an $n \times n$ identity matrix and c with n number of ones. Our objective function is now nonzero with the nonzero elements corresponding to free variables. All free variables are also restricted as $x_j \geq 0$ with $(j = 1, 2, \ldots, n + r)$. The restriction is accomplished by telling SeDuMi to expect $|I_k| - 1 + r$ nonnegativity constraints since we have added r free variables to the set of

binding constraints without the i^{th}. This new data is given over to SeDuMi which will minimize the value of all free variables. Again, if there is no feasible region there is no possible linear combination of the columns of A to form b and b is necessary. If a feasible region does exist, we sum the elements of the optimal solution SeDuMi returns. This solution is really the coefficients of the linear combination. If this sum is within a certain tolerance of zero we say it is in fact zero. Therefore the optimal solution occurs when all free variables have zero value and the constraint corresponding to the i^{th} column is redundant. If there is a nonzero sum within the feasible region, then the linear combination takes some number of nonzero free variables to exist and so this constraint is necessary. This is repeated for all i up to but not including the last column of G so that the LMI gradient is not tested for dependence.

In the case where the LMI is binding and not smooth we must employ the idea presented in Lemma 2.3 and improved by Theorem 2.2. Recall, if (2.7)-(2.9) is feasible then Theorem 2.2 also describes a feasible region and visa versa. Define nv as the total number of calculated linear gradient vectors at x_k^*. We create the matrix B by first placing all binding linear gradients into B as columns. Then the matrix T is created such that each row is the transpose of $vec(Q_1^T A_i Q_1)\ \forall\ i = 1, 2, \ldots, n$. T is then appended onto B. The vector b is again the value of the i^{th} linear constraint. However, the elements of c are comprised of nv zeros appended with zeros equal to the size of $vec(A_0)$.

All binding gradients are used in testing for dependence, but only the constraints which have not yet been identified are tested. In the case where length(c) = length(b) we tell SeDuMi that there are $nv - 1$ nonnegativity constraints and r positive semidefinite constraints. We must now evaluate the feasible region. If there is feasibility in the dual, then a linear combination of the binding vectors exists and the i^{th} linear constraint is redundant. If the system is dual infeasible, then there is no such linear combination and the i^{th} linear constraint is necessary. In the case where length$(c) \leq$ length(b) SeDuMi will reject the input. To alleviate this problem so that we can use SeDuMi to solve this feasibility problem we employ nearly the same tactic used in the previous situation relating to Theorem 2.1 where length$(c) \leq$ length(b). The $r \times r$ identity matrix is added to the beginning of B so that column one is the first column of $\mathrm{I}_{r\times r}$ and the $r + 1$ column is the

first column of the previous B. c is appended in the same fashion but with a vector of ones with length(r). There are now $nv - 1 + r$ nonnegativity constraints with r of them being free variables that we will minimize with the objective function c. If the primal is now infeasible we say that the i^{th} gradient is not a linear combination of the other binding gradients and so it is necessary. If there is a feasible primal, we must again sum the entries in the optimal solution. Recall that the optimal solution is really a vector containing the coefficients of the linear combination. If this sum is within some tolerance of zero we say that all the coefficients were in fact zero. This implies that there is feasibility in the primal without free variables and the constraint is redundant. If the linear combination requires free variables (i.e. a nonzero sum) then the constraint is necessary.

By the methods described above, we have identified the k^{th} linear constraint and all linear constraints binding at x_k^* as redundant or necessary. All indices of the identified constraints are stored in a vector named Idf. This vector is used to ensure that constraints already identified are left out of subsequent calculation. For instance, we test all elements in the set I_k that are not in the set Idf. All constraints identified in subsequent rounds of calculation are appended to a growing list of such identified parameters.

4 Linear Redundancy Detection Algorithm

In this section we describe our algorithm for eliminating redundant linear constraints. We use techniques described in the previous sections. We also discuss implementation of the algorithm.

Pseudocode.

$\Psi = \emptyset$

For $k = 1 : q$

  If $k \notin \Psi$

    solve SDP_k to find optimal point x_k^*

    $\Psi = \Psi \cup \{k\}$

    If k is necessary

Stop

Else

Compute index set I_k of linear constraints binding at x_k^*

Determine binding and smoothness of LMI at x_k^*

Calculate gradients of all constraints binding at x_k^*

Determine linear independence of binding gradients

Let Υ be all linear gradients whos indices $\notin I_k$

If (LMI binding and smooth) or (LMI not binding)

For $i \in \Upsilon$

Check if i^{th} gradient is a nonnegative linear combination; see (2.5)-(2.6)

End

End

If (LMI binding and not smooth)

For $i \in \Upsilon$

Check if i^{th} gradient is a nonnegative linear combination; see (2.10)-(2.12)

End

End

$\Psi = \Psi \cup \{\text{all newly identified constraints}\}$

End

End

End

A more through description of the program

Note that all functions are written as MATLAB v7.1 m-files.

