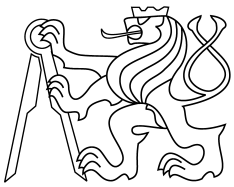
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

MASTER'S THESIS

Semidefinite Programming for Geometric Problems in Computer Vision

Pavel Trutman

pavel.trutman@fel.cvut.cz

April 29, 2017

Available at
http://cmp.felk.cvut.cz/~trutmpav/master-thesis/thesis/thesis.pdf

Thesis Advisor: Ing. Tomáš Pajdla, PhD.

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Acknowledgements

Author's declaration

I declare that I have work out the presented thesis independently and that I have listed all information sources used in accordance with the Methodical Guidelines about Maintaining Ethical Principles for Writing Academic Theses.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Abstract

Keywords:

Abstrakt

Keywords:

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Symbols and Abbreviations

$\mathrm{cl}(S)$	Closure of the set S.
$\mathrm{diag}(x)$	Diagonal matrix with components of x on the diagonal.
dom f	Domain of the function f.
Dom f	cl(dom f).
$f'(x)$	First derivative of the function $f(x)$.
$f''(x)$	Second derivative of the function $f(x)$.
$\mathcal{I}^n$	Identity matrix of size $n \times n$.
int S	Interior of the set S.
$\left\{\lambda_i(A)\right\}_{i=1}^n$	Set of all eigenvalues of the matrix $A \in \mathbb{R}^{n\times n}$.
LMI	Linear matrix inequality.
LP	Linear program.
$\mathcal{P}^n$	Cone of positive semidefinite $n \times n$ matrices.
QCQP	Quadratically constrained quadratic program.
$\mathbb{R}^n$	n-dimensional Euclidean space.
$\mathcal{S}^n$	Space of $n \times n$ real symmetric matrices.
SDP	Semidefinite programming.
$\mathrm{tr}(A)$	Trace of the matrix A.
$x^{(i)}$	i-th element of the vector x.
$x^\top$	Transpose of the vector x.

1. Introduction

2. Semidefinite programming

The goal of the semidefinite programming (SDP) is to optimize a linear function on a given set, which is an intersection of a cone of positive semidefinite matrices with an affine space. This set is called a spectrahedron and it is a convex set. As in SDP we are optimizing a convex function on a convex set, SDP is a special case of convex optimization.

Since SDP can be solved efficiently in polynomial time using interior-point methods, it has many applications in practise. For example, any linear program (LP) and quadratically constrained quadratic program (QCQP) can be written as a semidefinite program. However, this may be not the best idea to do as more efficient algorithms exist for solving LPs and QCQPs. On the other hand, there exist many useful applications of SDP, e.g. many NP-complete problems in combinatorial optimization can be approximated by semidefinite programs. One of the combinatorial problem worth mentioning is the MAX CUT problem (one of the Karp's original NP-complete problems [6]), for which M. Goemans and D. P. Williamson created the first approximation algorithm based on SDP [4]. Also in control theory, there are many problems based on linear matrix inequalities, which are solvable by SDP. Special application of SDP comes from polynomial optimization since global solution of polynomial optimization problems can be found by hierarchies of semidefinite programs. We will focus in details on this application in Section **??**.

2.1. Preliminaries on semidefinite programs

In this section, we introduce some notation and preliminaries about symmetric matrices and semidefinite programs. We will introduce further notation and preliminaries later on in the text when needed.

At the beginning, let us denote the inner product for two vectors x, $y \in \mathbb{R}^n$

$$\langle x, y \rangle \quad = \quad \sum_{i=1}^{n} x^{(i)} y^{(i)} \tag{2.1}$$

and the Frobenius inner product for two matrices X, $Y \in \mathbb{R}^{n \times m}$.

$$\langle X, Y \rangle \quad = \quad \sum_{i=1}^{n} \sum_{j=1}^{m} X^{(i,j)} Y^{(i,j)} \tag{2.2}$$

2.1.1. Symmetric matrices

Let $\mathcal{S}^n$ denotes the space of $n \times n$ real symmetric matrices.

For a matrix $M \in \mathcal{S}^n$, the notation $M \succeq 0$ means that M is positive semidefinite. $M \succeq 0$ if and only if any of the following equivalent properties holds.

1. $x^\top M x \geq 0$ for all $x \in \mathbb{R}^n$.
2. All eigenvalues of M are nonnegative.

The set of all positive semidefinite matrices is a cone. We will denote it as $\mathcal{P}^n$ and it is called a cone of positive semidefinite matrices.

For a matrix $M \in \mathcal{S}^n$, the notation $M \succ 0$ means that M is positive definite. $M \succ 0$ if and only if any of the following equivalent properties holds.

1. $M \succeq 0$ and $\operatorname{rank} M = n$.
2. $x^\top M x > 0$ for all $x \in \mathbb{R}^n$.
3. All eigenvalues of M are positive.

2.1.2. Semidefinite programs

The standard (primal) form of a semidefinite program in variable $X \in \mathcal{S}^n$ is defined as follows:

$$\begin{array}{rcll} p^* & = & \sup\limits_{X \in \mathcal{S}^n} & \langle C, X \rangle \\ & & \text{s.t.} & \langle A_i, X \rangle = b^{(i)} \quad (i = 1, \ldots, m) \\ & & & X \succeq 0 \end{array} \tag{2.3}$$

where C, A_1, …, $A_m \in \mathcal{S}^n$ and $b \in \mathbb{R}^m$ are given.

The dual form of the primal form is the following program in variable $y \in \mathbb{R}^m$.

$$\begin{array}{rcll} d^* & = & \inf\limits_{y \in \mathbb{R}^m} & b^\top y \\ & & \text{s.t.} & \sum\limits_{i=1}^{m} A_i y^{(i)} - C \succeq 0 \end{array} \tag{2.4}$$

The constraint

$$F(y) = \sum_{i=1}^{m} A_i y^{(i)} - C \succeq 0 \tag{2.5}$$

of the problem (2.4) is called a linear matrix inequality (LMI) in the variable y. The feasible region defined by LMI is called a spectrahedron. It can be shown, that this constraint is convex since if $F(x) \succeq 0$ and $F(y) \succeq 0$, then $\forall \alpha : 0 \leq \alpha \leq 1$ holds

$$F\big(\alpha x + (1-\alpha) y\big) = \alpha F(x) + (1-\alpha) F(y) \succeq 0. \tag{2.6}$$

The objective function of the problem (2.4) is linear, and therefore convex too. Because the semidefinite program (2.4) has convex objective function and convex constraint, it is a convex optimization problem and can be solved by standard convex optimization methods. To get a general picture, how a simple semidefinite problem may look like, see Figure 2.1.

The optimal solution y^* of any semidefinite program lies on the boundary of the feasible set, supposing the problem is feasible and the solution exists. The boundary of the feasible set is not smooth in general, but it is piecewise smooth as each piece is an algebraic surface.

Example 2.1 (Linear programming). Semidefinite programming can be seen as an extension to the linear programming since the componentwise inequalities between vectors in linear programming can be replaced by LMI. Consider a linear program in a standard form

$$\begin{array}{rcll} y^* & = & \arg\min\limits_{y \in \mathbb{R}^m} & b^\top y \\ & & \text{s.t.} & Ay - c \geq 0 \end{array} \tag{2.7}$$

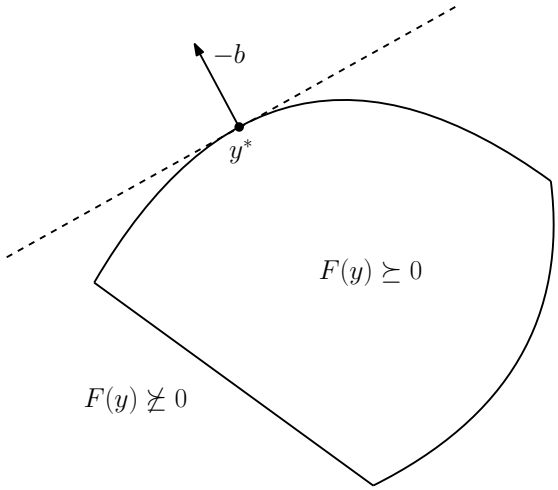


Figure 2.1. Example of a simple semidefinite problem for $y \in \mathbb{R}^2$. Boundary of the feasible set $\{y \mid F(y) \succeq 0\}$ is shown as a black curve. The minimal value of the objective function $b^\top y$ is attained at y^*.

with $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ and $A = \begin{bmatrix} a_1 & \cdots & a_m \end{bmatrix} \in \mathbb{R}^{n \times m}$. This program can be transformed into the semidefinite program (2.4) by assigning

$$C = \operatorname{diag}(c), \tag{2.8}$$
$$A_i = \operatorname{diag}(a_i). \tag{2.9}$$

2.2. State of the art review

An early paper by R. Bellman and K. Fan about theoretical properties of semidefinite programs [2] was issued in 1963. Later on, many researchers worked on the problem of minimizing the maximal eigenvalue of a symmetric matrix, which can be done by solving a semidefinite program. Selecting a few from many: J. Cullum, W. Donath, P. Wolfe [3], M. Overton [13] and G. Pataki [14]. In 1984, the interior-point methods for LPs solving were introduced by N. Karmarkar [5]. It was the first reasonably efficient algorithm that solves LPs in polynomial time with excellent behavior in practise. The interior-point algorithms were then extended to be able to solve convex quadratic programs.

