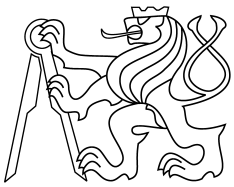
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

MASTER'S THESIS

Semidefinite Programming for Geometric Problems in Computer Vision

Pavel Trutman

pavel.trutman@fel.cvut.cz

September 1, 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

$\mathbb{C}$	Set of complex numbers.
$\mathrm{cl}(S)$	Closure of the set S.
$\deg(p)$	Total degree of the polynomial p.
$\mathrm{diag}(x)$	Diagonal matrix with components of x on the diagonal.
$\mathrm{dom}\ f$	Domain of the function f.
$\mathrm{Dom}\ f$	$\mathrm{cl}(\mathrm{dom}\ f)$.
$f'(x)$	First derivative of the function $f(x)$.
$f''(x)$	Second derivative of the function $f(x)$.
$\mathcal{I}(V)$	Vanishing ideal of the variety V.
$\mathcal{I}^n$	Identity matrix of size $n \times n$.
$\sqrt{I}$	Radical ideal of the ideal I.
$\sqrt[\mathbb{R}]{I}$	Real radical ideal of the ideal I.
$\langle f_1, f_2, \ldots, f_n \rangle$	Ideal generated by the polynomials $f_1, f_2, \ldots, f_n$.
$\mathrm{int}\ S$	Interior of the set S.
$\{\lambda_i(A)\}_{i=1}^n$	Set of all eigenvalues of the matrix $A \in \mathbb{R}^{n \times n}$.
LMI	Linear matrix inequality.
LP	Linear program.
$\mathbb{N}$	Set of natural numbers (including zero).
$\mathcal{N}_{\mathcal{B}}(f)$	Normal form of the polynomial f modulo ideal I with respect to the basis $\mathcal{B}$.
$\mathcal{P}^n$	Cone of positive semidefinite $n \times n$ matrices.
POP	Polynomial optimization.
QCQP	Quadratically constrained quadratic program.
$\mathbb{R}$	Set of real numbers.
$\mathbb{R}[x]$	Ring of polynomials with coefficients in $\mathbb{R}$ in n variables $x \in \mathbb{R}^n$.
$\mathcal{S}^n$	Space of $n \times n$ real symmetric matrices.
SDP	Semidefinite programming.
$\mathrm{tr}(A)$	Trace of the matrix A.
$\mathrm{vec}(p)$	Vector of the coefficients of the polynomial p with respect to some monomial basis.
$V_{\mathbb{C}}(I)$	Algebraic variety of the ideal I.
$V_{\mathbb{R}}(I)$	Real algebraic variety of the ideal I.
$x^{(i)}$	i-th element of the vector x.
$x^\top$	Transpose of the vector x.
$\lfloor x \rfloor$	$\max\{m \in \mathbb{Z} \mid m \leq x\}$; floor function.
$\mathcal{X}_f$	Multiplication matrix by the polynomial f.
$\mathbb{Z}$	Set of integers.

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 [11]), for which M. Goemans and D. P. Williamson created the first approximation algorithm based on SDP [7]. 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. Also systems of polynomial equations can be solved by hierarchies of semidefinite problems. This approach has the advantage that there exists a method, which allows us to compute real solutions only. Since in many applications, we are not interested in complex solutions, this method may be the right tool for polynomial systems solving. We will focus in details on SDP application on polynomial optimization and polynomial systems solving in Chapter 3.

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, $y \in \mathbb{R}^n$

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

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

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

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. $x^\top Mx \geq 0$ 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. $x^\top Mx > 0$ 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:

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

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 $y \in \mathbb{R}^m$.

$$\begin{aligned} d^* = \inf_{y \in \mathbb{R}^m} \ & b^\top y \\ \text{s.t.} \quad & \sum_{i=1}^{m} A_i y^{(i)} - C \succeq 0 \end{aligned} \tag{2.4}$$

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

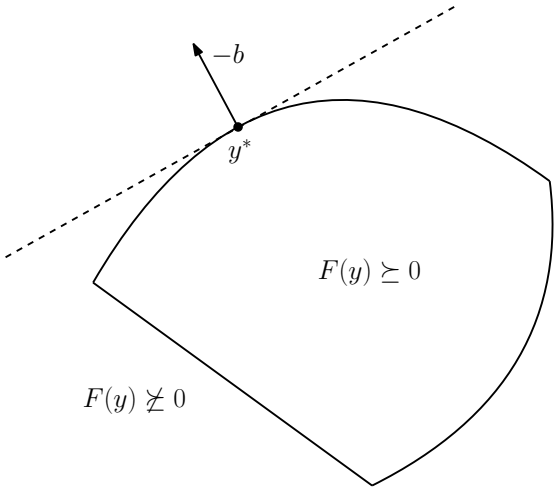


Figure 2.1. Example of a simple semidefinite problem for $y \in \mathbb{R}^2$. Boundary of the feasible set $\{y \mid F(y) \succeq 0\}$ is shown as a black curve. The minimal value of the objective function $b^\top y$ is attained at y^*.

vectors in linear programming can be replaced by LMI. Consider a linear program in a standard form

$$\begin{aligned} y^* = \arg \min_{y \in \mathbb{R}^m} \; & b^\top y \\ \text{s.t.} \quad & Ay - c \geq 0 \end{aligned} \tag{2.7}$$

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 [3] 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 [5], M. Overton [24] and G. Pataki [25]. In 1984, the interior-point methods for LPs solving were introduced by N. Karmarkar [10]. 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 [22] 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 [26] casts the standard semidefinite program into the homogeneous self-dual form, which has a trivial feasible point. SDPA [29] uses an infeasible interior-point method, which can initialized by an infeasible point. Some of the libraries (e.g. MOSEK [20]) 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 [23].

2.3. Nesterov's approach

In this section, we will follow Chapter 4 of [21] 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 [21].

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.

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

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

2. Negative logarithms.

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

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

3. Exponential functions.

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

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

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

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:

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

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

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

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

Example 2.3.

1. Linear functions.

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

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

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

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

   After substitution into (2.33) we obtain

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

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

3. Logarithmic barrier for a ray.

$$F(x) = -\ln x, \ \operatorname{dom} F = \{x \in \mathbb{R} \mid x > 0\} \tag{2.40}$$

$$F'(x) = -\frac{1}{x} \tag{2.41}$$

$$F''(x) = \frac{1}{x^2} \tag{2.42}$$

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

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

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

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

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

is a self-concordant barrier for convex set

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

with the parameter

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

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

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

we have

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

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

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

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

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

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

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

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

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** $x_F^* \leftarrow x_k$

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

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

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

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

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

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

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 $f(t,x)$

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

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

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

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

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

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:

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

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:

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

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

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

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

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

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

we have again

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

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.

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

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

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

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

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

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 $x_0 \in Q$.

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)

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

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:

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

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

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

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

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

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

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

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 $\mathcal{P}^n$:

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

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

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

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

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

and one of the barrier functions for this set Q is

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

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.

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

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

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

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

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

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

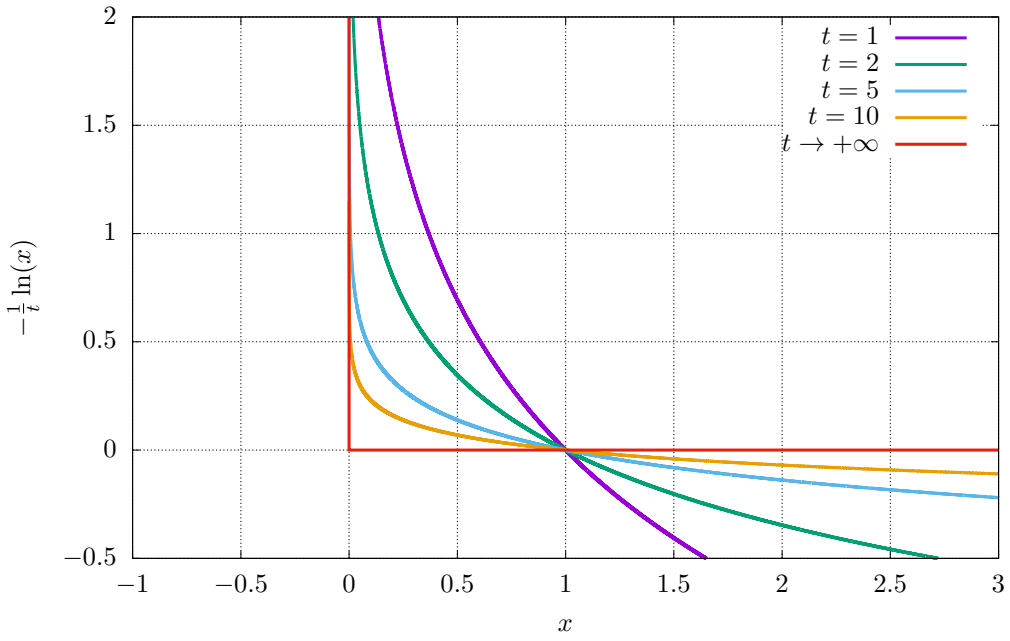


Figure 2.2. Illustration of the logarithmic barrier function for different values of t.

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

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

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.

