April 2010

Semidefinite Programming and its Application to the Sensor Network Localization Problem

Elizabeth Anne Hegarty
Worcester Polytechnic Institute

Project Number: MA-WJM-9800

Semidefinite Programming and its Application to the Sensor Network Localization Problem

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Elizabeth A. Hegarty

Date: April 29, 2010

Approved:

Professor William J. Martin, Major Advisor

Abstract

Semidefinite programming is a recently developed branch of convex optimization. It is an exciting, new topic in mathematics, as well as other areas, because of its algorithmic efficiency and broad applicability. In linear programming, we optimize a linear function subject to linear constraints. Semidefinite programming, however, addresses the optimization of a linear function subject to nonlinear constraints. The most important of these constraints requires that a combination of symmetric matrices be positive semidefinite. In spite of the presence of nonlinear constraints, several efficient methods for the numerical solution of semidefinite programming problems (SDPs) have emerged in recent years, the most common of which are interior point methods. With these methods, not only can SDPs be solved in polynomial time in theory, but the existing software works very well in practice.

About the same time these efficient algorithms were being developed, very exciting applications of semidefinite programming were discovered. With its use, experts were able to find approximations to NP-Complete problems that were .878 of the actual solution. This is amazing considering for some NP-Complete problems we cannot even find approximations that are 1% of the actual solution. Another amazing result was in coding theory. By applying semidefinite programming, the first advancement in 30 years was made on the upper bound of the maximum number of code words needed. Other great outcomes have arisen in areas such as eigenvalue optimization, civil engineering, sensor networks, and the financial sector, proving semidefinite programming to be a marvelous tool for all areas of study.

This project focuses on the understanding of semidefinite programming and its application to sensor networks. I began first by learning the theory of semidefinite programming and of the interior point methods which solve SDPs. Next, I surveyed many recent applications of semidefinite programming and chose the sensor network localization problem (SNLP) as a case study. I then implemented the use of an online tool -- the algorithm csdp, from the NEOS Solvers website -- to solve given SNLPs. I also developed MAPLE code to aid in the process of transforming a naturally stated problem into one suitable for NEOS input and for interpreting the csdp solution in a way understandable to a user unfamiliar with advanced mathematics.

Contents

1 Introduction

In the 1990s, a new development in mathematical sciences was made which allowed for broad applicability and great advances in many areas of study. This development was of semidefinite programming, a branch of convex optimization which can optimize linear functions given constraints that are nonlinear and involving positive semidefinite matrices. The results of this development were greater than could be imagined, with more exciting advancements continuing to be made today. Experts were able to find approximations to NP-Complete problems that were $.878$ of the actual solution. This is amazing considering for some NP-Complete problems we cannot even find approximations that are 1% of the actual solution. Another amazing result made possible by semidefinite programming was in the field of coding theory. With semidefinite programming, the first advancement in 30 years on the upper bound of the maximum number of code words needed was made possible. Other great outcomes have stemmed from semidefinite programming, such as its use in civil engineering and the financial sector, proving it to be a marvelous tool for all areas of study.

The general semidefinite programming problem (SDP) optimizes the trace of the product of two matrices, one being the variable matrix, X. One set of constraints require that the difference between a vector and a linear operator be zero, while the other set requires this difference to be nonnegative. The last constraint is that the variable matrix, X, be positive semidefinite. This is where we lose the linearity of our system. However, these constraints are still convex, allowing semidefinite programming to fall under convex optimization. Also, while SDPs are not linear programming problems (LPs), every LP can be relaxed into a SDP.

For every SDP, one can obtain the dual with the use of Lagrange multipliers. This ability led to adaptations of the Weak Duality Theorem and the Complementary Slackness Theorem to semidefinite programming. In the cases where we have strictly feasible interior points for the primal and dual problems, the Strong Duality Theorem can also be applied. These theorems aid in guaranteeing the solution obtained is the optimal solution. However, since semidefinite programming contains nonlinear constraints, one cannot use linear programming techniques to obtain this optimal solution. For this reason, methods for solution have been researched with many techniques being successful. In 1996, Christoph Helmberg, Franz Rendl, Robert Vanderbei and Henry Wolkowicz introduced an interior point method for SDPs, utilizing ideas of advanced linear programming. These ideas were of the central path and the path following method, which involve the use of barrier functions, Lagrangians, first- order optimality conditions, and Newton's method. Helmberg et al. were able to adapt all of these ideas to semidefinite programming, obtaining a very efficient algorithm for numerical solution. Many computer programs have been written for this algorithm and work very well in practice.

As noted previously, semidefinite programming's great appeal comes from its broad applicability. One very prominent application, which is worth elaborating on, is its use in sensor networks. Sensor networks are used to collect information on the environment they are placed in, however this information is only relevant if the positions of all the sensors are known. The problem of locating all of the sensors is identified as the Sensor Network Localization Problem (SNLP). Through the use of distance measurements and some known sensor positions, semidefinite programming can locate all the positions of the sensors in the system. Through use of computer programs, SNLPs can be solved quickly and efficiently.

This application, along with many others, can allow for semidefinite programming to become significant even to those not involved in advanced areas of study. While semidefinite programming seems like just another topic in mathematics whose purpose in everyday life and that of the common person seems meager, many of these applications can impact our lives. Its application in coding theory could lead to the shortening of codes which are used in almost everything from CDs to cell phones to the internet. The shortened code length would result in our whole world working much faster. Semidefinite programming and the SNLP can be utilized for military purposes, allowing for aerial deployment and the collection of information, such as air content and positioning of enemy lines, during ground warfare. The ability to solve the SNLP with semidefinite programming also intrigues many people for its use with firefighters. Being able to know the positions of firefighters inside a burning building and the environmental conditions they are in could lead to much greater safety for all involved because more informed decisions could be made. This application has become a very prevalent research topic at Worcester Polytechnic Institute, hoping to develop easy to use techniques to prevent firefighter injuries and casualties.

This report focuses on the understanding of semidefinite programming and its use in the field of sensor networks. I finish this section by introducing some notation. In the second chapter, I discuss the ideas of semidefinite programming. I begin by presenting the theory behind SDPs and then explain the interior point method developed by Helmberg et al. for numerical solution. In the third chapter, I discuss the uses of SDPs by first presenting a survey of famous applications. I then explore semidefinite programming's use in the SNLP. Finally, in the last chapter, I present a guide for utilizing the NEOS Solvers (an online tool for optimization problems) for solving SNLPs and MAPLE codes to aid in the process.

1.1 Notation

The main point of this section is to introduce some notation and define some of the major terms this paper is going to use. Those presented here will be the most common, and while some may be a review, a good understanding of the ideas is needed in order to continue. Note that if an idea is not discussed here and is used later on, it will be defined at the time of its use.

To begin I will introduce the notation for the inner product. This is $\langle x, y\rangle$ and, as part of its definition, $\langle x, y\rangle = x^T y$. From here, I will define the trace of a matrix. This is the addition of the diagonal entries of a square matrix. In this paper, we will often want to obtain the trace of the product of two matrices, one being the transpose of a matrix. Since, in semidefinite programming this trace will be the inner product we use, we will denote it as: $tr(F^T G) = \langle F, G\rangle$.

Next, while positive semidefinite matrices will be defined later, this paper will denote them using this symbol: $\succcurlyeq$. For example, if a matrix A is positive semidefinite we will say that $A \succcurlyeq 0$. We will do similarly for a positive definite matrix but instead use this symbol: $\succ$.

In the general SDP system, we utilize the terms $A(X)$ and $B(X)$. These will be linear operators which take our variable matrix X and a set of k matrices A_i (a set of m matrices B_i for the linear operator $B(X)$) and lists the traces of X with each matrix A_i in a vector. For each of these linear operators, there is an adjoint linear operator which takes the list and maps it back on to the matrices. We will denote these as $A^T(X)$ and $B^T(X)$. These adjoints are defined by this relation: $\langle A(X), y\rangle = \langle X, A^T(y) \rangle$.

We will begin our discussion with the basic theory. We will denote a vector by placing an arrow above it. For example, $\vec{u}$. However, once we get into defining the system, we can drop this notation because we do not run into situations in which this could get confusing. We will be using the terms a, b, y, and t in our systems which will all be vectors. The vectors a and y will be of length k and the vectors b and t will be of length m.

Lastly, we will quickly note a few things. We will use $\mathcal{C}$ to denote a cone, e will be the all ones vector, and $A \circ B$ will be the entrywise product of A and B.

2 Semidefinite Programming

Semidefinite programming is a newly developed branch of conic programming. To begin, we will discuss this over arching subject. Conic programming works in a Euclidean space. By a Euclidean space, we mean a vector space, E, over $\mathbb{R}$ with positive definite inner product. A positive definite inner product refers to the case where, for $x \in E$, $\langle x, x\rangle \geq 0$ and only equals zero if x is the zero vector. A **cone** in E is any subset closed under addition and multiplication by non-negative scalars. That is $\mathcal{C} \subseteq E$ is a cone if the following hold:

$$\forall\, x, y \in \mathcal{C}, x + y \in \mathcal{C}$$

$$\forall\, x \in \mathcal{C} \text{ and } \forall\, \beta \geq 0, \beta x \in \mathcal{C}$$

A **conic programming problem** optimizes a linear function over a linear section of some cone. A general conic programming problem takes the following form:

$$\max\ \langle C, X\rangle$$

$$st\ \ \langle A_i, X\rangle = b_i,\ \ 1 \leq i \leq m$$

$$x \in \mathcal{C}$$

Let us consider two special cases of conic programming. One is linear programming in which the vector space E is equal to $\mathbb{R}^n$. A general linear programming problem (LP) is:

$$\max\ c^T x$$

$$st\ \ a_i^T x = b_i$$

$$x \geq 0$$

One can see that the objective function is equivalent to the conic programming problem above, noting the definition of an inner product, $\langle x, y\rangle = x^T y$. The cone in which we are optimizing over requires all the entries of x are nonnegative.

Another special case is semidefinite programming, our main topic for discussion. In this case, the vector space is of $n\ x\ n$ symmetric matrices. The general semidefinite programming problem (SDP) is as follows:

$$\max\ \langle C, X\rangle$$

$$st\ \ \langle A_i, X\rangle = b_i$$

$$X \succcurlyeq 0$$

Here we take the inner product to be the trace of the product of two matrices, one being a transpose. For example, $\langle C, X\rangle = tr(C^T X)$. The cone we are use in the SDP is one which requires matrices to be positive semidefinite.

It can be shown that all LPs can be relaxed into SDPs. The idea is simple in that we think of the vectors used in LPs as just $1\ x\ n$ matrices, allowing us to refer to c as$C$, x as$X$, and a_i as A_i. With this idea we can see that $\langle C, X\rangle = tr(C^T X)$ will just become the trace of a $1\ x\ 1$ matrix, or just a single value, and this value will be the inner product of the vectors c and x. We can also see that the vector x, or the $1\ x\ n$ matrix X, will be positive semidefinite because we all the entries of x need to be nonnegative.

2.1 Background Theory

To discuss semidefinite programming further we need to introduce some background theory that will lead us to defining semidefinite matrices and understanding their properties.

Let A be a real symmetric $n\ x\ n$ matrix. We know that $\lambda \in \mathbb{C}$ (where $\mathbb{C}$ is the set of complex numbers) is an eigenvalue of A if there is a nonzero $\vec{v} \in \mathbb{C}^n\ s.t.\ A\vec{v} = \lambda\vec{v}$. If this holds then not only is λ an **eigenvalue** but $\vec{v}$ is an **eigenvector**.

If M is a $n\ x\ n$ matrix over the complex numbers, then $M^\dagger$ is the conjugate transpose of M. For example:

$$Let\ M = \begin{bmatrix} 1-i & i \\ 2 & 3 \end{bmatrix}\ then\ M^\dagger = \begin{bmatrix} 1+i & 2 \\ -i & 3 \end{bmatrix}$$

M is called **Hermitean** if and only if $M^\dagger = M$. In the case that M is a matrix over the Reals then it is Hermitean if and only if M is symmetric.

These two ideas are utilized in our first lemma:

Lemma 1: Every eigenvalue of any Hermitean matrix is real.

Let us prove this:

First we recall the definition of the inner product, $\langle\cdot,\cdot\rangle$ on $\mathbb{C}$ as $\langle\vec{u},\vec{v}\rangle = \vec{u}^T\vec{v} = \sum_{i=1}^n \bar{u}_i v_i$

(where $\bar{u}$ is the complex conjugate)

We know that M is Hermitean and suppose λ is an eigenvalue for some nonzero vector $\vec{u}$, then

$M\vec{u} = \lambda\vec{u}$.

Also, from the above definition, we can say:

$$\lambda\langle\vec{u},\vec{u}\rangle = \lambda(\vec{u}^T\vec{u}) = \vec{u}^T(\lambda\vec{u}) = \vec{u}^T(M\vec{u}) = (\vec{u}^T M^\dagger)\vec{u}$$

$$= (M\vec{u})^T\vec{u} = \langle M\vec{u},\vec{u}\ \rangle = \langle\lambda\vec{u},\vec{u}\rangle = \bar{\lambda}\langle\vec{u},\vec{u}\rangle$$

So we have that $\lambda\langle\vec{u},\vec{u}\rangle = \bar{\lambda}\langle\vec{u},\vec{u}\rangle$ which means that $\lambda = \bar{\lambda}$ since $\langle\vec{u},\vec{u}\rangle \neq 0$. This means that λ is real.

$$Note: a + bi = a - bi\ if\ and\ only\ if\ bi = 0$$

This means that every eigenvalue of a Hermitean matrix is real.■

Our second lemma discusses more properties using eigenvalues.

Lemma 2: If A is a real symmetric matrix, then eigenvectors belonging to distinct eigenvalues are orthogonal.

The proof is as follows:

Let $A\vec{u} = \lambda\vec{u}$ and $A\vec{v} = \mu\vec{v}$ where $\lambda \neq \mu$

Consider $\lambda\langle\vec{u},\vec{v}\rangle = (\lambda\vec{u})^T\vec{v}$:

$$\lambda\langle\vec{u},\vec{v}\rangle = (\lambda\vec{u})^T\vec{v} = (A\vec{u})^T\vec{u} = \vec{u}^T A^T\vec{v} = \vec{u}^T A\vec{v} = \vec{u}^T(\mu\vec{v}) = \mu\langle\vec{u},\vec{v}\rangle$$

So we have that $\lambda\langle\vec{u},\vec{v}\rangle = \mu\langle\vec{u},\vec{v}\rangle$ and since we know that $\lambda \neq \mu$ because they are distinct eigenvalues, then that means $\langle\vec{u},\vec{v}\rangle = 0$. This only occurs if $\vec{u}$ is orthogonal to $\vec{v}$. ■

This third lemma discusses linear transformations.

Lemma 3: Every linear transformation $T: V \to V$ has a real eigenvalue, where T is a finite-dimensional, real vector space.

Proof:

Coordinize V so that $T(\bar{x}) = A\bar{x}$ for some $n\ x\ n$ matrix A where n is also the dimension of V.

The characteristic polynomial for $T(\bar{x})$ is $\mathcal{X}_T(\lambda) = det(\lambda I - A)$ which is a polynomial of degree n in λ with real coefficients.

From the Fundamental Theorem of Algebra, we know that every non-constant polynomial has a root. This is true for complex numbers with polynomial of positive degree.

So for some $\theta \in \mathbb{C}$, $\mathcal{X}_T(\theta) = 0$. This means that $det(\theta I - A) = 0$.

So $\exists v \neq 0\ s.t. (\theta I - A)\bar{v} = \bar{0} \Longrightarrow A\bar{v} = \theta\bar{v}$

This means that $\bar{v}$ is an eigenvector for A, hence for$T$, and θ is an eigenvalue for A and T.

By Lemma 1, this means that θ is real. ■

These three lemmas lead us to our first main theorem.

Theorem 1: If A is a real symmetric $n\ x\ n$ matrix then A is orthogonally diagonalizable, i.e. there exists orthogonal P and diagonal D s.t. $A = PDP^T,\ i.e. AP = PD.$

The proof is done by induction:

Let $T: \mathbb{R}^n \rightarrow \mathbb{R}^n$ by $T(\bar{x}) = A\bar{x}$

Then T has an eigenvector $\bar{v}_1 \in \mathbb{R}^n$ so $A\bar{v}_1 = \theta\bar{v}_1$ for some $\theta \in \mathbb{R}$

Consider $W = \bar{v}_1{}^{\perp} = \{\bar{w} \in V: \langle \bar{w}, \bar{v}_1 \rangle = 0\}$

We claim: $\bar{w} \rightarrow A\bar{w}$ is a linear transformation $T^{'} = W \rightarrow W$

Proof: If $\bar{w} \perp \bar{v}_1 \Longrightarrow A\bar{w} \perp \bar{v}_1$

$\langle A\bar{w}, \bar{v}_1 \rangle = \langle \bar{w}, A\bar{v}_1 \rangle = \theta\langle \bar{w}, \bar{v}_1 \rangle = 0$

$\Longrightarrow A\bar{w} \perp \bar{v}_1$

By induction, W has an orthogonal basis of eigenvectors for $T^{'}$. ■

The next two theorems will be quickly noted allowing us to finally arrive at our definitions for positive semidefinite and positive definite matrices. The first theorem builds on Theorem 1 to further describe properties of symmetric $n\ x\ n$ matrices.

Theorem 2: Every symmetric $n\ x\ n$ matrix A is orthogonally similar to a diagonal matrix.

The second is the Spectral Decomposition Theorem.

Theorem 3: For any real symmetric $n\ x\ n$ matrix, there exists real numbers $\theta_1, \theta_2, \dots \dots \theta_n$ and orthonormal basis $v_1, v_2, \dots \dots v_n$ such that

$$A = \theta_1 v_1 v_1{}^T + \theta_2 v_2 v_2{}^T + \cdots + \theta_n v_n v_n{}^T = \sum_{j=1}^{n} \theta_j v_j v_j{}^T$$

We now have the background to introduce positive semidefinite matrices and discuss some of their properties.

