CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

CIIRC

CZECH INSTITUTE OF INFORMATICS ROBOTICS AND CYBERNETICS

MASTER'S THESIS

Semidefinite Programming for Geometric Problems in Computer Vision

Pavel Trutman

pavel.trutman@cvut.cz

December 29, 2017

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

Thesis Advisor: Ing. Tomáš Pajdla, PhD.

Czech Institute of Informatics, Robotics, and Cybernetics
Czech Technical University in Prague
Jugoslávských partyzánů 1580/3, 160 00 Prague 6, Czech Republic
phone: +420 723 309 712, www: https://www.ciirc.cvut.cz

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Pavel T r u t m a n

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Diploma Thesis: Semidefinite Programming for Geometric Problems in Computer Vision

Guidelines:

1. Review the state of the art in semidefinite programming [1,2,3] and its use for solving variations of so called minimal problems in computer vision [4,5].
2. Suggest and develop a semidefinite solver for solving a variation of minimal problems.
3. Implement the solver, choose a relevant computer vision problem and investigate the performance of the solver in comparison to standard algebraic methods for solving the problem.

Bibliography/Sources:

[1] Y. Nesterov. Introductory lectures on convex optimization. Kluwer Academic Press, 2004.
[2] M. Laurent. SUMS OF SQUARES, MOMENT MATRICES AND OPTIMIZATION OVER POLYNOMIALS (http://homepages.cwi.nl/~monique/files/moment-ima-update-new.pdf).
[3] M. Laurent and P. Rostalski. The Approach of Moments for Polynomial Equations. In Handbook on Semidefinite, Conic and Polynomial Optimization, M. F. Anjos, J. B. Lasserre, eds., Springer 2012.
[4] C. Aholt, S. Agarwal, R. Thomas. A QCQP Approach to Triangulation, Computer Vision – ECCV 2012, Lecture Notes in Computer Science 7572 (2012), 654-667.
[5] F. Kahl, D. Henrion. Globally Optimal Estimates for Geometric Reconstruction Problems. ICCV 2005, (http://www2.maths.lth.se/vision/publdb/reports/pdf/kahl-henrion-ijcv-07.pdf).

Diploma Thesis Supervisor: Ing. Tomáš Pajdla, Ph.D.

Valid until: the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 6, 2017

Acknowledgements

I would like to express my thanks to my advisor Tomáš Pajdla for his guidance and valuable advices, which enabled me to finish this thesis. I would also like to thank Didier Herion for introducing me into semidefinite programming and polynomial optimization techniques and for his useful discussion and comments to my work. Special thanks go to my family for all their support.

Author's declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....
Signature

Abstract

Many problems in computer vision lead to polynomial systems solving. The state of the art algebraic methods for polynomial systems solving are able to efficiently solve the systems over complex numbers. In computer vision and robotics non-real solutions are then discarded, as they are not solutions of the original geometric problems. On this purpose, we review and implement the moment method for polynomial systems solving, which solves the problems over real numbers directly. We show that the moment method is applicable to the minimal problems from geometry of computer vision. For that, we give description of the calibrated camera pose problem and of the calibrated camera pose with unknown focal length problem. We compare our implementation of the moment with the state of the art methods on these two selected minimal problems on real 3D scenes.

Moreover, we review and implement a method for solving polynomial optimization problems, which can extend the moment method with inequality constraints. This method uses Lasserre's hierarchies to find the optimal values of the original optimization problems. We compare the performance of our implementation with the state of the art methods on synthetically generated polynomial optimization problems.

Since the semidefinite programs solving is a key element in the moment method and the polynomial optimization methods, we review and implement an interior-point algorithm for semidefinite programs solving. We compare the performance of our implementation with the state of the art methods on synthetically generated semidefinite programs.

Keywords: computer vision, polynomial systems solving, polynomial optimization, semidefinite programming, minimal problems

Abstrakt

Mnoho problémů v počítačovém vidění vede na řešení systémů polynomiálních rovnic. Současné metody na řešení systémů polynomiálních rovnic jsou schopny řešit tyto systémy v oboru komplexních čísel. V počítačovém vidění a robotice jsou nereálná řešení následně vyřazena, protože ta nejsou řešeními původních geometrických problémů. Z tohoto důvodu prozkoumáme a implementujeme metodu momentů pro řešení systémů polynomiálních rovnic, která řeší tyto problémy přímo v oboru reálných čísel. Ukážeme, že metoda momentů je použitelná na minimální problémy z geometrie počítačového vidění. Proto popíšeme problém nalezení polohy kalibrované kamery a problém nalezení polohy kalibrované kamery s neznámou ohniskovou vzdáleností. Na těchto dvou vybraných minimálních problémech a reálných 3D scénách porovnáme naší implementaci metody momentů se současnými metodami.

Dále prozkoumáme a implementujeme metodu na řešení polynomiálně optimalizačních problémů, která může rozšířit metodu momentů o omezení s nerovnostmi. Tato metoda využívá Lasserrových hierarchií k nalezení optimálních hodnot původních optimalizačních problémů. Na synteticky generovaných polynomiálně optimalizačních problémech porovnáme výkon naší implementace se současnými metodami.

Protože řešení semidefinitních programů je klíčovým elementem metody momentů a metod polynomiální optimalizace, prozkoumáme a implementujeme algoritmus vnitřních bodů na řešení semidefinitních programů. Na synteticky generovaných semidefinitních problémech porovnáme výkon naší implementace se současnými metodami.

Klíčová slova: počítačové vidění, řešení polynomiálních systémů, polynomiální optimalizace, semidefinitní programování, minimální problémy

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Symbols and Abbreviations

BA	Bundle adjustment.
$\mathbb{C}$	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 the vector x on the diagonal.
dom f	Domain of the function f.
Dom f	$\mathrm{cl}(\mathrm{dom}\ f)$.
$f'(x)$	First derivative of the function $f(x)$.
$f''(x)$	Second derivative of the function $f(x)$.
G-J elimination	Gauss-Jordan elimination.
$\mathcal{I}(V)$	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 $f_1, f_2, \ldots, f_n$.
int S	Interior of the set S.
$\ker(Q_\Lambda)$	Kernel of the quadratic form Q_Λ.
$\ker(M)$	Kernel of the matrix M.
$\{\lambda_i(A)\}_{i=1}^n$	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.
PnP problem	The perspective-n-point problem.
P3P problem	The perspective-three-point problem.
P3.5Pf problem	The perspective-three-and-half-point problem with unknown focal length.
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 $x \in \mathbb{R}^n$.
$\mathbb{R}[x]^*$	Dual vector space to the ring of polynomials $\mathbb{R}[x]$.
RANSAC	Random Sample Consensus.
$\mathcal{S}^n$	Space of $n \times n$ real symmetric matrices.
SDP	Semidefinite programming.
SfM	Structure from motion.
SGM method	Semi-global matching method.
$SO(3)$	Group of all rotations about the origin of three-dimensional space.

SVD	Singular value decomposition.
$\mathrm{tr}(A)$	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.
$\lceil x \rceil$	$\min\{m \in \mathbb{Z} \mid m \geq x\}$; ceiling function.
$\lfloor x \rfloor$	$\max\{m \in \mathbb{Z} \mid m \leq x\}$; floor function.
$\mathcal{X}_f$	Multiplication matrix by the polynomial f.
$\mathbb{Z}$	Set of integers.

1. Introduction

In geometry of computer vision, many problems are formulated as systems of polynomial equations. The state of the art methods are based on polynomial algebra, i.e. on Gröbner bases and multiplication matrices computation. Contrary to this approach, this work applies non-linear optimization techniques to solve the polynomial systems, which is a novel idea in the field of geometry of computer vision. Moreover, the application of the optimization techniques allows us to enrich the polynomial systems with polynomial inequalities or to solve polynomial optimization problems, i.e. optimizing a polynomial function with given polynomial constraints.

1.1. Motivation

Object recognition and localization, reconstruction of 3D scenes, self-driving cars, film production, augmented reality and robotics are only few of many applications of geometry of computer vision. Thus, one would like to solve geometric problems efficiently, since these problems often have to be solved in real-time applications. Typical geometric problems from computer vision are the minimal problems, which arise when estimating geometric models of scenes from given images. To be able to solve these problems computationally, they are often represented by systems of algebraic equations. Hence, one of the issues of computer vision is, how to solve systems of polynomial equations efficiently, which is the scope of this work.

The polynomial systems obtained from the geometric problems are often not trivial, but usually consist of many polynomial equations of high degree in several unknowns. From that reason, general algorithms for polynomial systems solving are not efficient for them, and therefore special solvers have been developed for different problems to solve these problems efficiently and robustly. Previously, these solvers were handcrafted, which is quite time demanding process that has to be done for each problem from scratch. Then, the process was automated by automatic generators [22, 23], which automatically generate efficient solver for a given type of the polynomial system. These solvers obtain the Gröbner basis of the system and then construct the multiplication matrix, from which solutions are extracted by eigenvectors computation. The side effect of this approach is that some non-real solutions often appear amongst real solutions, which are not solutions to the original geometric problem. Since the computation of the non-real solutions takes time, a method which would find real solutions only may be faster than the contemporary approach.

Some of the arisen systems may be overconstrained. Such systems have a solution when solved on precise data using precise arithmetic, but they have no solution when solved on real noisy data. However, these systems may be transformed into optimization problem by relaxing some of the constrains and by minimizing the error of these constraints. Therefore, an efficient polynomial optimization method may prove useful for overconstrained systems.

1.2. Contributions

To solve polynomial systems over real numbers only, we apply the moment method introduced by J. B. Lasserre et al. This method uses hierarchies of semidefinite programs to find a Gröbner basis of real radical ideal constructed from the ideal generated by the given polynomials. Then, a multiplication matrix is constructed and solutions are obtained from it. In this case, the multiplication matrix should have smaller size than a multiplication matrix obtained from the automatic generator, which can save some computation time. We implement this method in Python and MATLAB and examine its properties on several minimal problems from geometry of computer vision on real 3D scenes. We show that this method is applicable on problems from computer vision.

The second contribution of this work is, that we describe and review a method for polynomial optimization problems. This method solves hierarchies of semidefinite programs to find the optimal value. An application of this method can, for example, be a solver of overconstrained polynomial systems. We implement our own implementation of this method in Python and compare it to the state of the art methods on synthetic polynomial optimization problems.

Since semidefinite programs solving is a key element in both previously mentioned methods, we review and describe an interior-point method for semidefinite programs solving. To be able to use this method in implementations of the moment method and the polynomial optimization method, we implement this interior-point method in Python. To verify our implementation we compare it to the state of the art semidefinite solvers on synthetic semidefinite programs.

1.3. Thesis structure

In this work, we first review an interior-point method for semidefinite programs solving. To do so, general properties of self-concordant functions and barriers need to be introduced, since they are key elements in convex optimization. Then, a specialized barrier function for semidefinite programming will be described. We describe our implementation of the semidefinite programs solver and compare it to the state of the art methods.

Secondly, we focus on polynomial optimization. After an introduction to polynomial algebra and moment matrices, we describe and implement a method, which solves polynomial optimization problems by relaxations of semidefinite programs. Then, we review the moment method and describe its implementation in Python.

To compare the implementation of the moment method to the state of the art methods, we introduce two minimal problems from computer vision on which we perform the experiments. The minimal problems are the estimation (i) of the calibrated camera pose and (ii) of the calibrated camera pose with unknown focal length. We show that our implementation of the moment method is applicable to these selected geometric problems from computer vision.

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. SDP, which is optimizing a convex function on a convex set, 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 not be 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 [21]), for which M. Goemans and D. P. Williamson created the first approximation algorithm based on SDP [14]. 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. Systems of polynomial equations can also be solved by hierarchies of semidefinite problems. This approach has the advantage that there exists a method that allows us to compute real solutions only. Since in many applications, we are not interested in non-real solutions, this method may be the right tool for polynomial systems solving. We will focus in details on SDP application in 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, $y \in \mathbb{R}^n$ by

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

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

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

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 true:

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 the 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 true:

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 $\forall \alpha, 0 \leq \alpha \leq 1$ there 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. See Figure 2.1 to get a general picture, how a simple semidefinite problem may look like.

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 when the componentwise inequalities between

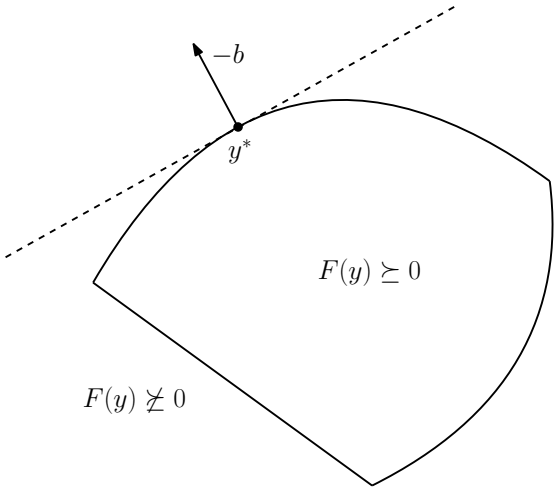

Figure 2.1. Example of a simple semidefinite problem for $y \in \mathbb{R}^2$. Boundary of the feasible set $\{y \mid F(y) \succeq 0\}$ is shown as a black curve. The minimal value of the objective function $b^\top y$ is attained at y^*.

vectors in linear programming are replaced by LMI. Consider a linear program in the 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 [5] 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 [10], M. Overton [39] and G. Pataki [40]. In 1984, the interior-point methods for LPs solving were introduced by N. Karmarkar [20]. 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 [37] 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 in methods used 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 every 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 [46] casts the standard semidefinite program into the homogeneous self-dual form, which has a trivial feasible point. SDPA [50] uses an infeasible interior-point method, which can initialized by an infeasible point. Some of the libraries (e.g. MOSEK [34]) 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 [38].

2.3. Interior point method

In this section, we will follow Chapter 4 of [36] 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. Look into [36] for the proofs and more details.

2.3.1. Self-concordant functions

Definition 2.2 (Self-concordant function in $\mathbb{R}$). A closed convex function $f\colon \mathbb{R} \mapsto \mathbb{R}$ 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 the self-concordant functions, we provide several examples.

Example 2.3.

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 2.4 (Self-concordant function in $\mathbb{R}^n$). A closed convex function $f\colon \mathbb{R}^n \mapsto \mathbb{R}$ 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 2.5 ([36, Theorem 4.1.1]). 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 2.6 ([36, Corollary 4.1.2]). Let function f be self-concordant with some constant M_f and let $\alpha > 0$. 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 2.6, we can see that any self-concordant function can be transformed into the standard self-concordant function by scaling.

Theorem 2.7 ([36, Theorem 4.1.3])**.** 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 2.7), we introduce 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 the following minimization problem

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

with self-concordant function $f(x)$. Algorithm 2.1 describes an 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 solution 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 approximation to the 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 $\operatorname{Dom} f$ as the closure of $\operatorname{dom} f$, i.e. $\operatorname{Dom} f = \operatorname{cl}(\operatorname{dom} f)$.

Definition 2.8 (Self-concordant barrier [36, Definition 4.2.2]**).** 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 2.8, 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 2.9.

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.

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

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

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

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 $\left\{x \in \mathbb{R} \mid x > 0\right\}$.

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

Theorem 2.10 ([36, Theorem 4.2.1])**.** 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 2.11 ([36, Theorem 4.2.2])**.** Let F_i be a 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 2.12 ([36, Theorem 4.2.5])**.** 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 a convex set, which is important for solving convex minimization problems. It is called the analytic center of convex set and we will focus on its properties.

Definition 2.13 ([36, Definition 4.2.3])**.** 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 2.14 ([36, Theorem 4.2.6])**.** 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 2.12 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 a 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. [36, Scheme 4.2.25]

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 to 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 2.15 ([36, Theorem 4.2.10])**.** 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 the 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 2.10 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 2.16 ([36, Theorem 4.2.8]). 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 2.17 ([36, Theorem 4.2.9]). 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:

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

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

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. [36, Scheme 4.2.23]

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 approximation to the 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

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

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 with interior formed by positive definite matrices and on its boundary lie matrices that 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 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

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

as the self-concordant barrier function for the set $\mathcal{P}^n$, 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 2.18 ([36, Theorem 4.3.3]). Function $F(X)$ is an n-self-concordant barrier for $\mathcal{P}^n$.