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

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

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:

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

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

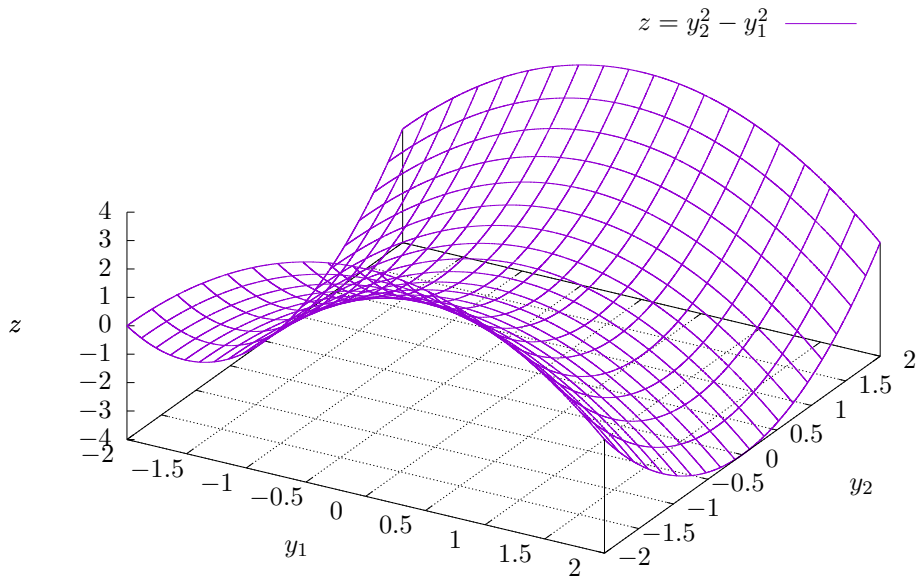


Figure 2.3. Hyperbolic paraboloid $z = y_2^2 - y_1^2$.

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.

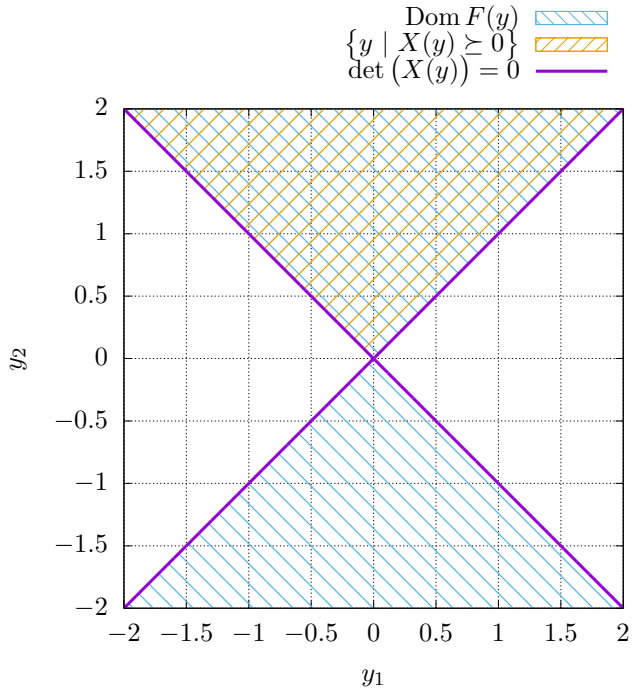


Figure 2.4. Illustration of the sets Dom $F(y)$ and $\{y \mid X(y) \succeq 0\}$.

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 [27]. 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 [26] and MOSEK [20], 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 [28] and BLAS [18]. 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 and polynomial systems solving, which will be described in Chapter 3. 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

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

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 [21]. 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 usage 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 information 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.

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

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

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

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

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

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

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

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

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

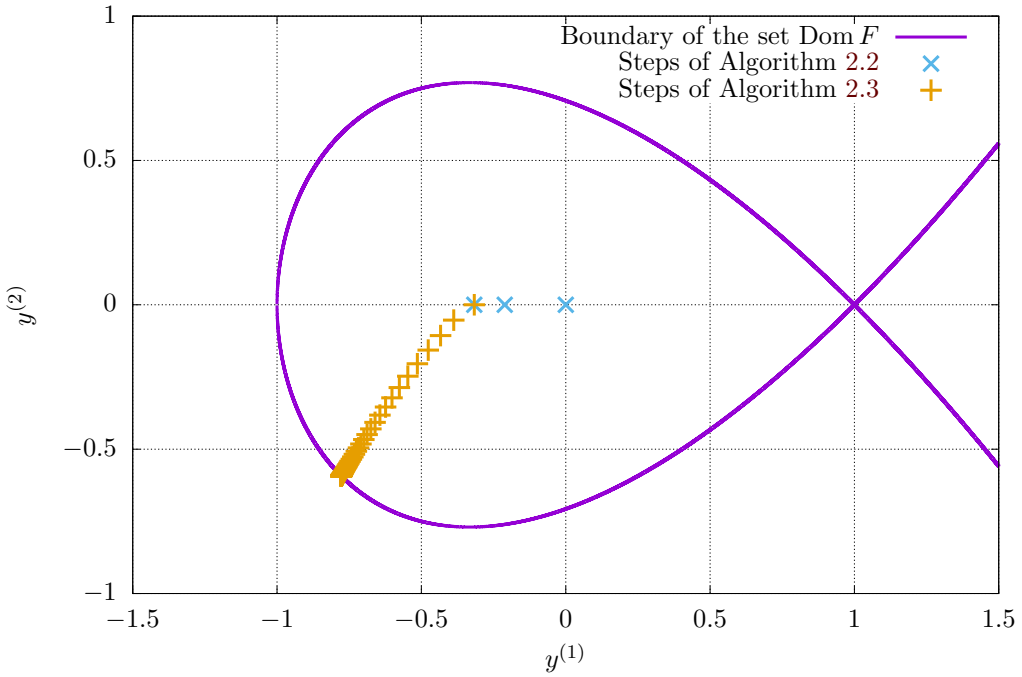


Figure 2.5. Graph of the semidefinite optimization problem stated in Example 2.6.

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 [26] and MOSEK [20]. 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.