Positive semidefinite (PSD): A real symmetric matrix A is positive semidefinite if $\forall \vec{v} \in \mathbb{R}^n,\ \vec{v}^T A\vec{v} \geq 0$.

Positive definite (PD): A real symmetric matrix B is positive definite (PD) if $\forall \vec{v} \in \mathbb{R}^n,\ \vec{v} \neq 0,\ \vec{v}^T B\vec{v} > 0$.

Here are some examples:

$$(PD)A_1 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{vmatrix}, \quad (PSD)A_2 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{vmatrix}, \quad (not\ PSD)A_3 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -3 \end{vmatrix}$$

We can prove these are true with the defining equations. We will show this for the first matrix:

First we can see that A_1 is a real symmetric matrix.

Second, we need $\forall \vec{v} \in \mathbb{R}^n,\ \vec{v} \neq 0, \vec{v}^T A \vec{v} > 0$.

We define $\vec{v} = \begin{vmatrix} v_1 \\ v_2 \\ v_3 \end{vmatrix}$ and we have $A_1 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{vmatrix}$.

So $\vec{v}^T A_1 \vec{v} = {v_1}^2 + 2{v_2}^2 + 3{v_3}^2$ which is >0 unless $v_1, v_2, v_3 = 0$. So A_1 is PD.

This brings us to our fourth theorem.

Theorem 4: Let A be a symmetric $n\ x\ n$ real matrix then

1. *A is PSD if and only if all eigenvalues of A are non-negative*
2. *A is PD if and only if all eigenvalues of A are positive*

We will do the proof of the first part of the theorem (the second part is very similar)

($\Longrightarrow$) Assume that A is PSD and $A\vec{u} = \lambda\vec{u},\ \vec{u} \neq \vec{0}$. Then have $\vec{u}^T A \vec{u} = \lambda \vec{u}^T \vec{u}$ by multiplying both sides by $\vec{u}^T$. But $\vec{u}^T A \vec{u} \geq 0$ because A is PSD and $\vec{u}^T \vec{u} > 0$ so this means that $\lambda \geq 0$.

($\Longleftarrow$) Assume all eigenvalues are non-negative. We can write A as $A = PDP^T$ from our first theorem. So D is diagonal with all entries non-negative. Also note, for any vector $\vec{u}$, $\vec{u}^T D \vec{u} = \sum D_{ii} {u_i}^2$ which will always be positive.

We can now write $\vec{v}^T A \vec{v}$ as $\vec{v}^T PDP^T \vec{v}$ and see that

$$\vec{v}^T A \vec{v} = \vec{v}^T PDP^T \vec{v} = (P^T \vec{v})^T D (P^T \vec{v}) \geq 0$$

which means that A is PSD. ■

At this point, we will introduce the idea of a principal submatrix which is composed from a larger, square matrix. A principal submatrix of a $n\ x\ n$ matrix A is any $m\ x\ m$ submatrix obtained from A by deleting $n - m$ rows and the corresponding columns. Our last theorem shows the relationship between these principal submatrices and PSD matrices.

Theorem 5: Everyone principal submatrix of a positive semidefinite matrix A is also positive semidefinite.

These theorems and lemmas provide the background for semidefinite programming. With their understanding, we will be able to further understand the ideas presented next.

2.2 Semidefinite Programming Theory

With all of the background introduced, we can continue our discussion of semidefinite programming. The general system is as follows:

SDP

$Max\ \langle C, X\rangle$

$s.t. A(X) = a$

$B(X) \leq b$

$X \succcurlyeq 0$

$$Where\ A(X) = \begin{vmatrix} tr(A_1X) \\ tr(A_2X) \\ \vdots \\ tr(A_kX) \end{vmatrix}\ and\ B(X) = \begin{vmatrix} tr(B_1X) \\ tr(B_2X) \\ \vdots \\ tr(B_kX) \end{vmatrix}$$

We can obtain the dual of this system fairly easily. The process is explained [1].

We start by letting v^* equal the optimal value for the SDP and introducing Lagrange multipliers $y \in \mathbb{R}^k$ for the equality constraint and $t \in \mathbb{R}^m$ for the inequality constraint.

We can then say that

$$v^* = max_{X \succcurlyeq 0} min_{t \geq 0, y}\ trCX + y^T\big(a - A(X)\big) + t^T(b - B(X)).$$

This will be less than the optimal solution for the dual which is equal to

$$min_{t \geq 0, y} max_{X \succcurlyeq 0}\ tr\big(C - A^T(y) - B^T(t)\big)X + a^Ty + b^Tt.$$

Note that the inner max over X is bounded from above only if $A^T(y) + B^T(t) - C \geq 0$ and the maximum happens when $tr\big(C - A^T(y) - B^T(t)\big)X = 0$. So we can just say that we need to min over the rest of the function when this term is zero. This presents us with our D-SDP shown below.

D-SDP

$Min\ \ a^Ty + b^Tt$

$s.t. A^T(y) + B^T(t) - C \succcurlyeq 0$

$t \geq 0$

$$Where\ A^T(y) = \sum_{i=1}^{k} y_iA_i\ and\ B^T(t) = \sum_{i=1}^{l} t_iB_i$$

Recall that in linear programming, we have a Weak Duality Theorem. This theorem can be adapted to semidefinite programming. The Weak Duality Theorem for SDP is stated below.

Weak Duality Theorem: If X is feasible for the SDP and (y, t, Z) is feasible for the D-SDP,

then $\langle C, X\rangle \leq y^T a + t^T b$.

The proof of this is as follows:

First we need to note that for X and (y, t, Z) to be feasible, then $X \succcurlyeq 0, Z \succcurlyeq 0$ meaning that

$\langle Z, X\rangle \geq 0$.

We can obtain C from the first constraint in the dual SDP and plug that into $\langle C, X\rangle$:

$\langle C, X\rangle = \langle\left(\sum_{i=1}^{k} y_i A_i + \sum_{i=1}^{l} t_i B_i - Z\right), X\rangle = \sum_{i=1}^{k} y_i\langle A_i, X\rangle + \sum_{i=1}^{l} t_i\langle B_i, X\rangle - \langle Z, X\rangle$

But we know, since X and $(y, t, Z)$are both feasible, then the constraints hold. That means that:

$$\langle A_i, X\rangle = a \ and \ \langle B_i, X\rangle \leq b$$

So we can write:

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i\langle A_i, X\rangle + \sum_{i=1}^{l} t_i\langle B_i, X\rangle - \langle Z, X\rangle \leq \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle Z, X\rangle$$

Since we have that $\langle Z, X\rangle \geq 0$, then we can conclude that:

$$\sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle Z, X\rangle \leq \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i = y^T a + t^T b$$

So we have that $\langle C, X\rangle \leq y^T a + t^T b$. ■

Similarly, we have a form of the Complementary Slackness Theorem for semidefinite programming. It is as follows:

Complementary Slackness Theorem: If X is optimal for SDP, (y, t, Z) is optimal for D-SDP and $\langle C, X\rangle = y^T a + t^T b$ then

1. $Tr(ZX) = 0$
2. *For every i, $1 \leq i \leq l$, $t_i = 0$* or $\langle B_i, X\rangle = b_i$

Here is the proof:

1. $\langle C, X\rangle = \langle\left(\sum_{i=1}^{k} y_i A_i + \sum_{i=1}^{l} t_i B_i - Z\right), X\rangle = \sum_{i=1}^{k} y_i\langle A_i, X\rangle + \sum_{i=1}^{l} t_i\langle B_i, X\rangle - \langle Z, X\rangle$
   We can move things around to arrive at:

$$\langle Z, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle - \langle C, X\rangle$$

We have that $\langle A_i, X\rangle = a_i \text{ and } \langle B_i, X\rangle \le b_i$ so

$$\langle Z, X\rangle \le \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i - \langle C, X\rangle = y^T a + t^T b - \langle C, X\rangle$$

But $\langle C, X\rangle = y^T a + t^T b$ from assumption that X and (y, t, Z) are optimal.
So $\langle Z, X\rangle \le y^T a + t^T b - \langle C, X\rangle = 0$ which means that $\langle Z, X\rangle \le 0$.
But since $Z \succcurlyeq 0$ and $X \succcurlyeq 0$ then $\langle Z, X\rangle \ge 0$
So $\langle Z, X\rangle = Tr(ZX) = 0$

2. We know that $\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle - \langle Z, X\rangle$ but from above, we have that $\langle Z, X\rangle = 0$ so we can drop this term.
This leaves us with $\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle$
But since $B(X) \le b$, then

$$\langle C, X\rangle = \sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle \le \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i = y^T a + t^T b$$

But by assumption that X and (y, t, Z) are optimal, then $\langle C, X\rangle = y^T a + t^T b$.
This means that this must hold:

$$\sum_{i=1}^{k} y_i \langle A_i, X\rangle + \sum_{i=1}^{l} t_i \langle B_i, X\rangle = \sum_{i=1}^{k} y_i a_i + \sum_{i=1}^{l} t_i b_i$$

Since we know by the constraints that $\langle A_i, X\rangle = a_i$, it obviously follows that

$$\sum_{i=1}^{k} y_i \langle A_i, X\rangle = \sum_{i=1}^{k} y_i a_i$$

What we really need is that

$$\sum_{i=1}^{l} t_i \langle B_i, X\rangle = \sum_{i=1}^{l} t_i b_i$$

By the constraints, we have $\langle B_i, X\rangle \le b_i$. This gives us two cases:

   1. If $\langle B_i, X\rangle < b_i$, then we must have that $t_i = 0, \forall i$ in order for the equality to hold
   2. If $\langle B_i, X\rangle = b_i$, then the equality obviously holds

So for $\langle C, X\rangle = y^T a + t^T b$, which means that X and (y, t, Z) are optimal solutions for SDP

and D-SDP, we need either $t_i = 0\ \forall i$ or $\langle B_i, X\rangle = b_i$ which was the claim of the theorem. ■

The Strong Duality Theorem can also be adapted but only for certain situations. It states that $\langle C, X\rangle = y^T a + t^T b$ for optimal X and (y, t, Z). This will only hold if we have strictly feasible interior points for both SDP and D-SDP [1].

2.3 The Interior Point Method

Since semidefinite programming loses its linearity, we cannot use linear programming methods to solve. Many methods were developed, however, the most popular of which are interior point methods. Understanding the procedure of these interior point methods is important because they are very efficient and much of the solving software available to solve SDPs utilizes this method. To begin, we will revisit linear programming and discuss some of its advanced topics presented by Robert Vanderbei in his book, *Linear Programming: Foundations and Extensions* [2]. These include the central path and the path following method. We will then see how these ideas can be applied to semidefinite programming as explained by Helmberg et al. in *An Interior-Point Method for Semidefinite Programming* [1].

2.3.1 The Central Path

To begin, let us look at a simple linear programming example. Suppose we are given the problem:

$$
\begin{array}{lll}
(\textit{Primal}) & & \text{Max } x_1 - 2x_2 \\
& \text{s.t.} & -x_2 \leq -1 \\
& & x_1 \leq 2 \\
& & x_1,\, x_2 \geq 0
\end{array}
$$

Its corresponding dual is:

$$
\begin{array}{lll}
(\textit{Dual}) & & \text{Min } -y_1 + 2y_2 \\
& \text{s.t.} & y_2 \geq 1 \\
& & y_1 \geq -2 \\
& & y_1, y_2 \geq 0
\end{array}
$$

By adding slack variables we obtain:

$$
\begin{array}{lll}
(\textit{Primal}) & & \text{Max } x_1 - 2x_2 \\
& \text{s.t.} & -x_2 + p = -1 \\
& & x_1 + q = 2 \\
& & x_1,\, x_2 \geq 0
\end{array}
$$

(*Dual*) Min $-y_1 + 2y_2$

s.t. $y_2 - t_1 = 1$

$y_1 \quad - t_2 = -2$

$y_1, y_2 \geq 0$

This now has the following matrix form:

(*Primal*) Max $c^T x$

s.t. $Ax = b$

$x \geq 0$

(*Dual*) Min $b^T y$

s.t. $A^T y \geq c$

$y \geq 0$

This example will be used throughout our discussion of the central path and the path following method to help illustrate the ideas being presented.

2.3.1.1 Barrier Functions

Barrier (or penalty) **functions** are a standard tool for converting constrained optimization problems to unconstrained problems. This allows us to avoid any problems the constraints may present when solving. For example, in the case of an LP, the system's feasible region is represented by a polytope. When trying to solve the system, we can get stuck in the polytope's corners, slowing down our algorithm. With increasing numbers of feasible solutions, the number of corners can grow exponentially. Applying the barrier function can eliminate this problem by not allowing us to get close enough to these corners.

Applying the ideas of a barrier function calls us to add a term to the current objective function. The new function we create by doing this is called the barrier function. For example, if we let $f(x)$ be our objective function and $h(x)$ be the term we are adding, your barrier function will be: $B(x) = f(x) + h(x)$. When using a barrier function, we incorporate the constraints into the term we are adding, allowing us to eliminate them from the system. To do this, we make the term model a penalty for approaching the edge of the feasible region. In this way, we will not be given the potential to violate the constraints or to slow down our algorithm.

To create this penalty, we use a function that becomes infinite at the boundary of feasible region. Ideally, one would like to use a function that is 0 inside the feasible region, allowing the barrier function to be exactly the original objective function, and that is infinite when it leaves the feasible region. The only problem in doing this is that the barrier function is now discontinuous which means we cannot use simple mathematics to investigate it [2].For this reason, we use a function that goes to 0 inside the feasible region and that becomes negative infinity as it goes outside. This function will now be continuous, smoothing out the discontinuity at the boundaries [2], and will allow for differentiability.

Picture

There are several functions that can be applied in order to obtain the above result. However, one of the most popular is the logarithm function. It has become more and more popular because it performs very well [3]. We apply it to a system by introducing a new term in the objective function for each variable. The term will be a constant times the logarithm of the given variable. As that variable goes to 0, the function will get more and more similar to the original objective function [2]. However, if we get close to the boundaries, where the variables are zero, the logarithmic terms will begin to go to negative infinity. In this way, we can obtain our penalty that will keep us from violating the constraints. This barrier function is termed the **logarithmic barrier function**.

For our system, when we apply these ideas of the logarithm barrier function (and choose $\mu > 0$ because it is a maximization problem), we end up with:

(*Primal*) Max $x_1 - 2x_2 + \mu \log x_1 + \mu \log x_2 + \mu \log p + \mu \log q$

s.t. $x_2 - 1 = p$

$-x_1 \quad + 2 = q$

(*Dual*) Min $-y_1 + 2y_2 + \mu \log y_1 + \mu \log y_2 + \mu \log t_1 + \mu \log t_2$

s.t. $y_2 - t_1 = 1$

$y_1 \quad - t_2 = -2$

This is not equivalent to the original system but it is very close. Notice: as μ gets small (i.e. close to zero), these objective functions will get closer and closer to the original objective functions. Also notice that we no longer have one system but a whole family of systems [2]. Depending on what μ's we choose, we will obtain different systems to solve, each with its own optimal solutions. We end up with multiple systems all similar but depending on μ.

2.3.1.2 Lagrangians

Next we want to consider the use of **Lagrangians**. These ideas are presented in Robert Vanderbei's book, *Linear Programming: Foundations and Extensions* [2], and described below. Let's say we are given the problem:

$$Max\ F(x)$$

$$s.t. g(x) = 0$$

Remember from calculus that a way to find the maximum of a function is to set its derivative equal to0. So a natural first instinct to solve this problem would be to take the gradient of F (since F is not necessarily in one dimension) and setting it equal to 0. Unfortunately, the given constraint prevents us from proceeding in this way since we wouldn't be taking that into account.

To think of how to solve for the maximum given a constraint, let us look at the problem geometrically. The gradient of $F(x)$ represents the direction in which f increases the fastest. When incorporating the constraint, we maximize $F(x)$ not when the gradient of f is 0 but when it is perpendicular to the set of feasible solutions of the system. That is, the set $\{x: g(x) = 0\}$. However, for any given point in the feasible set, the gradient of g is orthogonal to the feasible set at the given point. So we now have the gradient of F at a maximum and the gradient of g orthogonal to the feasible set. This means, in order for a point to be a maximum, the gradients must be proportional. That is, for x' a critical point, $\nabla\mathrm{F}(x') = y\nabla g(x')$, for some real number y . Typically y is called a **Lagrange multiplier**.

Now, if we move into a system that has multiple constraints, each constraint will represent, geometrically, a hypersurface. The feasible region will be the intersection of all the hypersurfaces. Because of this, the space perpendicular to the feasible set is given by the span of the gradients. Since we need the gradient of F to be perpendicular to the feasible set, this means it must be in this span. We end up with this system of equations for a critical point:

$$g(x') = 0,$$

$$\nabla\mathrm{F}(x') = \sum_{i=1}^{m} y_i \nabla g(x')$$

These are the ideas behind the Lagrangian function. We can interpret these results algebraically. First we present the Lagrangian function: $L(x, y) = F(x) - \sum_i y_i g_i(x)$. Having this, we look for its critical points for both x and y. We do this by setting all the first derivatives equal to zero since now we do not have any constraints to worry about (unlike above). This will give us:

$$\frac{\partial L}{\partial x_j} = \frac{\partial F}{\partial x_j} - \sum_i y_i \frac{\partial g_i}{\partial x_j} = 0$$

$$\frac{\partial L}{\partial y_i} = -g_i = 0$$

By writing these equations into vector notation, we will obtain the exact equations we did geometrically. We can rewrite our first equation as:

$$\frac{\partial F}{\partial x_j} = \sum_i y_i \frac{\partial g_i}{\partial x_j}$$

In vector form, for the critical point, this is nothing else but $\nabla F(x^{'}) = \sum_{i=1}^{m} y_i \nabla g(x^{'})$, our first equation we derived geometrically. It is even easier to see that $-g_i = 0$ is equivalent to $g(x^{'}) = 0$ for the critical point. These equations we have introduced for the derivatives are typically called the **first-order optimality conditions**.