Example 2.19. 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 shown 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 2.20. To make it clearer, what is the difference between the set Q and $\operatorname{Dom} F(y)$, let us present an 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

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

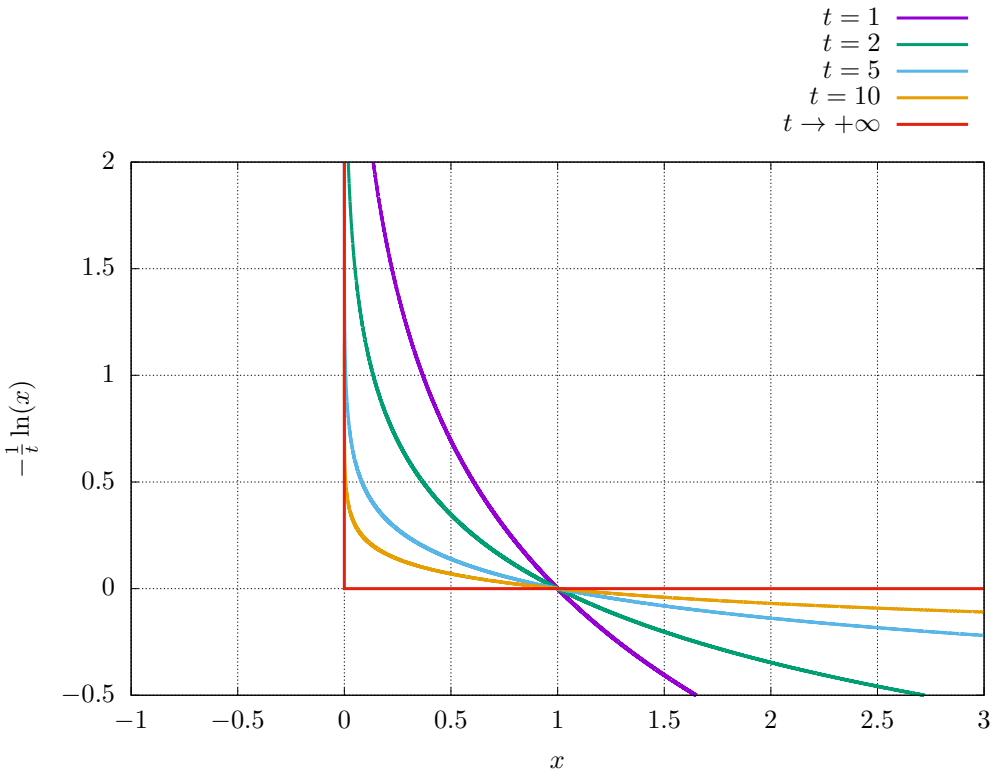

Figure 2.2. Illustration of the logarithmic barrier function for different values of t.

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 blue. 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) as

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

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:

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

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

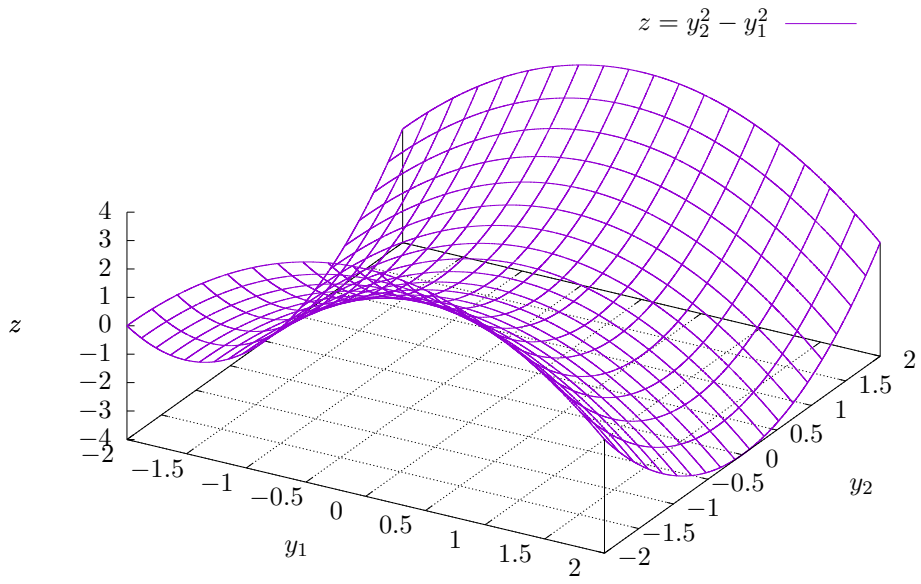

Figure 2.3. Hyperbolic paraboloid $z = y_2^2 - y_1^2$.

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 as follows:

- 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 $\operatorname{tr}\big(X_j(y)^{-1} A_{j,u}\big)$ for $u = 1, \ldots, m$ is needed mn operations,
- the computation of $\operatorname{tr}\Big(\big(X_j(y)^{-1} A_{j,u}\big)\big(X_j(y)^{-1} A_{j,v}\big)\Big)$ 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 is kept 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.

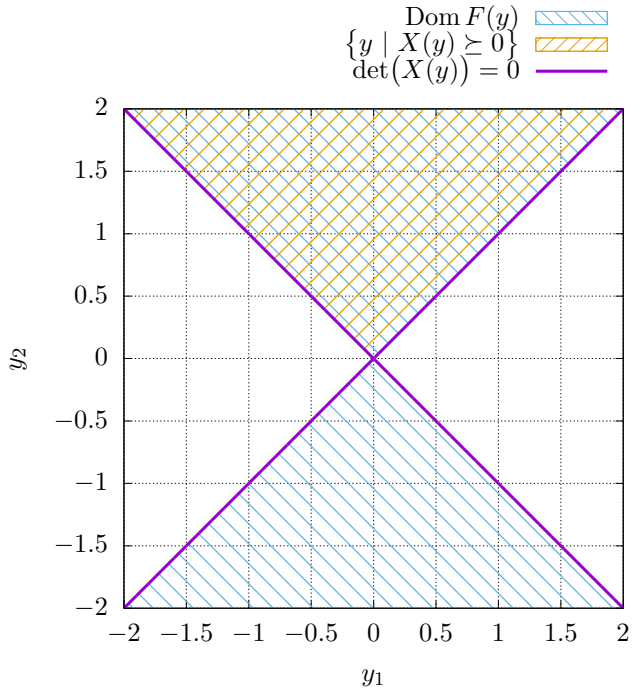

Figure 2.4. Illustration of the sets Dom $F(y)$ and $\{y \mid X(y) \succeq 0\}$.

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 [47]. 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 [46] and MOSEK [34], 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 effort 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 [3], ATLAS [48] and BLAS [31]. 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: 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 able to solve semidefinite programs in the 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 [36]. 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: # assuming 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 to the terminal.

[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 is 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 2.18. 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 a 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 2.21. 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 shown 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

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

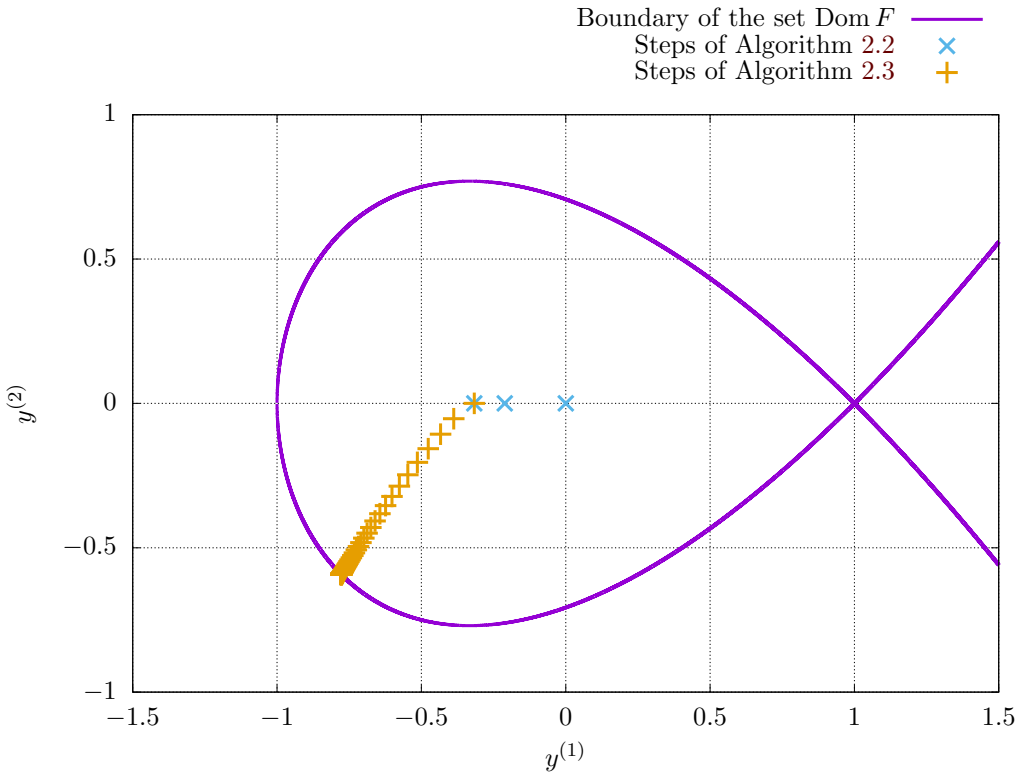

Figure 2.5. Graph of the semidefinite optimization problem stated in Example 2.21.

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 implementations. 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 [46] and MOSEK [34]. 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

Listing 2.3. Code for solving semidefinite problem stated in Example 2.21.

```
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())
```

requires $m(m+n)n^2$ arithmetic operations, where m is the size of the matrices in the LMI constraint and n is the number of variables. Since in typical applications of SDP, the size of the matrices grows with the number of variables, we have set $m = n$ to have just single parameter, which we call the size of the problem.

In our experiment, we have generated 50 unique LMI constraints for each size of the problem from 1 to 25. 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 $k = 1, \ldots, 25$ and unique LMI constraint $l = 1, \ldots, 50$, 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 $P_{k,l}$ 50 times. So, for each problem $P_{k,l}$ we have obtained execution times $\tau_{k,l,s}$ for $s = 1, \ldots, 50$. Because the influences mentioned above can only prolong the execution times, we have selected minimum of $\tau_{k,l,s}$ for each problem $P_{k,l}$.

$$\tau_{k,l} = \min_{s=1}^{50} \tau_{k,l,s} \tag{2.94}$$

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.

$$\tau_k = \frac{1}{50} \sum_{l=1}^{50} \tau_{k,l} \tag{2.95}$$

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** [46]	**MOSEK** [34]
1	0.010 70 s	0.020 20 s	0.002 07 s
2	0.014 50 s	0.027 70 s	0.002 32 s
3	0.015 60 s	0.029 60 s	0.002 46 s
4	0.017 60 s	0.033 30 s	0.002 81 s
5	0.020 00 s	0.035 60 s	0.003 21 s
6	0.023 30 s	0.037 80 s	0.003 49 s
7	0.027 80 s	0.038 00 s	0.003 49 s
8	0.032 80 s	0.040 20 s	0.003 76 s
9	0.039 70 s	0.040 50 s	0.004 25 s
10	0.046 80 s	0.043 70 s	0.004 85 s
11	0.054 60 s	0.044 70 s	0.005 54 s
12	0.064 40 s	0.045 80 s	0.005 81 s
13	0.073 90 s	0.047 50 s	0.006 20 s
14	0.085 20 s	0.049 00 s	0.006 99 s
15	0.096 20 s	0.051 30 s	0.007 49 s
16	0.111 00 s	0.052 70 s	0.007 73 s
17	0.124 00 s	0.055 50 s	0.008 86 s
18	0.144 00 s	0.057 80 s	0.009 50 s
19	0.162 00 s	0.059 90 s	0.010 30 s
20	0.181 00 s	0.062 10 s	0.010 40 s
21	0.207 00 s	0.066 50 s	0.012 10 s
22	0.233 00 s	0.068 80 s	0.012 90 s
23	0.260 00 s	0.070 20 s	0.014 00 s
24	0.287 00 s	0.072 30 s	0.014 70 s
25	0.324 00 s	0.075 90 s	0.017 30 s

Table 2.1. Execution times of different sizes of semidefinite problems solved by the 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 [3], ATLAS [48] and BLAS [31], 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 setup 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

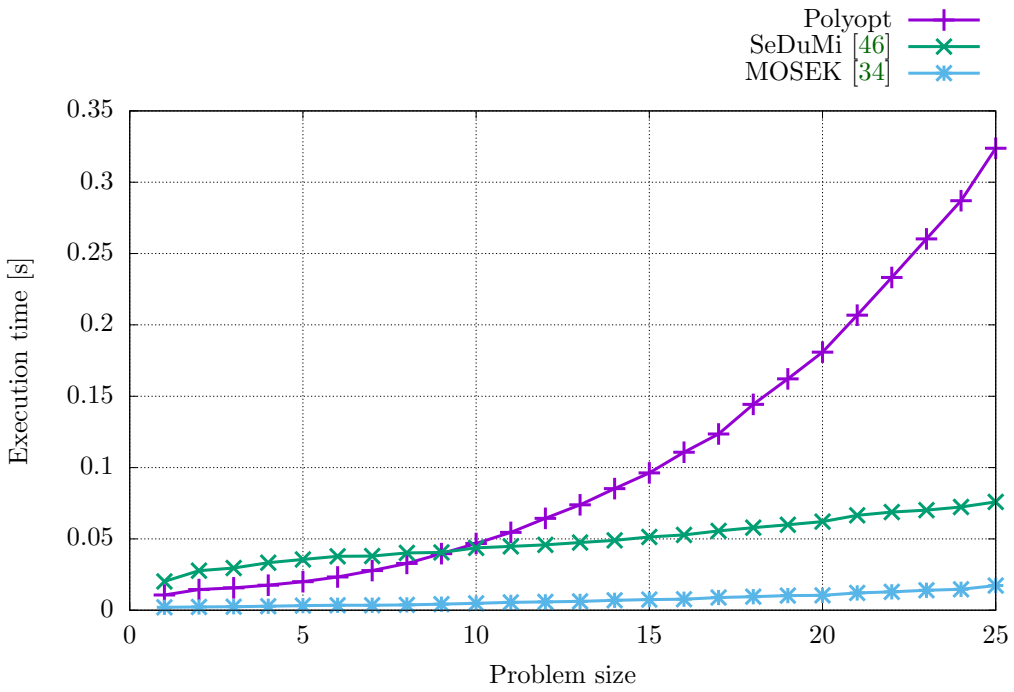

Figure 2.6. Graph of execution times based on the size of semidefinite problems solved by the selected toolboxes.

YALMIP [32] 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 Xeon E5-1650 v4 CPU 3.60GHz based computer with sufficient amount of free system memory. The installed version of Python was 3.5.3 and MATLAB R2017b 64-bit was used.

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 10.

2.6. Speed–accuracy trade-off

Obtaining a precise solution of a semidefinite program is quite time consuming, especially for problems with many unknowns, as we can see from Figure 2.6. However, in many applications we have no need for a precise solution. We only need a “good enough” solution, but obtained in a limited time period. Therefore one should be interested in a analysis describing this speed–accuracy trade-off.

2.6.1. Precision based analysis

The first experiment is an observation how many iterations, and therefore how much time, is required to find a solution based on the value of ε from Algorithm 2.3. This value of ε represents an accuracy of Algorithm 2.3 and sets up a threshold for the termination condition. The bigger the value of ε the less iterations the algorithm needs to terminate, and therefore less computation time is spent.

To evaluate the dependency we have generated unique semidefinite programs $P_{k,l}$ according to (2.93) for $k = 5, 10, 15, 20$ and $l = 1, \ldots, 1000$. For each of this problem and accuracy ε we have measured the number of iterations $N_{k,l,\varepsilon}$ of Algorithm 2.3 required to find the solution. Then, we have averaged the numbers across the unique problems.

$$N_{k,\varepsilon} = \frac{1}{1000} \sum_{l=1}^{1000} N_{k,l,\varepsilon} \tag{2.96}$$

The values of $N_{k,\varepsilon}$ are plotted in Figure 2.7 for different values of k and ε.