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,

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())
```

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

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

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 $y_0 \in \mathbb{R}^k$

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

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

$$\begin{aligned} &\min_{y \in \mathbb{R}^k} r_{k,l}^\top y \\ &\text{s.t.} \qquad \mathcal{I}^k + \sum_{i=1}^{k} A_{k,l,i} y^{(i)} \succeq 0 \\ &\qquad \begin{bmatrix} R^2 & y^{(1)} & y^{(2)} & \cdots & y^{(k)} \\ y^{(1)} & 1 & 0 & \cdots & 0 \\ y^{(2)} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y^{(k)} & 0 & 0 & \cdots & 1 \end{bmatrix} \succeq 0. \end{aligned} \tag{2.93}$$

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 $P_{k,l}$.

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

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.

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

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.

Problem size	Toolbox		
	Polyopt	**SeDuMi** [26]	**MOSEK** [20]
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.

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 [28] and BLAS [18], 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 [19] 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.

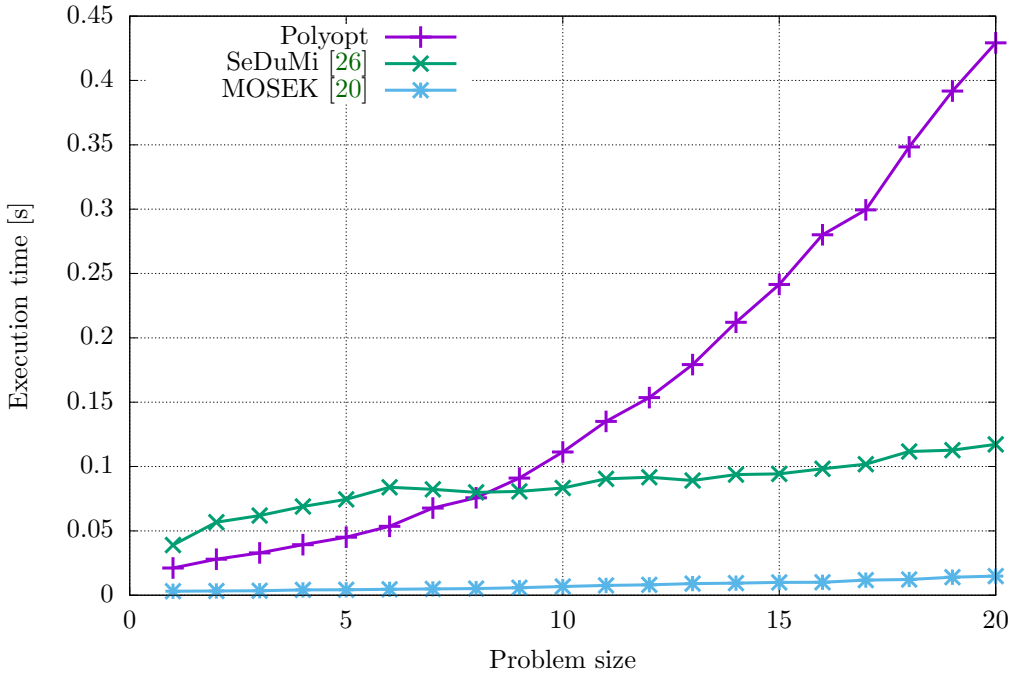


Figure 2.6. Graph of execution times based on the size of semidefinite problems solved by selected toolboxes.

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.

3. Optimization over polynomials

3.1. State of the art review

3.2. Algebraic preliminaries

In this whole chapter focused on polynomial optimization and polynomial systems solving, we will follow the notation from [4]. Just to keep this chapter self-contained, we will recall some basics of polynomial algebra.

3.2.1. The polynomial ring, ideals and varieties

Firstly, the ring of multivariate polynomials in n variables with coefficients in $\mathbb{R}$ is denoted as $\mathbb{R}[x]$, where $x = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^\top$. For $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{N}$, x^α denotes the monomial ${x_1}^{\alpha_1} \cdot {x_2}^{\alpha_2} \cdot \cdots \cdot {x_n}^{\alpha_n}$, with total degree $|\alpha| = \sum_{i=1}^{n} \alpha_i$, where $\alpha = \begin{bmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_n \end{bmatrix}^\top$. A polynomial $p \in \mathbb{R}[x]$ can be written as

$$p = \sum_{\alpha \in \mathbb{N}^n} p_\alpha x^\alpha \tag{3.1}$$

with total degree $\deg(p) = \max_{\alpha \in \mathbb{N}^n} |\alpha|$ for non-zero coefficients $p_\alpha \in \mathbb{R}$.

A linear subspace $I \subseteq \mathbb{R}[x]$ is an ideal if $p \in I$ and $q \in \mathbb{R}[x]$ implies $pq \in I$. Let $f_1, f_2, \ldots, f_m$ be polynomials in $\mathbb{R}[x]$. Then the set

$$\langle f_1, f_2, \ldots, f_m \rangle = \left\{ \sum_{j=1}^{m} h_j f_j \mid h_1, h_2, \ldots, h_m \in \mathbb{R}[x] \right\} \tag{3.2}$$

is called the ideal generated by $f_1, f_2, \ldots, f_m$. Given the ideal $I \in \mathbb{R}[x]$, the algebraic variety of I is the set

$$V_{\mathbb{C}}(I) = \big\{ x \in \mathbb{C}^n \mid f(x) = 0 \text{ for all } f \in I \big\} \tag{3.3}$$

and its real variety is

$$V_{\mathbb{R}}(I) = V_{\mathbb{C}}(I) \cap \mathbb{R}^n. \tag{3.4}$$

The ideal I is said to be zero-dimensional when its complex variety $V_{\mathbb{C}}(I)$ is finite. The vanishing ideal of a subset $V \subseteq \mathbb{C}^n$ is the ideal

$$\mathcal{I}(V) = \big\{ f \in \mathbb{R}[x] \mid f(x) = 0 \text{ for all } x \in V \big\}. \tag{3.5}$$

The radical ideal of the ideal $I \subseteq \mathbb{R}[x]$ is the ideal

$$\sqrt{I} = \big\{ f \in \mathbb{R}[x] \mid f^m \in I \text{ for some } m \in \mathbb{Z}^+ \big\}. \tag{3.6}$$

The real radical ideal of the ideal $I \subseteq \mathbb{R}[x]$ is the ideal

$$\sqrt[\mathbb{R}]{I} = \big\{ f \in \mathbb{R}[x] \mid f^{2m} + \sum_j h_j^2 \in I \text{ for some } h_j \in \mathbb{R}[x], m \in \mathbb{Z}^+ \big\}. \tag{3.7}$$

The following two theorems are stating the relations between the vanishing and (real) radical ideals.

Theorem 3.1 (Hilbert's Nullstellensatz). Let $I \in \mathbb{R}[x]$ be an ideal. The radical ideal of I is equal to the vanishing ideal of its variety, i.e.

$$\sqrt{I} = \mathcal{I}\big(V_{\mathbb{C}}(I)\big). \tag{3.8}$$

Theorem 3.2 (Real Nullstellensatz). Let $I \in \mathbb{R}[x]$ be an ideal. The real radical ideal of I is equal to the vanishing ideal of its real variety, i.e.

$$\sqrt[\mathbb{R}]{I} = \mathcal{I}\big(V_{\mathbb{R}}(I)\big). \tag{3.9}$$

The quotient ring $\mathbb{R}[x]/I$ is the set of all equivalence classes of polynomials in $\mathbb{R}[x]$ for congruence modulo ideal I

$$\mathbb{R}[x]/I = \big\{[f] \mid f \in \mathbb{R}[x]\big\}, \tag{3.10}$$

where the equivalence class $[f]$ is

$$[f] = \big\{f + g \mid q \in I\big\}. \tag{3.11}$$

Because $\mathbb{R}[x]/I$ is a ring, it is equipped with addition and multiplication on the equivalence classes:

$$[f] + [g] = [f + g] \tag{3.12}$$
$$[f][g] = [fg] \tag{3.13}$$

for $f, g \in \mathbb{R}[x]$.

For zero-dimensional ideal I, there is a relation between the dimension of $\mathbb{R}[x]/I$ and the cardinality of the variety $V_{\mathbb{C}}(I)$:

$$|V_{\mathbb{C}}(I)| \leq \dim\big(\mathbb{R}[x]/I\big). \tag{3.14}$$

Moreover, if I is a radical ideal, then

$$|V_{\mathbb{C}}(I)| = \dim\big(\mathbb{R}[x]/I\big). \tag{3.15}$$

Assume that the number of complex roots is finite and let $N = \dim\big(\mathbb{R}[x]/I\big)$, and therefore $|V_{\mathbb{C}}(I)| \leq N$. Consider a set $\mathcal{B} = \{b_1, b_2, \ldots, b_N\} \subseteq \mathbb{R}[x]$ for which the equivalence classes $[b_1], [b_2], \ldots, [b_N]$ are pairwise distinct and $\big\{[b_1], [b_2], \ldots, [b_N]\big\}$ is a basis of $\mathbb{R}[x]/I$. Then every polynomial $f \in \mathbb{R}[x]$ can be written in unique way as

$$f = \sum_{i=1}^{N} c_i b_i + p, \tag{3.16}$$

where $c_i \in \mathbb{R}$ and $p \in I$. The normal form of the polynomial f modulo I with respect to the basis $\mathcal{B}$ is the polynomial

$$\mathcal{N}_{\mathcal{B}}(f) = \sum_{i=1}^{N} c_i b_i. \tag{3.17}$$

3.2.2. Solving systems of polynomial equations by multiplication matrices

Systems of polynomial equations can be solved by computing eigenvalues and eigenvectors of so called multiplication matrices. Given $f \in \mathbb{R}[x]$, we define the multiplication operator (by f) $\mathcal{X}_f : \mathbb{R}[x]/I \to \mathbb{R}[x]/I$ as

$$\mathcal{X}_f([g]) = [f][g] = [fg]. \tag{3.18}$$

It can be shown that $\mathcal{X}_f$ is a linear mapping, and therefore can be represented by its matrix with respect to the basis $\mathcal{B}$ of $\mathbb{R}[x]/I$. For simplicity, we again denote this matrix $\mathcal{X}_f$ and it is called the multiplication matrix by f. When $\mathcal{B} = \{b_1, b_2, \ldots, b_N\}$ and we set $\mathcal{N}_{\mathcal{B}}(fb_j) = \sum_{i=1}^{N} a_{i,j} b_i$ for $a_{ij} \in \mathbb{R}$, then the multiplication matrix is

$$\mathcal{X}_f = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{bmatrix}. \tag{3.19}$$

Theorem 3.3 (Stickelberger theorem). Let I be a zero-dimensional ideal in $\mathbb{R}[x]$, let $\mathcal{B} = \{b_1, b_2, \ldots, b_N\}$ be a basis of $\mathbb{R}[x]/I$, and let $f \in \mathbb{R}[x]$. The eigenvalues of the multiplication matrix $\mathcal{X}_f$ are the evaluations $f(v)$ of the polynomial f at the points $v \in V_{\mathbb{C}}(I)$. Moreover, for all $v \in V_{\mathbb{C}}(I)$,

$$(\mathcal{X}_f)^\top [v]_{\mathcal{B}} = f(v)[v]_{\mathcal{B}}, \tag{3.20}$$

setting $[v]_{\mathcal{B}} = \begin{bmatrix} b_1(v) & b_2(v) & \cdots & b_N(v) \end{bmatrix}^\top$; that is, the vector $[v]_{\mathcal{B}}$ is a left eigenvector with eigenvalue $f(v)$ of the multiplication matrix $\mathcal{X}_f$.