We begin within a main file designated *redundancy_algo_shortcuts* which accepts the system data as two matrices; the first being an $m \times m(n+1)$ matrix A which stores the LMI information, and the second a $q \times (n+1)$ matrix L which stores all q linear constraints (each as a row). The number of linear constraints (q), size of LMI matrices (m), and number of variables (n) must also be input. First, three empty vectors are created. These are used to store all unique values of redundant linear constraints (rdf), all necessary linear constraints($necf$), and all constraints that

have been tested or identified by other binding constraints (Idf). A vector i is created to hold all natural numbers $k = 1, 2, \ldots, q$. The number of iterations taken is set to zero. While i is not empty, we set $k = i(1)$ and pass A, L, q, m, n, k, Idf to the function *SDPk_shortcuts*.

Within *SDPk_shortcuts* we make a new matrix $F = A$. We say that $b = [-L\{k\}(2 : n + 1)]^T$ and c is a column vector consisting of the first entry of $L\{p\}$ *where* $p = 1, 2, \ldots, k-1, k+1, \ldots, q$. We now set $A = L\{p\}(2 : n + 1)$. A matrix T is created such that $T = [vec(-F_1), \ldots, vec(-F_n)]$ and $A = [A; T]$. We store the value of k as $k2$, set $k.l = q - 1$ and $k.s = m$ and call SeDuMi. The optimal solution is stored as x. If the primal is feasible we evaluate the k^{th} constraint at x. If the primal is infeasible we say the constraint evaluated to -1. If the constraint evaluates to a negative or zero (within 0.00001) we say it is necessary and store its index in *nec*. Otherwise the index is stored in *rd*. Idf is updated and variables passed to the function *prop4_1*.

All constraints except the k^{th} are evaluated at the optimal solution. If a linear constraint evaluates to 0 ± 0.00001 we say that it is binding and log its index in I_k. We take the eigenvalues of $A(x^*)$. If $\exists\ \lambda = 0$, the LMI is binding and $lmibd = 1$. We also save the multiplicity of $\lambda = 0$ as R. If I_k is nonempty, the a_j of all linear constraints whos indices are $\in I_k$ are stored in a matrix lg and rounded to the nearest hundredth. We create a matrix V consisting of the eigenvectors of $A(x^*)$. If $\exists$ a $\lambda = 0$ we capture its corresponding vector $\in V$. All such vectors are stored in a matrix Q_1. If $R == 1$ we say matrix $Y = Q_1^T F_i Q_1$ for $i = 1, 2, \ldots, n$. $G = [lg, Y]$ is now a collection of all calculated gradients. If $lmibd == 0$ or ($lmibd == 1$ and $R == 1$) we say that c is the number of columns of G. If $c == 1$ the variable $allnec = 1$. If $c \leq n$, $D2 = diag(rref(G))$. If $\min(D2) > 0$, $allnec = 1$ We now exit *prop4_1* and return to *SDPk_shortcuts*.

We say that $[Iku, Ind]$ =setdiff(Ik, Idf) and $lgu = lg(:, Ind)$. If $allnec == 1$ the indices of newly identified constraints are added to *nec*. If ($lmibd == 0$ and $allnec == 0$) or ($lmibd == 1,\ allnec == 0,$ and $R == 1$) we pass into *Thm4_1*.

Designate nv as the number of vectors in G and nl the number of linear gradients. We test the i^{th} vector if $I_k(i) \in Iku$. We say that $A = G$ without the i^{th} column and b is the i^{th} column. We make a vector $c = zeros(nv - 1, 1)$. If length(c) >length(b), $k.l = nv - 1$ and we call SeDuMi. If the primal is feasible we log $I_k(i)$ in *rd*. Otherwise we log $I_k(i)$ in *nec*. If length(c) $\leq$length(b) we say that $A = [A, eye(r)],\ c = [c; ones(r, 1)],\ k.l = nv - 1 + r$ and we call SeDuMi. The entries in x

are summed. If the sum$(x) = 0 \pm 0.000001$, $I_k(i)$ is logged in *rd*. If primal infeasible or sum$(x) \neq 0$, $I_k(i)$ is logged in *nec*. We again return to *SDPk_shortcuts* and update the vectors *nec* and *rd* from the output of *Thm4_1*.