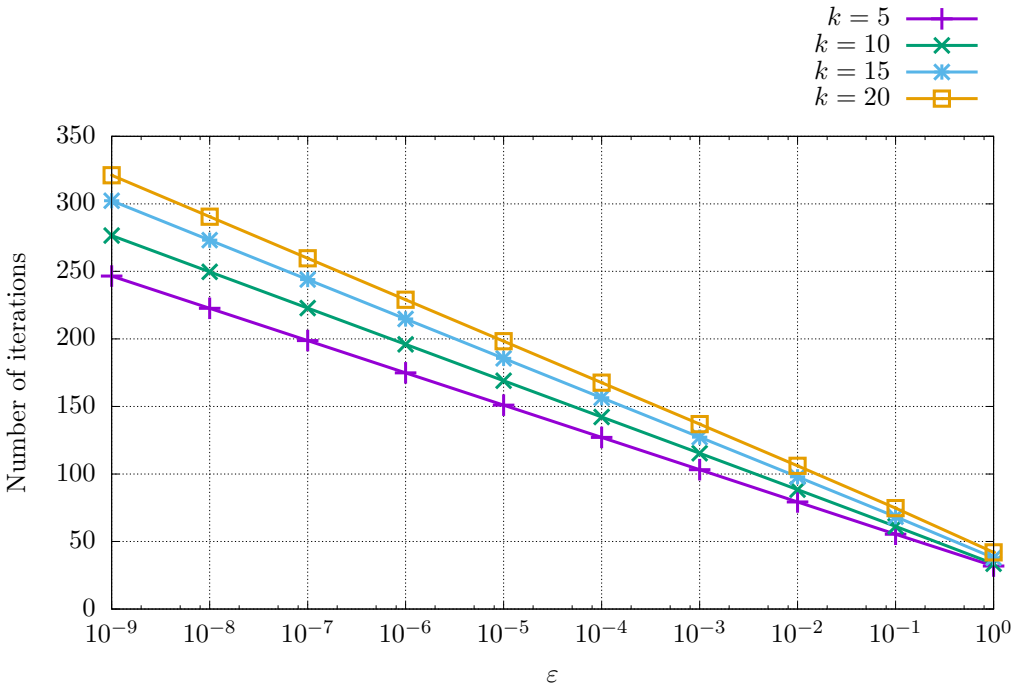

Figure 2.7. Graph of numbers of iterations required to solve the semidefinite problems by Algorithm 2.3 for different values of problem size k based on ε using the implementation from the Polyopt package.

From the plot we can see that the number of iterations, and therefore the execution time, required to solve a given problem of fixed problem size k is proportional to $\log(\varepsilon)$. This results is in accordance with the estimated number of steps from the equation (2.64).

2.6.2. Analysis based on the required distance from the solution

Algorithm 2.3 starts at the analytic center y_F^* of the feasible set of the problem and then follows the central path until it is sufficiently close to the optimal solution y^*.

The sizes of the steps of the algorithm are decreasing as the algorithm is closing to the solution. Therefore, we need only few iterations to get within half the distance $\|y^* - y_F^*\|$ from the solution, but many of them to be within 1 % of the distance $\|y^* - y_F^*\|$ from the solution. The situation for a simple semidefinite problem is shown in Figure 2.8.

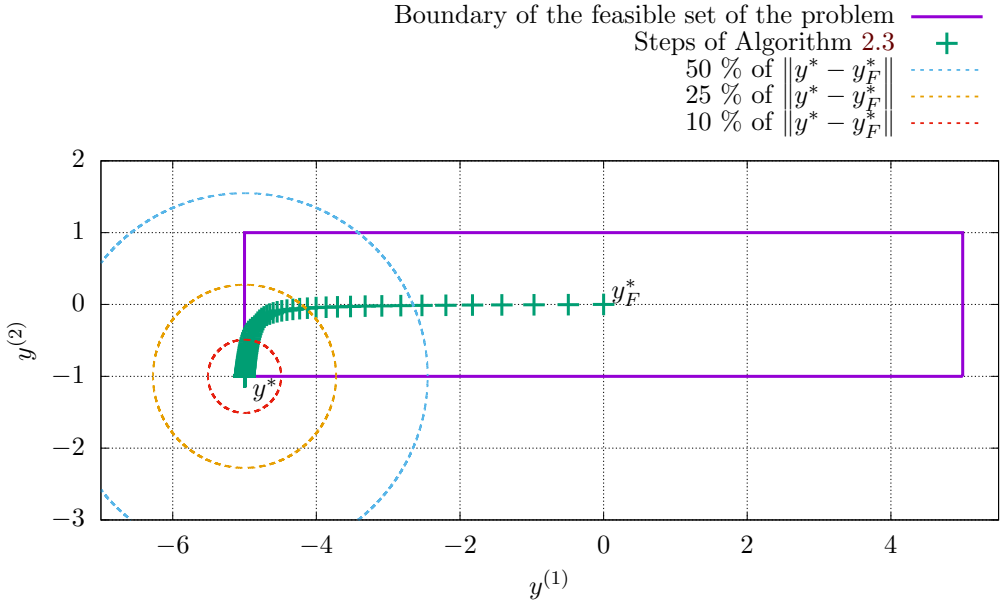

Figure 2.8. Example of a simple semidefinite problem with steps of Algorithm 2.3. The algorithm starts from the analytic center y_F^* and finishes at the optimal point y^*. Selected fractions of the distance $\|y^* - y_F^*\|$ are represented by concentric circles.

To see how many iterations are needed to get within some fraction of the distance $\|y^* - y_F^*\|$ from the optimal solution y^*, we have generated unique semidefinite problems $P_{k,l}$ for the problem sizes $k = 5, 10, 15, 20$ and $l = 1, \ldots, 1000$ according to (2.93). Then, we have solved each problem with accuracy $\varepsilon = 10^{-9}$ to obtain a precise approximation of the solution y^*. After that we have counted how many iterations $N_{k,l,\lambda}$ are needed to get within the distance $\lambda\|y^* - y_F^*\|$ from the solution y^* for the given problem and for selected values of λ. Then, we have averaged the numbers across the unique problems.

$$N_{k,\lambda} = \frac{1}{1000} \sum_{l=1}^{1000} N_{k,l,\lambda} \tag{2.97}$$

The values of $N_{k,\lambda}$ are plotted in Figure 2.9 for different values of k and λ.

From the experiments we can see that the number of iterations, and therefore the execution time, of Algorithm 2.3 required to get within some fraction λ of the distance $\|y^* - y_F^*\|$ for a given problem with fixed problem size k is almost proportional to $\log(\lambda)$.

2.7. Conclusions

In this chapter, we have reviewed the state of the art interior point algorithm for solving convex optimization problems and we have shown its application on semidefinite programming. We have implemented this algorithm in Python and verified its correctness on synthetically generated semidefinite programs. The computation time comparison to the state of the art toolboxes SeDuMi [46] and MOSEK [34] was given and it revealed that out implementation is significantly slower. The computation times

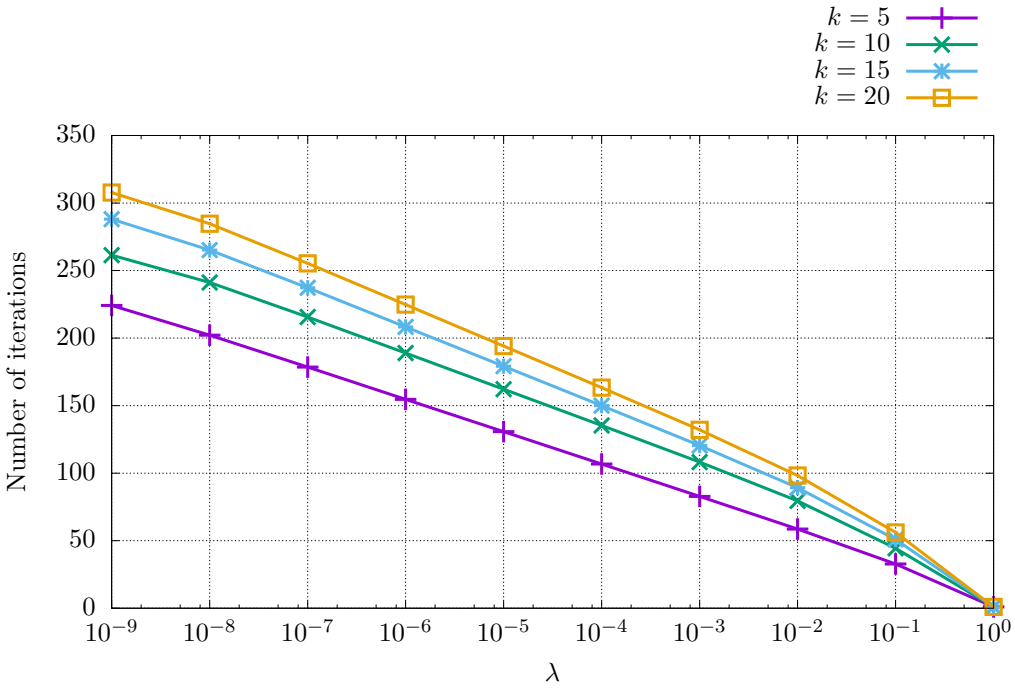

Figure 2.9. Graph of numbers of iterations required to get within the distance $\lambda\|y^* - y_F^*\|$ from the optimal solution y^* using Algorithm 2.3 for different values of problem size k based on λ using the implementation from the Polyopt package.

are comparable to the state of the art toolboxes only for small semidefinite problems, let us say for problems up to 10 unknowns. Then, we have investigated the speed–accuracy trade-off, which has shown that computation time can be saved when the precision of the solution is not crucial.

3. Optimization over polynomials

This chapter is devoted to the application of semidefinite programming in polynomial algebra. Firstly, we introduce basic notation from the polynomial algebra and the state of the art method for solving systems of polynomial equations using so called multiplication matrices. Then, we introduce the theory of moment matrices, since moment matrices will be used to relax the polynomial problems into the semidefinite ones. After that, we will focus on polynomial optimization, i.e. optimizing a polynomial function given polynomial constrains. We will present a method how to use hierarchies of semidefinite problems to solve a polynomial optimization problem. We implement this method and compare it to the state of the art optimization toolboxes. In the last section of this chapter, we will introduce the moment method for polynomial systems solving. This method also uses hierarchies of semidefinite problems to solve the polynomial systems with the advantage that only real solutions are found. When solving polynomial systems arisen from geometry of computer vision, we are typically interested only in the real solutions. Polynomial optimization method may prove to be a useful tool to eliminate the non-real solutions.

3.1. Algebraic preliminaries

In this chapter, which is focused on polynomial optimization and polynomial systems solving, we will follow the notation from [9]. Just to keep this chapter self-contained, we will recall some basics of polynomial algebra.

3.1.1. The polynomial ring, ideals and varieties

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 ${x_1}^{\alpha_1} {x_2}^{\alpha_2} \cdots {x_n}^{\alpha_n}$, with a 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 a 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 (i) $p \in I$ and $q \in \mathbb{R}[x]$ implies $pq \in I$ and (ii) $p, q \in I$ implies $p + q \in I$. 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 an ideal $I \in \mathbb{R}[x]$, the algebraic variety of I is the set

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

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 [9, Section 4.2, Theorem 6]**).** 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 [6, Theorem 4.1.4]**).** 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

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

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 a 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

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

3.1.2. Solving systems of polynomial equations using 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) $\mathcal{X}_f \colon \mathbb{R}[x]/I \to \mathbb{R}[x]/I$ 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 $\lambda_{x_i} = \big\{\lambda_j(\mathcal{X}_{x_i})\big\}_{j=1}^{N}$ 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 the 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 3.4. 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.

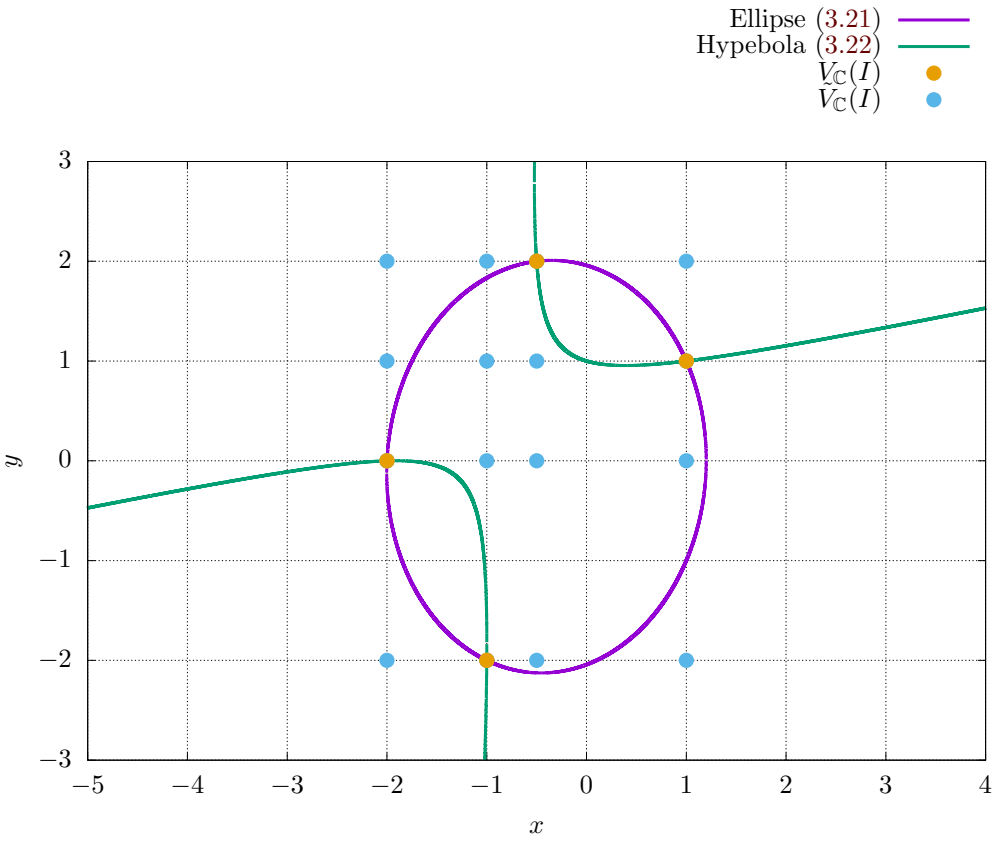

Figure 3.1. The intersection of the ellipse (3.21) and the hyperbola (3.22) with solutions found by the eigenvalue and the eigenvector methods using multiplication matrices.

First of all, we have to compute the Gröbner basis [4] of the ideal, for example using the F_4 Algorithm [11]. 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

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

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

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

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

and

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

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

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

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

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

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

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

Therefore, there are $4 \times 4 = 16$ possible solutions of 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.

3.2. 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 [29]. Moreover, we will introduce the only minimal basics to be able to proceed with polynomial optimization and polynomial systems solving. Detailed information can be found in [29] too.

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

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

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 $\alpha \in \mathbb{N}^n_d$.

Definition 3.5 (Riesz functional [26, page 38]**).** Given a sequence $y^{(\alpha)} = y_\alpha$ for $\alpha \in \mathbb{N}^n$, we define the Riesz linear functional $\ell_y \colon \mathbb{R}[x] \to \mathbb{R}$ 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.

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

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 3.6. 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 3.7 (Moment matrix [29, page 53]**).** 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 3.8 (Truncated moment matrix [29, page 53]**).** 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 $\alpha, \beta \in \mathbb{N}^n_s$.

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 3.9. For $n = 2$, the moment matrices for different orders are:

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

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

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

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 a 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 3.10 (Localizing matrix [29, page 53]**).** 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 3.11. For $n = 2$ and a polynomial $q(x) = x_1x_2 + 2x_1 + 3$, the localizing matrix of order one is

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

3.3. 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).

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

Let the feasibility set P of the optimization problem (3.58) be a 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 minima only.

3.3.1. State of the art review

It is known that the polynomial optimization problem (3.58) is NP-hard, and therefore several authors have proposed to approximate the problem (3.58) by a hierarchy of semidefinite relaxations. The first idea of applying a convex optimization technique to minimize unconstrained polynomial is from [44] by N. Z. Shor. Then, Y. Nesterov in [35] has shown a description of the cones of polynomials that are representable as a sum of squares and by exploitation of the duality between the moment cones and the cones of non-negative polynomials he has shown the characterization of a moment cone by LMIs when the non-negative polynomials can be written as a sum of squares. A breakthrough in polynomial optimization was done by J. B. Lasserre [25] who, by application of algebraic results by M. Putinar [43], showed that the polynomial optimization problems can be approximated by a sequence of semidefinite program relaxations, optima of which converge to the optimum of the polynomial optimization problem. Nowadays, efficient algorithms and their implementations for solving semidefinite programs exist as described in Section 2.2, and therefore polynomial optimization problems in a form (3.58) can be solved efficiently.