It is possible to apply these ideas to the barrier function.

Given the general logarithmic barrier function for an LP: $c^T x + \mu \sum_j \log x_j + \mu \sum_i \log w_i$

and having $Ax + w = b$ as the constraints, the Lagrangian is:

$$L(x, w, y) = \ c^T x + \mu \sum_j \log x_j + \mu \sum_i \log w_i + y^T (b - Ax - w)$$

In general, the first-order optimality conditions will be:

$$\frac{\partial L}{\partial x_j} = c_j + \mu \frac{1}{x_j} - \sum y_j a_{ij} = 0, \quad j = 1,2, \ldots \ldots, n$$

$$\frac{\partial L}{\partial w_i} = \mu \frac{1}{w_i} - y_i = 0, \quad i = 1,2, \ldots \ldots, m$$

$$\frac{\partial L}{\partial y_i} = b_i - \sum_j a_{ij} x_j - w_i = 0, \quad i = 1,2, \ldots . m$$

In the case of our example, implementing the Lagrangian process is as follows:

$$L_P(x_1, x_2, p, q) = x_1 - 2x_2 + \mu \log x_1 + \mu \log x_2 + \mu \log p + \mu \log q + y_1(1 + x_2 - p) + y_2(2 - x_1 - q)$$

$$L_D(y_1, y_2, t_1, t_2) = -y_1 + 2y_2 + \mu \log y_1 + \mu \log y_2 + \mu \log t_1 + \mu \log t_2 + x_1(1 - y_2 + t_2) + x_2(-2 + y_1 + t_2)$$

Following through with the primal, the corresponding first derivatives are:

$$\frac{\partial L_P}{\partial x_1} = 1 + \mu \frac{1}{x_1} - y_2$$

$$\frac{\partial L_P}{\partial y_2} = -2 + \mu \frac{1}{x_2} + y_1$$

$$\frac{\partial L_P}{\partial p} = \mu \frac{1}{p} - y_1$$

$$\frac{\partial L_P}{\partial q} = \mu \frac{1}{q} - y_2$$

$$\frac{\partial L_P}{\partial x_1} = 1 + x_2 - p$$

$$\frac{\partial L_P}{\partial x_2} = 2 - x_1 - q$$

These can be simplified into the above general format for the first-order optimality conditions.

We can rewrite these general equations for the first-order optimality conditions into matrix form. In this form, X and W will be diagonal matrices whose diagonal entries are the components of the vector x and w, respectively. The matrix form is shown below.

$A^T y - \mu X^{-1} e = c$

$y = \mu W^{-1} e$

$Ax + w = b$

We introduce a vector $z = \mu X^{-1} e$, allowing us to rewrite this as

$$Ax + w = b$$

$$A^T y - z = c$$

$$z = \mu X^{-1} e$$

$$y = \mu W^{-1} e$$

By multiplying the third equation through by X and the fourth equation through by W, we will get a primal-dual symmetric form for these equations [2]. In these equations, we introduce Z and Y which have the same properties of X and W. They are diagonal matrices whose diagonal entries are the components of their respective vector. With these modifications, the equations are

$$Ax + w = b$$

$$A^T y - z = c$$

$$XZe = \mu e$$

$$YWe = \mu e$$

The last two equations are called the μ-complementary conditions because if you let $\mu = 0$ then they are the usual complementary slackness conditions that need to be satisfied at optimality [2].

In the case of our example, once we write down the problem in terms of these newly developed equations, we can use simple algebra and substitution to obtain a solution.

2.3.1.3 The Central Path and the Path-Following Method

When discussing the barrier problem and Lagrangians, we must first explore the idea of the central path. When we introduced the logarithmic barrier function, we included a constant μ that scaled the logarithm term. Remember we noticed that because the barrier function depends on μ, a whole set of functions were created that were parameterized by μ. For each μ, we typically have a unique solution which is obtained somewhere in the interior of the feasible region. As μ goes to zero, the barrier function goes to the original objective function so, as we take smaller and smaller μ values and obtain the corresponding solutions, we will move closer and closer to the optimal solution of the original objective function. If all these solutions are plotted, one will notice a path is created through the feasible region leading to the optimal solution. This path is called the **central path**.

To find these unique solutions for each μ, we use the first-order optimality conditions. After utilizing the barrier function and the Lagrangian we arrived at these for the dual. Applied to the primal we have:

$$Ax + w = b$$

$$A^T y - z = c$$

$$XZe = \mu e$$

$$YWe = \mu e$$

The solution to these first-order optimality conditions will be the unique solution for a given μ, written as $(x_\mu, w_\mu, y_\mu, z_\mu)$. Once we have these solutions for increasingly smaller μ values, we can then plot them inside the feasible region to obtain the central path.

This idea of the central path is essential in the path-following method. This method allows us to start from any point and move towards the central path in order to reach the optimal solution. The procedure is as follows:

1. Estimate a value for μ at the current point
2. Compute a new reasonable value for μ that is smaller than the current value (but not too small)
3. Compute step directions $(\Delta x, \Delta w, \Delta y, \Delta z)$ which point towards the point $(x_\mu, w_\mu, y_\mu, z_\mu)$ on the central path
4. Calculate a step length parameter θ so that the new point $(x + \Delta x, w + \Delta w, y + \Delta y, z + \Delta z)$ still respects the constraints.
5. Replace the old point with the newly calculated one

We repeat this process until we get sufficiently close to the optimal solution [2]. The next sections explain this process in more detail.

2.3.1.3.1 Estimating μ

We are going to begin at a point with all variables positive. Starting with the first two steps, we need to understand how to estimate a good value for μ. If μ is too big then we might go to the center of the feasible set but if μ is too small, we may stray too far from the central path. This would result in us getting stuck at a place that is suboptimal. So we need to find a good compromise. One way to do this is to obtain the current value of μ and then just choose something smaller. If we know that the point we are starting with is on the central path then there are a lot of ways to figure out what μ is. One way is to use the first-order optimality conditions. Since we have from our set of equations for the primal that:

$$XZe = \mu e$$

$$YWe = \mu e$$

we can solve these for μ and then average the values. Doing this will give us the following equation:

$$\mu = \frac{z^T x + y^t w}{n + m}$$

If we know that our starting point is definitely on the path, we can use this equation to get the exact value for μ. However, we can still use it find μ when the point is not on the central path. It will just be an estimate. Since we are trying to get closer to optimality and we know this occurs as μ gets smaller, we need a μ that is smaller than the current value. So we take the above equation and multiply it by a fraction. Let this fraction be represented by δ. So for any point we can estimate our new μ value by:

$$\mu = \delta \frac{z^T x + y^t w}{n + m}$$

2.3.1.3.2 Step Directions and Newton's Method

Now as we move on to step three, we need to understand how we compute a step direction. First, though, we must discuss **Newton's method**. Newton's method allows one to find the point where a given function equals zero using differentiability and smoothness properties. It solves this problem as follows: given any point, find a step direction such that if we go in that direction and find a new point, the function evaluated at that new point will be close to zero. Ideally, given a function $F(x)$ and a point p, we want to find a direction Δp such that $F(p + \Delta p) = 0$. This method works well when the function is linear, however, if it is nonlinear, it is not possible to solve. To accommodate this, if given a nonlinear function, we implement the Taylor series and use the first two terms to approximate the function. So we have:

$$F(p + \Delta p) \approx F(p) + F^{'}(p)\Delta p$$

This can also be applied to a set of functions where:

$$F(p) = \begin{bmatrix} F_1(p) \\ F_2(p) \\ \vdots \\ F_N(p) \end{bmatrix}, p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_N \end{bmatrix} and\ F^{'}(p) = \begin{bmatrix} \frac{\partial F_1}{\partial p_1} & \cdots & \frac{\partial F_1}{\partial p_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_N}{\partial p_1} & \cdots & \frac{\partial F_N}{\partial p_N} \end{bmatrix}$$

Setting the function equal to zero, we will obtain the following equation and are then able to solve for Δp:

$$F^{'}(p)\Delta p = -F(p)$$

Once we know Δp we can then change our current point to $p + \Delta p$ and evaluate the function at the new point. This process repeats until the value of the function gets approximately equal to 0.

That is the process of Newton's method and it will be a guide for how we proceed to find the step direction. We want to find a direction such that if we move in that direction, the new point we end up at will be on the central path. Remember that on the central path, for the primal, we have the following first-order optimality conditions:

$$Ax + w = b$$

$$A^T y - z = c$$

$$XZe = \mu e$$

$$YWe = \mu e$$

So if our new point is on the central path we should have the following:

$$A(x + \Delta x) + (w + \Delta w) = b$$

$$A^T(y + \Delta y) - (z + \Delta z) = c$$

$$(X + \Delta X)(Z + \Delta Z)e = \mu e$$

$$(Y + \Delta Y)(W + \Delta W)e = \mu e$$

We rewrite these equations with all the unknowns on one side and the knowns on the other. Our current point is (x, w, y, z), so we know these values. This leaves us with:

$$A\Delta x + \Delta w = b - Ax - w$$

$$A^T \Delta y - \Delta z = c - A^T y + z$$

$$Z\Delta x + X\Delta z + \Delta X \Delta Z e = \mu e - XZe$$

$$W\Delta y + Y\Delta w + \Delta Y \Delta W e = \mu e - YWe$$

If we let $\rho = b - Ax - w$ and $\sigma = c - A^T y + z$, we can simplify the equations. Also, we want to have a linear system. In order to do this, we drop the nonlinear terms. (It may be surprising that we can do this but it is just a result of Newton's method. As ΔX and ΔZ get negligible, which occurs when we utilize Newton's method, their product goes to zero even faster, making the term $\Delta X \Delta Z e$ become irrelevant. Similarly this happens for ΔY and ΔW in the last equation.)

Modifying the equations as such, we end up with

$$A\Delta x + \Delta w = \rho$$

$$A^T \Delta y - \Delta z = \sigma$$

$$Z\Delta x + X\Delta z = \mu e - XZe$$

$$W\Delta y + Y\Delta w = \mu e - YWe$$

It can easily be shown that this is exactly the Newton equation we talked about above. Let

$$p = \begin{bmatrix} x \\ w \\ y \\ z \end{bmatrix}, F(p) = \begin{bmatrix} Ax + w - b \\ A^T w - z - c \\ XZe - \mu e \\ YWe - \mu e \end{bmatrix}, \Delta p = \begin{bmatrix} \Delta x \\ \Delta w \\ \Delta y \\ \Delta z \end{bmatrix}$$

We will obtain $F^{'}(p) = \begin{bmatrix} A & I & 0 & 0 \\ 0 & 0 & A^T & -I \\ Z & 0 & 0 & X \\ 0 & Y & W & 0 \end{bmatrix}$ when we compute all the partial derivatives.

Plugging these all into the equation $F^{'}(p)\Delta p = -F(p)$ we will get exactly the equations we have above.

Once we have these equations, we can then solve them for $(\Delta x, \Delta w, \Delta y, \Delta z)$ and obtain our step direction.

2.3.1.3.3 The Step Length Parameter

Our next step is to choose a step length parameter. Typically this is called θ and it is applied to the step direction. For example, our new point will be $\tilde{x} = x + \theta \Delta x$. When we obtained the equations above we were assuming the parameter was 1. Although, depending on our problem, when we take this step with the parameter equal to 1, we may violate some of the constraints and end up with a new solution where some of the variables are less than or equal to zero. We need to make sure this won't occur for any variable. Sticking with x as our example, this means we need $x_j + \theta \Delta x_j > 0,\ j = 1,2, \dots \dots, n$. Note: this is relevant only when Δx_j is negative. This is because x_j is positive and if Δx_j was also positive, then their sum would already be positive. Using this inequality, we can solve for θ:

$$x_j + \theta \Delta x_j > 0$$

$$\theta \Delta x_j > -x_j$$

$$\theta < \frac{-x_j}{\Delta x_j}$$

We need this inequality to hold for w, y, and z also. So we set θ equal to the minimum as follows:

$$\theta = min \left\{ \frac{-x_j}{\Delta x_j}, \frac{-w_i}{\Delta w_i}, \frac{-y_i}{\Delta y_i}, \frac{-z_j}{\Delta z_j} \right\}$$

Although, in order to guarantee all these variables will be strictly positive, we want strict inequality meaning θ must be smaller than the minimum. In order to obtain this, we multiply by r which will be a value less than 1 but very close to 1. In this way, we will choose θ using the following equation:

$$\theta = r\ min \left\{ \frac{-x_j}{\Delta x_j}, \frac{-w_i}{\Delta w_i}, \frac{-y_i}{\Delta y_i}, \frac{-z_j}{\Delta z_j} \right\}$$

Once we have all these pieces in place, we can get our step direction, scale it by θ and move to our new point. We then repeat this process until we get sufficiently close to the optimal solution.

2.3.2 Interior Point Method for Semidefinite Programming

The ideas just presented are adapted to semidefinite programming in the paper of Christoph Helmberg, Franz Rendl, Robert Vanderbei, and Henry Wolkowicz, *An Interior Point Method for Semidefinite Programming* [1]. They propose an interior point method algorithm for optimization problems over semidefinite matrices. Their algorithm follows closely to the path following method with necessary adaptations to semidefinite programming. The algorithm is described in this section.

Helmberg et al. work with the following primal and dual semidefinite problems [1]:

(*Primal*) $Max \;\; Tr\,(CX)$

s.t. $a - A(x) = 0$

$b - B(x) \geq 0$

$X \succcurlyeq 0$

(*Dual*) $Min \;\; a^T y + b^T t$

s.t. $A^T(y) + B^T(t) - C \succcurlyeq 0$

$y \epsilon R^k, t \epsilon R^m_+$

We can apply our original example, introduced in the last section, to this notation as such:

If we denote $C = \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$ and $X = \begin{pmatrix} x_1 & 0 \\ 0 & x_2 \end{pmatrix}$,

the objective function of the primal can be written as $Tr\,(CX) = x_1 - 2x_2$

If we let $b = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$,

the inequality constraints can be written as $b - B(x) = \begin{pmatrix} -1 + x_2 \\ 2 - x_1 \end{pmatrix}$

which is greater than or equal to $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ since we need $x_2 \geq 1$and $x_1 \leq 2$.

So we can rewrite the primal in this format as:

$Max \;\; Tr\,(CX)$

s.t. $b - B(x) \geq 0$

$X \succcurlyeq 0$

Similar to the way we constructed the inequality constraints, we can construct equality constraints if they were included in the problem. This would result in adding a term $a - A(x) = 0$ (In our case, we do not have any equality constraints so we would not include this).

We can do the same for the dual:

$t = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, b = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$ making $b^T t = (-1 \quad 2)\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -y_1 + 2y_2$ ($a^T y$ does not appear in this case)

$C = \begin{pmatrix} 1 & 0 \\ 0 & -2 \end{pmatrix}$ so $B^T(t) - C = \begin{pmatrix} y_2 - 1 & 0 \\ 0 & -y_1 + 2 \end{pmatrix} \succcurlyeq 0$ since it is symmetric and the diagonal entries are greater than zero. This will give us the dual problem as:

$$Min \;\; a^T y + b^T t$$

s.t. $B^T(t) - C \succcurlyeq 0$

$y \epsilon R^k, t \epsilon R^m_+$

Similarly, if equality constraints were included in the system, we would have to take those into account when converting to the dual. This would result in a term being added in the objective function ($a^T y$) and a term being added in the inequality constraint ($A^T(y)$). Since we do not have any equality constraints, these are not included above.

This construction of the primal and dual system is the same used by Helmberg et al. They begin their algorithm by defining their system as stated previously and again here:

(*Primal*) $Max \;\; Tr\,(CX)$

s.t. $a - A(X) = 0$

$b - B(X) \geq 0$

$X \succcurlyeq 0$

(*Dual*) $Min \;\; a^T y + b^T t$

s.t. $A^T(y) + B^T(t) - C \succcurlyeq 0$

$y \epsilon R^k, t \epsilon R^m_+$

For their algorithm they require a matrix X that strictly satisfies the above inequalities ($i.e. \;\; b - B(X) \geq 0$) and is positive definite (that is it is symmetric and all of its eigenvalues are positive). Also they assume that the equality constraints on X are linearly independent (if not, we could simply row reduce the linear system).

At first, the linear operators, A and B, were defined for symmetric matrices. However, it is realized that they will have to be applied to non-symmetric matrices. This problem is resolved by writing a matrix, M, in terms of its symmetric and non-symmetric parts:

$$M = symmetric + skew = \frac{1}{2}(M + M^T) + \frac{1}{2}(M - M^T)$$

and then letting the skew, or non-symmetric, part map to 0. Observe:

$$\frac{1}{2}(M - M^T) = 0 \langle \rightarrow \rangle M = M^T$$

In this way, when we apply A or B to the non-symmetric matrix M, we have the definition that $A(M) = A(M^T)$ $(similarly\ B(M) = B(M^T))$.