Therefore, we can create the multiplication matrix $\mathcal{X}_{x_i}$ for the variable x_i and then the eigenvalues of $\mathcal{X}_{x_i}$ correspond to the x_i-coordinates of the points $V_{\mathbb{C}}(I)$. This means that the solutions of the whole system can be found by computing eigenvalues $\lambda_{x_i} = \{\lambda_j(\mathcal{X}_{x_i})\}_{j=1}^{N}$ of the multiplication matrix $\mathcal{X}_{x_i}$ for all variables x_i. Then $V_{\mathbb{C}}(I)$ is a subset of the Cartesian product $\lambda_{x_1} \times \lambda_{x_2} \times \cdots \times \lambda_{x_n}$ and one has to select only the points that are solutions. However, this method becomes inefficient for large n, the number of variables, since n multiplication matrices have to be constructed and their eigenvalues computed.

For this reason, the second property of multiplication matrices is used. The roots can be recovered from left eigenvectors of $\mathcal{X}_f$ when all left eigenspaces of $\mathcal{X}_f$ have dimension one. This is the case when the values $f(v)$ for $v \in V_{\mathbb{C}}(I)$ are pairwise distinct and when the ideal I is radical. In that case, each left eigenvector of $\mathcal{X}_f$ corresponds to one solution $v \in V_{\mathbb{C}}(I)$ and the values of the eigenvectors are the evaluations $b_i(v)$ for $b_i \in \mathcal{B}$, and therefore when the variable $x_i \in \mathcal{B}$, we can readily obtain its value.

Example 3.1. Let us have a system of two polynomial equations.

$$-20x^2 + xy - 12y^2 - 16x - y + 48 = 0 \tag{3.21}$$

$$12x^2 - 58xy + 3y^2 + 46x - 47y + 44 = 0 \tag{3.22}$$

The first equation represents an ellipse and the second one a hyperbola as you can see in Figure 3.1. Let us solve the system using multiplication matrices.

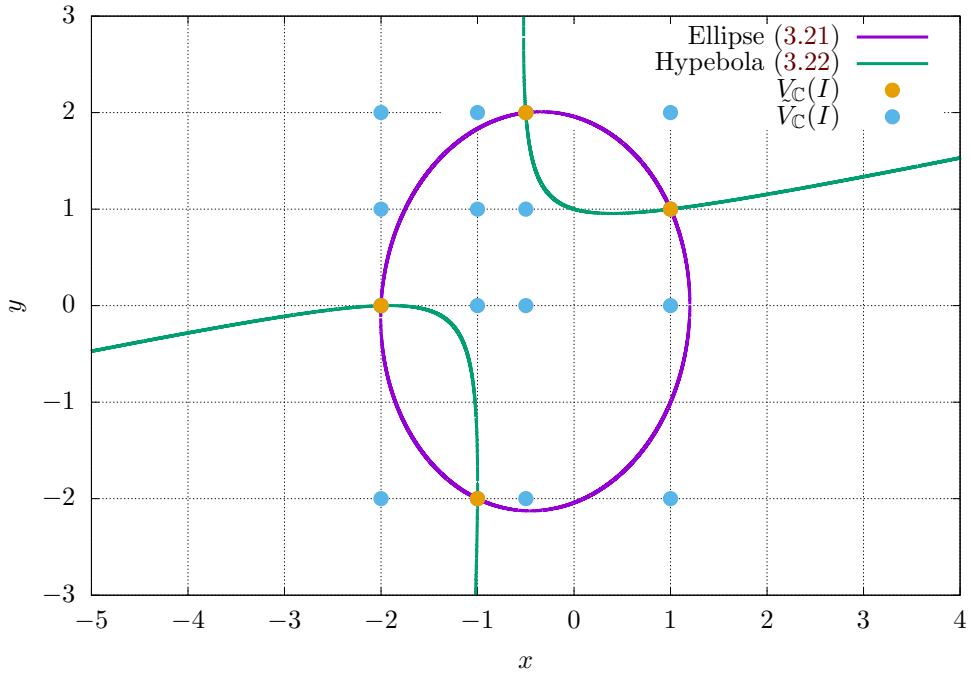


Figure 3.1. The intersection of the ellipse (3.21) and the hyperbola (3.22) with solutions found by the eigenvalue and the eigenvector methods using multiplication matrices.

First of all, we have to compute the Gröbner basis [2] of the ideal, for example using the F_4 Algorithm [6]. We have got the following basis:

$$164x^2 + 99y^2 + 126x + 15y - 404, \tag{3.23}$$

$$41xy + 3y^2 - 16x + 34y - 52, \tag{3.24}$$

$$41y^3 - 15y^2 + 48x - 170y + 96. \tag{3.25}$$

Now, we can select the monomial basis $\mathcal{B}$

$$\mathcal{B} = \begin{bmatrix} 1 & y & x & y^2 \end{bmatrix}^\top \tag{3.26}$$

and construct the multiplication matrices $\mathcal{X}_x$ and $\mathcal{X}_y$ accordingly, knowing that

$$\mathcal{X}_x\Big(\big[1\big]\Big) = \big[x\big] = 1\big[x\big], \tag{3.27}$$

$$\mathcal{X}_x\Big(\big[y\big]\Big) = \big[xy\big] = -\frac{3}{41}\big[y^2\big] + \frac{26}{41}\big[x\big] - \frac{34}{41}\big[y\big] + \frac{52}{41}\big[1\big], \tag{3.28}$$

$$\mathcal{X}_x\Big(\big[x\big]\Big) = \big[x^2\big] = -\frac{99}{164}\big[y^2\big] - \frac{63}{82}\big[x\big] - \frac{15}{164}\big[y\big] + \frac{101}{41}\big[1\big], \tag{3.29}$$

$$\mathcal{X}_x\Big(\big[y^2\big]\Big) = \big[xy^2\big] = -\frac{37}{41}\big[y^2\big] + \frac{20}{41}\big[x\big] + \frac{18}{41}\big[y\big] + \frac{40}{41}\big[1\big], \tag{3.30}$$

and

$$\mathcal{X}_y\Big(\big[1\big]\Big) = \big[y\big] = 1\big[y\big], \tag{3.31}$$

$$\mathcal{X}_y\Big(\big[y\big]\Big) = \big[y^2\big] = 1\big[y^2\big], \tag{3.32}$$

$$\mathcal{X}_y\Big(\big[x\big]\Big) = \big[xy\big] = -\frac{3}{41}\big[y^2\big] + \frac{26}{41}\big[x\big] - \frac{34}{41}\big[y\big] + \frac{52}{41}\big[1\big], \tag{3.33}$$

$$\mathcal{X}_y\Big(\big[y^2\big]\Big) = \big[y^3\big] = \frac{15}{41}\big[y^2\big] - \frac{48}{41}\big[x\big] + \frac{170}{41}\big[y\big] - \frac{96}{41}\big[1\big]. \tag{3.34}$$

Then, the multiplication matrices are:

$$\mathcal{X}_x = \begin{bmatrix} 0 & \frac{52}{41} & \frac{101}{41} & \frac{40}{41} \\ 0 & -\frac{34}{41} & -\frac{15}{164} & \frac{18}{41} \\ 1 & \frac{26}{41} & -\frac{63}{82} & \frac{20}{41} \\ 0 & -\frac{3}{41} & -\frac{99}{164} & -\frac{37}{41} \end{bmatrix}, \tag{3.35}$$

$$\mathcal{X}_y = \begin{bmatrix} 0 & 0 & \frac{52}{41} & -\frac{96}{41} \\ 1 & 0 & -\frac{34}{41} & \frac{170}{41} \\ 0 & 0 & \frac{26}{41} & -\frac{48}{41} \\ 0 & 1 & -\frac{3}{41} & \frac{15}{41} \end{bmatrix}. \tag{3.36}$$

The eigenvalues of $\mathcal{X}_x$ and $\mathcal{X}_y$ are

$$\left\{\lambda_i(\mathcal{X}_x)\right\}_{i=1}^4 = \left\{-2;\ -1;\ -\frac{1}{2};\ 1\right\}, \tag{3.37}$$

$$\left\{\lambda_i(\mathcal{X}_y)\right\}_{i=1}^4 = \{-2;\ 0;\ 1;\ 2\}. \tag{3.38}$$

Therefore, there are $4 \times 4 = 16$ possible solutions to the system and we must verify, which of them are true solutions. Let us denote the set of all possible solutions as $\tilde{V}_{\mathbb{C}}(I)$. These possible solutions are shown in Figure 3.1 by the blue color.

Secondly, we compute the left eigenvectors of the multiplication matrix $\mathcal{X}_x$ such that their first coordinates are ones, as it corresponds to the constant polynomial $b_1 = 1$. We obtain following four eigenvectors corresponding to four different solutions:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -2 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ -\frac{1}{2} \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \\ -1 \\ 4 \end{bmatrix}. \tag{3.39}$$

Since the second and the third coordinate corresponds to $b_2 = y$ and $b_3 = x$ respectively, we have got four solutions to the system of polynomials (3.21) and (3.22):

$$V_{\mathbb{C}}(I) = \left\{ \begin{bmatrix} 1 \\ 1 \end{bmatrix}; \begin{bmatrix} -2 \\ 0 \end{bmatrix}; \begin{bmatrix} -\frac{1}{2} \\ 2 \end{bmatrix}; \begin{bmatrix} -1 \\ -2 \end{bmatrix} \right\}. \tag{3.40}$$

These solutions are shown by the orange color in Figure 3.1.

3.3. Moment matrices

Polynomial optimization and solving systems of polynomial equations via hierarchies of semidefinite programs is based on the theory of measures and moments. But to keep the scope simple, we will avoid to introduce this theory. However, since it provides better understanding of the matter, interested reader may look into [16]. Moreover, we will introduce the only minimal basics to be able to proceed with polynomial optimization and polynomial systems solving. More and detailed information can be found in [16] too.

Now, let us start with the theory about moment matrices, which are crucial for application of SDP on polynomial optimization and polynomial systems solving. Recall that a polynomial has a form (3.1). Since such a polynomial may have infinite number of coefficients, let us introduce a polynomial $p \in \mathbb{R}[x]$ of the degree $d \in \mathbb{N}$:

$$p(x) = \sum_{\alpha \in \mathbb{N}_d^n} p_\alpha x^\alpha, \tag{3.41}$$

where $\mathbb{N}_d$ are natural numbers (including zero) up to the number d. This polynomial has at most $\binom{n+d}{n}$ non-zero coefficients, since there are $\binom{n+d}{n}$ monomials in n variables up to degree d. We will use the notation $\mathrm{vec}(p)$ for the vector of the coefficients of the polynomial p with respect to some monomial basis $\mathcal{B}$:

$$\mathrm{vec}(p)^{(\alpha)} = p_\alpha \tag{3.42}$$

for $\alpha \in \mathbb{N}_d^n$.

Definition 3.1 (Riesz functional). Given a sequence $y^{(\alpha)} = y_\alpha$ for $\alpha \in \mathbb{N}^n$, we define the Riesz linear functional $\ell_y : \mathbb{R}[x] \to \mathbb{R}$ such that

$$\ell_y(x^\alpha) = y_\alpha \tag{3.43}$$

for all $\alpha \in \mathbb{N}^n$.

The linearity of the Riesz functional allows us to apply it on polynomials.

$$\ell_y\big(p(x)\big) = \ell_y\left(\sum_{\alpha\in\mathbb{N}_d^n} p_\alpha x^\alpha\right) = \sum_{\alpha\in\mathbb{N}_d^n} p_\alpha \ell_y(x^\alpha) = \sum_{\alpha\in\mathbb{N}_d^n} p_\alpha y_\alpha \tag{3.44}$$

From the equation above, we can see that Riesz functional substitutes a new variable y_α for each monomial x^α, and therefore we can interpret the Riesz functional as an operator that linearizes polynomials.

Example 3.2. Given polynomial $p \in \mathbb{R}[x_1, x_2]$

$$p(x) = x_1^2 + 3x_1x_2 - 7x_2 + 9 \tag{3.45}$$

with $\deg(p) = 2$, the vector of its coefficients with respect to monomial basis

$$\mathcal{B} = \begin{bmatrix} x_1^2 & x_1x_2 & x_2^2 & x_1 & x_2 & 1 \end{bmatrix}^\top \tag{3.46}$$

is

$$\mathrm{vec}(p) = \begin{bmatrix} 1 & 3 & 0 & 0 & -7 & 9 \end{bmatrix}^\top. \tag{3.47}$$