In these days, many implementations of polynomial optimization problem solver exists. Probably the most common is the MATLAB toolbox Gloptipoly [18], which can use many different semidefinite program solvers to solve the SDP problem arising from the relaxations. Then, there is the GpoSolver [16], which uses MATLAB to describe the problem and then generates C++ code, which solves the problem for given parameters and which can be included into an existing codebase. Also the SOSTOOLS [42] is a toolbox implemented in MATLAB, which can be used to solve the sum of squares optimization problems.

3.3.2. Lasserre's LMI hierarchy

The global minimum of a polynomial optimization problem in form (3.58) can be found by hierarchies of semidefinite programs. This was introduced by J. B. Lasserre in [25]. He has shown that the polynomial optimization problem can be 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 a 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 3.12 (Lasserre's LMI hierarchy converges [17, Proposition 3.3]**).** For $r \in \mathbb{N}$ there 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 3.13 (Generic finite convergence [17, Proposition 3.4]**).** In the finite-dimensional space of coefficients of the polynomials p_k, $k = 0, 1, \ldots, m$, defining the 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).

We know from Theorem 3.13 that the finite convergence of the 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 3.14 (Certificate of finite convergence [17, Proposition 3.5]**).** Let y^* be the solution of the problem (3.61) at a given relaxation order $r \geq \max\{r_1, \ldots, r_m\}$. 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 a relaxation order r big enough, for which Theorem 3.14 is satisfied, we know, we have finished and we can extract the global optima. However, if we expect that there is only one global optimum, we can check a simpler condition.

Theorem 3.15 (Rank-one moment matrix [17, Proposition 3.6]**).** The condition of Theorem 3.14 is satisfied if

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

If the condition of Theorem 3.15 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 relaxation 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 3.16. Let us set up some polynomial optimization problem for demonstration purposes. We use the same ellipse and hyperbola from Example 3.4 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 optimal points

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

with value of the objective function

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

The illustration of the problem is depicted in Figure 3.2.

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

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

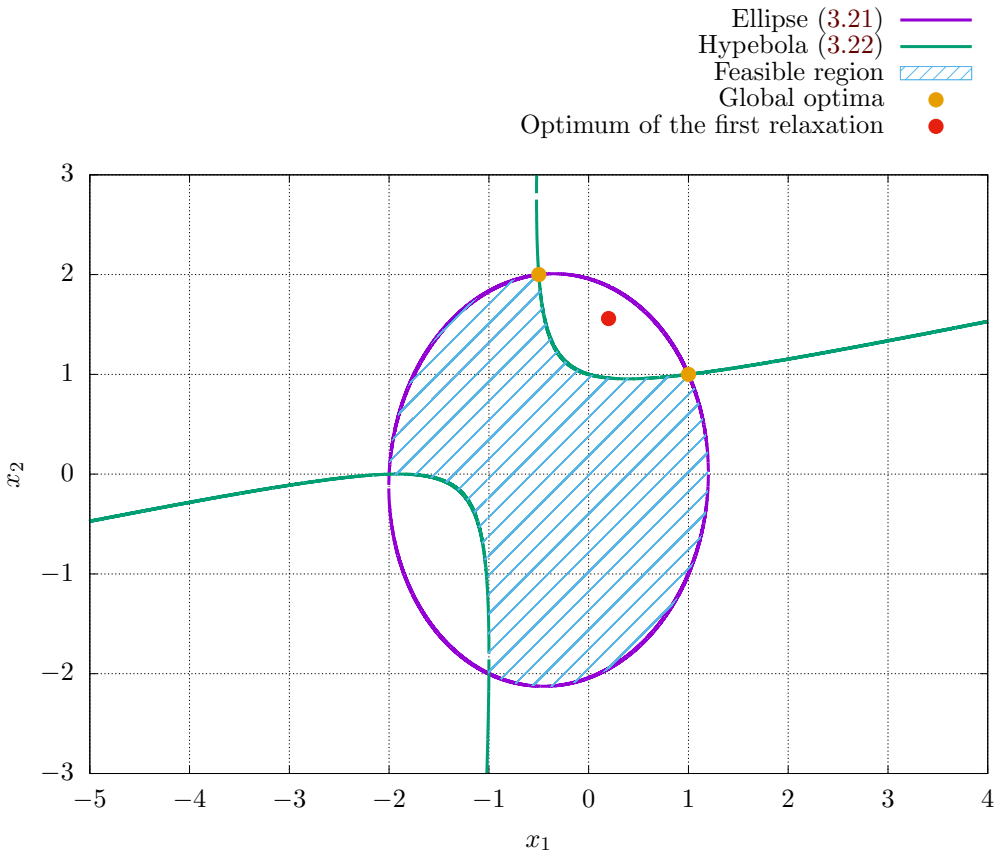

Figure 3.2. Feasible region and the expected global minima of the problem (3.67).

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. Ranks of different sizes of the moment matrix are

$$\operatorname{rank} M_0(y^*) = 1, \tag{3.73}$$

$$\operatorname{rank} M_1(y^*) = 2. \tag{3.74}$$

Since the condition of Theorem 3.14 is not satisfied, we continue with the second relaxation.

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

$$
\begin{aligned}
p_2^* = \min_{y \in \mathbb{R}^{\mathbb{N}_4^2}} \; & -y_{10} - \tfrac{3}{2} y_{01} \\
\text{s.t.} \; & y_{00} = 1 \\
& \begin{bmatrix}
y_{00} & y_{10} & y_{01} & y_{20} & y_{11} & y_{02} \\
y_{10} & y_{20} & y_{11} & y_{30} & y_{21} & y_{12} \\
y_{01} & y_{11} & y_{02} & y_{21} & y_{12} & y_{03} \\
y_{20} & y_{30} & y_{21} & y_{40} & y_{31} & y_{22} \\
y_{11} & y_{21} & y_{12} & y_{31} & y_{22} & y_{13} \\
y_{02} & y_{12} & y_{03} & y_{22} & y_{13} & y_{04}
\end{bmatrix} \succeq 0 \qquad (3.75) \\
& \begin{bmatrix}
-20y_{20} + y_{11} - 12y_{02} - 16y_{10} - y_{01} + 48y_{00} & -20y_{30} + y_{21} - 12y_{12} - 16y_{20} - y_{11} + 48y_{10} & -20y_{21} + y_{12} - 12y_{03} - 16y_{11} - y_{02} + 48y_{01} \\
-20y_{30} + y_{21} - 12y_{12} - 16y_{20} - y_{11} + 48y_{10} & -20y_{40} + y_{31} - 12y_{22} - 16y_{30} - y_{21} + 48y_{20} & -20y_{31} + y_{22} - 12y_{13} - 16y_{21} - y_{12} + 48y_{11} \\
-20y_{21} + y_{12} - 12y_{03} - 16y_{11} - y_{02} + 48y_{01} & -20y_{31} + y_{22} - 12y_{13} - 16y_{21} - y_{12} + 48y_{11} & -20y_{22} + y_{13} - 12y_{04} - 16y_{12} - y_{03} + 48y_{02}
\end{bmatrix} \succeq 0 \\
& \begin{bmatrix}
12y_{20} - 58y_{11} + 3y_{02} + 46y_{10} - 47y_{01} + 44y_{00} & 12y_{30} - 58y_{21} + 3y_{12} + 46y_{20} - 47y_{11} + 44y_{10} & 12y_{21} - 58y_{12} + 3y_{03} + 46y_{11} - 47y_{02} + 44y_{01} \\
12y_{30} - 58y_{21} + 3y_{12} + 46y_{20} - 47y_{11} + 44y_{10} & 12y_{40} - 58y_{31} + 3y_{22} + 46y_{30} - 47y_{21} + 44y_{20} & 12y_{31} - 58y_{22} + 3y_{13} + 46y_{21} - 47y_{12} + 44y_{11} \\
12y_{21} - 58y_{12} + 3y_{03} + 46y_{11} - 47y_{02} + 44y_{01} & 12y_{31} - 58y_{22} + 3y_{13} + 46y_{21} - 47y_{12} + 44y_{11} & 12y_{22} - 58y_{13} + 3y_{04} + 46y_{12} - 47y_{03} + 44y_{02}
\end{bmatrix} \succeq 0
\end{aligned}
$$

The minimum of the problem is

$$p_2^* = -2.5, \qquad (3.76)$$

which is in correspondence with the optimal points

$$\begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \left\{ \begin{bmatrix} -\frac{1}{2} \\ 2 \end{bmatrix} ; \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}. \qquad (3.77)$$

Ranks of different sizes of the moment matrix are

$$\operatorname{rank} M_0(y^*) = 1, \qquad (3.78)$$
$$\operatorname{rank} M_1(y^*) = 2, \qquad (3.79)$$
$$\operatorname{rank} M_2(y^*) = 2, \qquad (3.80)$$

and therefore Theorem 3.14 is satisfied and we do not have to continue with the next relaxation. We can verify that the found optimal points (3.77) are the same as the expected optimal points (3.68) of the original problem (3.67). At the end, we verify that Theorem 3.12 holds.

$$p_1^* \leq \quad p_2^* \leq p^* \qquad (3.81)$$
$$-2.54 \leq -2.5 \leq -2.5 \qquad (3.82)$$

3.3.3. 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 constraints for a given relaxation order, which is quite straightforward.

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) as follows

$$y^{(\alpha)} = x^\alpha \qquad (3.83)$$

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

$$M_r(y) = \zeta\zeta^\top, \tag{3.84}$$

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:

$$y_i^{(\alpha)} = x_i^\alpha, \tag{3.85}$$

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

$$y = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{3.86}$$

and then the moment matrix is

$$M_r(y) = \frac{1}{N}\sum_{i=1}^{N} \zeta_i\zeta_i^\top, \tag{3.87}$$

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 the 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 3.17. For $x \in \mathbb{R}^2$ the polynomial

$$p(x) = -20x_1^2 + x_1x_2 - 12x_2^2 - 16x_1 - x_2 + 48 \tag{3.88}$$

is represented in Python as a 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}
```

The 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: # assuming the polynomials pi, the vectors xi and the relaxation
       order r is 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 to the terminal by setting `problem.setPrintOutput(True)`. By calling of the function `problem.getFeasiblePoint(xs)` we obtain 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 a ball with radius `R` by calling `problem.getFeasiblePointFromRadius(R)`. When the polynomial problem is solved, the rank of the moment matrix can be obtained by calling `problem.momentMatrixRank()`.

Example 3.18. The Python code for the polynomial optimization problem (3.67) from Example 3.16 is shown in Listing 3.2.

3.3.4. Comparison 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 [18]. 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 [46]. 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 $k = 1, \ldots, 50$ of a type

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} p_{n,d,k}(x) \\ &\text{s.t.} \quad 1 - \sum_{i=1}^{n} x_i^2 \geq 0, \end{aligned} \tag{3.89}$$

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 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

Listing 3.2. Code for solving polynomial optimization problem stated in Example 3.16.

```
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([[-1], [0]])
x3 = array([[-1], [1]])
x4 = array([[0], [-1]])
x5 = array([[0], [0]])
x6 = array([[1], [0]])