Now, if $lmibd == 1$ and $R > 1$ we pass into *Thm5_1*. Define $m1$ as the number of columns in Q_1. Each vector $\in Iku$ is tested. Define B as a matrix consisting of all vectors but the i^{th} and b as the i^{th}. Let $T = [vec(Q_1^T F_i Q_1)]\ \forall\ t = 1, 2, \ldots, n$. Augment B such that $B = [B, T]$. Let $c = [zeros(nv - 1 + m^2)]$. If length$(c)$ >length(b), $k.l = nv - 1$ and $k.s = m1$ and we enter SeDuMi. If the dual is feasible we add the index $I_k(i)$ to *rd*, otherwise to *nec*. If length(c) $\leq$length(b) we say that $B = [eye(r), B],\ c = [ones(r, 1); c],\ k.l = nv - 1 + r,\ k.s = m1$ and enter SeDuMi. We sum the entries in x. If sum$(x) = 0 \pm 0.000001$, $I_k(i)$ is logged in *rd*. If primal infeasible or sum$(x) \neq 0$, $I_k(i)$ is logged in *nec*. We again return to *SDPk_shortcuts* and update the vectors *nec* and *rd* from the output of *Thm5_1*.

SDPk_shortcuts is now complete and we return to *redundancy_algo_shortcuts*. *rdf* and *necf* are updated with the values stored in *rd* and *nec*. *Idf* is then updated with the values in *rdf* and *necf*. We now delete $i(1)$ and all indices stored in I_k from the vector i. The number of iterations taken is increased by one and the while loop ended.

The total number of iterations taken to identify all constraints, the redundant linear constraints, the necessary linear constraints, and the total computation time are the displayed output of *redundancy_algo_shortcuts*.

5 Numerical Experimentation

This section gives numerical experiments to determine the effectiveness of our algorithm. All the experiments were performed on computers running with Intel Pentium 4 processors, 3.39GHz, and 2.00GB of RAM. These computers were part of a college campus LAN and so used some processing power for system processes. MATLAB v7.1.0.246 employed for all work.

With the exception of the first test problem (Example 1), all the problems were randomly generated as follows: The LMI $A_0 + \sum_{i=1}^{n} x_i A_i \succeq 0$ is generated such that A_0 is an $m \times m$ diagonal

matrix with each entry in $U(0,1)$. Each A_i is a symmetric $m \times m$ sparse matrix with approximately $0.8m^2$ nonzeros and each entry is the sum of one or more normally distributed random samples. Each linear constraint $b_j + a_j^T x \geq 0$ is randomly generated with b_j and each entry of a_j in $U(0,1)$.

Example 5.1. The following are the constraints of an example SDP

$$A(x) := \begin{bmatrix} 5 & -1 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} + x_1 \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \succeq 0$$

$$27 + 16x_1 - 8x_2 \geq 0 \quad (1)$$
$$14 + 3x_1 - 8x_2 \geq 0 \quad (2)$$
$$5 - x_1 - x_2 \geq 0 \quad (3)$$
$$4.6 - 0.9x_1 - x_2 \geq 0 \quad (4)$$
$$5.4 - 1.1x_1 - x_2 \geq 0 \quad (5)$$
$$3 + x_1 + x_2 \geq 0 \quad (6)$$
$$4 + 2x_1 + x_2 \geq 0 \quad (7)$$
$$2.5 + 0.5x_1 + x_2 \geq 0 \quad (8)$$
$$3 + x_1 - x_2 \geq 0 \quad (9)$$
$$37 + x_1 - 14x_2 \geq 0 \quad (10)$$
$$6 - x_1 - x_2 \geq 0 \quad (11)$$
$$10 - x_1 + 2x_2 \geq 0 \quad (12)$$
$$11 + 5x_1 + 2x_2 \geq 0 \quad (13)$$