The Riesz functional of $p(x)$ is

$$\ell\big(p(x)\big) = y_{20} + 3y_{11} - 7y_{01} + 9y_{00}. \tag{3.48}$$

Definition 3.2 (Moment matrix). A symmetric matrix M indexed by $\mathbb{N}^n$ is said to be a moment matrix (or generalized Hankel matrix) if its (α, β)-entry depends only on the sum $\alpha + \beta$ of the indices. Given sequence $y^{(\alpha)} = y_\alpha$ for $\alpha \in \mathbb{N}^n$, the moment matrix $M(y)$ has form

$$M(y)^{(\alpha,\beta)} = y_{\alpha+\beta} \tag{3.49}$$

for $\alpha, \beta \in \mathbb{N}^n$.

Definition 3.3 (Truncated moment matrix). Given sequence $y^{(\alpha)} = y_\alpha$ for $\alpha \in \mathbb{N}^n$, the truncated moment matrix $M_s(y)$ of order $s \in \mathbb{N}$ has form

$$M_s(y)^{(\alpha,\beta)} = y_{\alpha+\beta} \tag{3.50}$$

for $\alpha, \beta \in \mathbb{N}_s^n$.

The moment matrices are linear in y and symmetric, we can see that

$$M_s(y) \in \mathcal{S}^{\binom{n+s}{n}} \tag{3.51}$$

since $\binom{n+s}{n}$ is the number of monomials in n variables up to degree s.

Example 3.3. For $n = 2$ the moment matrices for different orders are:

$$M_0(y) = \begin{bmatrix} y_{00} \end{bmatrix}, \tag{3.52}$$

$$M_1(y) = \left[\begin{array}{c|cc} y_{00} & y_{10} & y_{01} \\ \hline y_{10} & y_{20} & y_{11} \\ y_{01} & y_{11} & y_{02} \end{array}\right], \tag{3.53}$$

$$M_2(y) = \left[\begin{array}{cc|ccc} y_{00} & y_{10} & y_{01} & y_{20} & y_{11} & y_{02} \\ y_{10} & y_{20} & y_{11} & y_{30} & y_{21} & y_{12} \\ \hline y_{01} & y_{11} & y_{02} & y_{21} & y_{12} & y_{03} \\ \hline y_{20} & y_{30} & y_{21} & y_{40} & y_{31} & y_{22} \\ y_{11} & y_{21} & y_{12} & y_{31} & y_{22} & y_{13} \\ y_{02} & y_{12} & y_{03} & y_{22} & y_{13} & y_{04}. \end{array}\right] \tag{3.54}$$

All the elements in the blocks separated by the dashed lines have the same degree. Moreover, we can see that the moment matrices of smaller order are nothing more than submatrices of the moment matrices of bigger order.

And just one example for $n = 3$:

$$M_1(y) = \left[\begin{array}{c|ccc} y_{000} & y_{100} & y_{010} & y_{001} \\ \hline y_{100} & y_{200} & y_{110} & y_{101} \\ y_{010} & y_{110} & y_{020} & y_{011} \\ y_{001} & y_{101} & y_{011} & y_{002} \end{array}\right]. \tag{3.55}$$

Definition 3.4 (Localizing matrix). Given sequence $y^{(\alpha)} = y_\alpha$ for $\alpha \in \mathbb{N}^n$ and polynomial $q(x) \in \mathbb{R}[x]$, its localizing matrix $M_s(qy)$ of order s has form

$$M_s(qy)^{(\alpha,\beta)} = \sum_\gamma q_\gamma y_{\alpha+\beta+\gamma} \tag{3.56}$$

for $\alpha, \beta \in \mathbb{N}^n_s$.

Notation $M_s(qy)$ emphasis that the localizing matrix is bilinear in q and y.

Example 3.4. For $n = 2$ and a polynomial $q(x) = x_1x_2 + 2x_1 + 3$, the localizing matrix is

$$M_1(qy) = \left[\begin{array}{c|cc} y_{11} + 2y_{10} + 3y_{00} & y_{21} + 2y_{20} + 3y_{10} & y_{12} + 2y_{11} + 3y_{01} \\ \hline y_{21} + 2y_{20} + 3y_{10} & y_{31} + 2y_{30} + 3y_{20} & y_{22} + 2y_{21} + 3y_{11} \\ y_{12} + 2y_{11} + 3y_{01} & y_{22} + 2y_{21} + 3y_{11} & y_{13} + 2y_{12} + 3y_{02} \end{array}\right]. \tag{3.57}$$

3.4. Polynomial optimization

The task of the polynomial optimization (POP) is to optimize a polynomial function on a set, which is given by a set of polynomial inequalities. For given polynomials

$p_0, \ldots, p_m \in \mathbb{R}[x]$, we can define a standard polynomial optimization problem in a form (3.58).

$$\begin{aligned} p^* = \min_{x \in \mathbb{R}^n} \; & p_0(x) \\ \text{s.t.} \quad & p_k(x) \geq 0 \quad (k = 1, \ldots, m) \end{aligned} \tag{3.58}$$

Let the feasibility set P of the optimization problem (3.58) be compact (closed and bounded) basic semialgebraic set, defined as

$$P = \big\{ x \in \mathbb{R}^n \mid p_k(x) \geq 0, \; k = 1, \ldots, m \big\}. \tag{3.59}$$

Since the set P is compact, the minimum p^* is attained at a point $x^* \in P$. On the other hand, we do not assume convexity of neither the polynomial p_0 nor the set P, and therefore the problem (3.58) may have several local minima and several global minima in general case. We are, of course, interested in the global minimum only.

3.4.1. Lasserre's LMI hierarchy

The global minimum can be found by hierarchies of semidefinite programs. This was introduced by J. B. Lasserre in [13]. He has shown that the polynomial optimization problem (3.58) can equivalently written as the following semidefinite program (3.60).

$$\begin{aligned} p^* = \inf_{y \in \mathbb{R}^{\mathbb{N}^n}} \; & \sum_{\alpha \in \mathbb{N}^n} p_{0\alpha} y_\alpha \\ \text{s.t.} \quad & y_0 = 1 \\ & M(y) \succeq 0 \\ & M(p_k y) \succeq 0 \quad (k = 1, \ldots, m) \end{aligned} \tag{3.60}$$

This infinite-dimensional semidefinite program is not solvable by computers, and therefore consider Lasserre's LMI hierarchy (3.61) for relaxation order $r \in \mathbb{N}$.

$$\begin{aligned} p_r^* = \inf_{y \in \mathbb{R}^{\mathbb{N}^n_{2r}}} \; & \sum_{\alpha \in \mathbb{N}^n_{2r}} p_{0\alpha} y_\alpha \\ \text{s.t.} \quad & y_0 = 1 \\ & M_r(y) \succeq 0 \\ & M_{r-r_k}(p_k y) \succeq 0 \quad (k = 1, \ldots, m) \end{aligned} \tag{3.61}$$

Where $r_k = \left\lceil \frac{\deg(p_k)}{2} \right\rceil$ and $r \geq \max\{r_1, \ldots, r_m\}$. The semidefinite program (3.61) is a relaxed version of the program (3.60) or of the initial polynomial optimization problem (3.58).

Theorem 3.4 (Lasserre's LMI hierarchy converges [8]). For $r \in \mathbb{N}$ holds

$$p_r^* \leq p_{r+1}^* \leq p^* \tag{3.62}$$

and

$$\lim_{r \to +\infty} p_r^* = p^*. \tag{3.63}$$

The semidefinite program (3.61) can be solved by the state of the art semidefinite program solvers or by the Polyopt package as described in Section 2.4. Solving the relaxed semidefinite programs for increasing relaxation order r gives us tighter and tighter lower bounds on the global minimum of the original problem (3.58).

Theorem 3.5 (Generic finite convergence [8]). In the finite-dimensional space of coefficients of polynomials p_k, $k = 0, 1, \ldots, m$, defining problem (3.58), there is a low-dimensional algebraic set, which is such that if we choose an instance of the problem (3.58) outside of this set, the Lasserre's LMI relaxations have finite convergence, i.e. there exists a finite $r^* \in \mathbb{N}$ such that $p_r^* = p^*$ for all $r \in \mathbb{N} : r \geq r^*$.

This means that in general it is enough to compute one finite relaxed semidefinite program (3.61) of the relaxation order big enough to obtain the global optimum of the polynomial optimization problem (3.58). Only in some exceptional and somewhat degenerate problems the finite convergence does not occur and the optimum can not be obtained by computing finite-dimensional semidefinite program in the form (3.61).

From Theorem 3.5 we know, that the finite convergence of Lasserre's LMI hierarchy is ensured generically for some relaxation order r, which is a priory not known to us. The verification that the finite convergence occurred provides us the following theorem.

Theorem 3.6 (Certificate of finite convergence [8]). Let y^* be the solution of the problem (3.61) at a given relaxation order $r > \max\{r_1, \ldots, r_m\}$. If

$$\operatorname{rank} M_{r-\max\{r_1,\ldots,r_m\}}(y^*) = \operatorname{rank} M_r(y^*) \tag{3.64}$$

then $p_r^* = p^*$.

So when we find relaxation order r big enough for which Theorem 3.6 is satisfied, we know, we are done and we can extract the global optimum. However, in practise another condition is checked.

Theorem 3.7 (Rank-one moment matrix [8]). The condition of Theorem 3.6 is satisfied if

$$\operatorname{rank} M_r(y^*) = 1. \tag{3.65}$$

If the condition of Theorem 3.7 holds, the global optimum of the problem (3.58) can be easily recovered as

$$x^* = \begin{bmatrix} y_{10\ldots0} & y_{01\ldots0} & \cdots & y_{00\ldots1} \end{bmatrix}^\top. \tag{3.66}$$