# 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)
```

variables $n = 1, \ldots, 7$ repeatedly for $s = 1, \ldots, 50$ 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 $P_{n,d,k}$.

$$\tau_{n,k} = \min_{s=1}^{50} \tau_{n,k,s} \tag{3.90}$$

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 $n = 1, \ldots, 7$.

$$\tau_n = \frac{1}{50} \sum_{k=1}^{50} \tau_{n,k} \tag{3.91}$$

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** [18]
1	3	0.008 63 s	0.018 70 s
2	6	0.020 00 s	0.020 10 s
3	10	0.043 20 s	0.020 20 s
4	15	0.088 10 s	0.022 00 s
5	21	0.171 00 s	0.022 90 s
6	28	0.311 00 s	0.024 60 s
7	36	0.518 00 s	0.025 20 s

Table 3.1. Execution times of the polynomial optimization problems in different number of variables with the relaxation order $r = 1$ solved by the 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 $d = 1, \ldots, 7$ repeatedly for $s = 1, \ldots, 50$ 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 as follows

$$\tau_{d,k} = \min_{s=1}^{50} \tau_{d,k,s}, \tag{3.92}$$

$$\tau_d = \frac{1}{50} \sum_{k=1}^{50} \tau_{d,k}. \tag{3.93}$$

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 Xeon E5-1650 v4 CPU 3.60GHz based computer with sufficient amount of free system memory. The installed version of Python was 3.5.3 and MATLAB R2017b 64-bit was used.

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.

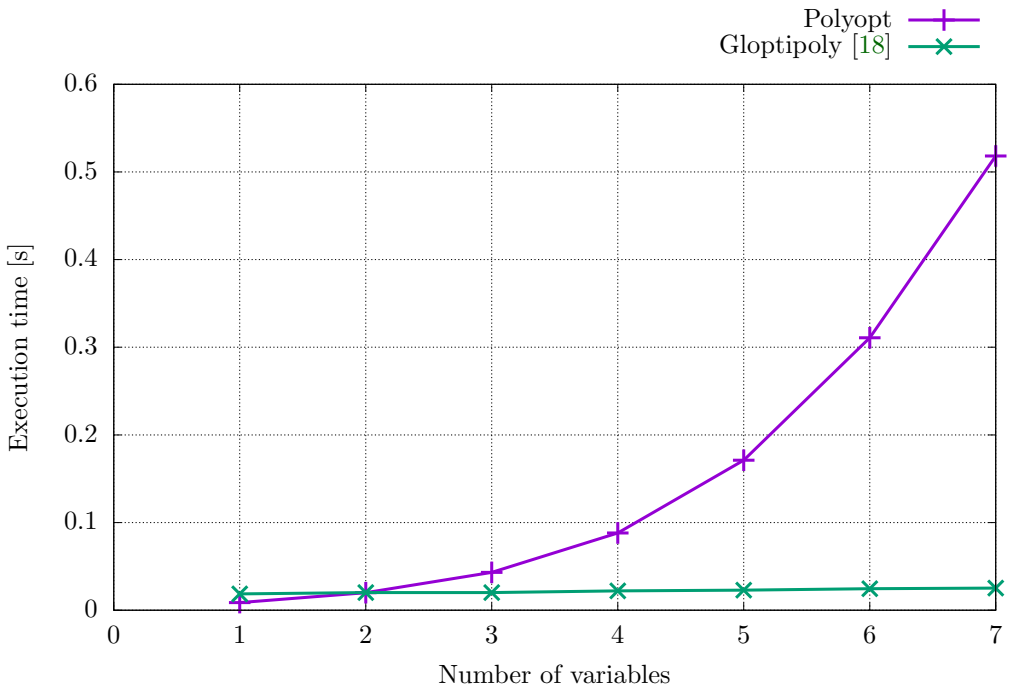

Figure 3.3. Graph of execution times of the polynomial optimization problems with the relaxation order $r = 1$ based on the number of variables solved by the selected toolboxes.

3.4. Solving systems of polynomial equations over the real numbers

The goal of this section is to solve the system of polynomial equations (3.94) without computation of the non-real solutions. Let $x \in \mathbb{R}^n$ and $f_1, f_2, \ldots, f_m \in \mathbb{R}[x]$.

$$\begin{aligned} f_1(x) &= 0 \\ f_2(x) &= 0 \\ &\vdots \\ f_m(x) &= 0 \end{aligned} \tag{3.94}$$

We will present and describe one of the state of the art method for solving polynomial systems over the real numbers. We will implement the method, so we will be able to apply it on some selected problems from geometry of computer vision and compare it with the state of the art methods used in computer vision.

3.4.1. State of the art review

Solving polynomial equations efficiently is a key element in geometry of computer vision. 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 by automatic generators [22, 23]. The automatic generator is based on Gröbner bases [4], which are typically computed by the Buchberger Algorithm [7] or by its successor, by the F_4 Algorithm [11]. When the Gröbner basis is found, the multiplication matrix is constructed and the solutions are found by the eigenvalue computation of the multiplication matrix. The disadvantage of this approach is that non-real solutions appear, which have no sense in

Degree	Relaxation order	Dimension of the SDP	Toolbox	
			Polyopt	Gloptipoly [18]
1	1	6	0.0176 s	0.0205 s
2	1	6	0.0174 s	0.0196 s
3	2	15	0.0933 s	0.0257 s
4	2	15	0.0932 s	0.0258 s
5	3	28	0.3730 s	0.0299 s
6	3	28	0.3770 s	0.0302 s
7	4	45	1.2100 s	0.0391 s

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

the geometry point of view and are discarded in the end. In 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, in which we are interested.

Therefore, one should be interested in a method, which would solve the polynomial system over the real numbers only. One of the methods is the moment method introduced in [27] and extended to the complex numbers in [28] both by J. B. Lasserre, M. Laurent and P. Rostalski. The moment method is summarized and enriched with examples in [30], which we will follow in this section.

3.4.2. The moment method

The moment method is based on obtaining of the real radical ideal $\sqrt[\mathbb{R}]{I}$ of the original ideal I. The real radical ideal is found by computing the kernel of a moment matrix, which is obtained by solving a hierarchy of semidefinite programs. Once the real radical ideal is found, its Gröbner basis is constructed and the solutions can be extracted in a standard way.

Since the moment method is based on positive linear forms and real radical ideals, let us introduce some basics from the theory about positive linear forms and their connection to the real radical ideals.

Positive linear forms

Let the dual vector space of the polynomial ring $\mathbb{R}[x]$ is denoted as $\mathbb{R}[x]^*$. Given a linear form $\Lambda \in \mathbb{R}[x]^*$, consider the quadratic form $Q_\Lambda \colon \mathbb{R}[x] \mapsto \mathbb{R}$ such that

$$Q_\Lambda(f) = \Lambda\big(f^2\big) \tag{3.95}$$

with kernel

$$\ker(Q_\Lambda) = \big\{ f \in \mathbb{R}[x] \mid \Lambda(fg) = 0 \ \forall g \in \mathbb{R}[x] \big\}. \tag{3.96}$$

Definition 3.19 (Positivity). Linear form $\Lambda \in \mathbb{R}[x]^*$ is said to be positive if $\Lambda\big(f^2\big) \geq 0$ for all $f \in \mathbb{R}[x]$, i.e. if the quadratic form Q_Λ is positive semidefinite.

How the positive linear forms are connected to real radical ideals shows the following theorem.

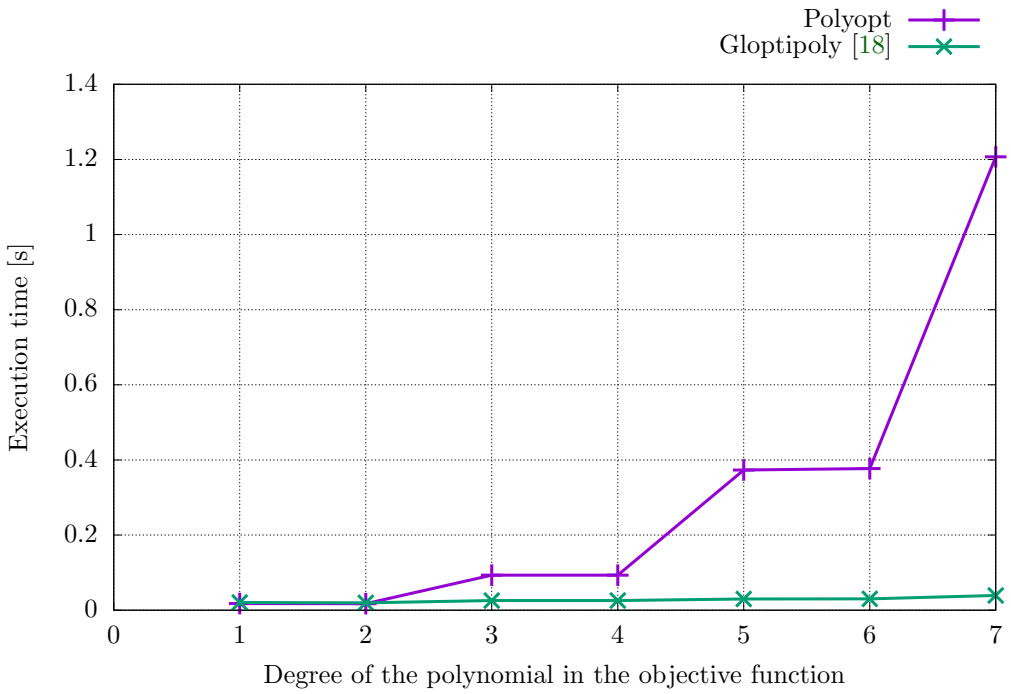

Figure 3.4. Graph of execution times of the polynomial optimization problems in $n = 2$ variables based on the degree of the polynomial in the objective function solved by the selected toolboxes.

Theorem 3.20 ([30, Lemma 2.1])**.** Let $\Lambda \in \mathbb{R}[x]^*$. Then, $\ker(Q_\Lambda)$ is an ideal in $\mathbb{R}[x]$, which is real radical ideal when Λ is positive.

We need to extend the theory about moments and moment matrices introduced in Section 3.2 and apply it to the linear forms. Therefore, we use a new definition of the moment matrix, which is equivalent to Definition 3.7.

Definition 3.21 (Moment matrix of Λ [30, Definition 2.3])**.** 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 linear form $\Lambda \in \mathbb{R}[x]^*$, the moment matrix $M(\Lambda)$ has form

$$M(\Lambda)^{(\alpha,\beta)} = \Lambda\big(x^\alpha x^\beta\big) \tag{3.97}$$

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

The moment matrix $M(\Lambda)$ has some interesting properties. For $p \in \mathbb{R}[x]$ the equation

$$Q_\Lambda(p) = \Lambda\big(p^2\big) = \operatorname{vec}(p)^\top M(\Lambda) \operatorname{vec}(p) \tag{3.98}$$

holds, and therefore $M(\Lambda)$ is the matrix of the quadratic form Q_Λ in some monomial basis. This concludes that Λ is positive if and only if $M(\Lambda) \succeq 0$.

The second interesting property is that a polynomial p is from $\ker(Q_\Lambda)$ if and only if its coefficient vector $\operatorname{vec}(p)$ is from $\ker\big(M(\Lambda)\big)$. Therefore, we identify both $\ker(Q_\Lambda)$ and $\ker\big(M(\Lambda)\big)$ with a set of polynomials hereafter. Thus by Theorem 3.20, $\ker\big(M(\Lambda)\big)$ is an ideal, which is real radical ideal when $M(\Lambda) \succeq 0$.

Example 3.22. For $n = 2$, let us have the linear form $\Lambda \in \mathbb{R}[x]^*$ defined by

$$\Lambda(1) = 1, \tag{3.99}$$
$$\Lambda(x_1 x_2) = 1, \tag{3.100}$$
$$\Lambda\big(x_1^{\alpha_1} x_2^{\alpha_2}\big) = 0 \text{ for all other monomials.} \tag{3.101}$$

Then, the moment matrix $M(\Lambda)$ (rows and columns indexed by 1, x_1, x_2, x_1^2, x_1x_2, x_2^2, ...) is

$$M(\Lambda) = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{3.102}$$

with $\operatorname{rank} M(\Lambda) = 4$ and with kernel

$$\ker\big(M(\Lambda)\big) = \big\langle x_1^2, x_2^2, 1 - x_1x_2 \big\rangle . \tag{3.103}$$

Since the kernel is not a radical ideal, Λ is not positive.

Theorem 3.23 ([30, Lemma 2.2])**.** Let $\Lambda \in \mathbb{R}[x]^*$ and let $\mathcal{B}$ be a set of monomials. Then, $\mathcal{B}$ indexes a maximal linearly independent set of columns of $M(\Lambda)$ if and only if $\mathcal{B}$ corresponds to a basis of $\mathbb{R}[x]/\ker\big(M(\Lambda)\big)$. That is,

$$\operatorname{rank} M(\Lambda) = \dim\Big(\mathbb{R}[x]/\ker\big(M(\Lambda)\big)\Big). \tag{3.104}$$

Which means that the monomial basis $\mathcal{B}$ of the quotient ring $\mathbb{R}[x]/\ker\big(M(\Lambda)\big)$ can be selected by looking at the maximal linearly independent set of columns of $M(\Lambda)$.

The following theorem shows, how to construct linear form Λ, such that the kernel of its moment matrix $\ker\big(M(\Lambda)\big)$ is a vanishing ideal of some selected points from $\mathbb{R}^n$.

Theorem 3.24 ([30, Lemma 2.3])**.** Let $\Lambda_{v_i} \in \mathbb{R}[x]^*$ is the evaluation at $v_i \in \mathbb{R}^n$. Let $\mathcal{B}_\infty$ is monomial basis containing all monomials, $\mathcal{B}_\infty^{(\alpha)} = x^\alpha$ for all $\alpha \in \mathbb{N}^n$. If Λ is a conic combination of evaluations at real points,

$$\Lambda = \sum_{i=1}^{r} \lambda_i \Lambda_{v_i}, \tag{3.105}$$

where $\lambda_i > 0$ and v_i are pairwise distinct, then moment matrix constructed as

$$M(\Lambda) = \sum_{i=1}^{r} \lambda_i [v_i]_{\mathcal{B}_\infty} [v_i]_{\mathcal{B}_\infty}^\top \tag{3.106}$$

has $\operatorname{rank} M(\Lambda) = r$ and $\ker\big(M(\Lambda)\big) = \mathcal{I}(v_1, v_2, \ldots, v_r)$.

Next theorem shows the converse implication to Theorem 3.24 so there is an equivalence in the end.

Theorem 3.25 (Finite rank moment matrix theorem [30, Theorem 2.6]**).** Assume that $\Lambda \in \mathbb{R}[x]^*$ is positive with $\operatorname{rank} M(\Lambda) = r < \infty$. Then,

$$\Lambda = \sum_{i=1}^{r} \lambda_i \Lambda_{v_i} \tag{3.107}$$

for some distinct $v_1, v_2, \ldots, v_r \in \mathbb{R}^n$ and some scalars $\lambda_i > 0$. Moreover, $\{v_1, v_2, \ldots, v_r\} = V_{\mathbb{C}}\Big(\ker\big(M(\Lambda)\big)\Big)$.

Now, we provide a semidefinite characterization of real radical ideals using positive linear forms. For that, we define the convex set

$$\mathcal{K} = \big\{\Lambda \in \mathbb{R}[x]^* \mid \Lambda(1) = 1, M(\Lambda) \succeq 0, \Lambda(p) = 0 \ \forall p \in I\big\}. \tag{3.108}$$

For any $\Lambda \in \mathcal{K}$, $\ker\big(M(\Lambda)\big)$ is a real radical ideal, which contains I, and therefore its real radical ideal $\sqrt[\mathbb{R}]{I}$, which implies

$$\dim\Big(\mathbb{R}[x]/\ker\big(M(\Lambda)\big)\Big) \leq \dim\Big(\mathbb{R}[x]/\sqrt[\mathbb{R}]{I}\Big). \tag{3.109}$$

When the real variety $V_{\mathbb{R}}(I)$ is finite, then $\ker\big(M(\Lambda)\big)$ is zero-dimensional and

$$\operatorname{rank} M(\Lambda) \leq |V_{\mathbb{R}}(I)|. \tag{3.110}$$

The equaility holds only for special elements $\Lambda \in \mathcal{K}$ named generic linear forms, which are described by the following definition.

Definition 3.26 (Generic linear forms [30, Definition 2.4]**).** Let $\mathcal{K}$ be defined as in (3.108) and assume $|V_{\mathbb{R}}(I)| < \infty$. A linear form $\Lambda \in \mathcal{K}$ is said to be generic if $M(\Lambda)$ has maximum rank, i.e. if $\operatorname{rank} M(\Lambda) = |V_{\mathbb{R}}(I)|$.

The last theorem in this part is about equivalent condition on generic linear forms, which is crucial for computation of $\sqrt[\mathbb{R}]{I}$ using linear forms.

Theorem 3.27 ([30, Lemma 2.4]**).** Assume $|V_{\mathbb{R}}(I)| < \infty$. An element $\Lambda \in \mathcal{K}$ is a generic linear form if and only if $\ker\big(M(\Lambda)\big) \subseteq \ker\big(M(\Lambda')\big)$ for all $\Lambda' \in \mathcal{K}$. Moreover, $\ker\big(M(\Lambda)\big) = \sqrt[\mathbb{R}]{I}$ for all generic linear forms $\Lambda \in \mathcal{K}$.

Truncated positive linear forms

Using the results of the previous section, we should be able to solve any system of polynomial equations with finite number of real solutions. From the given polynomial system (3.94) generating an ideal I we construct the set $\mathcal{K}$ (3.108), from which we find a generic linear form Λ (Definition 3.26). Then, by Theorem 3.27, we find the real radical ideal $\sqrt[\mathbb{R}]{I} = \ker\big(M(\Lambda)\big)$. Next, monomial basis $\mathcal{B}$ of the quotient ring $R[x]/\sqrt[\mathbb{R}]{I}$ is found using Theorem 3.23. In the end, the multiplication matrices are obtained and the real solutions of (3.94) are found from their eigenvectors.

However, since we deal with infinite-dimensional spaces $\mathbb{R}[x]$ and $\mathbb{R}[x]^*$, this method is not applicable computationally. Therefore, we restrict ourselves to the finite-dimensional subspaces $\mathbb{R}[x]_s$ and $\big(\mathbb{R}[x]_{2s}\big)^*$, where $[x]_s$ denotes the vectors of all monomials up to degree $s \in \mathbb{N}$. Again we define the quadratic form $Q_\Lambda : \mathbb{R}[x]_s \mapsto \mathbb{R}$ of the linear form $\Lambda \in \big(\mathbb{R}[x]_{2s}\big)^*$ such that

$$Q_\Lambda(f) = \Lambda\big(f^2\big) \tag{3.111}$$

with matrix $M_s(\Lambda)$ denoted as truncated moment matrix of order s of Λ defined below.

Definition 3.28 (Truncated moment matrix of Λ [30]). Given linear form $\Lambda \in \big(\mathbb{R}[x]_{2s}\big)^*$, the truncated moment matrix $M_s(\Lambda)$ of order $s \in \mathbb{N}$ has form

$$M_s(\Lambda)^{(\alpha,\beta)} = \Lambda\big(x^\alpha x^\beta\big) \tag{3.112}$$

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

Linear form Λ is positive if and only if $\Lambda\big(f^2\big) \geq 0$ for all $f \in \mathbb{R}[x]_s$, which is equivalent to the condition $M_s(\Lambda) \succeq 0$. As before, we identify $\ker Q_\Lambda$ and $\ker\big(M_s(\Lambda)\big)$ as a subset of $\mathbb{R}[x]_s$.

Theorem 3.29 (Flat extension theorem [30, Theorem 2.8]). Let $\Lambda \in \big(\mathbb{R}[x]_{2s}\big)^*$ and assume that $M_s(\Lambda)$ is a flat extension of $M_{s-1}(\Lambda)$, i.e.

$$\operatorname{rank} M_s(\Lambda) = \operatorname{rank} M_{s-1}(\Lambda). \tag{3.113}$$

Then, one can extend (uniquely) Λ to $\tilde{\Lambda} \in \big(\mathbb{R}[x]_{2s+2}\big)^*$ is such a way that $M_{s+1}(\tilde{\Lambda})$ is a flat extension of $M_s(\Lambda)$, thus $\operatorname{rank} M_{s+1}(\tilde{\Lambda}) = \operatorname{rank} M_s(\Lambda)$.

This theorem is a crucial one for the moment method, since is allows to conclude the information about the infinite moment matrix $M(\Lambda)$ from its finite part $M_s(\Lambda)$. The following theorem describes how this can be done.

Theorem 3.30 ([30, Theorem 2.9]). Let $\Lambda \in \big(\mathbb{R}[x]_{2s}\big)^*$ and assume (3.113) holds true. Then, one can extend Λ to $\tilde{\Lambda} \in \mathbb{R}[x]^*$ in such a way that $M(\tilde{\Lambda})$ is a flat extension of $M_s(\Lambda)$, and the ideal $\ker\big(M(\tilde{\Lambda})\big)$ is generated by the polynomials in $\ker\big(M_s(\Lambda)\big)$, i.e.

$$\operatorname{rank} M(\tilde{\Lambda}) = \operatorname{rank} M_s(\Lambda), \tag{3.114}$$

$$\ker\big(M(\tilde{\Lambda})\big) = \big\langle \ker\big(M_s(\Lambda)\big)\big\rangle. \tag{3.115}$$

Moreover, any monomial set $\mathcal{B}$ indexing a basis of the column space of $M_{s-1}(\Lambda)$ is a basis of the quotient space $\mathbb{R}[x]/\ker\big(M(\tilde{\Lambda})\big)$. If, moreover, $M_s(\Lambda) \succeq 0$, then the ideal $\big\langle \ker\big(M_s(\Lambda)\big)\big\rangle$ is real radical ideal and Λ is of the form