Now taking the system with all adjustments above integrated, we rewrite it into equality form introducing Z as a slack variable. Applying the ideas of the barrier problem to the dual system we have:

$$Min\ \ a^T y + b^T t - \mu(\log detZ + e^T \log t)$$

$$s.t.\ \ A^T(y) + B^T(t) - C = Z$$

$$t \geq 0, Z \succcurlyeq 0$$

Now we formulate the Lagrangian of this barrier function:

$$L_\mu(X, y, t, Z) = \ a^T y + b^T t - \mu(\log detZ + e^T \log t) + \langle Z + C - A^T(y) - B^T(t), X \rangle$$

Before taking the gradient to obtain the first-order derivatives, we will introduce the adjoint identity presented by Helmberg et al.:

$$\langle A(X), y \rangle = \langle X, A^T(y) \rangle$$

Utilizing this property we can write out the last expression in the Lagrangian as:

$$\langle Z + C - A^T(y) - B^T(t), X \rangle = \langle Z, X \rangle + \langle C, X \rangle - \langle A(X), y \rangle - \langle B(X), t \rangle.$$

With these modifications, our Lagrangian is now:

$$L_\mu(X, y, t, Z) = \ a^T y + b^T t - \mu(\log detZ + e^T \log t) + \langle Z, X \rangle + \langle C, X \rangle - \langle A(X), y \rangle - \langle B(X), t \rangle.$$

Now, taking the partial derivatives with this adjustment above, we end up with the following first-order optimality conditions:

$$\nabla_X L_\mu = Z + C - A^T(y) - B^T(t) = 0$$

$$\nabla_y L_\mu = a - A(X) = 0$$

$$\nabla_t L_\mu = b - B(X) - \mu t^{-1} = 0$$

$$\nabla_Z L_\mu = X - \mu Z^{-1} = 0$$

Using the last two equations, we can derive an equation for the µ value associated with a given point on the central trajectory, similar to what we did previously in the path following method. Our goal is to extend these properties in a reasonable way to get a μ value for solutions not on the central path. We start by solving the first and second equations for µ:

$$b - B(X) = \mu t^{-1} \rightarrow \textit{divide through by } t^{-1} \rightarrow \mu = t^T(b - B(X))$$

$$X = \mu Z^{-1} \rightarrow \textit{divide through by } Z^{-1} \rightarrow \mu = tr(ZX)$$

We average the two equations by adding the equations together and dividing by the sum of the corresponding lengths to arrive at:

$$\mu = \frac{tr(ZX) + t^T(b - B(X))}{n + m}$$

The algorithm being presented by Helmberg et al. proceeds from here in a fashion similar to that of the point-following method. It takes a point (X, y, t, Z) where $X, Z \succ 0,\ t \geq 0,\ \textit{and } b - B(x) > 0$. Then using the above equation for µ, estimates the current value for µ and divides it by 2. Dividing by 2 will guarantee that the new µ will be less than the current value of µ. The algorithm next computes directions $(\Delta X, \Delta y, \Delta t, \Delta Z)$ such that when applied to the original point, the new point, $(X + \Delta X, y + \Delta y, t + \Delta t, Z + \Delta Z)$, is on the central path at the determined µ.

Remember from the discussion of the path following method, the way to find the new point was to solve the first-order optimality conditions for the step direction. Again, not all the first-order optimality conditions are linear so we are unable to solve the system directly. We must rewrite the last two equations (which are the two that are nonlinear) by applying a linearization. To do this, we find the linear approximations to our nonlinear functions and utilize those to solve our system. Many choices could be made for our linearization. Helmberg et al. list these seven possibilities [1]:

$$\mu I - Z^{1/2} X Z^{1/2} = 0$$

$$\mu I - X^{1/2} Z X^{1/2} = 0$$

$$\mu Z^{-1} - X = 0$$

$$\mu X^{-1} - Z = 0$$

$$ZX - \mu I = 0$$

$$XZ - \mu I = 0$$

$$ZX + XZ - 2\mu I = 0$$

The first two, in semidefinite programming, involve matrix square roots making them not very good choices computationally. The third doesn't contain any information about the primal variables and the fourth doesn't contain any information about the dual variables so these two lead to poor results also. The next two result in the same step and contain equal information on the primal and dual variables. The last one has the same two properties just described for the previous two, and also preserves symmetry. Helmberg et al. choose to use the fifth option, $ZX - \mu I = 0$, which we will follow [1].

Applying this to the first-order optimality conditions we get:

$$\nabla_X L_\mu = Z + C - A^T(y) - B^T(t) = 0$$

$$\nabla_y L_\mu = a - A(X) = 0$$

$$\nabla_t L_\mu = t \circ (b - B(X)) - \mu e = 0$$

$$\nabla_Z L_\mu = ZX - \mu I = 0$$

We can rewrite these equations in a more compact form by defining the functions $F_\mu(s)$, F_d, F_p, F_{tB} and F_{ZX} as follows:

$$F_\mu(s) = F_\mu(X, y, t, Z) = \begin{pmatrix} Z + C - A^T(y) - B^T(t) \\ a - A(X) \\ t \circ \big(b - B(X)\big) - \mu e \\ ZX - \mu I \end{pmatrix} = \begin{pmatrix} F_d \\ F_p \\ F_{tB} \\ F_{ZX} \end{pmatrix}$$

Now, with this above formulization, the algorithm defines s^* as the solution to $F_\mu(s) = 0$ which satisfies the Karush-Kuhn-Tucker conditions (first-order optimality conditions) above, and is the optimal solution to the barrier problem. We want to step towards s^* by a length $\Delta s = (\Delta X, \Delta y, \Delta t, \Delta Z)$. If our new step is optimal, the above equations should hold for our new point. That is:

$$(Z + \Delta Z) + C - A^T(y + \Delta y) - B^T(t + \Delta t) = 0$$

$$a - A(X + \Delta X) = 0$$

$$(t + \Delta t) \circ \big(b - B(X + \Delta X)\big) - \mu e = 0$$

$$(Z + \Delta Z)(X + \Delta X) - \mu I = 0$$

Since we know our original point, and we want to find this optimal step, we put all unknowns on one side and the knowns on the other like we did for the point-following method. In order to do this, we first apply the property that $A(d + e) = A(d) + A(e)$. We obtain:

$$(Z + \Delta Z) + C - A^T(y) - A^T(\Delta y) - B^T(t) - B^T(\Delta t) = 0$$

$$a - A(X) - A(\Delta X) = 0$$

$$-t \circ B(\Delta X) - \Delta t \circ B(\Delta X) + t \circ b + \Delta t \circ b - t \circ B(X) - \Delta t \circ B(X) - \mu e = 0$$

$$ZX + Z\Delta X + \Delta Z X + \Delta Z \Delta X - \mu I = 0$$

We replace the last entry by $ZX + Z\Delta X + \Delta ZX - \mu I$ by using the ideas of Taylor's approximation: As ΔX and ΔZ become negligible, their product goes to zero even faster. We can do similarly for the third equation by getting rid of the $\Delta t \circ B(\Delta X)$ term.

Now by putting the knowns and unknowns onto separate sides we have:

$$\Delta Z - A^T(\Delta y) - B^T(\Delta t) = -Z - C + A^T(y) + B^T(t)$$

$$-A(\Delta X) = -a + A(X)$$

$$-t \circ B(\Delta X) + \Delta t \circ b - \Delta t \circ B(\Delta X) = -t \circ b + t \circ B(X) + \mu e$$

$$Z\Delta X + \Delta ZX = -ZX + \mu I$$

Simplified this is:

$$\Delta Z - A^T(\Delta y) - B^T(\Delta t) = -Z - C + A^T(y) + B^T(t)$$

$$-A(\Delta X) = -a + A(X)$$

$$-t \circ B(\Delta X) + \Delta t \circ \big(b - B(X)\big) = -t \circ (b - B(X)) + \mu e$$

$$Z\Delta X + \Delta ZX = -ZX + \mu I$$

Notice from the definitions above:

$$-Z - C + A^T(y) + B^T(t) = -F_d$$

$$-a + A(X) = -F_p$$

$$-t \circ \big(b - B(X)\big) + \mu e = -F_{tB}$$

$$-ZX + \mu I = -F_{ZX}$$

We can rewrite these equations in the form of a matrix to show the system equals $-F_\mu$:

$$\begin{pmatrix} \Delta Z - A^T(\Delta y) - B^T(\Delta t) \\ -A(\Delta X) \\ -t \circ B(\Delta X) + \Delta t \circ (b - B(X)) \\ Z\Delta X + \Delta ZX \end{pmatrix} = -\begin{pmatrix} F_d \\ F_p \\ F_{tB} \\ F_{ZX} \end{pmatrix} = -F_\mu$$

Notice that this is simply the equation for Newton's Method. We arrived at this same conclusion previously for a general case when trying to understand how to find a step direction. In order to get a step direction $\Delta s = (\Delta X, \Delta y, \Delta t, \Delta Z)$ towards s^*, we solve the equation

$F_\mu + \nabla F_\mu(\Delta s) = 0 \rightarrow \nabla F_\mu(\Delta s) = -F_\mu$ for Δs.

So now that we have these equations for the step direction set up, the next step is to actually obtain it. We can begin by solving the first equation for ΔZ. We will arrive at:

$$\Delta Z = -F_d + A^T(\Delta y) + B^T(\Delta t)$$

Now that we have an expression for ΔZ we can plug this into the last equation to obtain an expression for ΔX:

$$Z\Delta X + \left(-F_d + A^T(\Delta y) + B^T(\Delta t)\right)X = -F_{ZX}$$

Expanding, we obtain:

$$Z\Delta X - F_d X + A^T(\Delta y)X + B^T(\Delta t)X = -F_{ZX}$$

and bringing everything without a ΔX to the other side:

$$Z\Delta X = -F_{ZX} + F_d X - A^T(\Delta y)X - B^T(\Delta t)X$$

Multiply through by Z^{-1}; since Z is a diagonal matrix with strictly positive entries on the diagonal, we may write:

$$\Delta X = -Z^{-1}F_{ZX} + Z^{-1}F_d X - Z^{-1}A^T(\Delta y)X - Z^{-1}B^T(\Delta t)X.$$

Simplifying and grouping terms together:

$$\Delta X = -Z^{-1}F_{ZX} + Z^{-1}F_d X - Z^{-1}(A^T(\Delta y) + B^T(\Delta t))X$$

From the original equations, we had that $ZX - \mu I = F_{ZX}$. If we multiply this through by Z^{-1} we obtain $X - \mu I Z^{-1} = Z^{-1}F_{ZX}$. By plugging this in to our above equation for ΔX, we arrive at:

$$\Delta X = \mu Z^{-1} - X + Z^{-1}F_d X - Z^{-1}(A^T(\Delta y) + B^T(\Delta t))X$$

This is our equation for ΔX.

Now if we plug this into the second first-order optimality condition, we can get an expression for Δy and Δt:

$$-A\left(\mu Z^{-1} - X + Z^{-1}F_d X - Z^{-1}\left(A^T(\Delta y) + B^T(\Delta t)\right)X\right) = -F_p$$

Expanding this and dropping the negative:

$$A(Z^{-1}\mu) - A(X) + A(Z^{-1}F_d X) - A(Z^{-1}A^T(\Delta y)X) - A(Z^{-1}B^T(\Delta t)X) = F_p$$

Remember that $-F_p = -a + A(X)$ so if we apply this we get:

$$A(Z^{-1}\mu) - A(X) + A(Z^{-1}F_d X) - A(Z^{-1}A^T(\Delta y)X) - A(Z^{-1}B^T(\Delta t)X) = a - A(X)$$

The two $A(X)$ expressions cancel and by rearranging terms we arrive at:

$$A(Z^{-1}\mu) + A(Z^{-1}F_d X) - a = A(Z^{-1}A^T(\Delta y)X) + A(Z^{-1}B^T(\Delta t)X)$$

At this point, the authors introduce two linear operators, O_{11} and O_{12}, and a vector v_1, defined as follows:

$$O_{11}(\cdot) = A(Z^{-1}A^T(\cdot)X)$$

$$O_{12}(\cdot) = A(Z^{-1}B^T(\cdot)X)$$

$$v_1 = A(Z^{-1}\mu) + A(Z^{-1}F_dX) - a$$

This is done so that the expression for Δy and Δt can be written more compactly. By applying the above definitions to our equation we obtain this expression for Δy and Δt:

$$O_{11}(\Delta y) + O_{12}(\Delta t) = v_1$$

We can also substitute our expression for ΔX into our third equation for the step direction to obtain another expression for Δy and Δt. In this case, two more operators, O_{21} and O_{22}, and another vector, v_2, are introduced. They are defined as:

$$O_{21}(\cdot) = B(Z^{-1}A^T(\cdot)X)$$

$$O_{22}(\cdot) = \big(b - B(X)\big) \circ t^{-1} \circ (\cdot) + B(Z^{-1}B^T(\cdot)X)$$

$$v_2 = \mu t^{-1} - b + \mu B(Z^{-1}) + B(Z^{-1}F_dX)$$

and allow us to write this expression for Δy and Δt as:

$$O_{21}(\Delta y) + O_{22}(\Delta t) = v_2$$

Now that we have all the equations for our steps, we can begin to solve them. Notice that the expressions for Δy and Δt are all written in terms of things we know. So we first solve these for Δy and Δt. Then once we have those values, we can plug them into our equation for ΔZ. Finally, once we have the value for ΔZ, we can put all of these values into our expression for ΔX and solve. At this point, we finally have obtained our step direction.

We note, however, that when we solve for ΔX we will not necessarily obtain a symmetric matrix. For this reason, we will only use its symmetric part. We do this by remembering that a matrix can be written in terms of its symmetric part plus its skew part like below:

$$M = symmetric + skew = \frac{1}{2}(M + M^T) + \frac{1}{2}(M - M^T)$$

So by applying this to ΔX we have:

$$\Delta X = \frac{1}{2}(\Delta X + \Delta X^T) + \frac{1}{2}(\Delta X - \Delta X^T)$$

To use just the symmetric part, we say that $\Delta\tilde{X} = \frac{1}{2}(\Delta X + \Delta X^T)$ with $\Delta\tilde{X}$ just denoting the new value we will have for our step direction.

We now have our step direction $\Delta s = (\Delta X, \Delta y, \Delta t, \Delta Z)$, but as previously noted in our discussion of the path following method, we cannot automatically use this to step to our new point because we may violate some constraints. In this case, we may violate the condition that t and $b - B(X)$ has to be nonnegative and that X and Z have to be positive definite. So we need to find a step length parameter such that these conditions hold. Helmberg et al. do this by implementing a line search to get the parameters α_p and α_d [1]. This process finds the minimum value of the objective function in one dimension [4].

Once we have the step length parameters, we can step to our new point which will be:

$$X + \alpha_p \Delta X,$$

$$y + \alpha_d \Delta y,$$

$$t + \alpha_d \Delta t,$$

$$Z + \alpha_d \Delta Z$$

This process repeats until we obtain a point that satisfies the primal and dual feasibility and that gives a sufficiently small duality gap. Until then, we update µ according to our new point and continue the process to obtain our step directions and step length parameters.

Returning to our expressions we derived for Δy and Δt, it can be shown that these equations can be condensed into a system that is positive definite. Helmberg et al. present this and we will explore it here.

Recall the definition of an adjoint linear operator:

$$\langle A(X), y \rangle = \langle X, A^T(y) \rangle$$

Now we can see that O_{11} and O_{22} are self-adjoint and that O_{21} is the adjoint of O_{12}. Helmberg et al. go on further to state that our two new equations for Δy and Δt become a system of symmetric linear equations and this system is positive definite [1]. We can show this by first defining a new operator. We will call it O and is defined as follows:

$$O(X) = \begin{pmatrix} A(X) \\ B(X) \end{pmatrix}$$

Now using the adjoint definition from above, we can figure out what the adjoint of O is.

$$\left\langle O(X), \begin{pmatrix} y \\ t \end{pmatrix} \right\rangle = \left\langle \begin{pmatrix} A(X) \\ B(X) \end{pmatrix}, \begin{pmatrix} y \\ t \end{pmatrix} \right\rangle = \langle X, A^T(y) + B^T(t) \rangle$$

So the adjoint is $A^T(y) + B^T(t)$.

We first want to show that O is symmetric. For linear operators this is done by showing that the operator is equal to its adjoint. So we want to show that

$$\left\langle O(X), \binom{y}{t} \right\rangle = \left\langle X, O^T \binom{y}{t} \right\rangle = \left\langle X, O \binom{y}{t} \right\rangle$$

$$\left\langle O(X), \binom{y}{t} \right\rangle = \left\langle \binom{A(X)}{B(X)}, \binom{y}{t} \right\rangle = trace\left(\binom{A(X)}{B(X)}, \binom{y}{t}^T \right) = A(X)y + B(X)t.$$

Since we already know that $\langle A(X), y \rangle = \langle X, A(y) \rangle$ and similarly for B we have

$$A(X)y + B(X)t = XA(y) + XB(t) = \left\langle X, O\binom{y}{t} \right\rangle.$$

So we have that $\left\langle O(X), \binom{y}{t} \right\rangle = \left\langle X, O\binom{y}{t} \right\rangle$. This means that $O = O^T$ and therefore O is symmetric.

We now want to rewrite the system in terms of our new operator O. We start with

$$O_{11}(\Delta y) + O_{12}(\Delta t) = v_1,$$

$$O_{21}(\Delta y) + O_{22}(\Delta t) = v_2.$$

Writing this all out, by the definition of O_{ij} , we have

$$A(Z^{-1}A^T(\Delta y)X) + A(Z^{-1}B^T(\Delta t)X) = v_1$$

$$B(Z^{-1}A^T(\Delta y)X) + B(Z^{-1}B^T(\Delta t)X) + \big(b - B(X)\big) \circ t^{-1} \circ (\Delta t) = v_2$$

We combine these in matrix form as:

$$\begin{pmatrix} A(Z^{-1}A^T(\Delta y)X) + A(Z^{-1}B^T(\Delta t)X) \\ B(Z^{-1}A^T(\Delta y)X) + B(Z^{-1}B^T(\Delta t)X) \end{pmatrix} + \begin{pmatrix} 0 \\ \big(b - B(X)\big) \circ t^{-1} \circ (\Delta t) \end{pmatrix} = \binom{v_1}{v_2}.$$

Simplifying this we arrive at

$$\begin{pmatrix} A(Z^{-1}A^T(\Delta y)X + Z^{-1}B^T(\Delta t)X) \\ B(Z^{-1}A^T(\Delta y)X + Z^{-1}B^T(\Delta t)X) \end{pmatrix} + \begin{pmatrix} 0 \\ \big(b - B(X)\big) \circ t^{-1} \circ (\Delta t) \end{pmatrix} = \binom{v_1}{v_2}.$$

By the definition of O, we can see that the first matrix is nothing more than

$$O(Z^{-1}A^T(\Delta y)X + Z^{-1}B^T(\Delta t)X).$$

We can also notice that

$$Z^{-1}A^T(\Delta y)X + Z^{-1}B^T(\Delta t)X = Z^{-1}\big(A^T(\Delta y) + B^T(\Delta t)\big)X$$

By the definition of the adjoint of O, we see that this term is nothing more than $Z^{-1}O^T\binom{\Delta y}{\Delta t}X$.