As usual, we are interested in the complexity estimation. Given the polynomial optimization problem (3.58) in n variables, we obtain a relaxed semidefinite program (3.61) for a relation order r. This program is in $N = \binom{n+2r}{n}$ variables, which is equal to the number of monomials in n variables up to degree $2r$. If n is fixed, for example when solving given polynomial optimization problem, then N grows in $\mathcal{O}(r^n)$, that is polynomially in r. If the relaxation order r is fixed then N grows in $\mathcal{O}(n^r)$, that is polynomially in the number of variables n.

Example 3.5. Let us set up some polynomial optimization problem for demonstration purposes. We use the same ellipse and hyperbola from Example 3.1 to define us the feasible set, while minimizing the objective function $-x_1 - \frac{3}{2}x_2$.

$$\begin{aligned} p^* = \min_{x \in \mathbb{R}^2} \; & -x_1 - \tfrac{3}{2}x_2 \\ \text{s.t.} \quad & -20x_1^2 + x_1x_2 - 12x_2^2 - 16x_1 - x_2 + 48 \geq 0 \\ & 12x_1^2 - 58x_1x_2 + 3x_2^2 + 46x_1 - 47x_2 + 44 \geq 0 \end{aligned} \tag{3.67}$$

We expect that the problem has two global optima attained at

$$\begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \left\{ \begin{bmatrix} -\frac{1}{2} \\ 2 \end{bmatrix} ; \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\} \tag{3.68}$$

with value of the objective function

$$p^* = -2.5. \tag{3.69}$$

The illustration of the problem is depicted in Figure 3.2.

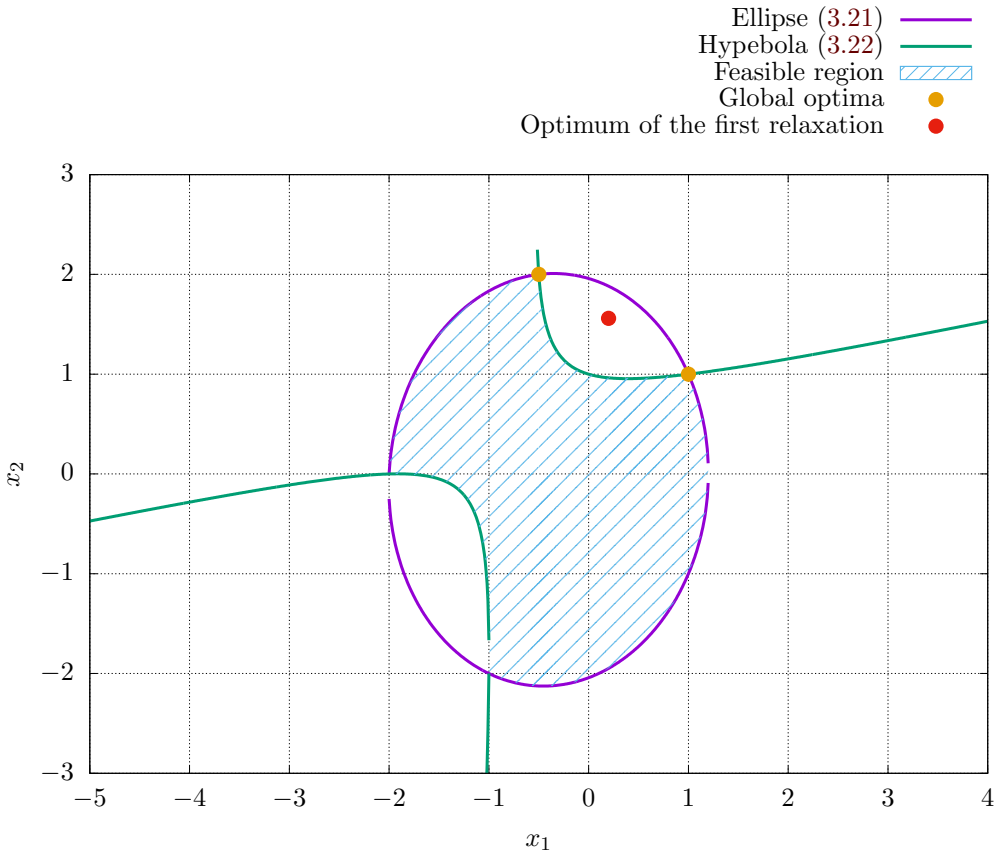


Figure 3.2. Feasible region and the expected global minima of the problem (3.67).

Firstly, we start with the relaxation order $r = 1$. The relaxed semidefinite problem is following.

$$\begin{aligned} p_1^* = \min_{y \in \mathbb{R}^{\mathbb{N}_2^2}} & -y_{10} - \tfrac{3}{2} y_{01} \\ \text{s.t.} \quad & y_{00} = 1 \\ & \begin{bmatrix} y_{00} & y_{10} & y_{01} \\ y_{10} & y_{20} & y_{11} \\ y_{01} & y_{11} & y_{02} \end{bmatrix} \succeq 0 \\ & \begin{bmatrix} -20y_{20} + y_{11} - 12y_{02} - 16y_{10} - y_{01} + 48y_{00} \end{bmatrix} \succeq 0 \\ & \begin{bmatrix} 12y_{20} - 58y_{11} + 3y_{02} + 46y_{10} - 47y_{01} + 44y_{00} \end{bmatrix} \succeq 0 \end{aligned} \tag{3.70}$$

By solving this problem, we obtain a possible solution

$$\begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \begin{bmatrix} 0.20 \\ 1.56 \end{bmatrix}, \tag{3.71}$$

$$p_1^* = -2.54, \tag{3.72}$$

which is not feasible. The moment matrix has rank

$$\operatorname{rank} M_1(y^*) = 2. \tag{3.73}$$

Since the condition of Theorem 3.7 is not satisfied, we continue with the second relaxation.

The second relaxation for $r = 2$ is below.

$$\begin{aligned}
p_2^* = \min_{y \in \mathbb{R}^{\mathbb{N}_4^2}} \; & -y_{10} - \tfrac{3}{2} y_{01} \\
\text{s.t.} \quad & y_{00} = 1 \\
& \begin{bmatrix}
y_{00} & y_{10} & y_{01} & y_{20} & y_{11} & y_{02} \\
y_{10} & y_{20} & y_{11} & y_{30} & y_{21} & y_{12} \\
y_{01} & y_{11} & y_{02} & y_{21} & y_{12} & y_{03} \\
y_{20} & y_{30} & y_{21} & y_{40} & y_{31} & y_{22} \\
y_{11} & y_{21} & y_{12} & y_{31} & y_{22} & y_{13} \\
y_{02} & y_{12} & y_{03} & y_{22} & y_{13} & y_{04}
\end{bmatrix} \succeq 0 \\
& \begin{bmatrix}
-20y_{20} + y_{11} - 12y_{02} - 16y_{10} - y_{01} + 48y_{00} & -20y_{30} + y_{21} - 12y_{12} - 16y_{20} - y_{11} + 48y_{10} & -20y_{21} + y_{12} - 12y_{03} - 16y_{11} - y_{02} + 48y_{01} \\
-20y_{30} + y_{21} - 12y_{12} - 16y_{20} - y_{11} + 48y_{10} & -20y_{40} + y_{31} - 12y_{22} - 16y_{30} - y_{21} + 48y_{20} & -20y_{31} + y_{22} - 12y_{13} - 16y_{21} - y_{12} + 48y_{11} \\
-20y_{21} + y_{12} - 12y_{03} - 16y_{11} - y_{02} + 48y_{01} & -20y_{31} + y_{22} - 12y_{13} - 16y_{21} - y_{12} + 48y_{11} & -20y_{22} + y_{13} - 12y_{04} - 16y_{12} - y_{03} + 48y_{02}
\end{bmatrix} \succeq 0 \\
& \begin{bmatrix}
12y_{20} - 58y_{11} + 3y_{02} + 46y_{10} - 47y_{01} + 44y_{00} & 12y_{30} - 58y_{21} + 3y_{12} + 46y_{20} - 47y_{11} + 44y_{10} & 12y_{21} - 58y_{12} + 3y_{03} + 46y_{11} - 47y_{02} + 44y_{01} \\
12y_{30} - 58y_{21} + 3y_{12} + 46y_{20} - 47y_{11} + 44y_{10} & 12y_{40} - 58y_{31} + 3y_{22} + 46y_{30} - 47y_{21} + 44y_{20} & 12y_{31} - 58y_{22} + 3y_{13} + 46y_{21} - 47y_{12} + 44y_{11} \\
12y_{21} - 58y_{12} + 3y_{03} + 46y_{11} - 47y_{02} + 44y_{01} & 12y_{31} - 58y_{22} + 3y_{13} + 46y_{21} - 47y_{12} + 44y_{11} & 12y_{22} - 58y_{13} + 3y_{04} + 46y_{12} - 47y_{03} + 44y_{02}
\end{bmatrix} \succeq 0
\end{aligned} \tag{3.74}$$

The minimum of the problem is

$$p_2^* = -2.5, \tag{3.75}$$

which is in correspondence with the optimum points

$$\begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \left\{ \begin{bmatrix} -\frac{1}{2} \\ 2 \end{bmatrix}; \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}. \tag{3.76}$$

They are the same as the minima of the original problem (3.67) and it depends on the iterative algorithm and the chosen starting point in which of these minima we end up. The moment matrix has rank

$$\operatorname{rank} M_2(y^*) = 1, \tag{3.77}$$

and therefore the global optimum has been found and we do not have to continue with the next relaxation. At the end, we verify that Theorem 3.4 holds.

$$p_1^* \leq \quad p_2^* \leq p^* \tag{3.78}$$

$$-2.54 \leq -2.5 \leq -2.5 \tag{3.79}$$

3.4.2. Implementation details

To verify and for better understanding of the Lasserre's LMI hierarchies, we have implemented the approach described in the previous section into the package Polyopt. More information about the Polyopt package and how to install it are in Section 2.4.1.

For solving the semidefinite programs (3.61) as proposed by Lasserre, we can use the interior-point algorithm as described in Section 2.3 and implemented in the Polyopt package. All we have to do is to construct the moment matrix and the localizing matrices from the polynomial constrains for a given relaxation order, which is quite easy.

What may be slightly complicated is, how to find the initial feasible point for the interior-point algorithm. The vector y of the semidefinite program (3.61) can be constructed from a vector x of the polynomial optimization problem (3.58) followingly:

$$y^{(\alpha)} = x^{\alpha} \tag{3.80}$$

for $\alpha \in \mathbb{N}^n_{2r}$. If x is from the feasible region P as stated in (3.59), then y is a feasible point of (3.61). Then the moment matrix $M_r(y)$ has rank one, since

$$M_r(y) = \zeta\zeta^{\top}, \tag{3.81}$$

and therefore y is not strictly feasible point, as it is required by the `SDPSolver` class. Hence, we construct N feasible points y_i from N different feasible points x_i for $N \geq \binom{n+r}{n}$ followingly:

$$y_i^{(\alpha)} = x_i^{\alpha}, \tag{3.82}$$

for $i = 1, \ldots, N$ and $\alpha \in \mathbb{N}^n_{2r}$. Then the strictly feasible point y can be constructed as

$$y = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{3.83}$$

and then the moment matrix is

$$M_r(y) = \frac{1}{N}\sum_{i=1}^{N} \zeta_i\zeta_i^{\top}, \tag{3.84}$$

where ζ_i are linearly independent, if x_i are pairwise different. Then $M_r(y)$ has full rank if N is bigger or equal to the number of rows or columns of $M_r(y)$, which is $\binom{n+r}{n}$.

The polynomial optimization solver is implemented in the class `POPSolver` of the Polyopt package. This solver can solve polynomial problem in form (3.58) for relaxation order r and with given strictly feasible points $x_1, \ldots, x_N$ of the polynomial optimization problem (3.58). Firstly, we initialize the problem with the objective function p_0, the constraining polynomials $p_1, \ldots, p_m$ and with the relaxation order r. Then a strictly feasible point y_0 of the semidefinite problem is computed by the function `getFeasiblePoint()` from the strictly feasible points $x_1, \ldots, x_N$ of the polynomial optimization problem (3.58). Finally, the problem is solved by the function `solve()`, which returns the optimal point x^*. The minimal working example is shown in Listing 3.1. For the purpose of the Polyopt package, the polynomials in n variables are represented by dictionaries in Python. The values are the coefficients and the keys are n-tuples of integers, where each tuple represents the corresponding monomial and the integers represent the degrees of each variable of the monomial.

Example 3.6. For $x \in \mathbb{R}^2$ the polynomial

$$p(x) = -20x_1^2 + x_1x_2 - 12x_2^2 - 16x_1 - x_2 + 48 \tag{3.85}$$

is represented in Python as variable `p` in a following way.