$$\Lambda = \sum_{i=1}^{r} \lambda_i \Lambda_{v_i}, \tag{3.116}$$

where $\lambda_i > 0$ and $\{v_1, v_2, \ldots, v_r\} = V_{\mathbb{C}}\Big(\ker\big(M_s(\Lambda)\big)\Big) \subseteq \mathbb{R}^n$.

This result allows as to represent a real radical ideal $\sqrt[\mathbb{R}]{I}$ (infinite set of polynomials) by a finite truncated moment matrix $M_s(\Lambda)$. Moreover, the monomial basis $\mathcal{B}$ of the quotient ring $R[x]/\sqrt[\mathbb{R}]{I}$ can be readily obtained from it.

Now, the theory about positive linear forms and flat extensions of truncated moment matrices can be used to construct an algorithm for computing $\sqrt[\mathbb{R}]{I}$ from the generators of an ideal I operating on finite-dimensional subspaces $\mathbb{R}[x]_t$ only. Given $\langle f_1, f_2, \ldots, f_m\rangle = I$ from the polynomial system (3.94) to solve and $t \in \mathbb{N}$, we define the set

$$\mathcal{F}_t = \big\{ f_i x^\alpha \mid i = 1, 2, \ldots, m, |\alpha| \leq t - \deg(f_i) \big\} \tag{3.117}$$

of prolongations up to degree t of the polynomials f_i. The truncated analogue of the set $\mathcal{K}$ is defined as

$$\mathcal{K}_t = \Big\{ \Lambda \in \big(\mathbb{R}[x]_t\big)^* \mid \Lambda(1) = 1, M_{\lfloor t/2 \rfloor}(\Lambda) \succeq 0, \Lambda(f) = 0 \ \forall f \in \mathcal{F}_t \Big\}. \tag{3.118}$$

Since the set $\mathcal{K}_t$ is an intersection of a cone of positive semidefinite matrices with an affine space, the set is a spectrahedron. This property allows us to use a SDP solver to find an element of this set, named generic truncated linear form. The required properties of this element describes the following theorem, which is the truncated analogue of Theorem 3.27.

Theorem 3.31 (Generic truncated linear forms [30, Lemma 2.6]**).** The following assertions are equivalent for $\Lambda \in \big(\mathbb{R}[x]_t\big)^*$:

1. $\operatorname{rank} M_{\lfloor t/2 \rfloor}(\Lambda) \geq \operatorname{rank} M_{\lfloor t/2 \rfloor}(\Lambda')$ for all $\Lambda' \in \mathcal{K}_t$.
2. $\ker\big(M_{\lfloor t/2 \rfloor}(\Lambda)\big) \subseteq \ker\big(M_{\lfloor t/2 \rfloor}(\Lambda')\big)$ for all $\Lambda' \in \mathcal{K}_t$.
3. The linear form Λ lies in the relative interior of the convex set $\mathcal{K}_t$.

Then, Λ is called a generic element of $\mathcal{K}_t$ and the kernel $\mathcal{N}_t = \ker\big(M_{\lfloor t/2 \rfloor}(\Lambda)\big)$ is independent of the particular choice of the generic element $\Lambda \in \mathcal{K}_t$.

Theorem 3.32 ([30, Theorem 2.10]**).** We have

$$\mathcal{N}_t \subseteq \mathcal{N}_{t+1} \subseteq \ldots \subseteq \sqrt[\mathbb{R}]{I}, \tag{3.119}$$

with equality $\sqrt[\mathbb{R}]{I} = \langle \mathcal{N}_t \rangle$ for t large enough.

Now, we are almost ready to write down the algorithm, which will be described in the following section. All we need is the stopping criterion, which will describe Theorem 3.33 and the certificate of termination, which is ensured by Theorem 3.34.

Theorem 3.33 ([30, Theorem 2.11]**).** Let $I = \langle f_1, f_2, \ldots, f_m \rangle$ be an ideal in $\mathbb{R}[x]$, $D = \max_{i=1}^m \deg(f_i)$, and $d = \left\lceil \frac{D}{2} \right\rceil$. Let $\Lambda \in \mathcal{K}_t$ be a generic element and assume that at least one of the following two conditions holds:

$$\operatorname{rank} M_s(\Lambda) = \operatorname{rank} M_{s-1}(\Lambda) \text{ for some } D \leq s \leq \left\lfloor \frac{t}{2} \right\rfloor, \tag{3.120}$$

$$\operatorname{rank} M_s(\Lambda) = \operatorname{rank} M_{s-d}(\Lambda) \text{ for some } d \leq s \leq \left\lfloor \frac{t}{2} \right\rfloor. \tag{3.121}$$

Then, $\sqrt[\mathbb{R}]{I} = \big\langle \ker\big(M_s(\Lambda)\big) \big\rangle$, and any basis of the column space of $M_{s-1}(\Lambda)$ is a basis of the quotient space $\mathbb{R}[x]/\sqrt[\mathbb{R}]{I}$.

Theorem 3.34 ([30, Theorem 2.12]**).** Let I be an ideal in $\mathbb{R}[x]$.

1. If $V_{\mathbb{R}}(I) = \emptyset$, then $\mathcal{K}_t = \emptyset$ for t large enough.
2. If $1 \leq |V_{\mathbb{R}}(I)| < \infty$, then for t large enough, there exists an integer s for which (3.121) holds for all $\Lambda \in \mathcal{K}_t$.

The moment matrix algorithm

Algorithm 3.4 describes the moment matrix algorithm for computing real solutions of the system of polynomial equations.

In each iteration of the algorithm, a generic element Λ of the set $\mathcal{K}_t$ has to be found. As we mentioned above, we can view $\mathcal{K}_t$ as a spectrahedron, and therefore we would like to use the knowledge of semidefinite programming to find Λ. Thus, we represent the set $\mathcal{K}_t$ as the feasible region of a semidefinite program and then we can use any of the state of the art SDP solvers to find any relative interior point of the feasible region.

Algorithm 3.4. The moment matrix algorithm for computing real roots. [30, Algorithm 1]

Input:

$f_1, f_2, \ldots, f_m$ generators of an ideal $I = \langle f_1, f_2, \ldots, f_m \rangle$ with $|V_{\mathbb{R}}(I)| < \infty$

Output:

$V_{\mathbb{R}}(I)$ a set of real solutions

$D \leftarrow \max_{i=1}^{m} \deg(f_i)$
$d \leftarrow \left\lceil \frac{D}{2} \right\rceil$
$t \leftarrow D$
done $\leftarrow$ **false**
while not *done* **do**
  $\mathcal{F}_t \leftarrow \left\{ f_i x^\alpha \mid i = 1, 2, \ldots, m, |\alpha| \leq t - \deg(f_i) \right\}$
  $\Lambda \leftarrow$ any generic element of the set $\mathcal{K}_t = \left\{ \Lambda \in \left(\mathbb{R}[x]_t\right)^* \mid \Lambda(1) = 1, M_{\lfloor t/2 \rfloor}(\Lambda) \succeq 0, \Lambda(f) = 0 \ \forall f \in \mathcal{F}_t \right\}$
  if $\operatorname{rank} M_s(\Lambda) = \operatorname{rank} M_{s-1}(\Lambda)$ for some $D \leq s \leq \left\lfloor \frac{t}{2} \right\rfloor$ **or** $\operatorname{rank} M_s(\Lambda) = \operatorname{rank} M_{s-d}(\Lambda)$ for some $d \leq s \leq \left\lfloor \frac{t}{2} \right\rfloor$ **then**
    $J \leftarrow \left\langle \ker\left(M_s(\Lambda)\right) \right\rangle$
    $\mathcal{B} \leftarrow$ a monomial set indexing a basis of the column space of $M_{s-1}(\Lambda)$
    $G \leftarrow$ a basis of the ideal J
    $\mathcal{X} \leftarrow$ a multiplication matrix (computed using $\mathcal{B}$ and G)
    $V_{\mathbb{R}}(I) \leftarrow V_{\mathbb{C}}(J)$ (computed via the eigenvectors of $\mathcal{X}$)
    done $\leftarrow$ **true**
  else
    $t \leftarrow t + 1$
  end if
end while
return $V_{\mathbb{R}}(I)$

Since we are interested in any feasible point of $\mathcal{K}_t$, we will optimize a constant function. The following semidefinite program (3.122) fulfils all our requirements and by solving it, we obtain a generic linear form Λ.

$$\begin{aligned} \min_{\Lambda \in (\mathbb{R}[x]_t)^*} \quad & 0 \\ \text{s.t.} \quad & \Lambda(1) = 1 \\ & M_{\lfloor t/2 \rfloor}(\Lambda) \succeq 0 \\ & \Lambda\left(f_i x^\alpha\right) = 0 \quad \forall i \ \forall |\alpha| \leq t - \deg(f_i) \end{aligned} \tag{3.122}$$

Given the linear form Λ, ranks of its moment matrices $M_s(\Lambda)$ for many values of s have to be computed in order to check the stopping conditions (3.120) and (3.121). This may become very challenging task, since we need to compute ranks of matrices consisting of numerical values. This is typically done by singular values computation, but is has to be treated very carefully.

When one of the stopping conditions holds for some s, there are as many linear independent columns of $M_{s-1}(\Lambda)$ as many real solutions there are. We select the monomials indexing these columns and they form the basis $\mathcal{B}$ of the quotient ring $\mathbb{R}[x]/\sqrt[\mathbb{R}]{I}$.

Then, we select any variable x_i, for which we construct the multiplication matrix $\mathcal{X}_{x_i}$. For the construction, the polynomials from $\ker\left(M_s(\Lambda)\right)$ are used since for each $b \in \mathcal{B}$

the monomial bx_i can be rewritten as

$$bx_i = \sum_j \lambda_j b_j + q, \tag{3.123}$$

where $b_j \in \mathcal{B}$, $\lambda_j \in \mathbb{R}$ and $q \in \ker\big(M_s(\Lambda)\big)$.

Having the multiplication matrix $\mathcal{X}_{x_i}$ constructed, the real solutions of the polynomial system (3.94) are found by the eigenvectors computation.

3.4.3. Implementation details

To understand Algorithm 3.4 and the theory behind deeply, we have decided to write our own implementation of the algorithm. Firstly, just to verify our understanding of the algorithm, we have implemented it in MATLAB using optimization toolbox YALMIP [32] in conjunction with the state of the art SDP solver MOSEK [34]. Secondly, to be able to use our own SDP solver from the Polyopt package, presented in Section 2.4, we have implemented Algorithm 3.4 in Python into the Polyopt package. Both implementations are described below.

Implementation in MATLAB with MOSEK toolbox

The implementation in MATLAB is quite straightforward. In each iteration over t, the set $\mathcal{F}_t$ is extended with new polynomials of higher degree. Then, the semidefinite program (3.122) is built. To avoid the notation of linear forms, the program can be rewritten into notation of moment matrices as known from Section 3.2 using LMI and affine constraints only. The fact that $M_{\lfloor t/2 \rfloor}(y)$ is a moment matrix is implied by its structure. Using this, we get a equivalent semidefinite program (3.124).

$$\begin{aligned} y^* = \arg \min_{y \in \mathbb{R}^{\mathbb{N}^n_t}} \; & 0 \\ \text{s.t.} \quad & y_0 = 1 \\ & M_{\lfloor t/2 \rfloor}(y) \succeq 0 \\ & \operatorname{vec}\big(f_i x^\alpha\big)^\top y = 0 \quad \forall i \; \forall |\alpha| \leq t - \deg(f_i) \end{aligned} \tag{3.124}$$

This program is solved by MOSEK [34] using the YALMIP [32] toolbox as an interface. The result from MOSEK is the moment matrix $M_{\lfloor t/2 \rfloor}(y^*)$, since we are not really interested in the y^* values. Then, if the stopping condition (3.120) or (3.121) holds for some s, we are ready to construct the multiplication matrix. Since M_s is a matrix of numerical values, we firstly impose its rank, which we have computed when checking the stopping conditions. This is done via the singular value decomposition (SVD) by annulling the singular values close to zero. We construct the basis $\mathcal{B}$ by selecting the pivot columns of the Gauss-Jordan (G-J) elimination of the matrix M_{s-1}. Because we have the SVD computed already, we use it to compute $\ker(M_s)$. We perform the G-J elimination on it with columns indexed by $\mathcal{B}$ permuted to the rightmost side. By this we have all required monomials expressed in terms of linear combinations of the monomials from $\mathcal{B}$ as stated in (3.123). Finally, the multiplication matrix is constructed and the solutions are extracted via eigenvectors computation. In case that some variable is not present in $\mathcal{B}$ and can not be read directly from the eigenvectors, it can computed easily by substituting $\mathcal{B}$ into the eliminated version of $\ker(M_s)$.

Polyopt package implementation

The implementation from the Polyopt package follows the implementation in MATLAB. The part where they differ is, how to solve the semidefinite program, which is in a form (3.124). Because the SDP solver from the Polyopt package is able to solve only semidefinite programs with LMI constraints as stated in (2.83), we need to eliminate the affine constraints. We use these affine constraints to eliminate as much variables as possible, so only the LMI constraints remain. The new equivalent semidefinite program has form (3.125).

$$\begin{aligned} \bar{y}^* = \arg \min_{\bar{y} \in \mathbb{R}^m} \; & 0 \\ \text{s.t.} \quad & A_0 + \sum_{i=1}^{m} A_i \bar{y}^{(i)} \succeq 0 \end{aligned} \tag{3.125}$$

This problem should be now easily solved by the SDP solver from the Polyopt package as presented in Section 2.4.

Unfortunately, since we lowered the dimension of the semidefinite problem by elimination of some variables, the size $r = \binom{n+t}{n}$ of the matrix $A(\bar{y})$

$$A(\bar{y}) = A_0 + \sum_{i=1}^{m} A_i \bar{y}^{(i)} \tag{3.126}$$

from the LMI constraint stayed the same as before the elimination. This causes that $A(\bar{y})$ is singular independently on the values of $\bar{y}$. Therefore, for each value of $\bar{y}$ there is at least one eigenvalue of $A(\bar{y})$ zero, and thus there is no interior point of the feasible region of the problem (3.125). This is no problem for the moment method algorithm, because we are looking for a point from its relative interior. But since the SDP solver from the Polyopt package is an interior-point method, it can not solve a problem, which has no feasible strictly interior point. This issue can be solved by a method called facial reduction [41], which shrinks the matrix $A(\bar{y})$ and removes the superfluous dimensions so that there will be interior points in the feasible region. However, we did not implemented this method, but it may be a possible improvement to this implementation. Instead of this, we have constructed new semidefinite program (3.127).

$$\begin{aligned} \tau^*, \bar{y}^* = \arg \min_{\tau \in \mathbb{R}, \bar{y} \in \mathbb{R}^m} \; & \tau \\ \text{s.t.} \quad & A_0 + \sum_{i=1}^{m} A_i \bar{y}^{(i)} + \mathcal{I}^r \tau \succeq 0 \\ & \tau \geq 0 \end{aligned} \tag{3.127}$$

Feasible region of this problem has strictly interior points, since for each value of $\bar{y}$ we can find τ large enough that $A(\bar{y}) + \mathcal{I}^r \tau$ has all eigenvalues positive. Moreover, we can easily find a starting point for the SDP solver, for example when fixing $\bar{y}_0 = \begin{bmatrix} 0 & 0 & \cdots & 0 \end{bmatrix}^\top$, any

$$\tau_0 > -\min \left\{ \left\{ \lambda_i \big(A(\bar{y}_0) \big) \right\}_{i=1}^{r}; 0 \right\} \tag{3.128}$$

is suitable starting point for the algorithm.

When we apply the Polyopt SDP solver on the program (3.127), we obtain some optimal τ^*. If τ^* is zero up to the numerical precision, we recover y^* from $\bar{y}^*$ by back

substitution. Then, the moment matrix $M_{\lfloor t/2 \rfloor}(y^*)$ is reconstructed and the solutions are computed in similar way as in the MATLAB implementation. If τ^* is not zero, then the semidefinite program (3.124) has empty feasible region, which means that the original polynomial system (3.94) has no real solutions.

Usage

The class `PSSolver` from the Polyopt package provides Python implementation of the polynomial solver using the moment method. The class method `solve()` finds the real solutions of the polynomial system (3.94). Minimal working example provides Listing 3.3. Detailed information about the execution can be enabled by setting `problem.setPrintOutput(True)`.

Listing 3.3. Typical usage of the class `PSSolver` of the Polyopt package.