In 1988, Y. Nesterov and A. Nemirovski [11] did an important breakthrough. They showed that interior-point methods developed for LPs solving can be generalized to all convex optimization problems. All that is required, is the knowledge of a self-concordant barrier function for the feasible set of the problem. Y. Nesterov and A. Nemirovski have shown that a self-concordant barrier function exists for every convex set. However, their proposed universal self-concordant barrier function and its first and second derivatives are not easily computable. Fortunately for SDP, which is an important class of convex optimization programs, computable self-concordant barrier functions are known, and therefore the interior-point methods can be used.

Nowadays, there are many libraries and toolboxes that one can use for solving semidefinite programs. They differ to each other in used methods and their implementations. Before starting solving a problem, one should know the details of the problem to solve and choose the library accordingly to it as not every method and its implementation is suitable for the given problem.

Most methods are based on interior-point methods, which are efficient and robust for

general semidefinite programs. The main disadvantage of these methods is that they need to store and factorize usually large Hessian matrix. Most modern implementations of the interior-point methods do not need the knowledge of an interior feasible point in advance. SeDuMi [15] casts the standard semidefinite program into the homogeneous self-dual form, which has a trivial feasible point. SDPA [18] uses an infeasible interior-point method, which can initialized by an infeasible point. Some of the libraries (e.g. MOSEK [9]) have started out as LPs solvers and were extended for QCQPs solving and convex optimization later on.

Another type of methods used in SDP are the first-order methods. They avoid storing and factorizing Hessian matrices, and therefore they are able to solve much larger problems than interior-point methods, but at some cost in accuracy. This method is implemented, for instance, in the SCS solver [12].

2.3. Nesterov's approach

In this section, we will follow Chapter 4 of [10] by Y. Nesterov, which is devoted to the convex optimization problems. This chapter describes the state of the art interior-point methods for solving convex optimization problems. We will extract from it the only minimum, just to be able to introduce an algorithm for semidefinite programs solving. We will present some basic definitions and theorems, but we will not prove them. For the proofs and more details look into [10].

2.3.1. Self-concordant functions

Definition 2.1 (Self-concordant function in $\mathbb{R}$). A closed convex function $f : \mathbb{R} \mapsto \mathbb{R}$ is self-concordant if there exist a constant $M_f \geq 0$ such that the inequality

$$|f'''(x)| \leq M_f f''(x)^{3/2} \tag{2.10}$$

holds for all $x \in \operatorname{dom} f$.

For better understanding of self-concordant functions, we provide several examples.

Example 2.2.

1. Linear and convex quadratic functions.

$$f'''(x) = 0 \text{ for all } x \tag{2.11}$$

   Linear and convex quadratic functions are self-concordant with constant $M_f = 0$.

2. Negative logarithms.

$$f(x) = -\ln(x) \text{ for } x > 0 \tag{2.12}$$
$$f'(x) = -\frac{1}{x} \tag{2.13}$$
$$f''(x) = \frac{1}{x^2} \tag{2.14}$$
$$f'''(x) = -\frac{2}{x^3} \tag{2.15}$$
$$\frac{|f'''(x)|}{f''(x)^{3/2}} = 2 \tag{2.16}$$

   Negative logarithms are self-concordant functions with constant $M_f = 2$.

3. Exponential functions.

$$f(x) = e^x \tag{2.17}$$
$$f''(x) = f'''(x) = e^x \tag{2.18}$$
$$\frac{|f'''(x)|}{f''(x)^{3/2}} = e^{-x/2} \to +\infty \quad \text{as } x \to -\infty \tag{2.19}$$

Exponential functions are not self-concordant functions.

Definition 2.2 (Self-concordant function in $\mathbb{R}^n$). A closed convex function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is self-concordant if function $g : \mathbb{R} \mapsto \mathbb{R}$

$$g(t) = f(x + tv) \tag{2.20}$$

is self-concordant for all $x \in \operatorname{dom} f$ and all $v \in \mathbb{R}^n$.

Now, let us focus on the main properties of self-concordant functions.

Theorem 2.1. Let functions f_i be self-concordant with constants M_i and let $\alpha_i > 0$, $i = 1, 2$. Then the function

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) \tag{2.21}$$

is self-concordant with constant

$$M_f = \max\left\{\frac{1}{\sqrt{\alpha_1}}M_1, \frac{1}{\sqrt{\alpha_2}}M_2\right\} \tag{2.22}$$

and

$$\operatorname{dom} f = \operatorname{dom} f_1 \cap \operatorname{dom} f_2. \tag{2.23}$$

Corollary 2.1. Let function f be self-concordant with some constant M_f and $\alpha > 0$. Then the function $\phi(x) = \alpha f(x)$ is also self-concordant with the constant $M_\phi = \frac{1}{\sqrt{\alpha}}M_f$.

We call function $f(x)$ as the standard self-concordant function if $f(x)$ is some self-concordant function with the constant $M_f = 2$. Using Corollary 2.1, we can see that any self-concordant function can be transformed into the standard self-concordant function by scaling.

Theorem 2.2. Let function f be self-concordant. If $\operatorname{dom} f$ contains no straight line, then the Hessian $f''(x)$ is nondegenerate at any x from $\operatorname{dom} f$.

For some self-concordant function $f(x)$, for which we assume, that $\operatorname{dom} f$ contains no straight line (which implies that all $f''(x)$ are nondegenerate, see Theorem 2.2), we denote two local norms as

$$\|u\|_x = \sqrt{u^\top f''(x) u} \tag{2.24}$$
$$\|u\|_x^* = \sqrt{u^\top f''(x)^{-1} u}. \tag{2.25}$$

Consider following minimization problem

$$x^* = \arg\min_{x \in \operatorname{dom} f} f(x) \tag{2.26}$$

as the minimization of the self-concordant function $f(x)$. Algorithm 2.1 is describing the iterative process of solving the optimization problem (2.26). The algorithm is divided into two stages by the value of $\|f'(x_k)\|^*_{x_k}$. The splitting parameter β guarantees quadratic convergence rate for the second part of the algorithm. The parameter β is chosen from interval $(0, \bar{\lambda})$, where

$$\bar{\lambda} = \frac{3-\sqrt{5}}{2}, \tag{2.27}$$

which is a root of the equation

$$\frac{\lambda}{(1-\lambda)^2} = 1. \tag{2.28}$$

Algorithm 2.1. Newton method for minimization of self-concordant functions.