So we have:

$$O\left(Z^{-1}O^T\binom{\Delta y}{\Delta t}X\right) + \begin{pmatrix} 0 \\ \big(b - B(X)\big) \circ t^{-1} \circ (\Delta t) \end{pmatrix} = \binom{v_1}{v_2}$$

which represents our original system.

We now want to sketch the proof of the statement of Helmberg et al. that this new system is positive definite [1]. In order to do this, we need to show that the system is symmetric and that all its diagonal entries are positive. We have already shown that O is symmetric and we know that $\left(b - B(X)\right) \circ t^{-1} \circ (\Delta t) > 0$. This is because we have the conditions that t and $b - B(X)$ have to be positive. So we just need that $\left\langle O\left(Z^{-1} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X\right), \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} \right\rangle > 0$.

$$\left\langle O\left(Z^{-1} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X\right), \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} \right\rangle = \left\langle Z^{-1} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X, O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} \right\rangle$$

$$= trace\left(Z^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X^{\frac{1}{2}} X^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} Z^{\frac{1}{2}}\right) = \left\langle Z^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X^{\frac{1}{2}}, Z^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X^{\frac{1}{2}} \right\rangle$$

Since, by our constraints, $Z \succ 0, X \succ 0$ we know that these terms will not contribute anything negative to the terms above. Also, since $O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} = A^T(y) + B^T(t)$ and we know, from our constraints, that $A^T(y) + B^T(t) > C$ and C is positive, then $O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix}$ will be positive. This means that $\left\langle O\left(Z^{-1} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X\right), \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} \right\rangle = \left\langle Z^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X^{\frac{1}{2}}, Z^{\frac{1}{2}} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X^{\frac{1}{2}} \right\rangle > 0$ and that our diagonal entries are positive.

So in conclusion we can say that our new system

$$O\left(Z^{-1} O^T \begin{pmatrix} \Delta y \\ \Delta t \end{pmatrix} X\right) + \begin{pmatrix} 0 \\ \left(b - B(X)\right) \circ t^{-1} \circ (\Delta t) \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

has a positive definite coefficient matrix.

2.3.3 Descent Direction

Even though we have found a good method for obtaining a direction that will lead to the optimal solution, it is important to know how good this direction actually is. Helmberg et al. prove that it forms a descent direction [1]. They do this by utilizing a well defined merit function. This merit function determines the progress of the algorithm we described above. The function they define is:

$$\begin{aligned} f_\mu(X, y, t, Z) = \langle Z, X \rangle - \mu \log \det(XZ) + t^T\left(b - B(X)\right) \\ - \mu e^T \log\left(t \circ \left(b - B(X)\right)\right) + \frac{1}{2}\left\|F_p\right\|^2 + \frac{1}{2}\left\|F_d\right\|^2 \end{aligned}$$

This function is the difference between the objective functions of the dual and primal barrier functions if you have a feasible point. So it is convex over the set of feasible points. It can be shown that this function is bounded below by $(n + m)\mu(1 - \log \mu)$ because the minimum of

$(x - \mu \log x)$, when $x > 0$, occurs when $x = \mu$. Furthermore, $F_\mu(s) = 0$ if and only if

$f_\mu = (n+m)\mu(1-\log\mu)$. Also, by the following lemma (the technical proof of which we omit), we see that Δs is actually a descent direction for f_μ.

Lemma: The directional derivative of f_μ in the direction of Δs satisfies $\langle \nabla_s f_\mu, \Delta s\rangle \leq 0$ with equality holding if and only if $F_\mu(s) = 0$.

With the above realizations and this lemma, we can conclude that an inaccurate line search in terms of the merit function and starting with a feasible point is enough to assure convergence. When starting with a randomly chosen infeasible point, however, this may not hold.

3 Survey of Applications

Semidefinite programming has many applications, and, through use of the interior point method just described, its algorithmic efficiency intrigues many experts. It has proven to be very important in trying to approximate solutions to some NP-complete problems and also very applicable to real world situations. In this section I will begin by surveying some of the most interesting and famous applications of semidefinite programming. I will then focus the rest of the section on one application, the sensor network localization problem.

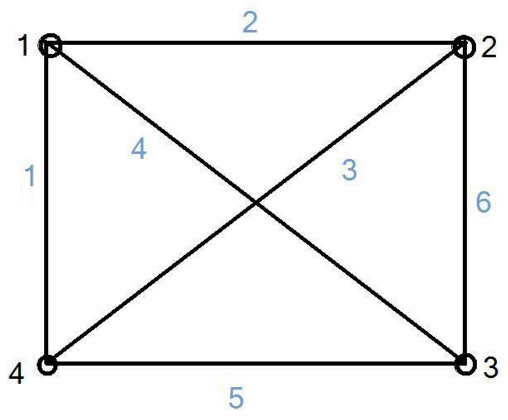
In this section, I have noted five famous and exciting applications of semidefinite programming. The first two deal with NP-Complete problems and obtaining an approximation to the actual solution. The third involves coding theory. The last two are kept brief. While they do have important results, adequate time was not available to study them as fully.

3.1 Max-Cut

The first application I will discuss provided a very surprising and exciting result, and because of this, I have decided a thorough explanation is owed. This result was the product of the work of Goemans and Williamson in which they applied semidefinite programming theory to the Max-Cut problem. Through investigation of the problem with semidefinite programming they were able to develop a method for obtaining an answer that is $.878$ of the actual solution [5]. This was pretty amazing since the problem is known to be NP-Complete [6].

The Max-Cut problem is described as follows, utilizing an example.

Given the following graph, we want to find an edge cut of maximum total weight. A **cut** in a graph with vertex set V is a partition $(S:\bar{S})$ of V where $S \cup \bar{S} = V$ and $S \cap \bar{S} = \emptyset$. We say an edge straddles a cut if it has one end in S and the other in $\bar{S}$. If the edge of the graph has a specified weight, then the weight, $wt(S:\bar{S})$, of cut $(S:\bar{S})$ is defined as the sum of the weights of all edges having one end in S and one edge in $\bar{S}$. Let us consider the graph below for an example. If we made a cut down the middle (from top to bottom) of this graph, the cut would have weight 14.



To attack the Max-Cut, we construct the adjacency matrix of the graph. The entries are the weights between each pair of vertices. We will use the above graph for an example as we try to formulate this matrix. In the adjacency matrix, each row and corresponding column represents a vertex. For example, row 1 represents vertex 1 and so does column 1. The (i, j) entry is the weight of the edge between the corresponding two vertices, and if no such edge exists, we set it equal to zero. So the $(1,2)$ entry in the matrix would be the weight between vertex 1 and 2. For this graph, the adjacency matrix is:

$$A = \begin{bmatrix} 0 & 2 & 4 & 1 \\ 2 & 0 & 6 & 3 \\ 4 & 6 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

with the weights between two of the same vertices being 0.

We also construct a diagonal matrix D whose ith diagonal entry is defined as the sum of all weights on edges incident to vertex i. For example, in the case of the above graph, we have

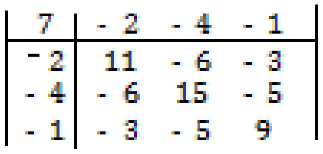
$$D = \begin{bmatrix} 7 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 \\ 0 & 0 & 15 & 0 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$

Next we obtain the Laplacian: $L = D - A$. For our example this is:

$$L = \begin{vmatrix} 7 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 \\ 0 & 0 & 15 & 0 \\ 0 & 0 & 0 & 9 \end{vmatrix} - \begin{vmatrix} 0 & 2 & 4 & 1 \\ 2 & 0 & 6 & 3 \\ 4 & 6 & 0 & 5 \\ 1 & 3 & 5 & 0 \end{vmatrix} = \begin{vmatrix} 7 & -2 & -4 & -1 \\ -2 & 11 & -6 & -3 \\ -4 & -6 & 15 & -5 \\ -1 & -3 & -5 & 9 \end{vmatrix}$$

We are going to associate, to each cut, a matrix of all $1s$ and $-1s$. We let S represent the set of the vertices which are included in the cut. $\bar{S}$ will be the set of all the vertices not included in the cut. We then can create a vector X s.t. $x_i = \begin{cases} 1, & i \in S \\ -1, & i \in \bar{S} \end{cases}$. So for example if we choose a cut in the above graph such that it included vertices 1 and 2 then our X vector would be: $X = \begin{vmatrix} 1 \\ 1 \\ -1 \\ -1 \end{vmatrix}$ with $S = \{1, 2\}$ and $\bar{S} = \{3, 4\}$.

Notice that if we choose a cut with $|s| = 1$, our Laplacian can is partitioned in a particular way. For example if $S = 1$, in the graph above:



$$\left|\begin{array}{c|ccc} 7 & -2 & -4 & -1 \\ \hline -2 & 11 & -6 & -3 \\ -4 & -6 & 15 & -5 \\ -1 & -3 & -5 & 9 \end{array}\right|$$

The value of the cut is 7, since the edges straddling the cut have weights $-2, -4$, and -1. The rest of the matrix represents the Laplacian for the inverse of the cut, that is, the rest of the graph once the cut is made at vertex 1.

So for any cut, we can partition the Laplacian into four parts, $L_S, C, C^T, L_{\bar{S}}$ with C representing the cut edges. So our Laplacian will become a block matrix represented as:

$$\begin{vmatrix} L_S & C \\ C^T & L_{\bar{S}} \end{vmatrix}$$

Notice that $e^T L_S e = wt(S\colon \bar{S})$ with e being the all ones vector. Also, along the same lines

$e^T Ce = -wt(S\colon \bar{S})$ and $e^T L_{\bar{S}} e = wt(S\colon \bar{S})$.

Since X is a block matrix made up of the all one vectors, if we apply these properties we can get a formula including the weight of the cut:

$$X^T LX = |e^T \quad -e| \begin{vmatrix} L_S & C \\ C^T & L_{\bar{S}} \end{vmatrix} \begin{vmatrix} e^T \\ -e \end{vmatrix} = |(e^T L_S - eC^T) \quad (e^T C - eL_{\bar{S}})| \begin{vmatrix} e^T \\ -e \end{vmatrix} =$$
$$= e^T L_S e - e^T C^T e \ - e^T Ce + e^T L_{\bar{S}} e = 4wt(S\colon \bar{S})$$

We can notice that this also equals $trace(X^T LX)$. Now we can implement the property that $trace(MN) = trace(NM)$ where M, N are two square matrices allowing for

$$trace(X^T LX) = trace(LX^T X) = \langle L, X^T X \rangle$$

Remember that the goal of this problem is to find the maximum cut we can make. We can now write down this problem formally:

$$Max \ \langle L, X^T X \rangle$$

$$s.t. X \in \{1, -1\}^n$$

However, this problem is NP-Complete.

It can be relaxed into an LP problem as follows:

$$Max \ \frac{1}{4} \langle L, Y \rangle$$

$$s.t. -1 \le x_{ij} \le 1$$

but this does not yield a more impressive result.

On the other hand, with $Y = X^T X$, we have an SDP relaxation:

$$Max\frac{1}{4}\langle L, Y\rangle$$

$$s.t. x_{ii} = 1\ \forall i$$

$$Y \succcurlyeq 0$$

This relaxation can give a pretty close approximation to the optimal solution. Goemans and Williamson realized that the solution that results from this relaxation is $.878$ of the MAXCUT [5]. They did this as follows in their paper, *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems using Semidefinite Programming* [5].

Take the optimal solution Y and let $m = rankY$, we know that $\langle L, Y\rangle \geq 4maxcut$. Since $x_{ii} = 1\ \forall i$, we know that all the points are on the unit sphere in $\mathbb{R}^m$. If we let U equal a matrix of all the unit vectors centered at the center of the unit sphere then $y_{ij} = u_i \cdot u_j$ and $Y = UU^T$.

Now comes the clever part. We now take a random vector r that splits the sphere into two hemispheres and define $S = \{i: u_i \cdot r > 0\}$ and $\bar{S} = \{i: u_i \cdot r < 0\}$. We then calculate, for this r, the probability that $u_i \cdot r > 0$ and $u_j \cdot r < 0$. This can be written as:

$$Prob\big[sgn(r \cdot u_i) \neq sgn(r \cdot u_j)\big]$$

Fortunately, the m-dimensional computation reduces to a computation in 2-D. This refers to the probability that a random plane separates the two vectors. This probability is proportional to the angle between them [5]. So if we know that angle, we can solve for this probability. Looking at the unit circle, we find:

$$Prob\big[sgn(r \cdot u_i) \neq sgn(r \cdot u_j)\big] = \frac{1}{\pi}\arccos(u_i \cdot u_j)$$

If we let $E(w)$ be the expected value of the max cut, we obtain:

$$E(w) = \frac{1}{\pi}\sum_{i<j} w_{ij} \arccos(u_i \cdot u_j)$$

Since $-1 \leq u_i \cdot u_j \leq 1$, we can employ first-year calculus to see that $\frac{1}{\pi}\arccos(u_i \cdot u_j) \geq \frac{1}{2}\alpha(1 - u_i \cdot u_j)$ where

$\alpha = \min\frac{2}{\pi}(\frac{\theta}{1-cos\theta})$. We can incorporate this bound to estimate the expected value:

$$E(w) \geq \frac{1}{2}\alpha\sum_{i<j} w_{ij}(1 - u_i \cdot u_j)$$

We can estimate this further by realizing that if $\alpha = \min\frac{2}{\pi}(\frac{\theta}{1-cos\theta})$ then $\alpha > 2/\pi\sin(\theta)$ which is greater than $.87856$. We know that the expected value will be greater than the optimal value so we can say that $E(w) \geq \alpha(optimal\ value) \geq .878(optimal\ value)$. This means that this SDP relaxation can get us to 87.8% of the optimal value [5]. This is an amazing result for a NP-Complete problem.

3.2 Max-2-SAT

The Max-2-SAT problem is another NP-Complete problem [7] that has had semidefinite programming applied to obtain a better approximation for solution. While it is based on a different idea, it can be relaxed into a Max-Cut problem [5] allowing us to find a solution that is also $.878$ of the optimal solution.

The Max-2-SAT problem is, given a string of clauses that contain two Boolean variables, what is the maximum number of clauses that can be satisfied? Mathematically, an example of this problem is below. In this instance, $\bar{x}$ refers to the negation of x, $\wedge$ refers to *"and"*, and $\vee$ refers to *"or"*.

$$Given\ (x_1, x_2, x_3, x_4), how\ many\ clauses\ can\ be\ satisfied\ in\ the\ following:$$

$$(x_1 \vee \bar{x}_3) \wedge (x_2 \vee x_4) \wedge (x_1 \vee \bar{x}_4) \wedge (x_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \wedge \bar{x}_2) \wedge (x_3 \vee x_4)$$

In general, the Max-2-SAT problem is NP-Complete. However, as stated previously, it can be generalized into a version of the Max-Cut problem. This is done by Goemans and Williamson in their paper, *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems using Semidefinite Programming* [5].

First the problem is represented by the integer quadratic system below:

$$\sum_{i<j} \left[a_{ij} \left(1 - y_i y_j\right) + b_{ij} \left(1 + y_i y_j\right) \right]$$

$$subject\ to\ y_i \in \{-1, 1\} \quad \forall i \in V$$

They then relax the system into the following SDP formula:

$$max \sum w_j\ v(C_j)$$

$$subject\ to\ y_i \in \{-1, 1\} \quad \forall i \in \{0, 1, \dots \dots, n\}$$

where $v(C_j)$ are linear combinations of $1 + y_i y_j$ and $1 - y_i y_j$.

One can see that this problem looks very similar to that of the Max-Cut problem. Using the same SDP approaches as for the Max-Cut, we can arrive at the conclusion that the solution will be $.878$ of the actual solution.

3.3 Schrijver Coding Theory

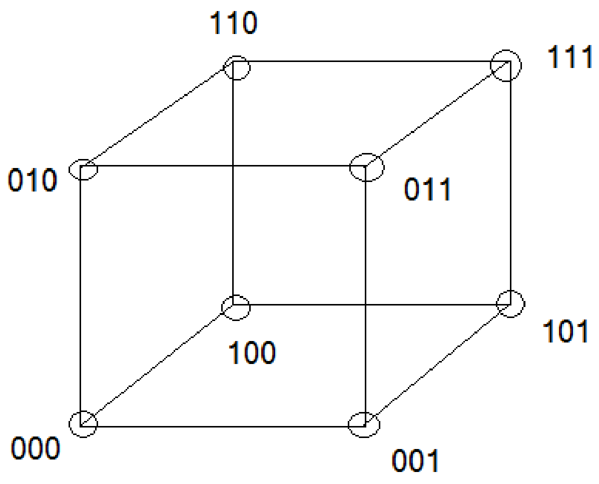
Semidefinite Programming has been used in coding theory also. Coding theory is extremely useful and needed in most aspects of our everyday lives. When you shop online, coding theory is used to distort your credit card number so that no one can steal that information. It is used to make CDs and DVDs and maybe most importantly, our cell phones.