```
1: import polyopt
2:
3: # assuming the polynomials fi are already defined
4: problem = polyopt.PSSolver([f1, ..., fm])
5: solution = problem.solve()
```

Example 3.35. The Python code solving the system of polynomial equations consisting of equation (3.21) and (3.22) from Example 3.4 is shown in Listing 3.4.

Listing 3.4. Code for solving system of polynomial equations stated in Example 3.35.

```
1: import polyopt
2:
3: # polynomials of the system
4: f1 = {(0, 0): 48, (1, 0): -16, (2, 0): -20, (1, 1): 1, (0, 1): -1,
       (0, 2): -12}
5: f2 = {(0, 0): 44, (1, 0): 46, (2, 0): 12, (1, 1): -58, (0, 1): -47,
        (0, 2): 3}
6:
7: # initialize the solver
8: problem = polyopt.PSSolver([f1, f2])
9:
10: # enable outputs
11: problem.setPrintOutput(True)
12:
13: # solve!
14: solution = problem.solve()
```

3.4.4. Comparison with the state of the art methods

Efficiency and numerical stability of the new algorithm implementations should be compared to the contemporary state of the art implementations. We would like to compare our implementation with another implementations of the moment method. A MATLAB toolbox called Bermeja should use this method as mentioned in [30], however we were unable to obtain it despite the provided link. Different, but still on moment matrices and SDP solvers built, algorithm uses the MATLAB toolbox Gloptipoly [18]. Then, there are polynomial system solvers based on the Gröbner bases computations,

e.g. the F_4 Algorithm [11], which compute all complex solutions. The most challenging will probably be the comparison with solvers specialized for the given task, for example generated by some automatic generators [22, 23].

To compare these solvers in general, one would generate random polynomial systems, solve them by the selected solvers and compare their results and computational times. But in the end, we are interested in performance on real geometric problems from computer vision computed on data captured from real 3D scenes, and therefore we skip these random polynomial system experiments and refer to the experiments on real geometric problems performed in Chapter 4.

3.5. Conclusions

Firstly, the basics of polynomial algebra and moment matrices have been reviewed.

Secondly, we focused on polynomial optimization. We have reviewed the state of the art method for solving polynomial optimization problems by using hierarchies of semidefinite problems. We have implemented this method in Python and the description of it was given. We have verified the implementation on synthetically generated polynomial optimization problems and compared its performance to the state of the art toolbox Gloptipoly [18]. Since the SDP solver is the most time consuming part of the method, the computation time of our implementation is comparable to Gloptipoly only for small POP problems, which generates semidefinite programs with up to 10 variables, e.g. for POP problems with three unknowns and polynomials up to degree two.

Thirdly, the moment method for solving systems of polynomial equations was reviewed. We have implemented this method in Python with the SDP solver from Chapter 2 and in MATLAB with MOSEK [34] SDP solver. No experiments has been performed, since we will evaluate the performance on minimal problems from computer vision in the next chapter.

4. Minimal problems in geometry of computer vision

Many problems from geometry of computer vision can be modeled by systems of polynomial equations. A problem that requires only the minimal subset of data points to solve the problem is called a minimal problem. A typical example is the 5-point algorithm [45] for relative pose estimation between two cameras given five image correspondences only. In many applications, solvers of these minimal problems are used in the Random Sample Consensus (RANSAC) algorithm [12], where the minimal problems has to be solved repeatedly for a large amount of input data. Thus, these solvers are required to be fast and efficient. The state of the art method is to generate these solvers by automatic generators [22, 23], which are based on Gröbner bases construction and eigenvectors of multiplication matrices computation. In these solvers both real and non-real solutions are computed, but the non-real solutions are discarded, since they have no geometric meaning.

In Section 3.4, we have proposed and implemented an algorithm, which does not need to compute the superfluous non-real solutions, and therefore may be faster than the standard solvers generated by the automatic generator. In this section, we compare the speed and the numerical stability of the state of the art solvers with our implementations of the moment method algorithm for polynomial system solving. For this reason, we have selected few minimal problems from geometry of computer vision, on which we will compare the selected solvers.

4.1. Dataset description

First of all, we describe the scene, which we have chosen for our experiments. It is a real scene of a sculpture of Buddha head taken for the LADIO [2] project. The reconstructed 3D model can be seen at webpage `https://skfb.ly/67ZxD` and the source images are publicly available at Github in repository `alicevision/dataset_buddha`. For the reader we show the reconstructed surface of the sculpture in Figure 4.1.

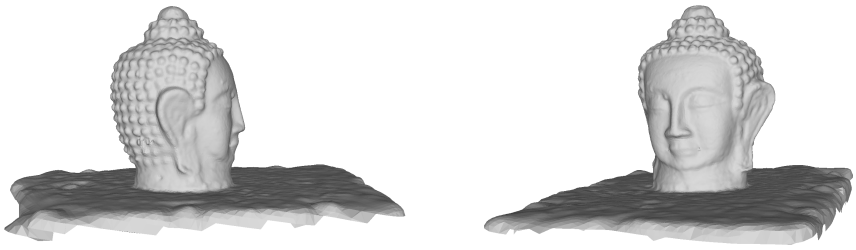

Figure 4.1. Sculpture of Buddha head. Surface representing a point cloud reconstructed from the taken images.

There are 67 taken images of the sculpture, from which 145 001 spatial points were reconstructed using a scene reconstruction pipeline from the photogrammetric framework

named Alice Vision [1]. This complex pipeline consists of SIFT [33] feature detection, RANSAC [12] outlier detection framework using epipolar geometry and incremental structure from motion (SfM) algorithm. This algorithm starts with epipolar geometry between two cameras triangulating the corresponding 2D features into 3D points. Then, new cameras are iteratively added and resectioned based on the 2D-to-3D correspondences using the perspective-n-point (PnP) [51] algorithm in a RANSAC [12] framework. Pose of each added camera is then refined by a non-linear optimization. After that new 3D points are triangulated and by a bundle adjustment (BA) extrinsic and intrinsic parameters of all cameras as well as the position of all 3D points are refined. Then, next iteration of the SfM algorithm is performed until all cameras are estimated. Next step of the reconstruction pipeline is the retrieval of the depth value for each pixel for each reconstructed cameras. The method used in the pipeline is the semi-global matching (SGM) [19] method. After that mesh is created from the reconstructed point cloud by the 3D Delaunay tetrahedralization, which is then textured.

Usage this complex reconstruction pipeline provides us good ground truth values for our experiments with minimal number of outliers.

All the experiments were executed on Intel Xeon E5-1650 v4 CPU 3.60GHz based computer with sufficient amount of free system memory. The installed version of Python was 3.5.3 and MATLAB R2017b 64-bit was used.

4.2. Calibrated camera pose

Computation of calibrated camera pose (its rotation and location with respect to the global coordinate system) is one of the typical problems in computer vision. The pose can be computed from at least three known 3D points and their perspective projection into the image plane, thus the problem is called the perspective-three-point (P3P) problem and it is known since 1841 from [15], but its modern and complete description can be found in [13].

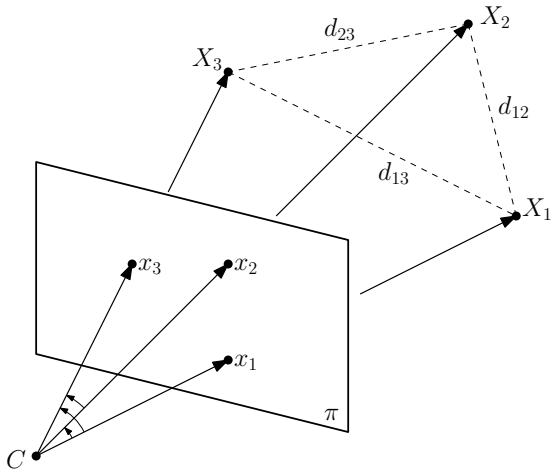

Figure 4.2. Scheme of the P3P problem. A pose of a calibrated camera can be computed from three known 3D points X_1, X_2, X_3 and their projections x_1, x_2, x_3 into the image plane π. The camera projection center is denoted as C. Distances d_{12}, d_{23}, d_{13} denote the distances between the respective 3D points.

The problem is stated followingly: Given three 3D points $X_1, X_2, X_3 \in \mathbb{R}^3$ in the global coordinate system and their projections $x_1, x_2, x_3 \in \mathbb{R}^2$ respectively into the image plane in the image coordinate system, we are looking for a camera projection

center position $C \in \mathbb{R}^3$ and a camera rotation matrix $R \in SO(3)$ – all rotations in 3D space around the origin, such that the projection equation

$$\lambda_i \begin{bmatrix} x_i \\ 1 \end{bmatrix} = K \begin{bmatrix} R & -RC \end{bmatrix} \begin{bmatrix} X_i \\ 1 \end{bmatrix} \tag{4.1}$$

holds for $i = 1, 2, 3$ and $\lambda_i \in \mathbb{R}/\{0\}$, where $K \in \mathbb{R}^{3\times3}$ is known calibration matrix of the camera. The situation is depicted in Figure 4.2.

It has been shown that for a general case this problem can be solved by finding roots of a quartic equation in variable $\xi \in \mathbb{R}$

$$a_4\xi^4 + a_3\xi^3 + a_2\xi^2 + a_1\xi + a_0 = 0 \tag{4.2}$$

with coefficients $a_0, \ldots, a_4 \in \mathbb{R}$, which can be computed by the formulae below.

$$a_4 = -4d_{23}^4 d_{12}^2 d_{13}^2 c_{23}^2 + d_{23}^8 - 2d_{23}^6 d_{12}^2 - 2d_{23}^6 d_{13}^2 + d_{23}^4 d_{12}^4 + 2d_{23}^4 d_{12}^2 d_{13}^2 + d_{23}^4 d_{13}^4 \tag{4.3}$$

$$\begin{aligned} a_3 = {} & 8d_{23}^4 d_{12}^2 d_{13}^2 c_{12} c_{23}^2 + 4d_{23}^6 d_{12}^2 c_{13} c_{23} - 4d_{23}^4 d_{12}^4 c_{13} c_{23} + 4d_{23}^4 d_{12}^2 d_{13}^2 c_{13} c_{23} \\ & - 4d_{23}^8 c_{12} + 4d_{23}^6 d_{12}^2 c_{12} + 8d_{23}^6 d_{13}^2 c_{12} - 4d_{23}^4 d_{12}^2 d_{13}^2 c_{12} - 4d_{23}^4 d_{13}^4 c_{12} \end{aligned} \tag{4.4}$$

$$\begin{aligned} a_2 = {} & -8d_{23}^6 d_{12}^2 c_{13} c_{12} c_{23} - 8d_{23}^4 d_{12}^2 d_{13}^2 c_{13} c_{12} c_{23} + 4d_{23}^8 c_{12}^2 - 4d_{23}^6 d_{12}^2 c_{13}^2 \\ & - 8d_{23}^6 d_{13}^2 c_{12}^2 + 4d_{23}^4 d_{12}^4 c_{13}^2 + 4d_{23}^4 d_{12}^4 c_{23}^2 - 4d_{23}^4 d_{12}^2 d_{13}^2 c_{23}^2 + 4d_{23}^4 d_{13}^4 c_{12}^2 + 2d_{23}^8 \\ & - 4d_{23}^6 d_{13}^2 - 2d_{23}^4 d_{12}^4 + 2d_{23}^4 d_{13}^4 \end{aligned} \tag{4.5}$$

$$\begin{aligned} a_1 = {} & 8d_{23}^6 d_{12}^2 c_{13}^2 c_{12} + 4d_{23}^6 d_{12}^2 c_{13} c_{23} - 4d_{23}^4 d_{12}^4 c_{13} c_{23} + 4d_{23}^4 d_{12}^2 d_{13}^2 c_{13} c_{23} - 4d_{23}^8 c_{12} \\ & - 4d_{23}^6 d_{12}^2 c_{12} + 8d_{23}^6 d_{13}^2 c_{12} + 4d_{23}^4 d_{12}^2 d_{13}^2 c_{12} - 4d_{23}^4 d_{13}^4 c_{12} \end{aligned} \tag{4.6}$$

$$a_0 = -4d_{23}^6 d_{12}^2 c_{13}^2 + d_{23}^8 - 2d_{23}^4 d_{12}^2 d_{13}^2 + 2d_{23}^6 d_{12}^2 + d_{23}^4 d_{13}^4 + d_{23}^4 d_{12}^4 - 2d_{23}^6 d_{13}^2 \tag{4.7}$$

Where the distances d_{12}, d_{23} and d_{13} are

$$d_{12} = \|X_1 - X_2\|, \tag{4.8}$$

$$d_{23} = \|X_2 - X_3\|, \tag{4.9}$$

$$d_{13} = \|X_1 - X_3\| \tag{4.10}$$

and the coefficients c_{12}, c_{23} and c_{13} are cosines of the angles between the respective projection rays, and they can be directly computed from the projected points coordinates.

$$c_{12} = \frac{x_1^\top K^{-\top} K^{-1} x_2}{\|K^{-1}x_1\| \|K^{-1}x_2\|} \tag{4.11}$$

$$c_{23} = \frac{x_2^\top K^{-\top} K^{-1} x_3}{\|K^{-1}x_2\| \|K^{-1}x_3\|} \tag{4.12}$$