Input:

f a self-concordant function to minimize
$x_0 \in \operatorname{dom} f$ a starting point
$\beta \in (0, \bar{\lambda})$ a parameter of size of the region of quadratic convergence
ε a precision

Output:

x^* an optimal solution to the minimization problem (2.26)

1: $k \leftarrow 0$
2: **while** $\|f'(x_k)\|^*_{x_k} \geq \beta$ **do**
3: $\quad x_{k+1} \leftarrow x_k - \frac{1}{1+\|f'(x_k)\|^*_{x_k}} f''(x_k)^{-1} f'(x_k)$
4: $\quad k \leftarrow k+1$
5: **end while**
6: **while** $\|f'(x_k)\|^*_{x_k} > \varepsilon$ **do**
7: $\quad x_{k+1} \leftarrow x_k - f''(x_k)^{-1} f'(x_k)$
8: $\quad k \leftarrow k+1$
9: **end while**
10: **return** $x^* \leftarrow x_k$

The first while loop (lines 2 – 5) represents damped Newton method, where at each iteration we have

$$f(x_k) - f(x_{k+1}) \geq \beta - \ln(1+\beta) \text{ for } k \geq 0, \tag{2.29}$$

where

$$\beta - \ln(1+\beta) > 0 \text{ for } \beta > 0, \tag{2.30}$$

and therefore the global convergence of the algorithm is ensured. It can be shown that the local convergence rate of the damped Newton method is also quadratic, but the presented switching strategy is preferred as it gives better complexity bounds.

The second while loop of the algorithm (lines 6 – 9) is the standard Newton method with quadratic convergence rate.

The algorithm terminates when the required precision ε is reached.

2.3.2. Self-concordant barriers

To be able to introduce self-concordant barriers, let us denote $\operatorname{Dom} f = \operatorname{cl}(\operatorname{dom} f)$.

Definition 2.3 (Self-concordant barrier). Let $F(x)$ be a standard self-concordant function. We call it a ν-self-concordant barrier for set $\operatorname{Dom} F$, if

$$\sup_{u \in \mathbb{R}^n} \left(2u^\top F'(x) - u^\top F''(x)u\right) \quad \leq \quad \nu \tag{2.31}$$

for all $x \in \operatorname{dom} F$. The value ν is called the parameter of the barrier.

The inequality (2.31) can be rewritten into the following equivalent matrix notation:

$$F''(x) \quad \succeq \quad \frac{1}{\nu} F'(x) F'(x)^\top. \tag{2.32}$$

In Definition 2.3, the hessian $F''(x)$ is not required to be nondegenerate. However, in case that $F''(x)$ is nondegenerate, the inequality (2.31) is equivalent to

$$F'^\top(x) F''(x)^{-1} F'(x) \quad \leq \quad \nu. \tag{2.33}$$

Let us explore, which basic functions are self-concordant barriers.

Example 2.3.

1. Linear functions.

$$F(x) \quad = \quad \alpha + a^\top x, \ \operatorname{dom} F = \mathbb{R}^n \tag{2.34}$$
$$F''(x) \quad = \quad 0 \tag{2.35}$$

   From (2.32) and for $a \neq 0$ follows, that linear functions are not self-concordant barriers.

2. Convex quadratic functions.
   For $A = A^\top \succ 0$:

$$F(x) \quad = \quad \alpha + a^\top x + \frac{1}{2} x^\top A x, \ \operatorname{dom} F = \mathbb{R}^n \tag{2.36}$$
$$F'(x) \quad = \quad a + Ax \tag{2.37}$$
$$F''(x) \quad = \quad A \tag{2.38}$$

   After substitution into (2.33) we obtain

$$(a + Ax)^\top A^{-1}(a + Ax) \quad = \quad a^\top A^{-1} a + 2a^\top x + x^\top A x, \tag{2.39}$$

   which is unbounded from above on $\mathbb{R}^n$. Therefore, quadratic functions are not self-concordant barriers.

3. Logarithmic barrier for a ray.

$$F(x) \quad = \quad -\ln x, \ \operatorname{dom} F = \left\{x \in \mathbb{R} \mid x > 0\right\} \tag{2.40}$$
$$F'(x) \quad = \quad -\frac{1}{x} \tag{2.41}$$
$$F''(x) \quad = \quad \frac{1}{x^2} \tag{2.42}$$

   From (2.33), when $F'(x)$ and $F''(x)$ are both scalars, we get

$$\frac{F'(x)^2}{F''(x)} \quad = \quad \frac{x^2}{x^2} \quad = \quad 1. \tag{2.43}$$

   Therefore, the logarithmic barrier for a ray is a self-concordant barrier with parameter $\nu = 1$ on domain $\left\{x \in \mathbb{R} \mid x > 0\right\}$.

Now, let us focus on the main properties of self-concordant barriers.

Theorem 2.3. Let $F(x)$ be a self-concordant barrier. Then the function $c^\top x + F(x)$ is a self-concordant function on $\operatorname{dom} F$.

Theorem 2.4. Let F_i be ν_i-self-concordant barriers, $i = 1, 2$. Then the function

$$F(x) \quad = \quad F_1(x) + F_2(x) \tag{2.44}$$

is a self-concordant barrier for convex set

$$\operatorname{Dom} F \quad = \quad \operatorname{Dom} F_1 \cap \operatorname{Dom} F_2 \tag{2.45}$$

with the parameter

$$\nu \quad = \quad \nu_1 + \nu_2. \tag{2.46}$$

Theorem 2.5. Let $F(x)$ be a ν-self-concordant barrier. Then for any $x \in \operatorname{Dom} F$ and $y \in \operatorname{Dom} F$ such that

$$(y - x)^\top F'(x) \quad \geq \quad 0, \tag{2.47}$$

we have

$$\|y - x\|_x \quad \leq \quad \nu + 2\sqrt{\nu}. \tag{2.48}$$

There is one special point of convex set, which is important for solving convex minimization problem. It is called the analytic center of convex set and we will focus on its properties.

Definition 2.4. Let $F(x)$ be a ν-self-concordant barrier for the set $\operatorname{Dom} F$. The point

$$x_F^* \quad = \quad \arg \min_{x \in \operatorname{Dom} F} F(x) \tag{2.49}$$

is called the analytic center of convex set $\operatorname{Dom} F$, generated by the barrier $F(x)$.

Theorem 2.6. Assume that the analytic center of a ν-self-concordant barrier $F(x)$ exists. Then for any $x \in \operatorname{Dom} F$ we have

$$\|x - x_F^*\|_{x_F^*} \quad \leq \quad \nu + 2\sqrt{\nu}. \tag{2.50}$$

This property clearly follows from Theorem 2.5 and the fact, that $F'(x_F^*) = 0$.

Thus, if $\operatorname{Dom} F$ contains no straight line, then the existence of x_F^* (which leads to nondegenerate $F''(x_F^*)$) implies, that the set $\operatorname{Dom} F$ is bounded.

Now, we describe the algorithm and its properties for obtaining an approximation to the analytic center. To find the analytic center, we need to solve the minimization problem (2.49). For that, we will use the standard implementation of the damped Newton method with termination condition

$$\|F'(x_k)\|_{x_k}^* \quad \leq \quad \beta, \text{ for } \beta \in (0, 1). \tag{2.51}$$

The pseudocode of the whole minimization process is shown in Algorithm 2.2.

Theorem 2.7. Algorithm 2.2 terminates no later than after N steps, where

$$N \quad = \quad \frac{1}{\beta - \ln(1 + \beta)} \big(F(x_0) - F(x_F^*)\big). \tag{2.52}$$

Algorithm 2.2. Damped Newton method for analytic centers.

Input:

F a ν-self-concordant barrier
$x_0 \in \operatorname{Dom} F$ a starting point
$\beta \in (0,1)$ a centering parameter

Output:

x_F^* an approximation of the analytic center of the set $\operatorname{Dom} F$

1: $k \leftarrow 0$
2: **while** $\|F'(x_k)\|_{x_k}^* > \beta$ **do**
3: $\quad x_{k+1} \leftarrow x_k - \frac{1}{1+\|F'(x_k)\|_{x_k}^*} F''(x_k)^{-1} F'(x_k)$
4: $\quad k \leftarrow k+1$
5: **end while**
6: **return** $x_F^* \leftarrow x_k$

The knowledge of analytic center allows us to solve the standard minimization problem

$$x^* \quad = \quad \arg\min_{x \in Q} c^\top x \tag{2.53}$$

with bounded closed convex set $Q \equiv \operatorname{Dom} F$, which has nonempty interior, and which is endowed with a ν-self-concordant barrier $F(x)$. Denote

$$f(t,x) \quad = \quad tc^\top x + F(x), \text{ for } t \geq 0 \tag{2.54}$$

as a parametric penalty function. Using Theorem 2.3 we can see, that $f(t,x)$ is self-concordant in x. Let us introduce new minimization problem using the parametric penalty function $f(t,x)$

$$x^*(t) \quad = \quad \arg\min_{x \in \operatorname{dom} F} f(t,x). \tag{2.55}$$

This trajectory is called the central path of the problem (2.53). We will reach the solution $x^*(t) \to x^*$ as $t \to +\infty$. Moreover, since the set Q is bounded, the analytic center x_F^* of this set exists and

$$x^*(0) \quad = \quad x_F^*. \tag{2.56}$$

From the first-order optimality condition, any point of the central path satisfies equation

$$tc + F'\big(x^*(t)\big) \quad = \quad 0 \tag{2.57}$$

Since the analytic center lies on the central path and can be found by Algorithm 2.2, all we have to do, to find the solution x^*, is to follow the central path. This enables us an approximate centering condition:

$$\|f'(t,x)\|_x^* \quad = \quad \|tc + F'(x)\|_x^* \quad \leq \quad \beta, \tag{2.58}$$

where the centering parameter β is small enough.