The feasible region for this example is given in Figure 2

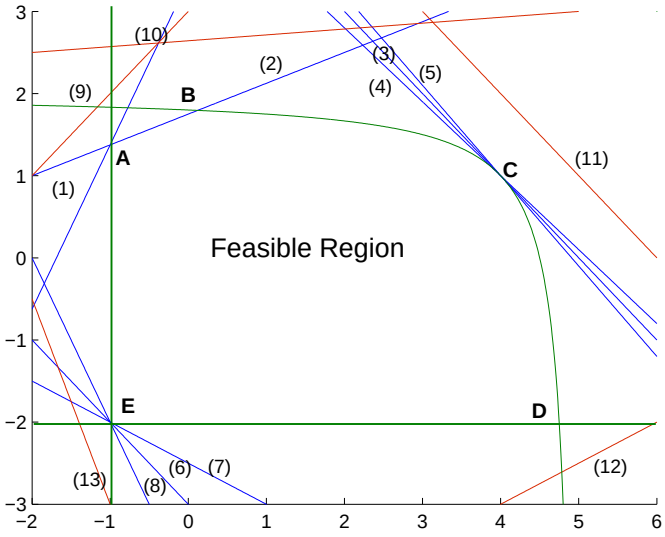

Figure 2: Our algorithm identifies all $q = 13$ linear constraints in the above example in 8 iterations, properly identifying only [2 4 5] as necessary and all other linear constraints as redundant.

Example 5.2. Here is another example

$$A(x) := \begin{bmatrix} 0.6 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0.7 \end{bmatrix} + x_1 \begin{bmatrix} 0 & -0.3 & -1 \\ -0.3 & 0 & 0.3 \\ -1 & 0.3 & 0 \end{bmatrix} + x_2 \begin{bmatrix} 0.2 & -0.4 & -1.2 \\ -0.4 & 0 & 0.5 \\ -1.2 & 0.5 & 0 \end{bmatrix} \succeq 0$$

$$\begin{aligned} -7 - 2x_1 + x_2 &\geq 0 \quad (1) & 3 + 7x_1 - 4x_2 &\geq 0 \quad (7) \\ 6 - x_1 + 7x_2 &\geq 0 \quad (2) & 8 - 7x_1 + 2x_2 &\geq 0 \quad (8) \\ 5 - 9x_1 + x_2 &\geq 0 \quad (3) & 3 - 4x_1 - 5x_2 &\geq 0 \quad (9) \\ 1 - 2x_1 - x_2 &\geq 0 \quad (4) & 6 + x_1 + 13x_2 &\geq 0 \quad (10) \\ 2 - 4x_1 - 21x_2 &\geq 0 \quad (5) & 10 + 25x_1 + x_2 &\geq 0 \quad (11) \\ 4 + x_1 + 14x_2 &\geq 0 \quad (6) & 5 - 10x_1 + 9x_2 &\geq 0 \quad (12) \end{aligned}$$

With a graphical representation in Figure 3

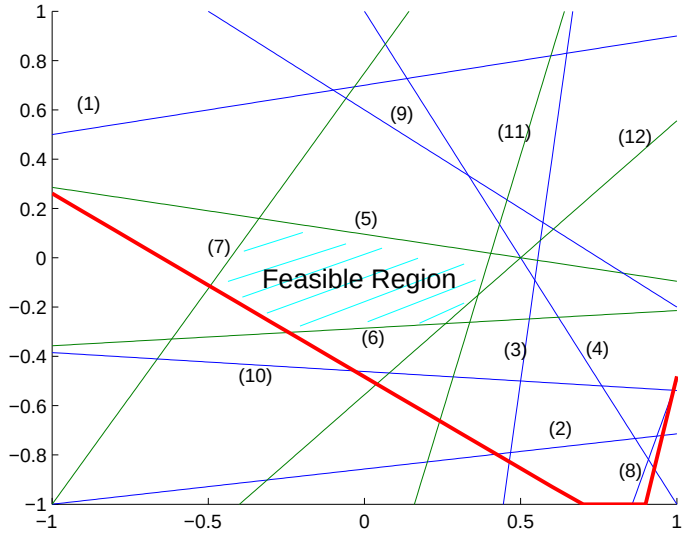

Figure 3: The $q = 12$ linear constraints are identified in 7 iterations. [5 6 7 11 12] are identified as necessary and all other linear constraints as redundant.

Computation times of our algorithm vs solving $SDP_k \ \forall \ k = (1, 2, \ldots, q)$

Example	q	m	n	R	N	$\#\ k\ sol$	$time1(s)$	$time2(s)$	$comp$
1	13	4	2	10	3	8	0.3862	0.4715	0.8191
2	12	3	2	7	5	7	0.2151	0.3496	0.6151
3	9	4	3	5	4	6	0.2636	0.3723	0.7081
4	9	5	3	3	6	7	0.2885	0.3387	0.8518
5	12	6	3	7	5	8	0.3577	0.5023	0.7122
6	12	10	3	10	2	11	0.6363	0.6867	0.9266
7	20	15	4	15	5	16	1.5471	1.7073	0.9062
8	30	20	4	27	3	27	2.4037	2.4825	0.9683
9	50	20	7	45	5	48	5.1342	5.0415	1.0184
10	70	25	7	62	8	63	9.4452	9.3181	1.0136
11	100	50	10	95	5	96	42.2854	40.9541	1.0325
12	500	150	15	484	16	490	4095.89	4097.78	0.9995