The basic premise of coding theory is to alternate a given code into a different one that is much harder to read by an average person. So in the case of the credit cards, we change all the numbers so that they no longer mean anything to a normal person. They can also insert many new numbers to completely alter the sequence. The number of errors to transform a given code, x, into a new one, y, is defined as $d(x, y)$. More formally:

$$d(x, y) = \#\{i: x_i \neq y_i\}$$

For example if you have the code (000111) and you alter it to become (001001) the number of errors, or changes, you performed were three.

There are a few more ideas to introduce. We will use an example to help illustrate these. Consider this unit cube with points at the given coordinates.



Let our code be: $C = \{x, y, z\} = \{000, 001, 011\}$. We now can write down the profile of each point. We see that for any point, the farthest is can be away from another point is 3 positions. So for each point we want to know how many points in the system are $0, 1, 2$, and 3 positions away from it. So for x we have:

$$Profile\ of\ x: [\#0\ away, \#1\ away, \#2\ away, \#3\ away] = [1, 1, 1, 0]$$

You can do similarly for y and z:

$$Profile\ of\ y: [1, 2, 0, 0]$$

$$Profile\ of\ z: [1, 1, 1, 0]$$

The inner distribution is the average of these profiles. In our case, this will be $\left[1, \frac{4}{3}, \frac{2}{3}, 0\right]$.

In code theory, you want to try to reduce the amount of errors to make a code. If we can cut down the length of codes used in our products, everything would work much faster. In 1973, Philippe Delsarte found that the inner distribution is a feasible solution to a certain LP and this provided the best known general-purpose upper-bound on the efficiency of the size of a code [8]. However, there has been a push to move away from this LP and move towards something that provides a better outcome. While Delsarte found the best known upper-bound, it was still not a very good one. Alexander Schrijver, in 2005, counted triples of code words, not doubles like done previously, and realized the problem could be made into a SDP. This process is described in his paper, *New Code Upper Bounds from the Terwilliger Algebra* [9]. This led to the first advancement on the upper bound in approximately 30 years and while the new results aren't a huge improvement, they still have helped coding theory become simpler.

3.4 Other Applications

There are other very exciting applications of semidefinite programming. I will introduce two more here. First is eigenvalue optimization which is used in many applications of applied mathematics and engineering [10]. While there are many techniques used to solve such problems, usually depending on the specific application, Adrian Lewis developed a technique that uses semidefinite programming. The most notable result from this was that he was able to separate variables in differential equations [8]. Second, semidefinite Programming has been applied in the financial sector for its use with moment problems [11].

3.5 The Sensor Network Localization Problem

One interesting application to a real world situation is that of sensor networks. It is this application that the rest of the report focuses on. Sensor networks are a group of sensors that are set up to obtain information about the environment they are placed in. They are low costing, multipurpose tools that can monitor the environment around them providing information on such things as sound levels, seismic activity, and temperature [12]. While this information can be very valuable, it is only useful if we know the locations of the sensors. In most cases, we only know some of the positions. While GPS seems like the obvious solution to this, it can be very expensive and not always reliable since GPS is not accepted everywhere [12]. The best solution is to utilize the tools we developed in the previous chapter and apply semidefinite programming to sensor networks. In this way, we can estimate the node positions using distance measurements between sensors. Formally, the problem is stated as: Given the true positions of some of the nodes and pair-wise distances between some nodes, how can the positions of all of the nodes be estimated [12]? Many times, semidefinite programming is chosen to solve the problem because the approach is known to find the positions in polynomial time whenever a unique solution exists [12]. The problem of estimating the node locations in this way is called the Sensor Network Localization Problem and is described in Clayton W. Commander, Michelle A Ragle, and Yinyu Ye's paper, *Semidefinite Programming and the Senor Network Localization Problem, SNLP* [12]. Their methods, as presented in their paper, are explained below.

3.5.1 Constructing the Problem

To begin, we will introduce some terms. The sensors whose locations we know at the start will be referred to as anchors while those whose locations are unknown are called referred to as nodes. Our task is to, given the locations of m anchors, find the location of n nodes in the system based on distance measurements we can collect.

We will follow the notation of the SNLP paper and denote the anchor points by $a_1, a_2, \ldots\ldots, a_m \in \mathbb{R}^d$ and the node points by $x_1, x_2, \ldots\ldots, x_n \in \mathbb{R}^d$. We will let d_{kj} be the Euclidean distances between points a_k and x_j for some k, j and d_{ij} be the Euclidean distances between x_i and x_j for $i < j$. We will define the set $N_a = \{(k,j) : d_{kj} \ \mathit{is\ specified}\}$ which denotes the anchor/node pairs whose distance is known and the set $N_x = \{(i,j) : i < j, d_{ij} \ \mathit{is\ specified}\}$ which denotes the node/node pairs whose distance is known. Also, the Euclidean norm of a vector x, defined as $\sqrt{\langle x, x\rangle}$, will be denoted as $\|x\|$.

With this notation we can formally state the Sensor Network Localization Problem (SNLP). It is to find the estimated position of $x_1, x_2, \ldots\ldots, x_n \epsilon \mathbb{R}^d$ such that

$$\|a_k - x_j\|^2 = d_{kj}^2, \forall\ (k,j) \in N_a \text{ and}$$

$$\|x_i - x_j\|^2 = d_{ij}^2, \forall\ (i,j) \in N_x$$

This problem seems to be very simple when written out in this form. However, there are a lot of hard to answer questions that come up when dealing with this problem. Two of the most important are, given an instance of the SNLP, does it have a realization in the required dimensions and if so, is it unique? If the problem is in two dimensions, a unique realization can be determined. However, even if we know there is a unique realization, it is NP-complete to compute it. Because of this, a direct method cannot be taken and we need to resort to other measures [12]. For this reason, we need to employ other methods for solution.

3.5.2 Methods for Solving

While there are several methods we can use to solve the problem, the most appealing has become semidefinite programming. In order to apply semidefinite programming to the SNLP we need to apply a relaxation. The applied relaxation will make the quadratic distance constraints (which are non-convex) into linear constraints. This is done as follows in Commander et al's paper.

We will start with the second equation. Let $X = [x_1, x_2, \ldots\ldots, x_n]$ be the $d \times n$ matrix we are trying to find and $e_{ij} \in \mathbb{R}$ be a column vector of length n (the number of nodes in the network) such that all entries are zero except the $i\text{-}th$ position which is 1 and the $j\text{-}th$ position which is -1. For example, say we have a network with four anchors and three nodes and we have a distance measurement between node 1 and node 3.

Then our column vector e_{ij} will be $\begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}$.

We can now say, $\forall\ (i,j) \in N_x$,

$$\left\|x_i - x_j\right\|^2 = e_{ij}^T X^T X e_{ij}$$

Let's show this with an example:

Let $i = 1, j = 3, d = 1, n = 3$. So we have $e_{ij} = \begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}$, $X = |x_1 \quad x_2 \quad x_3|$

$$e_{ij}^T X^T X e_{ij} = |1 \quad 0 \quad -1| \begin{vmatrix} x_1 \\ x_2 \\ x_3 \end{vmatrix} |x_1 \quad x_2 \quad x_3| \begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix} = (x_1 - x_3)|x_1 \quad x_2 \quad x_3| \begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}$$

$$= (x_1 - x_3)(x_1 - x_3) = (x_1 - x_3)^2 = x_1^2 - 2x_1x_3 + x_3^2$$

Now the other side:

$$\|x_1 - x_3\|^2 = \|x_1\|^2 + \|x_3\|^2 - 2x_1x_3$$

x_1 and x_3 are vectors of d dimension and in this case $d = 1$, so $\|x_1\|^2 = {x_1}^2$ and $\|x_3\|^2 = {x_3}^2$

Therefore,

$$\|x_1 - x_3\|^2 = \|x_1\|^2 + \|x_3\|^2 - 2x_1x_3 = x_1^2 + x_3^2 - 2x_1x_3$$

which is what we had above, meaning

$$\left\|x_i - x_j\right\|^2 = e_{ij}^T X^T X e_{ij}.$$

We can use this same idea to rewrite the other constraint. We will again introduce some notation. Let $(a_k; e_j)$ be a column vector with length equal to $d + n$ (the dimension plus the number of nodes). It will be made by putting a_k "on top of" e_j where a_k is the coordinates of the $k\text{-}th$ anchor and e_j is a vector with -1 as the $j\text{-}th$ entry and all others entries being zero. For example, say we have the same network from above with four anchors and three nodes. We have a distance measurement for anchor 1 and node 2 with anchor 1 being at the position $(2, 5)$.

Then our $(a_k; e_j)$ vector will be $\begin{vmatrix} 2 \\ 5 \\ \cdots \\ 0 \\ -1 \\ 0 \end{vmatrix}$

We can say $\forall\ (k,j) \in N_a$,

$$\left\|a_k - x_j\right\|^2 = \left(a_k; e_j\right)^T [I_d; X]^T [I_d; X](a_k; e_j)$$

We can show this using similar ideas as above:

Let $d = 1, j = 3, k = 1$. So we have: $(a_1; e_3) = \begin{vmatrix} a_1 \\ 0 \\ 0 \\ -1 \end{vmatrix}$ and $[I_d; X] = |1 \quad x_1 \quad x_2 \quad x_3|$

$$\left(a_k; e_j\right)^T [I_d; X]^T [I_d; X]\left(a_k; e_j\right) = |a_1 \quad 0 \quad 0 \quad -1| \begin{vmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \end{vmatrix} |1 \quad x_1 \quad x_2 \quad x_3| \begin{vmatrix} a_1 \\ 0 \\ 0 \\ -1 \end{vmatrix} = (a_1 - x_3)(a_1 - x_3)$$

$$= (a_1 - x_3)^2 = {a_1}^2 - 2a_1x_3 + {x_3}^2$$

The other side:

$$\|a_1 - x_3\|^2 = \|a_1\|^2 + \|x_3\|^2 - 2a_1x_3 = {a_1}^2 + {x_3}^2 - 2a_1x_3$$

(again because a_1 and x_3 are vectors of d dimension which is 1, $\|a_1\|^2 = {a_1}^2, \|x_3\|^2 = {x_3}^2$)

which is what we had above, therefore

$$\left\|a_k - x_j\right\|^2 = \left(a_k; e_j\right)^T [I_d; X]^T [I_d; X](a_k; e_j)$$

We can use these two new identities to rewrite the problem, but first we will make a few more adjustments. We will define Y as $X^T X$. If we plug this into our new identities we will end up with the following:

$$\left\|x_i - x_j\right\|^2 = e_{ij}^T X^T X e_{ij} = e_{ij}^T Y e_{ij}$$

$$\left\|a_k - x_j\right\|^2 = \left(a_k; e_j\right)^T [I_d; X]^T [I_d; X]\left(a_k; e_j\right) = \left(a_k; e_j\right)^T \begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix} (a_k; e_j)$$

The second matrix, $\begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix}$, derives as follows:

$$[I_d; X]^T [I_d; X] = \begin{bmatrix} I_d^T \\ X^T \end{bmatrix} [I_d; X] = \begin{bmatrix} I_d^T I_d & I_d^T X \\ X^T I_d & X^T X \end{bmatrix} = \begin{bmatrix} I_d & X \\ X^T & Y \end{bmatrix}$$

We now present the problem:

$$Find\ X \in \mathbb{R}^{dxn}\ and\ Y \in \mathbb{R}^{nxn}$$

$$such\ that$$

$$e_{ij}^T Y e_{ij} = d_{ij}^2, \forall (i,j) \in N_x,$$

$$\left(a_k; e_j\right)^T \begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix} \left(a_k; e_j\right) = d_{kj}^2, \forall (k,j) \in N_a$$

$$Y = X^T X$$

We can make this into an SDP by applying a relaxation to the last constraint. We will say that $Y \succcurlyeq X^T X$ which means that $Y - X^T X$ is positive semidefinite. It has been shown that this positive semidefinite matrix can be written as the following [12]:

$$Z = \begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix} \succcurlyeq 0$$

At this point, we can define a principle submatrix of Z. This will be the dxd matrix $Z_{1:d,1:d}$. We can now compose the SDP as done by Commander et al, which is a relaxation of the original SNLP, as follows:

$$Find\ Z \in \mathbb{R}^{(d+n)\times(d+n)}\ to$$

$$maximize\ 0$$

$$subject\ to$$

$$Z_{1:d,1:d} = I_d,$$

$$\langle (0; e_{ij})(0; e_{ij})^T, Z\rangle = d_{ij}^2, \forall (i,j) \in N_x,$$

$$\langle (a_k; e_j)(a_k; e_j)^T, Z\rangle = d_{kj}^2, \forall (k,j) \in N_a$$

$$Z \succcurlyeq 0$$

These constraints might look different but they are exactly what we had previously, just reworked into a new formulation. This is proved below:

For the first constraint:

$$\langle (0; e_{ij})(0; e_{ij})^T, Z\rangle = \langle \begin{pmatrix} 0 \\ e_{ij} \end{pmatrix} (0; e_{ij}), Z\rangle = \langle \begin{pmatrix} 0 & 0 \\ 0 & e_{ij} e_{ij}{}^T \end{pmatrix}, \begin{pmatrix} I_d & X \\ X^T & X^T X \end{pmatrix}\rangle$$
$$= Tr\begin{pmatrix} 0 & 0 \\ e_{ij} e_{ij}{}^T X^T & e_{ij} e_{ij}{}^T X^T X \end{pmatrix} = e_{ij} e_{ij}{}^T X^T X = e_{ij}{}^T X^T X e_{ij} = e_{ij}{}^T Y e_{ij}$$

which is what we had previously for the first constraint.

For the second constraint:

$$\langle (a_k; e_j)(a_k; e_j)^T, Z\rangle = \langle \begin{pmatrix} a_k \\ e_j \end{pmatrix} (a_k; e_j), Z\rangle = \langle \begin{pmatrix} a_k a_k{}^T & a_k e_j^T \\ e_j a_k{}^T & e_j e_j^T \end{pmatrix}, \begin{pmatrix} I_d & X \\ X^T & X^T X \end{pmatrix}\rangle$$
$$= Tr\begin{pmatrix} I_d a_k a_k{}^T + a_k e_j^T X^T & a_k a_k{}^T X + a_k e_j^T X^T X \\ e_j a_k{}^T I_d + e_j e_j^T X^T & e_j a_k{}^T X + e_j e_j^T X^T X \end{pmatrix}$$
$$= I_d a_k a_k{}^T + a_k e_j^T X^T + e_j a_k{}^T X + e_j e_j^T X^T X = (a_k{}^T I_d^T + e_j^T X^T)(I_d a_k + X e_j)$$
$$= (a_k; e_j)^T [I_d; X]^T [I_d; X](a_k; e_j)$$

which is what we had for the second constraint previously.

The dual of this SDP is:

$$minimize\ \langle I_d, V\rangle + \sum_{(i,j)\in N_x} y_{ij}\, d_{ij}^2 + \sum_{(k,j)\in N_a} w_{kj}\, d_{kj}^2$$

$$subject\ to$$

$$\begin{pmatrix} V & 0 \\ 0 & 0 \end{pmatrix} + \sum_{(i,j)\in N_x} y_{ij}\left(0; e_{ij}\right)\left(0; e_{ij}\right)^T + \sum_{(k,j)\in N_a} w_{kj}\left(a_k; e_j\right)\left(a_k; e_j\right)^T \succcurlyeq 0$$

So and Ye, in their paper *Theory of Semidefinite Programming for Sensor Network Localization* [13], provide many results of this representation of the SNLP. One important result, describes a set of instances where this SDP relaxation is exact. This occurs when the feasible solution, Z, has rank d. The theorem is stated below:

Theorem: Let $\bar{Z}$ be a feasible solution for SDP and $\bar{U}$ be an optimal slack matrix of SDP-D. Then, by the duality theorem for semidefinite programming, it follows that:

1. $\langle \bar{Z}, \bar{U}\rangle = 0$
2. $rank(\bar{Z}) + rank(\bar{U}) \leq d + n$
3. $rank(\bar{Z}) \geq d\ and\ rank(\bar{U}) \leq n$

This means, if you have an optimal slack matrix for the dual with rank n, then the rank of Z will be d [12]. This is because of the above inequalities. If $rank(\bar{U}) = n$ then you have $rank(\bar{Z}) + n \leq d + n$. This can be simplified to $rank(\bar{Z}) \leq d$. But by the 3rd condition, we also have that $rank(\bar{Z}) \geq d.$ So $rank(\bar{Z})$ must equal d. The main result from this is that we can say the original SNLP is equal to the SDP relaxation meaning the SNLP can be solved in polynomial time.

So and Ye have another important conclusion that states there exists a big group of localizable graphs that can be solved efficiently [13]. This theorem is stated below:

Theorem: Suppose that the network in question is connected. Then the following are equivalent:

1. *Problem SNLP is uniquely localizable*
2. *The max-rank solution matrix of SDP has rand d*
3. *The solution matrix, Z, of SDP satisfies $Y = X^T X$*

This result further emphasizes that the SNLP can be solved in polynomial time by using the SDP relaxation we derived above. This is as long as the SNLP is uniquely localizable. This holds for the reverse too. This theorem also introduces the fact that there is a group of graphs for which was can figure out the localization even though the general SNLP is NP-Complete [12].

Knowing that the solution to the SDP will provide the optimal solution for the SNLP, we can utitlize the interior point method of Helmberg et al. to solve.

4 NEOS Guide

For this project, the SNLPs were solved using the interior point method described earlier. This was done using computer software because, while we have the algorithm for solving the SNLP, it is not very easy to do by hand. Like many optimization problems, we need the use of computers to arrive at our solution. The NEOS Solver is optimization software that can solve most any type of optimization problem. It has applications in semidefinite programming which we can utilize to solve our SNLP. We can obtain the NEOS Solver online where it functions as a type of cloud computing. This means that when we enter the data for our problem, it gets sent off to a university or company that owns the software. There the problem is solved, the solution is sent back to NEOS and NEOS provides it to us. This is usually done through email, however if the problem does not take long to solve, it will show the answer in the browser window. In an effort to utilize the NEOS software and analyze our solution, I studied the NEOS program to understand the formats and the ideas behind it. I also created MAPLE codes to aid us in our use of NEOS.