Assuming $x \in \operatorname{dom} F$, one iteration of the path-following algorithm consists of two steps:

$$t_+ \quad = \quad t + \frac{\gamma}{\|c\|_x^*}, \tag{2.59}$$

$$x_+ \quad = \quad x - F''(x)^{-1}\big(t_+ c + F'(x)\big). \tag{2.60}$$

Theorem 2.8. Let x satisfy the approximate centering condition (2.58)

$$\|tc + F'(x)\|_x^* \quad \leq \quad \beta \tag{2.61}$$

with $\beta < \bar{\lambda} = \frac{3-\sqrt{5}}{2}$. Then for γ, such that

$$|\gamma| \quad \leq \quad \frac{\sqrt{\beta}}{1+\sqrt{\beta}} - \beta, \tag{2.62}$$

we have again

$$\|t_+c + F'(x_+)\|_{x_+}^* \quad \leq \quad \beta. \tag{2.63}$$

This theorem ensures the correctness of the presented iteration of the path-following algorithm. For the whole description of the path-following algorithm please see Algorithm 2.3.

Algorithm 2.3. Path following algorithm.

Input:

F a ν-self-concordant barrier

$x_0 \in \operatorname{dom} F$ a starting point satisfying $\|F'(x_0)\|_{x_0}^* \leq \beta$, e.g. the analytic center x_F^* of the set $\operatorname{Dom} F$

$\beta \in (0, 1)$ a centering parameter

γ a parameter satisfying $|\gamma| \leq \frac{\sqrt{\beta}}{1+\sqrt{\beta}} - \beta$

$\varepsilon > 0$ an accuracy

Output:

x^* an optimal solution to the minimization problem (2.53)

1: $t_0 \leftarrow 0$
2: $k \leftarrow 0$
3: **while** $\varepsilon t_k < \nu + \frac{(\beta+\sqrt{\nu})\beta}{1-\beta}$ **do**
4: $\quad t_{k+1} \leftarrow t_k + \frac{\gamma}{\|c\|_{x_k}^*}$
5: $\quad x_{k+1} \leftarrow x_k - F''(x_k)^{-1}\big(t_{k+1}c + F'(x_k)\big)$
6: $\quad k \leftarrow k + 1$
7: **end while**
8: **return** $x^* \leftarrow x_k$

Theorem 2.9. Algorithm 2.3 terminates no more than after N steps, where

$$N \quad \leq \quad \mathcal{O}\left(\sqrt{\nu} \ln \frac{\nu \|c\|_{x_F^*}^*}{\varepsilon}\right). \tag{2.64}$$

The parameters β and γ in Algorithm 2.2 and Algorithm 2.3 can be fixed. The reasonable values are:

$$\beta \quad = \quad \frac{1}{9}, \tag{2.65}$$

$$\gamma \quad = \quad \frac{\sqrt{\beta}}{1+\sqrt{\beta}} - \beta \quad = \quad \frac{5}{36}. \tag{2.66}$$

The union of Algorithm 2.2 and Algorithm 2.3 can be easily used to solve the standard minimization problem (2.53), supposing we have a feasible point $x_0 \in Q$.

2.3.3. Barrier function for semidefinite programming

In this section, we are going to show, how to find a self-concordant barrier for the semidefinite program (2.4), so that we can use Algorithm 2.2 and Algorithm 2.3 to solve it. For the purpose of this section, we are interested only in the constrains of the problem. The constrains are defining us the feasibility set Q:

$$Q \quad = \quad \left\{ y \in \mathbb{R}^m \mid A_0 + \sum_{i=1}^{m} A_i y^{(i)} \succeq 0 \right\}, \tag{2.67}$$

where $A_0, \ldots, A_m \in \mathcal{S}^n$. Let us denote $X(y) = A_0 + \sum_{i=1}^{m} A_i y^{(i)}$. If the matrix $X(y)$ is block diagonal

$$X(y) \quad = \quad \begin{bmatrix} X_1(y) & 0 & \cdots & 0 \\ 0 & X_2(y) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & X_k(y) \end{bmatrix} \tag{2.68}$$

with $X_j(y) \in \mathcal{S}^{n_j}$ for $j = 1, \ldots, k$ and $\sum_{j=1}^{k} n_j = n$, then the feasibility set Q can be expressed as

$$Q \quad = \quad \left\{ y \in \mathbb{R}^m \mid X_j(y) \succeq 0,\ j = 1, \ldots, k \right\}. \tag{2.69}$$

This rule allows us to easily add or remove some constraints without touching the others and to keep the sizes of the used matrices small, which can significantly speed up the computation.

Instead of the set Q, which is parametrized by y, we can directly optimize over the set of positive semidefinite matrices. This set $\mathcal{P}^n$ is defined as

$$\mathcal{P}^n \quad = \quad \left\{ X \in \mathcal{S}^n \mid X \succeq 0 \right\} \tag{2.70}$$

and it is called the cone of positive semidefinite $n \times n$ matrices. This cone is a closed convex set, which interior is formed by positive definite matrices and on its boundary lie matrices, which have at least one eigenvalue equal to zero.

Now, we are looking for a self-concordant barrier function, which will enable us to optimize over the cone $\mathcal{P}^n$. The domain of this function needs to contain the set $\mathcal{P}^n$ and the values of the function have to be growing to $+\infty$ as we are getting closer to the boundary of the set $\mathcal{P}^n$. This will create us a repelling force from the boundary of $\mathcal{P}^n$, when following the central path (2.55). Consider the function $F(X)$ as the self-concordant barrier function for the set $\mathcal{P}^n$:

$$F(X) \quad = \quad -\ln \prod_{i=1}^{n} \lambda_i(X), \tag{2.71}$$

where $X \in \operatorname{int} \mathcal{P}^n$ and $\left\{ \lambda_i(X) \right\}_{i=1}^{n}$ is the set of eigenvalues of the matrix X. To avoid the computation of eigenvalues, the function $F(X)$ can be also expressed as:

$$F(X) \quad = \quad -\ln \det(X). \tag{2.72}$$

Theorem 2.10. Function $F(X)$ is an n-self-concordant barrier for $\mathcal{P}^n$.

Example 2.4. Consider one-dimensional problem with linear constraint $x \geq 0$. Then, the set Q is

$$Q \quad = \quad \{x \in \mathbb{R} \mid x \geq 0\} \tag{2.73}$$

and one of the barrier functions for this set Q is

$$F(x) \quad = \quad -\ln(x). \tag{2.74}$$

Then, when following the central path (2.55), the function $F(x)$ allows us to reach the boundary of Q as t grows to $+\infty$. This situation is showed in Figure 2.2 for different values of t.