In the above table all situations were randomly generated. Table example 1 and 2 correspond to Example 5.1 and 5.2 respectively. q is the number of linear constraints, m the dimension of LMI constraints, n the number of variables, R is the number of redundant linear constraints, N the number of necessary linear constraints, $\#\ k\ sol$ is the number of SDP_k's our algorithm required to identify all linear constraints, $time1$ is cpu time to identify all linear constraints using our algorithm, $time2$ is cpu time to identify all linear constraints by solving all SDP_k, and $comp$ is simply $time1 \div time2$

The advantages and limitations of our algorithm are now fairly plain to see. In cases where a relatively high percent of the system's linear constraints are necessary computation time has been reduced by up to 39%. The time reduction would likely be even greater if $N \geq R$. This is because the chances for a redundant constraint to be binding are rather small. It is much more likely for a constraint to cut the feasible region than to be tangent to it. Thus, in random examples with few necessary linear constraints our algorithm cannot utilize the shortcuts described in Theorems 2.1 and 2.2. It then must solve SDP_k for nearly all k. The number of iterations required will, in almost all cases, be $< q$ but because of the checking statements, list updates, and the like, computation time may be slightly longer.

6 Conclusion

We have studied the problem of determining redundant linear constraints in a system involving linear matrix inequality constraints. This process is begun by solving an SDP_k and evaluating the optimal solution on the region $\mathcal{R}_k$. If the value of the k^{th} constraint is negative at the optimal solution we know that this constraint is necessary. If the constraint is redundant we first identify all other constraints binding at x_k^* and evaluate their attributes. Second, two shortcuts are implemented to determine the redundancy/necessity of these binding constraints. These shortcuts are performed by solving a simple SDP feasibility problem. Once this is complete the process is repeated for all k that have not been identified.

Our methods simplify the shortcut methods of Jibrin and Daniel by proving that matrix variable size can be reduced in certain feasibility problems. There is great utility to the implementation of

redundancy identifying techniques. Not only is the identification of redundant linear constraints interesting in its own right, but can lead to new insight into the situation at hand. We have shown that our algorithm does exhibit reduced computation time in situations where there are sufficient necessary linear constraints (accounting for problem size) as compared with solving $SDP_k \; \forall \; k$.

References

[1] Caron, R. J., A. Boneh, S. Boneh. "Redundancy." Advances in Sensitivity Analysis and Parametric Programming, Kluwer Academic Publishers. International Series on Operations Research and Management Science, vol. 6, 1997.

[2] Choi, C. Cris and Y. Ye. "Application of Semidefinite Programming to Circuit Partitioning." http://citeseer.ist.psu.edu/choi99application.html. 1999. June 2006.

[3] Formaneck, S.. "Improving the Semidefinite Coordinate Direction (SCD) and Semidefinite Stand-and-Hit (SSH) Methods for the Detection of Necessary Linear Matrix Inequalities." 2002.

[4] Hentenryck, P. Van and J. L. Imbert. "A Note on Redundant Linear Constraints." Brown University. Tech Report CS-92-11. 1992.

[5] Jibrin, S. and Stover, D. "Identifying Redundant Linear Constraints in Systems of Linear Matrix Inequality Constraints."

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

[7] Krishnan, K. and Mitchell, J. "A Linear Programming Approach to Semidefinite Programming Problems." Rensselaer Polytechnic Institute. 2001.

[8] McMaster University, "SeDuMi - optimization package over symmetric cones" Advanced Optimization Laboratory. http://sedumi.mcmaster.ca/

[9] Polik, Imre. “RE: Symmetry Question.” E-mail to Michael Zimmermann. 7 June 2006.

[10] VanAntwerp, J. G. and Braatz, R. D. “A Tutorial on Linear and Bilinear Matrix Inequalities.” Journal of Process Control. Elsevier Science Ltd. 2000.

[11] Vandengerghe, L. and Boyd, S. “Applications of Semidefinite Programming.” 1998.

[12] Vandenberghe, L. and Boyd, S. “Semidefinite Programming.”

[13] Wolkowicz, H., R. Saigal, and L. Vandenberghe, editors. “Handbook of Semidefinite Programming: Theory, Algorithms, and Applications.” Kluwer Academic Publishers, Boston, MA, 2000.

[14] Wolkowicz, H. “Semidefinite Programming” University of Waterloo. 2002.