```
p = {(2, 0): -20, (1, 1): 1, (0, 2): -12, (1, 0): -16, (0, 1): -1,
  (0, 0): 48}
```

Variable `p` is a dictionary indexed by tuples. The first integer in each tuple represents the degree of the variable x_1 in the given monomial and the second integer represents the degree of the variable x_2.

Listing 3.1. Typical usage of the class `POPSolver` of the Polyopt package.

```
1: import polyopt
2:
3: # supposing the polynomials pi and the vectors xi are already
       defined
4: problem = polyopt.POPSolver(p0, [p1, ..., pm], r)
5: y0 = problem.getFeasiblePoint([x1, ..., xN])
6: xStar = problem.solve(y0)
```

Detailed information about the execution of the SDP solver can be printed out by setting `problem.setPrintOutput(True)`. The function `problem.getFeasiblePoint(xs)` computes the feasible point y from the feasible points x_i stored in the list `xs`. If the feasible points x_i are not know, they can be generated randomly from ball with radius `R` by the function `problem.getFeasiblePointFromRadius(R)`. When the polynomial problem is solved, the rank of the moment matrix can be obtained by calling `problem.momentMatrixRank()`.

Example 3.7. The Python code for the polynomial optimization problem (3.67) from Example 3.5 is shown in Listing 3.2.

3.4.3. Comparision with the state of the art methods

To get an idea, how the implementation from the Polyopt package is performing, we have compared it to the state of the art optimization toolbox Gloptipoly [9]. Gloptipoly is a MATLAB toolbox, which uses the Lasserre hierarchy to transform the polynomial optimization problem to the relaxed semidefinite program. This semidefinite program is then solved by some state of the art semidefinite program solver, by default by SeDuMi [26]. We have to point out that we are comparing a Python implementation with a MATLAB one.

We have generated random instances of a polynomial optimization problem $P_{n,d,k}$ for $k = 1, \ldots, 30$ of a type:

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} p_{n,d,k}(x) \\ &\text{s.t.} \quad 1 - \sum_{i=1}^{n} x_i^2 \geq 0, \end{aligned} \tag{3.86}$$

where n is the number of variables, d is the degree of the polynomial $p_{n,d,k}$. The generated instances differ in the coefficients of $\mathrm{vec}(p_{n,d,k})$, which were generated randomly from uniform distribution $(-1; 1)$. Moreover, the Polyopt package requires the initial point of the generated semidefinite program, which was randomly generated by the function `getFeasiblePointFromRadius(1)` in advance for each problem $P_{n,d,k}$. Then the execution times of each problems were measured. One option was to measure only the execution times of the semidefinite programs solvers. But since this depends only on the size of the generated semidefinite program, the results would be similar to the experiments in Section 2.5. Therefore, we have measured the sum of times required to construct the semidefinite program and to solve the semidefinite program. For the

Listing 3.2. Code for solving polynomial optimization problem stated in Example 3.5