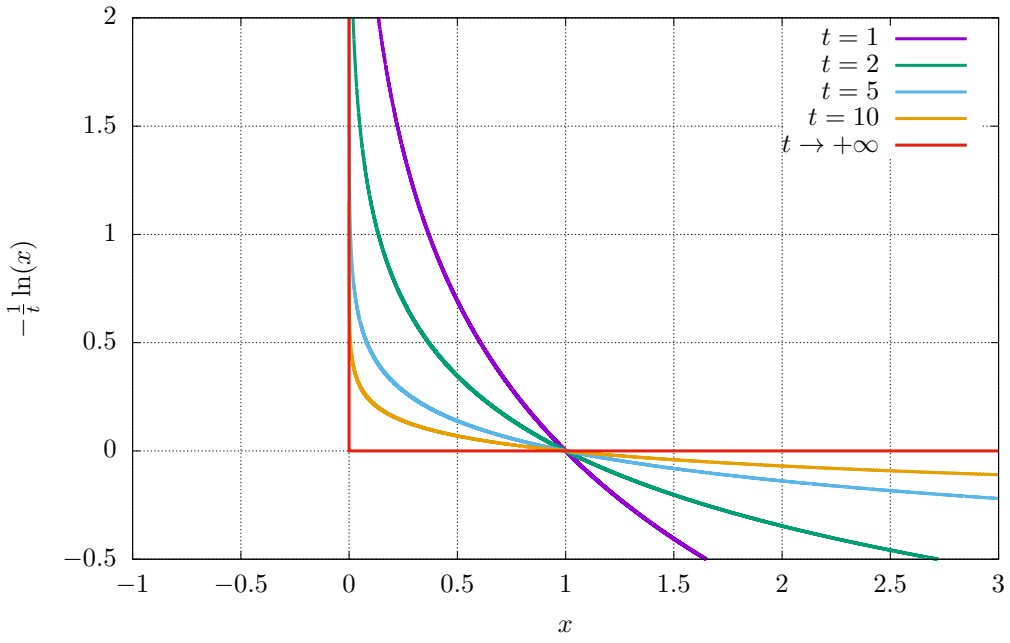


Figure 2.2. Illustration of the logarithmic barrier function for different values of t.

Note, that $\operatorname{Dom} F \supseteq \mathcal{P}^n$ because $\det(X) \geq 0$ when the number of negative eigenvalues of X is even. Therefore, the set $\operatorname{Dom} F$ is made by disjoint subsets, which one of them is $\mathcal{P}^n$. As Algorithm 2.2 and Algorithm 2.3 are interior point algorithms, when the starting point is from $\operatorname{int} \mathcal{P}^n$, then we never leave $\mathcal{P}^n$ during the execution of the algorithms and the optimal solution is found.

Similarly, the self-concordant barrier function for the set Q is a function

$$F(y) \quad = \quad -\ln \det\big(X(y)\big). \tag{2.75}$$

Example 2.5. To make it clearer, what is the difference between the set Q and $\operatorname{Dom} F(y)$, we have prepared this example. Let

$$X(y) \quad = \quad \begin{bmatrix} y_2 & y_1 \\ y_1 & y_2 \end{bmatrix}, \tag{2.76}$$

where $y = \begin{bmatrix} y_1 & y_2 \end{bmatrix}^\top$. The equation

$$z \quad = \quad \det(X(y)) \quad = \quad y_2^2 - y_1^2 \tag{2.77}$$

represents a hyperbolic paraboloid, which you can see in Figure 2.3. Therefore, the equation $z = 0$ is a slice of it, denoted by the purple color in Figure 2.4. The domain of the self-concordant barrier function is

$$\operatorname{Dom} F(y) \quad = \quad \left\{ y \mid \det\big(X(y)\big) \geq 0 \right\} \tag{2.78}$$

and is shaded by the blue color. We can see, that the set $\operatorname{Dom} F(y)$ consists of two disjoint parts. One of them is the set, where $X(y) \succeq 0$ (denoted by the orange color) and the second part is an area, where both eigenvalues of $X(y)$ are negative. Therefore, one has to pick his starting point x_0 from the interior of the set $Q = \left\{ y \in \mathbb{R}^2 \mid X(y) \succeq 0 \right\}$ to obtain the optimal solution from the set Q.

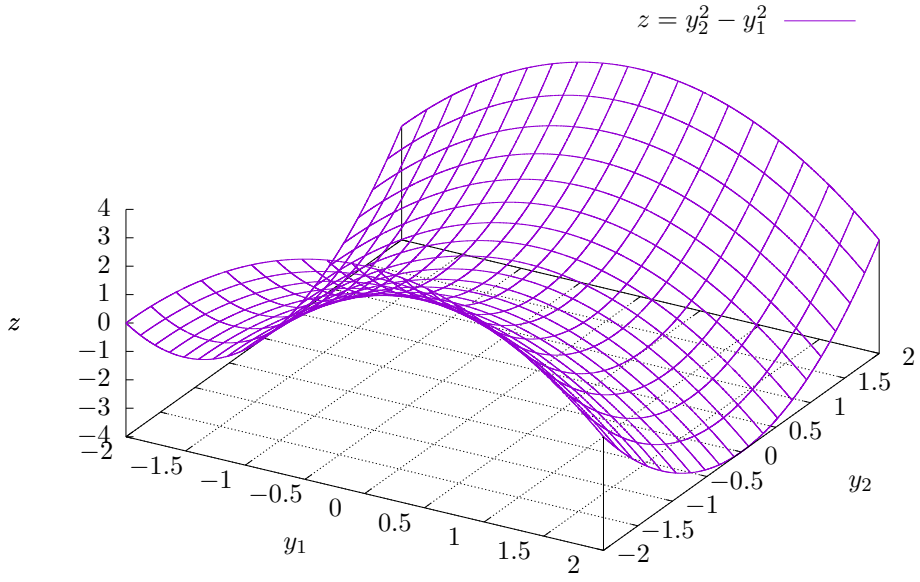


Figure 2.3. Hyperbolic paraboloid $z = y_2^2 - y_1^2$.

When the matrix X has the block diagonal form (2.68), we can rewrite the barrier function (2.75) into summation form

$$F(y) \quad = \quad -\sum_{j=1}^{k} \ln \det\big(X_j(y)\big). \tag{2.79}$$

For the purposes of Algorithm 2.2 and Algorithm 2.3, we need the first and the second partial derivatives of this function. Let us denote $X_j(y) = A_{j,0} + \sum_{i=1}^{m} A_{j,i} y^{(i)}$ for $j = 1, \ldots, k$, then the derivatives are:

$$\frac{\partial F}{\partial y^{(u)}}(y) \quad = \quad -\sum_{j=1}^{k} \operatorname{tr}\big(X_j(y)^{-1} A_{j,u}\big), \tag{2.80}$$

$$\frac{\partial^2 F}{\partial y^{(u)} \partial y^{(v)}}(y) \quad = \quad \sum_{j=1}^{k} \operatorname{tr}\Big(\big(X_j(y)^{-1} A_{j,u}\big)\big(X_j(y)^{-1} A_{j,v}\big)\Big), \tag{2.81}$$

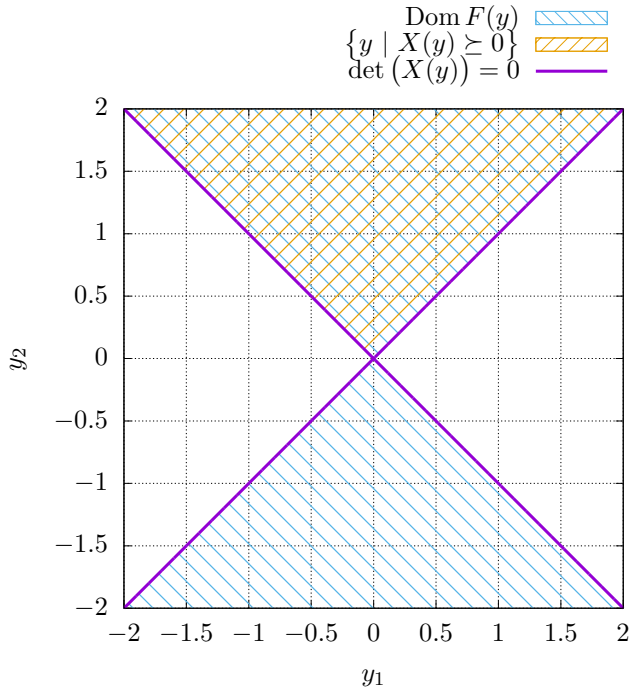


Figure 2.4. Illustration of the sets $\operatorname{Dom} F(y)$ and $\{y \mid X(y) \succeq 0\}$.

for $u, v = 1, \ldots, m$.