I will use a simple example in an effort to explain these programs and the process to follow to solve an SNLP. Our SNLP example consists of 5 anchors and 4 nodes. The positions of the anchors are $(0,0), (4,0), (3,6), (0,3)$ and $(6,3)$. We have 8 anchor-node measurements and 4 node-node measurements. We are trying to obtain the positions of the 4 nodes given the anchor positions and the measurements. We will continue to refer to this example to help illustrate the descriptions that follow.

4.1 NEOS Input and Output

While NEOS Solvers can make our work very easy, it requires specific input and provides solutions in a similarly specific output. The format of the input and output is dependent on the type of solver we are choosing to use. In our situation, with the SNLP, we employed the semidefinite programming solver for CSDP.

4.1.1 Input

I first looked at the NEOS input. The input is very specific and we can only formulate it once we have all the needed information. This means we must first understand what information we need and how to obtain it.

The input consists of six sections defined as follows [14]:

1. Comments: there can be as many lines of comments as the user wants however each line must begin with '"' or '*'. While comments can be used, it is often advised not to in order to avoid any confusion with the solver.
2. The number of constraint matrices
3. The number of blocks in the block diagonal matrices
4. List of numbers that give the size of the individual blocks. Note: Negative numbers are allowed and should be used for a block where the diagonals are the only nonzero entries.
5. The objective function vector

6. Entries of the constraint matrices with one entry on each line. Each line will have the format: matrix, block, row, column, value. Since all the matrices are assumed to be symmetric, only the upper triangle entries of the matrix are needed.

Now that we know what information is needed, we must understand how to obtain it. Let us first remember our SNLP problem. It is stated as an SDP in the following form:

$$maximize\ 0$$

$$subject\ to$$

$$Z_{1:d,1:d} = I_d,$$

$$\langle (0; e_{ij})(0; e_{ij})^T, Z\rangle = d_{ij}^2, \forall (i,j) \in N_x,$$

$$\langle (a_k; e_j)(a_k; e_j)^T, Z\rangle = d_{kj}^2, \forall (k,j) \in N_a$$

$$Z \succcurlyeq 0$$

We can rewrite the second and third constraints as:

$$e_{ij}^T Y e_{ij} = d_{ij}^2, \forall (i,j) \in N_x,$$

$$(a_k; e_j)^T \begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix} (a_k; e_j) = d_{kj}^2, \forall (k,j) \in N_a$$

by applying the definition of Z which is, $Z = \begin{pmatrix} I_d & X \\ X^T & Y \end{pmatrix}$. Remember that a_k is the position vector of the anchor k, e_j is the vector of length n of all zeros except at -1 in the jth position, and e_{ij} is the vector of length n of all zeros except 1 at the ith position and -1 at the jth position.

To begin we want to find the constraint matrices which are needed to provide information for the NEOS input lines $2, 3, 4$, and 6. The way to formulate these matrices is fairly simple but tedious. For the anchor-node constraints we must compute $(a_k; e_j)^T (a_k; e_j)$ for each distance known. For the node-node constraints we must compute $e_{ij}^T e_{ij}$ for each distance known. Utilizing the defined example above, we will demonstrate this process.

For the anchor-node constraint we will use the measurement of anchor 2 to node 4. This distance is 2. The constraint matrix will be:

$$\begin{vmatrix} 4 \\ 0 \\ \cdots \\ 0 \\ 0 \\ 0 \\ -1 \end{vmatrix} |4 \quad 0 \quad \vdots \quad 0 \quad 0 \quad 0 \quad -1| = \begin{vmatrix} 16 & 0 & 0 & 0 & 0 & -4 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -4 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

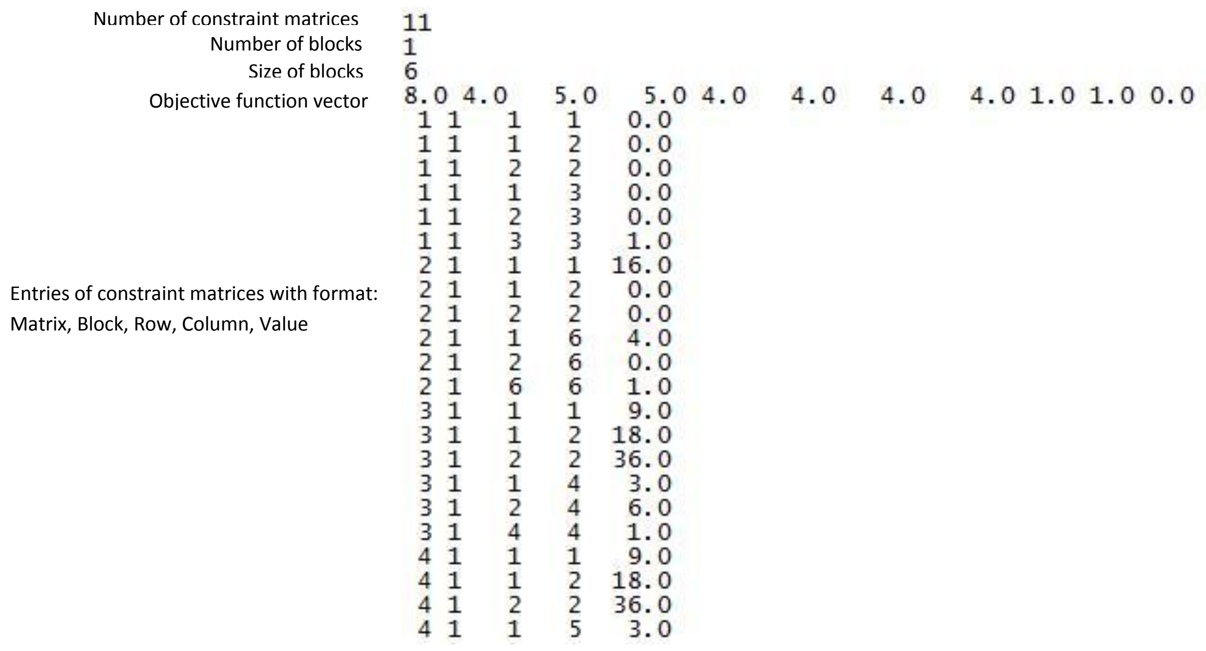
We will do similarly for the node-node constraints. We will use the node-node measurement of node 1 and node 2. The distance is 2. The constraint matrix will be:

$$\begin{vmatrix} 1 \\ -1 \\ 0 \\ 0 \end{vmatrix} \begin{vmatrix} 1 & -1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

These matrices will be the majority of our constraint matrices. The last three matrices will be used to force $Z_{1:d,1:d} = I_d$. The first of these will be a matrix of all zeros except for a 1 in the 1, 1 spot. The second will be all zeros except for a 1 in the 2, 2 spot. Finally, the third will also be all zeros except for a 1 in the 2, 1 spot. This is how we will construct all of our constraint matrices.

We now have almost all of the information we need to create the NEOS output. The last piece is the objective function vector needed in line 5. This will be made up of the known distances d_{ij}^2 and d_{kj}^2. They will be presented in the same order of the corresponding constraint matrices. So if we decide that our anchor-node example above is the first constraint matrix, then out first distance in the objective function vector will be 2, the corresponding distance. For the last three matrices, we will have 1, 1, 0 corresponding to the first, second, and third matrix. This will force the values in the matrices to the identity matrix values.

We now should be able to understand how to obtain the necessary information for the NEOS input. Here is part of the NEOS input for our simple example.



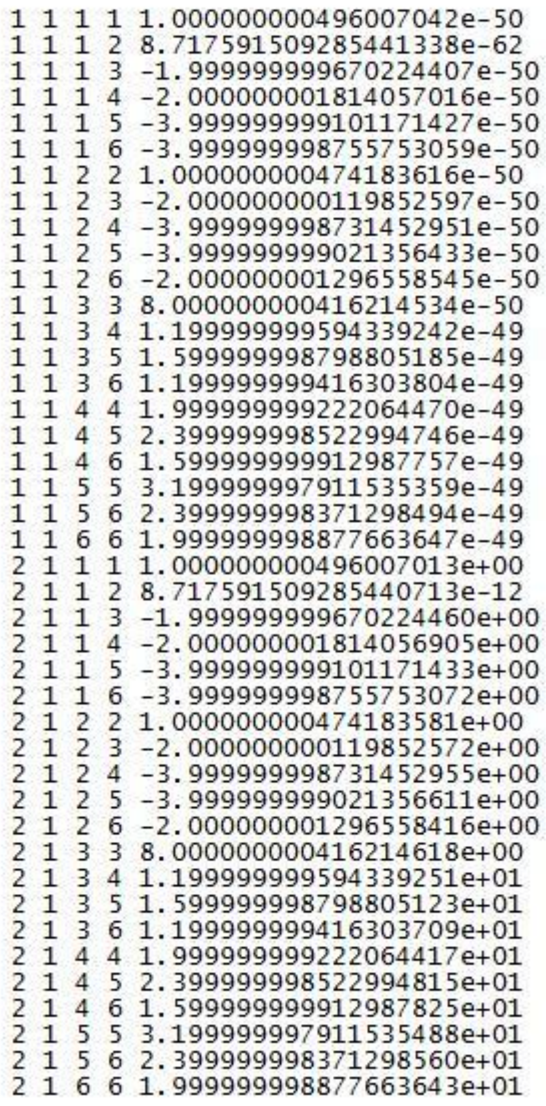
Number of constraint matrices
Number of blocks
Size of blocks
Objective function vector
Entries of constraint matrices with format:
Matrix, Block, Row, Column, Value

```
11
1
6
8.0 4.0   5.0   5.0 4.0   4.0   4.0   4.0 1.0 1.0 0.0
 1 1   1   1   0.0
 1 1   1   2   0.0
 1 1   2   2   0.0
 1 1   1   3   0.0
 1 1   2   3   0.0
 1 1   3   3   1.0
 2 1   1   1  16.0
 2 1   1   2   0.0
 2 1   2   2   0.0
 2 1   1   6   4.0
 2 1   2   6   0.0
 2 1   6   6   1.0
 3 1   1   1   9.0
 3 1   1   2  18.0
 3 1   2   2  36.0
 3 1   1   4   3.0
 3 1   2   4   6.0
 3 1   4   4   1.0
 4 1   1   1   9.0
 4 1   1   2  18.0
 4 1   2   2  36.0
 4 1   1   5   3.0
```

4.1.2 Output

Once we have the data in this format and we submit it to NEOS, we will obtain the solution back in the specific output format. The NEOS output will be in a similar format as the input however it just presents matrix values. These matrix values are of our solution matrix Z. Below is the output for our example.

Entries of our solution matrices with format: Matrix, Block, Row, Column, Value



```
1 1 1 1 1.000000000496007042e-50
1 1 1 2 8.717591509285441338e-62
1 1 1 3 -1.999999999670224407e-50
1 1 1 4 -2.000000001814057016e-50
1 1 1 5 -3.999999999101171427e-50
1 1 1 6 -3.999999998755753059e-50
1 1 2 2 1.000000000474183616e-50
1 1 2 3 -2.000000000119852597e-50
1 1 2 4 -3.999999998731452951e-50
1 1 2 5 -3.999999999021356433e-50
1 1 2 6 -2.000000001296558545e-50
1 1 3 3 8.000000000416214534e-50
1 1 3 4 1.199999999594339242e-49
1 1 3 5 1.599999998798805185e-49
1 1 3 6 1.199999999416303804e-49
1 1 4 4 1.999999999222064470e-49
1 1 4 5 2.399999998522994746e-49
1 1 4 6 1.599999999912987757e-49
1 1 5 5 3.199999997911535359e-49
1 1 5 6 2.399999998371298494e-49
1 1 6 6 1.999999998877663647e-49
2 1 1 1 1.000000000496007013e+00
2 1 1 2 8.717591509285440713e-12
2 1 1 3 -1.999999999670224460e+00
2 1 1 4 -2.000000001814056905e+00
2 1 1 5 -3.999999999101171433e+00
2 1 1 6 -3.999999998755753072e+00
2 1 2 2 1.000000000474183581e+00
2 1 2 3 -2.000000000119852572e+00
2 1 2 4 -3.999999998731452955e+00
2 1 2 5 -3.999999999021356611e+00
2 1 2 6 -2.000000001296558416e+00
2 1 3 3 8.000000000416214618e+00
2 1 3 4 1.199999999594339251e+01
2 1 3 5 1.599999998798805123e+01
2 1 3 6 1.199999999416303709e+01
2 1 4 4 1.999999999222064417e+01
2 1 4 5 2.399999998522994815e+01
2 1 4 6 1.599999999912987825e+01
2 1 5 5 3.199999997911535488e+01
2 1 5 6 2.399999998371298560e+01
2 1 6 6 1.999999998877663643e+01
```

Now that we understand the NEOS formats, we can utilize the software to solve our SNLPs.

4.2 MAPLE CODES

Even though it might be enough to understand the input and output files, converting our SNLP into this format and deciphering the output might be very tedious by hand, especially with many nodes and anchors. For this reason, I have developed MAPLE codes to perform this work. The first code takes all the sensor network data, provided in a very easy to use format, and converts it into the proper NEOS format. Once the solution is provided by NEOS, the second code takes the output, converts it into matrix form, and then plots the anchors and nodes onto a coordinate graph. We will continue to use our simple example to help explain the codes and the process for their use.

4.2.1 Data File

To begin, we receive, or compile ourselves, the data describing the SNLP in a very simple format. We write up the information into a text document called "data." The format of this file is as follows:

1. Number of anchors
2. Anchor positions with one position for each line in the format: node, x-coord., y-coord.
3. Number of anchor-node measurements
4. Anchor-node measurements in format: anchor, node, measurement
5. Number of node-node measurements
6. Node-node measurements in format: node, node, measurement

Here is the data file for our given example:

```
5
1    0    0
2    4    0
3    3    6
4    0    3
5    6    3
8
1    1    2.8284
2    4    2.0
3    2    2.2361
3    3    2.2361
4    1    2.2361
4    2    2.2361
5    4    2.2361
5    3    2.2361
4
1    2    2.0
```

2	3	2.0
3	4	2.0
4	1	2.0

4.2.2 MAPLE Code 1: Data to NEOS Input

Once we have our data file, we then open our first MAPLE code. This code takes our data file and converts it into the NEOS input format explained above. The MAPLE code is displayed below. A key thing to note is that prior executing the code, we must change the current directory in this code to the directory that our data file is stored in.

```
> # Input data for sensor network SDP and print csdp format as output.
 d:=2:  # Assume 2-dimensional for now
  # Open file "data" for reading. Contents are assumed to be in TEXT format.
 # File pointer for this file is fp.
 currentdir("C:/Users/bhegarty/Desktop");
  fp := fopen(data,READ,TEXT);
  # Get number of anchors.
 inlist := fscanf(fp,"%d");
  k := inlist[1];
  # Initialize all anchors to (0,0).
 Anchors := matrix(k,d,0):
  # For each anchor, read its label and (x,y)-coords.
 for i to k
  do
    inlist := fscanf(fp,"%d %f %f");
    if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading anchors`);
    i := k;
    else j := inlist[1]; Anchors[j,1] := inlist[2];   Anchors[j,2] := inlist[3];
    fi
  od;
  # Get number of anchor-node measurements.
 inlist := fscanf(fp,"%d");
  if nops( inlist ) < 1 then
   print(`ERROR: Ran out of data while reading number of anchor-node measurements.`);
 else
```

```
   AN := inlist[1];

    # Initialize all entries to zero.

   ANlist := matrix(AN,3,0):

    # For each measurement, store: anchor label, sensor label, distance.

   for i to AN

    do

     inlist := fscanf(fp,"%d %d %f");

     if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading AN measurements`);

     i := AN;

     else if inlist[1] > k then printf(`ERROR: in anchor-node pair %d, node index %d out of
range.\n`,

                                     i, inlist[1]);

     else ANlist[i,1] := inlist[1]; ANlist[i,2] := inlist[2]; ANlist[i,3] := inlist[3];

     fi  fi

    od;

  fi;

 # Get number of node-node measurements.

 inlist := fscanf(fp,"%d");

 if nops( inlist ) < 1 then

   print(`ERROR: Ran out of data while reading number of node-node measurements.`);

 else

   NN := inlist[1];

    # Initialize all entries to zero.

   NNlist := matrix(NN,3,0):

    # For each measurement, store: sensor label, sensor label, distance.

   for i to NN

    do

     inlist := fscanf(fp,"%d %d %f");

     if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading NN measurements`);

     i := NN;

     else NNlist[i,1] := inlist[1]; NNlist[i,2] := inlist[2]; NNlist[i,3] := inlist[3];

     # Keep convention that smaller node is first.

     if NNlist[i,1] > NNlist[i,2] then  # swap

        tempval := NNlist[i,1]; NNlist[i,1] := NNlist[i,2];  NNlist[i,2] := tempval;

     fi
     fi
    od;
 fi;
  # Now figure out the number of sensors (= biggest label that appeared).
  n := max( seq( ANlist[h,2],h=1..AN), seq( NNlist[h,2],h=1..NN) );
  # For debugging purposes, now report all data collected
  print(`k=`,k,`n=`,n,`AN=`,AN,`NN=`,NN);
 op(Anchors), op(ANlist), op(NNlist);
  #  Now we can give the neos input.  Start with an easy-to-locate line.
  print(`Now we give the input file for the csdp neos solver.`);
 printf(`neosneosneosneosneosneosneosneosneosneosneosneosneosneosneosneosneos\n`);
 printf(`%d\n1\n%d\n`,AN+NN+3,n+2): # num. constraints, num. blocks, block size
  # Now print objective coefficients
 printf(`%2.1f `,ANlist[1,3]^2*1.0):
 for i from 2 to AN do printf(`%5.1f `,ANlist[i,3]^2*1.0); od:
 printf(`   `):
 for i to NN do printf(`%5.1f `,NNlist[i,3]^2*1.0); od:
 printf(` 1.0 1.0 0.0\n`):
  for i to AN  # Build all constraints for anchor-node measurements
  do
   anc := ANlist[i,1]:  nod := ANlist[i,2]:
   printf(`%2d 1   1   1 %5.1f\n`,i,Anchors[anc,1]^2*1.0);
   printf(`%2d 1   1   2 %5.1f\n`,i,Anchors[anc,1]*Anchors[anc,2]*1.0);
   printf(`%2d 1   2   2 %5.1f\n`,i,Anchors[anc,2]^2*1.0);
   printf(`%2d 1   1  %2d %5.1f\n`,i,2+nod,Anchors[anc,1]*1.0);
   printf(`%2d 1   2  %2d %5.1f\n`,i,2+nod,Anchors[anc,2]*1.0);
   printf(`%2d 1  %2d  %2d %5.1f\n`,i,2+nod,2+nod,1.0);
  od:
   for i to NN # Build all constraints for node-node measurements
  do
   n1 := NNlist[i,1]:  n2 := NNlist[i,2]:
   printf(`%2d 1  %2d  %2d   1.0\n`,AN+i,2+n1,2+n1);
   printf(`%2d 1  %2d  %2d  -1.0\n`,AN+i,2+n1,2+n2);
   printf(`%2d 1  %2d  %2d   1.0\n`,AN+i,2+n2,2+n2);
```

```
od:
# Finally, force 2x2 identity matrix in upper-left
# corner of psd variable Z
printf(`%2d 1   1   1   1.0\n`,AN+NN+1);
printf(`%2d 1   2   2   1.0\n`,AN+NN+2);
printf(`%2d 1   1   2   1.0\n`,AN+NN+3);
quit
```