```
from numpy import *
import polyopt

# objective function
p0 = {(1, 0): -1, (0, 1): -3/2}

# constraint functions
p1 =  {(2, 0): -20, (1, 1): 1, (0, 2): -12, (1, 0): -16, (0, 1):
    -1, (0, 0): 48}
p2 =  {(2, 0): 12, (1, 1): -58, (0, 2): 3, (1, 0): 46, (0, 1): -47,
     (0, 0): 44}

# feasible points of the polynomial problem
x1 = array([[1], [1]])
x2 = array([[2], [2]])
x3 = array([[-1], [-1]])
x4 = array([[-2], [1]])
x5 = array([[1], [2]])
x6 = array([[0], [2]])

# degree of the relaxation
r = 2

# initialize the solver
problem = polyopt.POPSolver(p0, [p1, p2], r)

# obtain a feasible point of the SDP problem from the feasible
    points of the polynomial problem
y0 = problem.getFeasiblePoint([x1, x2, x3, x4, x5, x6])

# enable outputs
problem.setPrintOutput(True)

#solve!
xStar = problem.solve(y0)
```

Polyopt package we have measured the execution times of the functions `POPSolver()` and `solve()`. For the Gloptipoly toolbox the execution times of the functions `msdp()` and `msol()` were measured.

Firstly, we have fixed the degree of the polynomial $d = 2$, and therefore we have set the relaxation order $r = 1$. We have solved the problems $P_{n,2,k}$ for the number of variables $n = 1, \ldots, 5$ repeatedly for $s = 1, \ldots, 30$ to eliminate some fluctuation in the time measuring. The measured times were saved as $\tau_{n,k,s}$. Because the influences in time measuring can only prolong the execution times, we have selected minimum of $\tau_{n,k,s}$ for each problem $P_{n,d,k}$.

$$\tau_{n,k} = \min_{s=1}^{30} \tau_{n,k,s} \tag{3.87}$$

Since the execution times of the problems of the same sizes should be the same, the average of the computation times was computed for each number of variables $n = 1, \ldots, 5$.

$$\tau_n = \frac{1}{30} \sum_{k=1}^{30} \tau_{n,k} \tag{3.88}$$

These execution times τ_n were measured and computed separately for the Polyopt package and the Gloptipoly toolbox and are shown in Table 3.1 and Figure 3.3.

Number of variables	**Dimension of the SDP**	**Toolbox**	
		Polyopt	**Gloptipoly** [9]
1	3	0.0159 s	0.0340 s
2	6	0.0411 s	0.0348 s
3	10	0.0942 s	0.0348 s
4	15	0.207 s	0.0371 s
5	21	0.412 s	0.0394 s

Table 3.1. Execution times of polynomial optimization problems in different number of variables with relaxation order $r = 1$ solved by selected toolboxes.

Secondly, we have fixed the number of variables $n = 2$ and let the degree d of the polynomial $p_{n,d,k}$ vary. We have set the relaxation order as low as possible to $r = \lceil \frac{d}{2} \rceil$. We have solved the problems $P_{2,d,k}$ for degrees $d = 1, \ldots, 5$ repeatedly for $s = 1, \ldots, 30$ to eliminate some fluctuation in the time measuring. The measured times were saved as $\tau_{d,k,s}$. For the same reasons as stated in the first case, we have proccessed the measured times followingly:

$$\tau_{d,k} = \min_{s=1}^{30} \tau_{d,k,s} \tag{3.89}$$

$$\tau_d = \frac{1}{30} \sum_{k=1}^{30} \tau_{d,k} \tag{3.90}$$

These execution times τ_d were measured and computed separately for the Polyopt package and the Gloptipoly toolbox and are shown in Table 3.2 and Figure 3.4.

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.

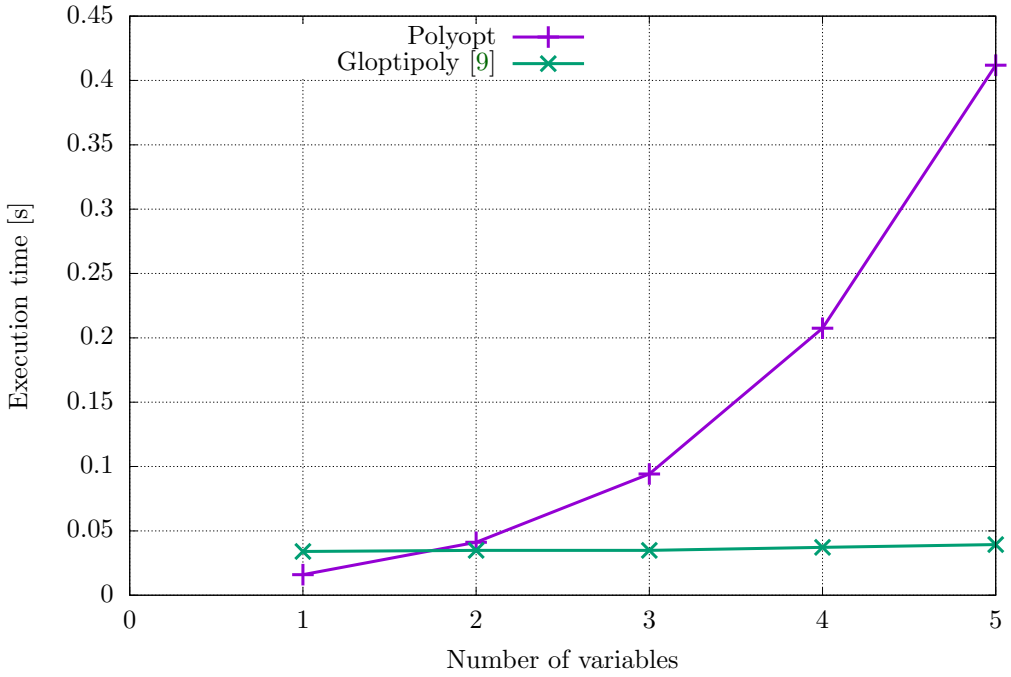


Figure 3.3. Graph of execution times of polynomial optimization problems with relaxation order $r = 1$ based on the number of variables solved by selected toolboxes.

Degree	Relaxation order	Dimension of the SDP	Toolbox	
			Polyopt	**Gloptipoly** [9]
1	1	6	0.0381 s	0.0356 s
2	1	6	0.0373 s	0.0341 s
3	2	15	0.217 s	0.0462 s
4	2	15	0.219 s	0.0491 s
5	3	28	0.919 s	0.0524 s

Table 3.2. Execution times of polynomial optimization problems for different degrees of the polynomial in the objective function in $n = 2$ variables solved by selected toolboxes.

From the graphs we can see that the Polyopt package is not comparable with Gloptipoly for high dimensions of the generated semidefinite program. This is in accordance with the results of Section 2.5, since the semidefinite programs solver is the most expensive part in the polynomial optimization. On the other hand, for really small polynomial optimization problems, the execution times of the Polyopt package are similar to the Gloptipoly execution times.

3.5. Solving systems of polynomial equations over the real numbers

Solving polynomial equations efficiently is a key element in computer vision geometry. For this reason, specialized solvers are constructed for the most common geometry problems to make the solvers efficient and numerically stable. Previously, these solvers were handcrafted, then the process was automated using automatic generators [12]. These automatic generators are based on Gröbner basis [**?**] and on computation of

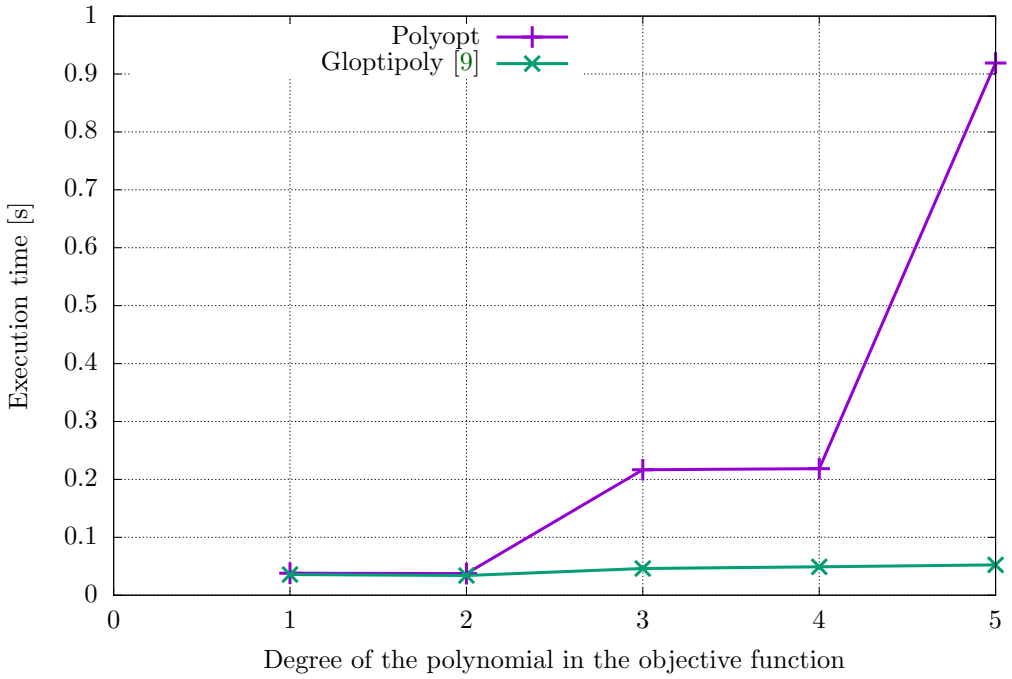


Figure 3.4. Graph of execution times of polynomial optimization problems in $n = 2$ variables based on the degree of the polynomial in the objective function solved by selected toolboxes.

eigenvectors of multiplication matrices. The disadvantage of this approach is that non-real solutions appear, and such solutions have no sense in geometry point of view and are discarded in the end. In the case that many non-real solutions are present and only few real solutions are obtained, the time consumed by computing the non-real solutions is much bigger than the time needed to compute the real solutions, which we are interested in.

Therefore, one should be interested in a method, which would solve the polynomial system over the real numbers only. One of the method is the moment method introduced in [14] and extended to the complex numbers in [15] both by J. B. Lasserre, M. Laurent and P. Rostalski. The moment method is summarized and enriched with examples in [17], which we will follow in this section.

The goal of this section is to solve the system of polynomial equations (3.91) without computation of the non-real solutions, where $x \in \mathbb{R}^n$ and $f_1, f_2, \ldots, f_m \in \mathbb{R}[x]$.

$$\begin{aligned} f_1(x) &= 0 \\ f_2(x) &= 0 \\ &\vdots \\ f_m(x) &= 0 \end{aligned} \tag{3.91}$$

3.5.1. The moment method

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. 20, 26

[2] Thomas Becker and Volker Weispfenning. *Gröbner Bases, A Computational Approach to Commutative Algebra*. Number 141 in Graduate Texts in Mathematics. Springer-Verlag, New York, NY, 1993. 31

[3] 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. 8

[4] David Cox, John Little, and Donald O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Springer, New York, USA, 2nd edition, 1997. 28

[5] 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. 8

[6] Jean-Charles Faugère. A new efficient algorithm for computing gröbner bases (f_4). *Journal of pure and applied algebra*, 139(1–3):61–88, July 1999. 31

[7] 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. 6

[8] Didier Henrion. Optimization on linear matrix inequalities for polynomial systems control, September 2014. 35, 36

[9] Didier Henrion, Jean-Bernard Lasserre, and Johan Löfberg. Gloptipoly 3: Moments, optimization and semidefinite programming. *Optimization Methods Software*, 24(4–5):761–779, August 2009. 40, 42, 43, 44

[10] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984. 8

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

[12] Zuzana Kukelova, Martin Bujnak, and Tomas Pajdla. Automatic generator of minimal problem solvers. In *Proceedings of The 10th European Conference on Computer Vision*, ECCV 2008, October 12–18 2008. 43

[13] Jean B. Lasserre. Global optimization with polynomials and the problem of moments. *Society for Industrial and Applied Mathematics Journal on Optimization*, 11:796–817, 2001. 35

[14] Jean Bernard Lasserre, Monique Laurent, and Philipp Rostalski. Semidefinite characterization and computation of zero-dimensional real radical ideals. *Foundations of Computational Mathematics*, 8(5):607–647, October 2008. 44

[15] Jean Bernard Lasserre, Monique Laurent, and Philipp Rostalski. A unified approach to computing real and complex zeros of zero-dimensional ideals. In *Emerging Applications of Algebraic Geometry*, pages 125–155, New York, 2009. Springer New York. 44

[16] Monique Laurent. Sums of squares, moment matrices and optimization over polynomials. In *Emerging Applications of Algebraic Geometry*, pages 157–270. Springer New York, 2009, Updated version from 2010. http://homepages.cwi.nl/~monique/files/moment-ima-update-new.pdf [Online; accessed 2017-05-05]. 32

[17] Monique Laurent and Philipp Rostalski. The approach of moments for polynomial equations. In *Handbook on Semidefinite, Conic and Polynomial Optimization*, pages 25–60, Boston, MA, 2012. Springer US. 44

[18] 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. 20, 26

[19] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. 26

[20] 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]. 9, 20, 23, 26, 27

[21] Yurii Nesterov. *Introductory lectures on convex optimization : A basic course*. Springer, 2004. 9, 21

[22] 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. 8

[23] 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. 9

[24] Michael Overton. On minimizing the maximum eigenvalue of a symmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 9:256–268, April 1988. 8

[25] Gabor Pataki. *On the multiplicity of optimal eigenvalues*. University of Michigan, Ann Arbor, MI (United States), December 1994. 8

[26] Jos F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999. 9, 20, 23, 26, 27, 40

[27] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, 2011. 20

[28] 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]. 20, 26

[29] 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. 9