The computation of the derivatives is the most expensive part of each step of Algorithm 2.2 and Algorithm 2.3. Therefore, the estimated number of arithmetic operations of computation of the derivatives is also the complexity of each step in the algorithms. The number of arithmetic operations for j-th constraint in form $\{y \mid X_j(y) \succeq 0\}$ is:

- the computation of $X_j(y) = A_{j,0} + \sum_{i=1}^{m} A_{j,i} y^{(i)}$ needs mn^2 operations,
- the computation of the inversion $X_j(y)^{-1}$ needs n^3 operations,
- to compute all matrices $X_j(y)^{-1} A_{j,u}$ for $u = 1, \ldots, m$ is needed mn^3 operations,
- to compute $\operatorname{tr}\left(X_j(y)^{-1} A_{j,u}\right)$ for $u = 1, \ldots, m$ is needed mn operations,
- the computation of $\operatorname{tr}\left(\left(X_j(y)^{-1} A_{j,u}\right)\left(X_j(y)^{-1} A_{j,v}\right)\right)$ for $u, v = 1, \ldots, m$ needs $m^2 n^2$ operations.

The most expensive parts requires mn^3 and $m^2 n^2$ arithmetic operations on each constraint. Typically, the value k, the number of constraints, is small and it keeps constant, when the semidefinite programs are generated as subproblems, when solving more complex problems, e.g. polynomial optimization. Therefore, we can say, that k is constant and we can omit it from the complexity estimation. To sum up, one step of Algorithm 2.2 and Algorithm 2.3 requires

$$\mathcal{O}\big(m(m+n)n^2\big) \tag{2.82}$$

arithmetic operations.

2.4. Implementation details

To be able to study the algorithms described previously in this section, we have implemented them in the programming language Python [16]. The full knowledge of the code allows us to trace the algorithms step by step and inspect their behaviors. Instead of using some state of the art toolboxes for semidefinite programming, e.g. SeDuMi [15] and MOSEK [9], which are more or less black boxes for us, the knowledge of the used algorithms allows us to decide, if the chosen algorithm is suitable for the given semidefinite problem or not. Moreover, if we would like to create some specialized solver for some class of semidefinite problems, we can easily reuse the code, edit it as required and build the solver very quickly. On the other hand, we can not expect that our implementation will be as fast as the implementation of some state of the art toolboxes, as much more time and people was used to develop them.

The implementation is compatible with Python version 3.5 and higher. The package NumPy is used for linear algebra computations. Please refer to the installation guide of NumPy for your system to ensure, that it is correctly set to use the linear algebra libraries, e.g. LAPACK [1], ATLAS [17] and BLAS [7]. The incorrect setting of these libraries causes significant drop of the performance. Other Python packages are required as well, e.g. SymPy and SciPy, but theirs settings are not so crucial for the performance of this implementation.

2.4.1. Package installation

The package with implementation of Algorithm 2.2 and Algorithm 2.3 is named Polyopt, as the semidefinite programming part of this package is only a tool, which is used for polynomial optimization, which will be described in Section **??**. The newest version of the package is available at `http://cmp.felk.cvut.cz/~trutmpav/master-thesis/polyopt/`. To install the package on your system, you have to clone and checkout the Git repository with the source codes of the package. To install other packages that are required, the preferred way is to use the pip[1] installer. The required packages are listed in the `requirements.txt` file. Then, install the package using the script `setup.py`. For the exact commands for the whole installation process please see Listing 2.1.

Listing 2.1. Installation of the package Polyopt.

```
1: git clone https://github.com/PavelTrutman/polyopt.git
2: cd polyopt
3: pip3 install -r requirements.txt
4: python3 setup.py install
```

To check, whether the installation was successful, run command `python3 setup.py test`, which will execute the predefined tests. If no error emerges, then the package is installed and ready to use.

2.4.2. Usage

The Polyopt package is created to be able to solve semidefinite programs in a form

$$\begin{aligned} y^* \quad = \quad & \arg\min_{y \in \mathbb{R}^m} \quad c^\top y \\ & \text{s.t.} \qquad A_{j,0} + \sum_{i=1}^{m} A_{j,i} y^{(i)} \quad \succeq \quad 0 \text{ for } j = 1, \dots, k, \end{aligned} \tag{2.83}$$

[1] The PyPA recommended tool for installing Python packages. See `https://pip.pypa.io`.

where $A_{j,i} \in \mathcal{S}^{n_j}$ for $i = 0, \dots m$ and $j = 1, \dots, k$, $c \in \mathbb{R}^m$ and k is the number of constraints. In addition, a strictly feasible point $y_0 \in \mathbb{R}^m$ must be given.

The semidefinite program solver is implemented in the class `SDPSolver` of the Polyopt package. Firstly, the problem is initialized by the matrices $A_{j,i}$ and the vector c. Then, the function `solve` is called with parameter y_0 as the starting point and with the method for the analytic center estimation. A choice from two methods is available, firstly, the method `dampedNewton`, which corresponds to Algorithm 2.2, and secondly, the method `auxFollow`, which is the implementation of the Auxiliary path-following scheme [10]. However, the `auxFollow` method is unstable and it fails in some cases, and therefore it is not recommended to use. The function `solve` returns the optimal solution y^*. The minimal working example is shown in Listing 2.2.

Listing 2.2. Typical use of the class `SDPSolver` of the Polyopt package.

```
1: import polyopt
2:
3: # supposing the matrices Aij and the vectors c and y0 are already
       defined
4: problem = polyopt.SDPSolver(c, [[A10, A11, ..., A1m], ..., [Ak0,
       Ak1, ..., Akm]])
5: yStar = problem.solve(y0, problem.dampedNewton)
```

Detailed info can be printed out during the execution of the algorithm. This option can be set by `problem.setPrintOutput(True)`. Then, in each iteration of Algorithm 2.2 and Algorithm 2.3, the values k, x_k and eigenvalues of $X_j(x_k)$ are printed.

If n, the dimension of the problem, is equal to 2, boundary of the set $\operatorname{Dom} F$ (2.78) and all intermediate points x_k can be plotted. This can be enabled by setting `problem.setDrawPlot(True)`. An example of such a graph is shown in Figure 2.5.

The parameters β and γ are predefined to the same values as in (2.65) and (2.66). These parameters can be set to different values by assigning to the variables `problem.beta` and `problem.gamma` respectively. The default value for the accuracy parameter ε is 10^{-3}. This value can be changed by overwriting the variable `problem.eps`.

The function `problem.getNu()` returns the ν parameter of the self-concordant barrier function used for the problem according to Theorem 2.10. When the problem is solved, we can obtain the eigenvalues of $X(y^*)$ by calling `problem.eigenvalues()`. We should observe, that some of them are positive and some of them are zero (up to the numerical precision). The zero eigenvalues mean, that we have reached the boundary of the set Q, because the optimal solution lies always on the boundary of the set Q.

It may happen, that the set $\operatorname{Dom} F$ is not bounded, but the optimal solution can be attained. In this case, the analytic center does not exists and the proposed algorithms can not be used. By adding a constraint

$$X_{k+1}(y) \quad = \quad \begin{bmatrix} R^2 & y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ y^{(1)} & 1 & 0 & \cdots & 0 \\ y^{(2)} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y^{(m)} & 0 & 0 & \cdots & 1 \end{bmatrix} \quad \text{for } R \in \mathbb{R}, \tag{2.84}$$

we bound the set by the ball with radius R. The constraint (2.84) is equivalent to

$$\|y\|_2^2 \quad \leq \quad R^2. \tag{2.85}$$

This will make the set $\operatorname{Dom} F$ bounded and the analytic center can by found in the standard way by Algorithm 2.2. When optimizing the linear function by Algorithm 2.3, the radius R may be set too small and the optimum may be found on the boundary of the constraint (2.84). Then, the found optimum is not the solution to the original problem and the algorithm has to be run again with bigger value of R. The optimum is found on the boundary of the constraint (2.84), if at least one of the eigenvalues of $X_{k+1}(y^*)$ is zero. In our implementation, the artificial bounding constraint (2.84) can be set by `problem.bound(R)`. When the problem is solved, we can list the eigenvalues of $X_{k+1}(y^*)$ by the function `problem.eigenvalues('bounded')`.