This program will then provide us with our NEOS input. For our example, this input is:

```
15
1
6
8.0 4.0   5.0   5.0   5.0   5.0   5.0   5.0 4.0   4.0   4.0   4.0 1.0 1.0 0.0
1 1   1   1   0.0
1 1   1   2   0.0
1 1   2   2   0.0
1 1   1   3   0.0
1 1   2   3   0.0
1 1   3   3   1.0
2 1   1   1  16.0
2 1   1   2   0.0
2 1   2   2   0.0
2 1   1   6   4.0
2 1   2   6   0.0
2 1   6   6   1.0
3 1   1   1   9.0
3 1   1   2  18.0
3 1   2   2  36.0
3 1   1   4   3.0
3 1   2   4   6.0
3 1   4   4   1.0
4 1   1   1   9.0
```

```
4 1  1  2  18.0

4 1  2  2  36.0

4 1  1  5   3.0

4 1  2  5   6.0

4 1  5  5   1.0

5 1  1  1   0.0

5 1  1  2   0.0

5 1  2  2   9.0

5 1  1  3   0.0

5 1  2  3   3.0

5 1  3  3   1.0

6 1  1  1   0.0

6 1  1  2   0.0

6 1  2  2   9.0

6 1  1  4   0.0

6 1  2  4   3.0

6 1  4  4   1.0

7 1  1  1  36.0

7 1  1  2  18.0

7 1  2  2   9.0

7 1  1  5   6.0

7 1  2  5   3.0

7 1  5  5   1.0

8 1  1  1  36.0

8 1  1  2  18.0

8 1  2  2   9.0

8 1  1  6   6.0

8 1  2  6   3.0

8 1  6  6   1.0

9 1  3  3   1.0

9 1  3  4  -1.0
```

```
9 1   4   4   1.0

10 1   4   4   1.0

10 1   4   5  -1.0

10 1   5   5   1.0

11 1   5   5   1.0

11 1   5   6  -1.0

11 1   6   6   1.0

12 1   3   3   1.0

12 1   3   6  -1.0

12 1   6   6   1.0

13 1   1   1   1.0

14 1   2   2   1.0

15 1   1   2   1.0
```

4.2.3 MAPLE Code 2: NEOS Output to Matrix and Graph

After submitting this to NEOS, we will receive the NEOS solution. It will be in the format explained above. For our example, we receive this output (as previously shown):

```
1 1 1 1 1.000000000496007042e-50

1 1 1 2 8.717591509285441338e-62

1 1 1 3 -1.999999999670224407e-50

1 1 1 4 -2.000000001814057016e-50

1 1 1 5 -3.999999999101171427e-50

1 1 1 6 -3.999999998755753059e-50

1 1 2 2 1.000000000474183616e-50

1 1 2 3 -2.000000000119852597e-50

1 1 2 4 -3.999999998731452951e-50

1 1 2 5 -3.999999999021356433e-50

1 1 2 6 -2.000000001296558545e-50

1 1 3 3 8.000000000416214534e-50

1 1 3 4 1.199999999594339242e-49

1 1 3 5 1.599999998798805185e-49
```

```
1 1 3 6 1.199999999416303804e-49

1 1 4 4 1.999999999222064470e-49

1 1 4 5 2.399999998522994746e-49

1 1 4 6 1.599999999912987757e-49

1 1 5 5 3.199999997911535359e-49

1 1 5 6 2.399999998371298494e-49

1 1 6 6 1.999999998877663647e-49

2 1 1 1 1.000000000496007013e+00

2 1 1 2 8.717591509285440713e-12

2 1 1 3 -1.999999999670224460e+00

2 1 1 4 -2.000000001814056905e+00

2 1 1 5 -3.999999999101171433e+00

2 1 1 6 -3.999999998755753072e+00

2 1 2 2 1.000000000474183581e+00

2 1 2 3 -2.000000000119852572e+00

2 1 2 4 -3.999999998731452955e+00

2 1 2 5 -3.999999999021356611e+00

2 1 2 6 -2.000000001296558416e+00

2 1 3 3 8.000000000416214618e+00

2 1 3 4 1.199999999594339251e+01

2 1 3 5 1.599999998798805123e+01

2 1 3 6 1.199999999416303709e+01

2 1 4 4 1.999999999222064417e+01

2 1 4 5 2.399999998522994815e+01

2 1 4 6 1.599999999912987825e+01

2 1 5 5 3.199999997911535488e+01

2 1 5 6 2.399999998371298560e+01

2 1 6 6 1.999999998877663643e+01
```

We then take this output, copy it, and paste it into a text file called "fromneos." After creating this file, we open MAPLE again and use our next code which will take the NEOS output and display it in matrix and graphical form. This code is shown below:

(Note again that the directory must be correct for the code to work.)

```
> currentdir("C:/Users/bhegarty/Desktop");

> # File pointer for this file is neosfp.

neosfp := fopen(fromneos,READ,TEXT);

# File pointer for data file is dfp.

dfp := fopen(data,READ,TEXT);

L := [];

# For each line, store it as a 5-tuple in L

ans := fscanf(neosfp,"%d %d %d %d");

while nops( ans ) > 0

  do

     xx := fscanf(neosfp,"%g");

     L := [  op(L), [ op(ans), op(xx) ]  ];

     ans := fscanf(neosfp,"%d %d %d %d");

  od:

fclose(neosfp);

> #  Now get original data to find anchors and graph:

# Get number of anchors.

 inlist := fscanf(dfp,"%d");

  k := inlist[1];

  # Initialize list of anchors to empty

 anchors := [];

  # For each anchor, store (x,y)-coords.

 for i to k

  do

    inlist := fscanf(dfp,"%d %f %f");

    if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading anchors`);

    i := k;

    else anchors:= [op(anchors), [inlist[2], inlist[3]]];

    fi

  od;
```

```
>  # Get number of anchor-node measurements.

 inlist := fscanf(dfp,"%d");

  if nops( inlist ) < 1 then

   print(`ERROR: Ran out of data while reading number of anchor-node measurements.`);

 else

   AN := inlist[1];

    # Initialize list of anchor node edges to empty

   aedges:=[];

   andist:=[];

    # For each measurement, store: anchor label, sensor label, distance.

   for i to AN

    do

     inlist := fscanf(dfp,"%d %d %f");

     if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading AN measurements`);

     i := AN;

     else if inlist[1] > k then printf(`ERROR: in anchor-node pair %d, node index %d out of
range.\n`, i, inlist[1]);

     else aedges:=[op(aedges), [inlist[1], inlist[2]]]; andist:= [op(andist), inlist[3]];

     fi  fi

    od;

  fi;

 >aedges, andist, indist, AN;

>  # Get number of node-node measurements.

 inlist := fscanf(dfp,"%d");

if nops( inlist ) < 1 then

   print(`ERROR: Ran out of data while reading number of node-node measurements.`);

 else

   NN := inlist[1];

># Initialize list of node node edges to empty

   xedges:=[];   nndist:=[];   # For each measurement, store: sensor label, sensor label, #
#distance.

   for i to NN

    do

     inlist := fscanf(dfp,"%d %d %f");

     if nops( inlist ) < 3 then print(`ERROR: Ran out of data while reading NN measurements`);
     i := NN;

     else xedges:=[op(xedges),[inlist[1], inlist[2]]]; nndist:=[op(nndist), inlist[3]];

     # Keep convention that smaller node is first.

     ####(Don't think this works because inlist[1] is more of a command, not a value

     ####  Does it matter that to have it in numerical order?)

         if inlist[1] > inlist[2] then #swap

         tempval:= inlist[1]; inlist[1]:= inlist[2]; inlist[1]:=tempval;

     fi

     fi

     od;

fi;

> # now get actual matrix from neos output

numdigs := 3;

numvals := nops(L); # How many entries in the list?


size := max(  seq( L[i][4], i=1..numvals) );  # find size of matrix

B := matrix(size,size, 0.):

for i to numvals

   do

     row := L[i][3];  col := L[i][4];

     B[ row, col] := round( 10^numdigs * L[i][5])/(10.^numdigs) ;

     B[ col, row] := round( 10^numdigs * L[i][5])/(10.^numdigs) ;

   od:

op(B);

getX := proc(B)  # Given matrix B, pull off coords of all points x_i

local   Pts, h, size;

  size  := linalg[coldim](B);

  Pts := [ seq( [B[1,h],B[2,h] ], h=3..size) ];

  RETURN( Pts );

end;

#get nodes from matrix

Points:= -getX(B);

> numanchors := nops( anchors );

numPoints  := nops( Points );
```

```
numaedges := nops( aedges );

numxedges := nops( xedges );

> anchors, Points, aedges, xedges;

> with(plots):

eps := 0.05;

# Start off with axes only and choose region of the plane to display. #(Later

# we can automate the choice of upper and lower limits.

listofplots := [ plot(-1,x=-2..5,y=-2..10,color="White") ];

for i to numanchors

 do

   # Plot a small black circle centered at (cx,cy).

   cx := anchors[i][1]; cy := anchors[i][2];

   acirc||i := implicitplot((x-cx)^2+(y-cy)^2 = eps^2,

   x = cx-2*eps .. cx+2*eps, y = cy-2*eps .. cy+2*eps, thickness = 3, color = "Black");

   # Add label

   atext||i := textplot([cx+.1, cy-.6, typeset(A[i])], color = "DarkBlue", font = [TIMES, ROMAN,
20], align = {right});

   # Now toss these two plots onto the end of the list.

   listofplots := [ op(listofplots), acirc||i, atext||i];

 od:

> for i to numPoints

 do

   # Plot a small blue circle centered at (cx,cy).

   cx := Points[i][1]; cy := Points[i][2];

   Pcirc||i := implicitplot((x-cx)^2+(y-cy)^2 = eps^2,

   x = cx-2*eps .. cx+2*eps, y = cy-2*eps .. cy+2*eps, thickness = 3, color = "Orange");

   # Add label

   Ptext||i := textplot([cx+.1, cy-.6, typeset(X[i])], color = "Orange", font = [TIMES, ROMAN,
20], align = {right});

   # Now toss these two plots onto the end of the list.

   listofplots := [ op(listofplots), Pcirc||i, Ptext||i];

 od:

> for i to numaedges

 do
```

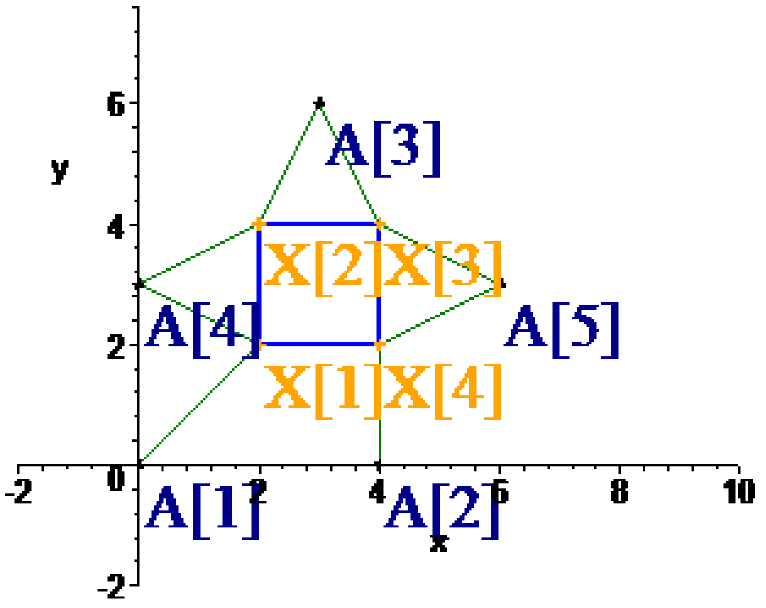
```
   listofplots := [ op(listofplots) , plot( [ anchors[aedges[i][1]], Points[aedges[i][2]]
],color="Green") ];

 od:

> for i to numxedges

 do

   listofplots := [ op(listofplots) , plot( [ Points[xedges[i][1]], Points[xedges[i][2]]
],color="Blue",thickness=2) ];

 od:

 display( listofplots);
```

Using this code for our example, we will receive the following representations of the solution:

Matrix:

$$\begin{bmatrix} 1.000000000 & 0. & 2.000000000 & 2.000000000 & 4.000000000 & 4.000000000 \\ 0. & 1.000000000 & 2.000000000 & 4.000000000 & 4.000000000 & 2.000000000 \\ 2.000000000 & 2.000000000 & 8.000000000 & 12.00000000 & 16.00000000 & 12.00000000 \\ 2.000000000 & 4.000000000 & 12.00000000 & 20.00000000 & 24.00000000 & 16.00000000 \\ 4.000000000 & 4.000000000 & 16.00000000 & 24.00000000 & 32.00000000 & 24.00000000 \\ 4.000000000 & 2.000000000 & 12.00000000 & 16.00000000 & 24.00000000 & 20.00000000 \end{bmatrix}$$

Graph:



5 Conclusion and Further Work

Vast benefits of semidefinite programming have already been noted by many experts, and I believe more are to be discovered in the future. This project focused on the understanding of this new and exciting topic in mathematics and how it can be applied to the real world. It began by giving an introduction into the basic theory behind semidefinite programming. It then discussed a method for SDP solution: the interior point method. Next, it surveyed a variety of applications, elaborating on that of the sensor networks. Lastly, it presented computer based tools that can be used to solve given SNLPs.

While this project covered the theory of semidefinite programming, its methods for its solution, and its application to sensor networks, more work can be done. I believe the next step is to investigate the SNLP with noisy data, referring to the utilization of measurements which are not exact. Also, the development of a user friendly and interactive graph, which displays the solution to the sensor network being studied, would be very advantageous to those not as familiar with these topics of advanced mathematics.

6 References

[1] C. Helmberg, F. Rendl, R. Vanderbei, and H. Wolkowicz, *An Interior-Point Method for Semidefinite Programming*, SIAM J. Optim., 6 (1996), pp.342-361.

[2] R. Vanderbei. Linear Programming: Foundations and Extensions. Springer, New York, (2008), pp.287-318.

[3] J. B. Lasserre, *Why the Logarithmic Barrier Function in Convex and Linear Programming*, Operations Research Letters, 27 (2000), pp.149-152.

[4] "Line Search Methods." Wolfram Mathematica: Documentation Center, Wolfram Research Inc, (2010). http://reference.wolfram.com/mathematica/tutorial/UnconstrainedOptimizationLineSearchMethods.html.

[5] M. Goemans and D. Williamson, *Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming*, JACM, 42 (1995), pp. 1115-1145.

[6] J. Wheeler, *An Investigation of the MaxCut Problem*, Internal Report, University of Iowa, 2004.

[7] A. Kojevnikov and A. S. Kulikov, *New Approach to Proving Upper Bounds for MAX-2-SAT*, SODA (2006), pp. 11-17.

[8] Professor W. J. Martin, 2009-2010, Personal communication through weekly meetings.

[9] A. Schrijver, *New Code Upper Bounds from the Terwilliger Agebra*, IEEE Transactions on Information Theory 51 (2005) pp. 2859-2866.

[10] A.S. Lewis, *The Mathematics of Eigenvalue Optimization*, Mathematical Programming, 97 (2003), pp. 155-176.

[11] I. Popescu and D. Bertsimas*, Moment Problems via Semidefinite Programming: Applications in Probability and Finance*, Working Paper, MIT, Cambridge, MA (2000).

[12] C. Commander, M. Ragle and Y. Ye, *Semidefinite Programming and the Sensor Network Localization Problem, SNLP*, Encyclopedia of Optimization, 2007.

[13] A. So and Y. Ye, *Theory of Semidefinite Programming for Sensor Networks Localization*, Mathematical Programming, 2006.

[14] *NEOS: Server for Optimization*, http://neos.mcs.anl.gov/neos/solvers/sdp:csdp/MATLAB_BINARY.html.