Example 2.6. Let us present a simple example to show a detailed usage of the package Polyopt. Let us have semidefinite program in a form

$$\begin{aligned} y^* &= \arg\min_{y\in\mathbb{R}^2} \quad y^{(1)} + y^{(2)} \\ &\qquad \text{s.t.} \quad \begin{bmatrix} 1+y^{(1)} & y^{(2)} & 0 \\ y^{(2)} & 1-y^{(1)} & y^{(2)} \\ 0 & y^{(2)} & 1-y^{(1)} \end{bmatrix} \succeq 0 \end{aligned} \tag{2.86}$$

with starting point

$$y_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^\top. \tag{2.87}$$

Listing 2.3 shows the Python code used to solve the given problem. The graph of the problem is showed in Figure 2.5. The analytic center of the feasible region of the problem is

$$y_F^* = \begin{bmatrix} -0.317 & 0 \end{bmatrix}^\top, \tag{2.88}$$

the optimal solution is attained at

$$y^* = \begin{bmatrix} -0.778 & -0.592 \end{bmatrix}^\top \tag{2.89}$$

and the objective function has value -1.37. The eigenvalues of $X(y^*)$ are

$$\left\{\lambda_i\big(X(y^*)\big)\right\}_{i=1}^{3} = \{2.32 \cdot 10^{-4};\ 1.32;\ 2.45\}. \tag{2.90}$$

2.5. Comparison with the state of the art methods

Because a new implementation of a well-known algorithm was made, one should compare many properties of this implementation with the contemporary state of the art methods. For that reason, we have generated some random instances of semidefinite problems. We have solved these problems by our implementation from the Polyopt package and by selected state of the art toolboxes, namely SeDuMi [15] and MOSEK [9]. Firstly, we have verified the correctness of the implementation by checking that the optimal solution is the same as the solution obtained by SeDuMi and MOSEK for each instance of data. We have also measured execution times of all three libraries and compared them in Table 2.1 and Figure 2.6.

Listing 2.3. Code for solving semidefinite problem stated in Example 2.6.

```
from numpy import *
import polyopt

# Problem statement
# min c1*y1 + c2*y2
# s.t. A0 + A1*y1 + A2*y2 >= 0
c = array([[1], [1]])
A0 = array([[1,  0,  0],
            [0,  1,  0],
            [0,  0,  1]])
A1 = array([[1,  0,  0],
            [0, -1,  0],
            [0,  0, -1]])
A2 = array([[0,  1,  0],
            [1,  0,  1],
            [0,  1,  0]])

# starting point
y0 = array([[0], [0]])

# create the solver object
problem = polyopt.SDPSolver(c, [[A0, A1, A2]])

# enable graphs
problem.setDrawPlot(True)

# enable informative output
problem.setPrintOutput(True)

# solve!
yStar = problem.solve(y0, problem.dampedNewton)

# print eigenvalues of X(yStar)
print(problem.eigenvalues())
```

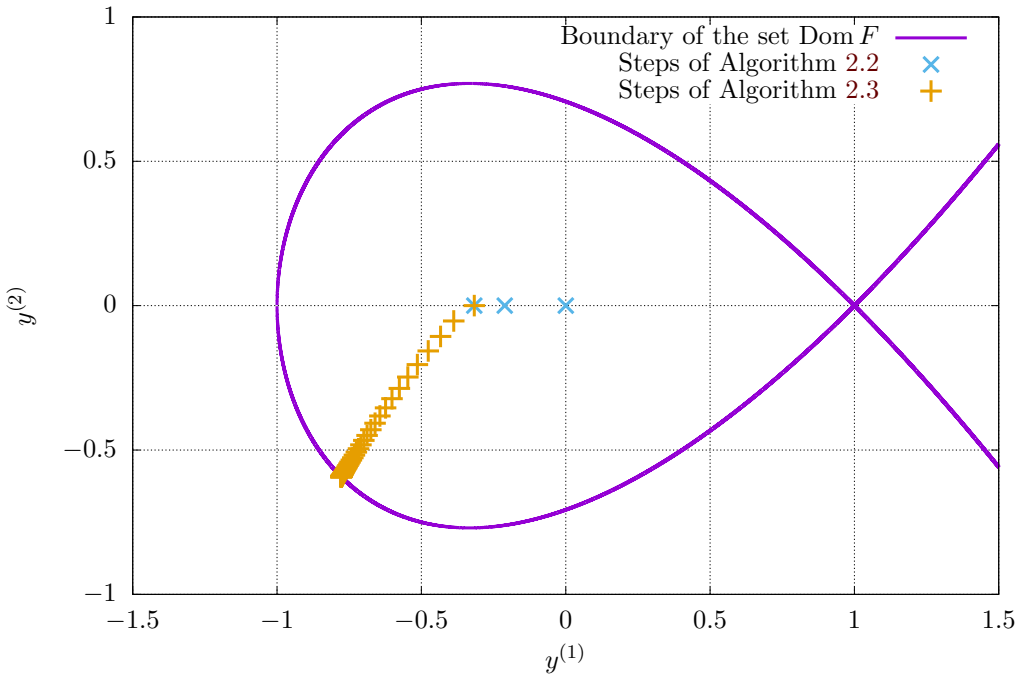


Figure 2.5. Graph of the semidefinite optimization problem stated in Example 2.6.

2.5.1. Problem description

Now, let us describe, how the random instances of the semidefinite problems were generated. From (2.82) we know that each step of Algorithm 2.2 and Algorithm 2.3 requires $m(m+n)n^2$ arithmetic operations, where m is the size of the matrices in the LMI constraint and n is the number of variables. Since in typical applications of SDP, the size of the matrices grows with the number of variables, we have set $m = n$ to have just single parameter, which we call the size of the problem.

In our experiment, we have generated 30 unique LMI constraints for each size of the problem from 1 to 20. Each unique constraint has form

$$X_{k,l}(y) \quad = \quad \mathcal{I}^k + \sum_{i=1}^{k} A_{k,l,i} y^{(i)} \tag{2.91}$$

for the size of the problem $k = 1, \ldots, 20$ and unique LMI constraint $l = 1, \ldots, 30$, where $A_{k,l,i} \in \mathcal{S}^k$. The matrices $A_{k,l,i}$ were filled with random numbers from uniform distribution $(-1; 1)$ with symmetricity of the matrices preserved. The package Polyopt requires the starting point y_0 to be given by the user in advance. But from the structure of the constraint (2.91) we can see that $y_0 \in \mathbb{R}^k$

$$y_0 \quad = \quad \begin{bmatrix} 0 & \cdots & 0 \end{bmatrix}^\top \tag{2.92}$$

is a feasible point. We used the point y_0 to initialize problems for Polyopt package, but we have let SeDuMi and MOSEK use their own initialization process. However, since the LMI constraints were randomly generated, there is no guarantee that the sets, which they define, are bounded. Therefore, we have added constraint (2.84) for $R = 10^3$, which guarantees that we are optimizing over bounded sets.

The objective function of the problem is generated randomly too. For each unique instance, we have generated random vector $r \in \mathbb{R}^n$ from uniform distribution $(-1; 1)$.

Then, the objective function to minimize is $r^\top y$. The final generated problem denoted as $P_{k,l}$ looks like

$$\begin{array}{lll} \min\limits_{y\in\mathbb{R}^k} & r_{k,l}^\top y & \\ \text{s.t.} & \mathcal{I}^k + \sum\limits_{i=1}^{k} A_{k,l,i} y^{(i)} & \succeq \quad 0 \\ & \begin{bmatrix} R^2 & y^{(1)} & y^{(2)} & \cdots & y^{(k)} \\ y^{(1)} & 1 & 0 & \cdots & 0 \\ y^{(2)} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y^{(k)} & 0 & 0 & \cdots & 1 \end{bmatrix} & \succeq \quad 0. \end{array} \tag{2.93}$$

2.5.2. Time measuring

To eliminate influences that negatively affect the execution times on CPU, such as other processes competing for the same CPU core, processor caching, data loading delays, etc., we have executed each problem $P_{k,l}$ 30 times. So, for each problem $P_{k,l}$ we have obtained execution times $\tau_{k,l,s}$ for $s = 1, \ldots, 30$. Because the influences mentioned above can only prolong the execution times, we have selected minimum of $\tau_{k,l,s}$ for each problem $P_{k,l}$.

$$\tau_{k,l} \quad = \quad \min_{s=1}^{30} \tau_{k,l,s} \tag{2.94}$$

Since the execution times of problems of the same sizes should be more or less the same, we have computed the average execution time τ_k for each size of the problem.

$$\tau_k \quad = \quad \frac{1}{30} \sum_{l=1}^{30} \tau_{k,l} \tag{2.95}$$

These execution times τ_k, where k is the size of the problem, were measured and computed separately for the Polyopt, SeDuMi and MOSEK toolboxes and are shown in Table 2.1 and Figure 2.6.

It has to be mentioned, that the Polyopt toolbox is implemented in Python, but the toolboxes SeDuMi and MOSEK were run from MATLAB with precompiled MEX files (compiled C, C++ or Fortran code) and therefore the execution times are not readily comparable. On the other side, the Python package NumPy uses common linear algebra libraries, like LAPACK [1], ATLAS [17] and BLAS [7], and we can presume that SeDuMi and MOSEK use them too.

Our intention was to measure only the execution time of the solving phase, not of the preparation time. In case of the Polyopt package, we measured the execution time of the function `solve()`. For SeDuMi and MOSEK, we have used MATLAB framework YALMIP [8] for defining the semidefinite programs and calling the solvers. The execution time of the YALMIP code is quite long, because YALMIP makes an analysis of the problem and compiles it into a standard form. Only after that, an external solver (SeDuMi or MOSEK) is called to solve the problem. Fortunately, YALMIP internally measures the execution time of the solver, so we have used this time in our statistics.

The experiments were executed on Intel Core i5-3210M CPU 2.50 GHz based computer with sufficient amount of free system memory. The installed version of Python was 3.5.3 and MATLAB R2016b 64-bit was used.

Problem size	Toolbox		
	Polyopt	**SeDuMi** [15]	**MOSEK** [9]
1	0.0212 s	0.0390 s	0.00307 s
2	0.0280 s	0.0567 s	0.00328 s
3	0.0329 s	0.0619 s	0.00348 s
4	0.0392 s	0.0690 s	0.00414 s
5	0.0451 s	0.0745 s	0.00424 s
6	0.0536 s	0.0839 s	0.00460 s
7	0.0677 s	0.0823 s	0.00488 s
8	0.0757 s	0.0799 s	0.00517 s
9	0.0910 s	0.0807 s	0.00578 s
10	0.111 s	0.0833 s	0.00678 s
11	0.135 s	0.0904 s	0.00765 s
12	0.154 s	0.0917 s	0.00805 s
13	0.179 s	0.0891 s	0.00901 s
14	0.212 s	0.0938 s	0.00937 s
15	0.241 s	0.0943 s	0.00993 s
16	0.280 s	0.0983 s	0.0101 s
17	0.299 s	0.102 s	0.0118 s
18	0.348 s	0.112 s	0.0122 s
19	0.392 s	0.113 s	0.0140 s
20	0.429 s	0.117 s	0.0148 s

Table 2.1. Execution times of different sizes of semidefinite problems solved by selected toolboxes.

2.5.3. Results

By the look of the graph in Figure 2.6, we can see that the MOSEK toolbox totally wins. The SeDuMi toolbox seems to have some constant overhead, but the execution time grows slowly with the increasing size of the problem. The Polyopt package accomplishes quite bad results compared to SeDuMi and MOSEK, especially for large sizes of problems. But this behavior was expected, as we know that the execution time should be proportional to k^4, where k is the size of the problem. However, due to SeDuMi overhead, the Polyopt package is faster than SeDuMi for problem sizes up to 8.

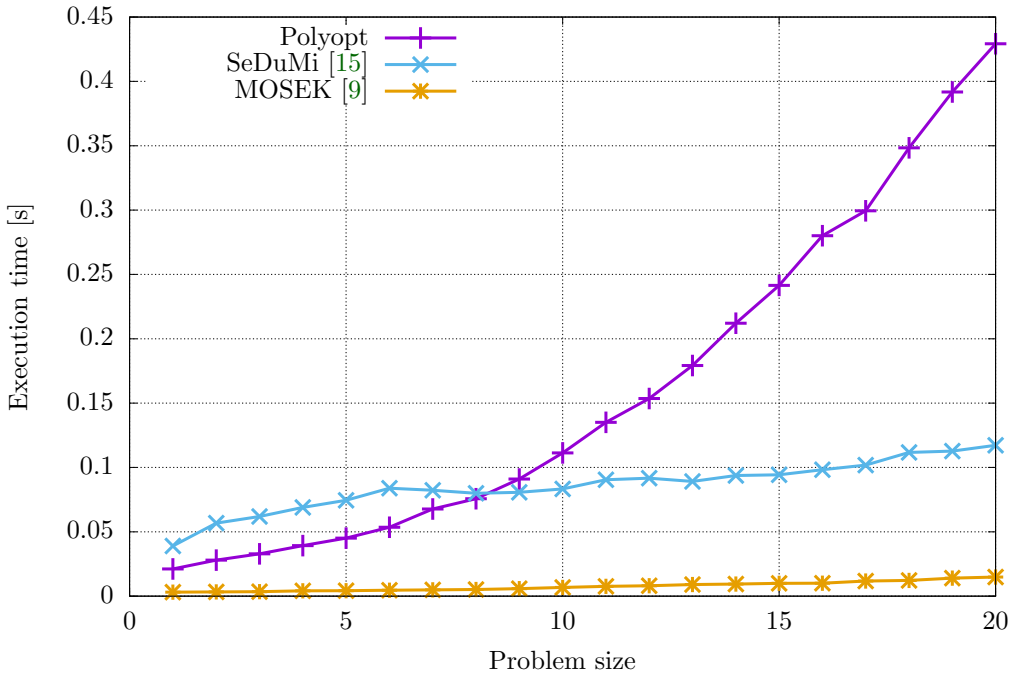


Figure 2.6. Graph of execution times based on the size of semidefinite problems solved by selected toolboxes.

3. Polynomial optimization

3.1. State of the art review

3.2. Theoretical background

3.3. Implementation details

3.4. Comparison with the state of the art methods

4. Conclusion

A. Contents of the enclosed CD

```
/
└── thesis
    └── thesis.pdf ..............digital copy of this thesis
```

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999. 19, 24

[2] Richard Bellman and Ky Fan. On systems of linear inequalities in hermitian matrix variables. In *Convexity: Proceedings of Symposia in Pure Mathematics*, volume 7, pages 1–11. American Mathematical Society Providence, 1963. 7

[3] Jane Cullum, W. E. Donath, and P. Wolfe. The minimization of certain nondifferentiable sums of eigenvalues of symmetric matrices. In *Nondifferentiable Optimization*, pages 35–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975. 7

[4] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, November 1995. 5

[5] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984. 7

[6] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972. 5

[7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979. 19, 24

[8] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. 24

[9] MOSEK ApS. *The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 28)*, 2015. http://docs.mosek.com/7.1/toolbox/index.html [Online; accessed 2017-04-25]. 8, 19, 21, 25, 26

[10] Yurii Nesterov. *Introductory lectures on convex optimization : A basic course*. Springer, 2004. 8, 20

[11] Yurii Nesterov and Arkadi Nemirovski. A general approach to polynomial-time algorithms design for convex programming. Technical report, Central Economical and Mathematical Institute, USSR Academy of Sciences, Moscow, USSR, 1988. 7

[12] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen P. Boyd. SCS: Splitting conic solver, version 1.2.6. https://github.com/cvxgrp/scs [Online; accessed 2017-04-22], April 2016. 8

[13] Michael Overton. On minimizing the maximum eigenvalue of a symmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 9:256–268, April 1988. 7

[14] Gabor Pataki. *On the multiplicity of optimal eigenvalues*. University of Michigan, Ann Arbor, MI (United States), December 1994. 7

[15] Jos F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999. 8, 19, 21, 25, 26

[16] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, 2011. 19

[17] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. http://www.cs.utsa.edu/~whaley/papers/spercw04.ps [Online; accessed 2017-04-18]. 19, 24

[18] Makoto Yamashita, Katsuki Fujisawa, Kazuhide Nakata, Maho Nakata, Mituhiro Fukuda, Kazuhiro Kobayashi, and Kazushige Goto. A high-performance software package for semidefinite programs: SDPA7. Technical report, Tokyo Japan, September 2010. 8