$$c_{13} = \frac{x_1^\top K^{-\top} K^{-1} x_3}{\|K^{-1}x_1\| \|K^{-1}x_3\|} \tag{4.13}$$

The equation (4.2) may have zero, two or four real roots, but some of them are discarded by checking three polynomial equations, that the law of cosines holds up to some numerical precision in triangles $\triangle(CX_iX_j)$ for $i, j = 1, 2, 3$ and $i \neq j$, i.e.

$$d_{12}^2 = \|X_1 - C\|^2 + \|X_2 - C\|^2 - 2c_{12}\|X_1 - C\|\|X_2 - C\|, \tag{4.14}$$

$$d_{23}^2 = \|X_2 - C\|^2 + \|X_3 - C\|^2 - 2c_{23}\|X_2 - C\|\|X_3 - C\|, \tag{4.15}$$

$$d_{13}^2 = \|X_1 - C\|^2 + \|X_3 - C\|^2 - 2c_{13}\|X_1 - C\|\|X_3 - C\|. \tag{4.16}$$

The camera pose (C and R) is then computed from each of the remaining solutions.

The P3P problem is probably the simplest problem, which could be chosen from the geometry of computer vision for comparison of the polynomial systems solvers, since only one polynomial of degree four in one variable is given.

In the experiment, we have selected all available 67 cameras, for each of them 1000 triplets of 2D–to–3D correspondences has been randomly chosen. For each triplet, the coefficients a_0, a_1, a_2, a_3, a_4 of the equation (4.2) has been precomputed. Then, the real roots ξ of the equation (4.2) has been found by the selected polynomial solvers. From ξ the camera location C and rotation R has been computed in a standard way. Then, the best tuple C and R minimizing the maximal reprojection error on all correspondences in the image for each camera and solver has been selected.

4.2.1. Performance of the polynomial solvers

We used the described P3P minimal problem to compare following polynomial systems solvers. Firstly, we would like to see the performance of some state of the art purely algebraic solver. A possible candidate is a solver generated by the automatic generator [22], which in case of one degree four polynomial equation in one variable is reduced to eigenvectors computation of multiplication matrix of size 4×4. Secondly, the implementation of the moment method from the Polyopt package has been tested. Thirdly, to be able to compare different implementations of the moment method with different implementation of the SDP solver, we have run the MATLAB implementation with MOSEK toolbox as described in Section 3.4.3. Lastly, the MATLAB toolbox Gloptipoly [18] was used to compare the solvers with another method with built-in optimization.

The histograms of the maximal reprojection errors for the selected tuples of C and R for each polynomial solver can be seen in Figure 4.3. For each estimated camera center position C, we have computed the error e_C of the camera position compared to the ground truth values

$$e_C = \|C - C_{GT}\|, \tag{4.17}$$

i.e. the distance of the estimated camera position to the ground truth position. The histograms of these position errors for each polynomial solver are in Figure 4.4. For each estimated camera rotation R, we have computed the residual rotation to the ground truth camera rotation and computed the angle e_R of this residual rotation as

$$e_R = \arccos\left(\frac{1}{2}\Big(\operatorname{tr}\big(R_{GT}^{-1}R\big) - 1\Big)\right). \tag{4.18}$$

The histograms of the angles of residual rotations for each polynomial solver are in Figure 4.5. We have also measured the execution time required to solve each instance of the equation (4.2) by each polynomial solver and histogram of these times can be found in Figure 4.6. The performance of our implemented polynomial solvers depends on the number of iterations of Algorithm 3.4. This represents the variable t from the algorithm, which is the degree of monomials, which are relaxed. The values of t at which the algorithm terminated are shown as histograms in Figure 4.7. For the Gloptipoly toolbox we have shown the values of two times the relaxation order, which has to be provided for the toolbox in advance. This value is equivalent to the final value of the variable t.

Further off, we are interested in the number of real solutions found by the polynomial solvers. Since the algebraic solver computes all complex solutions first and then the

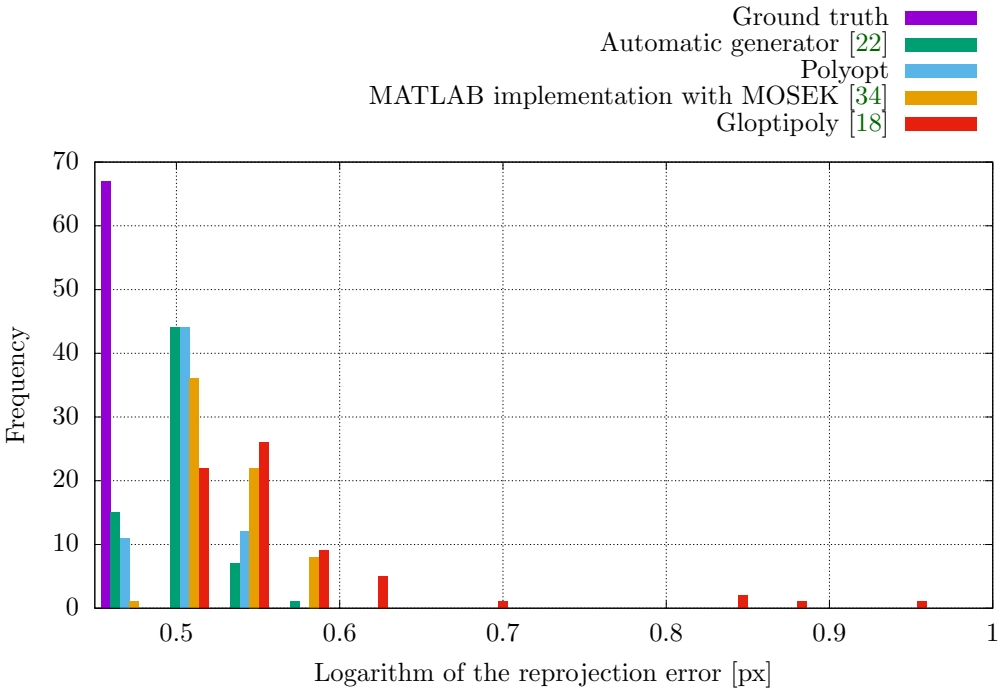

Figure 4.3. Histogram of the maximal reprojection errors of all correspondences in the image for the best camera positions and rotations estimated by the selected polynomial solvers for the P3P problem compared to the maximal reprojection errors computed for the ground truth camera positions and rotations.

non-real filters out, we are sure that this solver finds all real solutions. On the other hand, given our observation, the methods based on optimization do not recover all real solutions, despite the theory. The implementation from the Polyopt package has some numerical issues, which we were unable to remove in the time of writing this thesis, and therefore there is a small chance that the SDP solver fails in its computation and the solution is not found. Moreover, the number of real solutions is related to the rank of the moment matrix found by the SDP solver, which numerically depend on how good representative of the set $\mathcal{K}$ we have obtained. This issue is common for both Polyopt implementation and the MATLAB with MOSEK implementation. In case of the Gloptipoly toolbox, which is in the first case a polynomial optimization toolbox and not a polynomial system solver, the relaxation order has to be given in advance. If the relaxation order is not high enough, no solution is found without distinguishing the cases when there is no solution or just the relaxation order is not high enough. Surprisingly, when the relaxation order is too high, there are not found all the real solutions (typically one or none solution is found), because of reasons not known to us. Therefore, we have set the relaxation order for Gloptipoly to one fixed value, in this case to three, which is the most common relaxation order from all runs of the Polyopt package and the MATLAB with MOSEK implementation, as you can see from Figure 4.7. Since these solvers are typically used in RANSAC-like [12] algorithms, it may not be a big issue, when some of the real solutions are not found, of course depending the application. For the reasons stated above we present in Table 4.1 number of all complex solutions and number of real solutions found by each of the polynomial solvers.

We can see that for the P3P problem there is about 40 % of non-real solutions, which need not be computed. We observe that in practice the moment method based implementations do not found all of the real solutions. The Polyopt implementation

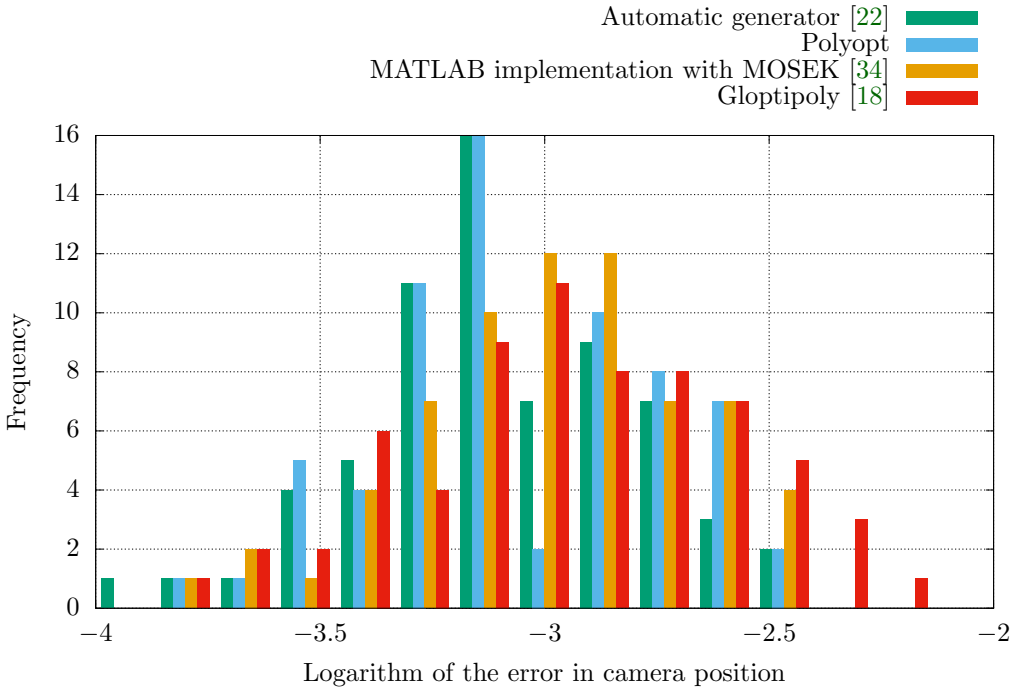

Figure 4.4. Histogram of the errors in estimated camera positions computed by the selected polynomial solvers for the P3P problem with respect to the ground truth camera positions.

found only 80 % of the solutions, but we belive that this can be improved up to 90 % success rate, which shows the MATLAB implementation. Very poor results performed the Gloptipoly toolbox, but they can be probably improved by correct setting of the relaxation order. On the other hand, from the histograms in Figures 4.3, 4.4, 4.5 we can see that the overall results for the P3P problem of the Polyopt implementation and the MATLAB implementation are comparable to the purely algebraic solver, despite they have not found all the real solutions. Regarding the computation times shown in Figure 4.6, we can see that the moment method based implementations are significantly slower than the pure algebraic solver. The best results from the moment method based solvers shows the Gloptipoly [18] toolbox, but note that we have set the relaxation order in advance, and therefore only one semidefinite program had to be solved for each instance.

4.3. Calibrated camera pose with unknown focal length

A computer vision problem slightly more complicated than the P3P problem is the calibrated camera pose estimation with unknown focal length. In this problem the pose and focal length of a calibrated camera is estimated from four known 2D–to–3D correspondences. This problem has seven degrees of freedom, and therefore is over-constrained with four points. A minimal solver for this problem has been presented by C. Wu in [49] using 3.5 points, i.e. ignoring one of the image coordinates for one of the points. Since then, the problem is called the P3.5Pf problem. However, this representation contains a degeneracy, which has been removed in [24] by application of clever parametrization and clever elimination techniques. We use this representation to test our implementations of the moment method. The scheme of the problem geometry can be seen in Figure 4.8.

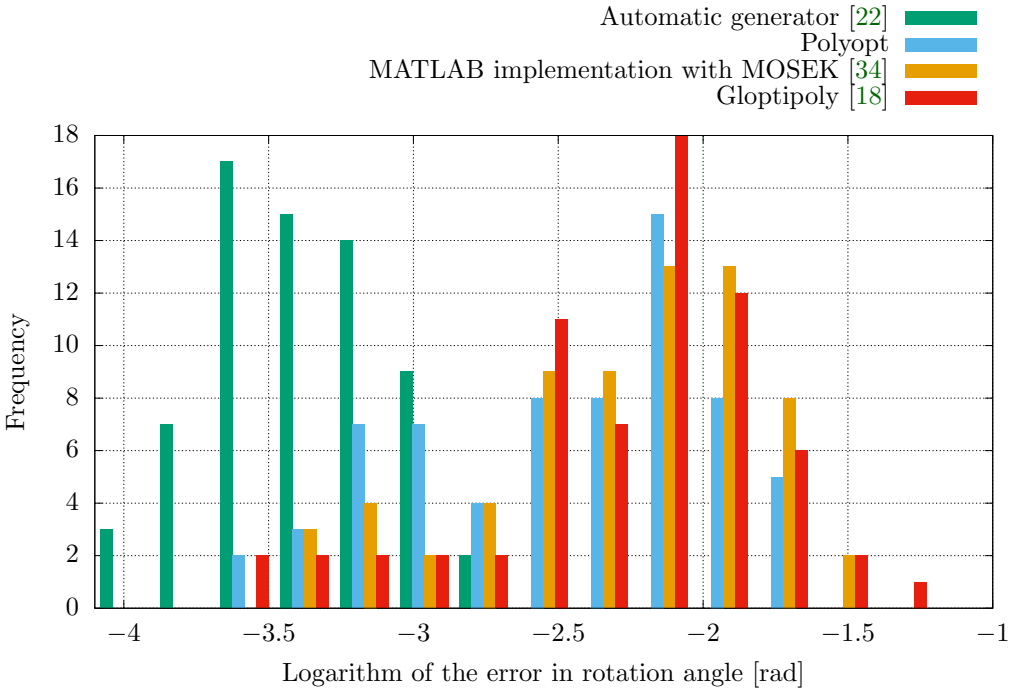

Figure 4.5. Histogram of the errors in rotation angles computed by the selected polynomial solvers for the P3P problem with respect to the ground truth camera rotations.

The solution of the P3.5Pf problem comes from the projection equation

$$\lambda_i \begin{bmatrix} x_i \\ 1 \end{bmatrix} = K_{known} K_f \begin{bmatrix} R & -RC \end{bmatrix} \begin{bmatrix} X_i \\ 1 \end{bmatrix} \tag{4.19}$$

for $i = 1, 2, 3, 4$ and $\lambda_i \in \mathbb{R}/\{0\}$, where $K_{known} \in \mathbb{R}^{3\times3}$ is the known calibration matrix up to the unknown focal length, which is stored in the matrix $K_f = \text{diag}\left(\begin{bmatrix} f & f & 1 \end{bmatrix}^\top \right)$. The matrix $R \in SO(3)$ is the rotation matrix and the vector $C \in \mathbb{R}^3$ is the projection centre of the camera. The 3D coordinates are given in vectors X_i and their projections in vectors x_i. We remove the known calibration matrix from the equation (4.19) by pre-calibrating of the image coordinates.

$$\tilde{x}_i = K_{known}^{-1} x_i \tag{4.20}$$

We group the unknowns f, R and C into one camera projection matrix $P \in \mathbb{R}^{3\times4}$ as follows:

$$K_f \begin{bmatrix} R & -RC \end{bmatrix} = P = \begin{bmatrix} P_1^\top \\ P_2^\top \\ P_3^\top \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix}. \tag{4.21}$$

Then, the projection equation (4.19) is transformed into

$$\lambda_i \begin{bmatrix} \tilde{x}_i \\ 1 \end{bmatrix} = P \begin{bmatrix} X_i \\ 1 \end{bmatrix}. \tag{4.22}$$

Each 2D–to–3D correspondence gives us two linearly independent equations in the

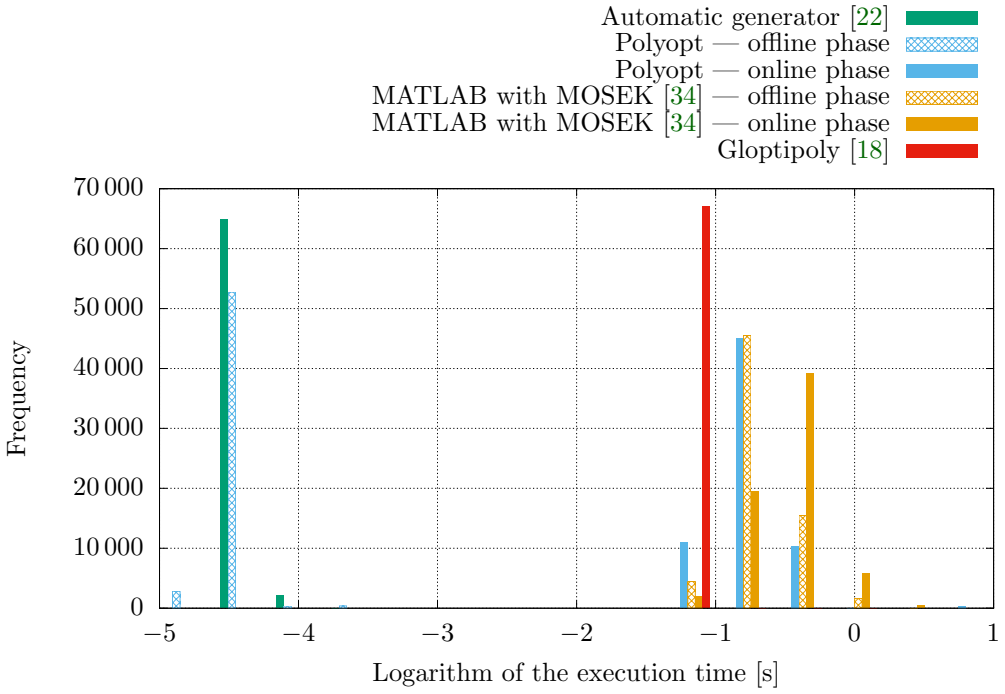

Figure 4.6. Histogram of the execution times required to compute the P3P problem by the selected polynomial solvers.

camera matrix.

$$P_1^\top \begin{bmatrix} X_i \\ 1 \end{bmatrix} - \tilde{x}_i^{(1)} P_3^\top \begin{bmatrix} X_i \\ 1 \end{bmatrix} = 0 \tag{4.23}$$

$$P_2^\top \begin{bmatrix} X_i \\ 1 \end{bmatrix} - \tilde{x}_i^{(2)} P_3^\top \begin{bmatrix} X_i \\ 1 \end{bmatrix} = 0 \tag{4.24}$$

That gives us eight linearly independent equations and by ignoring one of them we get a minimal problem. To fix the scale we add one additional equation

$$P_3^\top \begin{bmatrix} X_1 \\ 1 \end{bmatrix} = 1. \tag{4.25}$$

These eight equations can be written in a matrix form

$$\begin{bmatrix} -X_1^\top & -1 & 0 & 0 & \tilde{x}_1^{(1)} X_1 & \tilde{x}_1^{(1)} \\ 0 & 0 & -X_1^\top & -1 & \tilde{x}_1^{(2)} X_1 & \tilde{x}_1^{(2)} \\ -X_2^\top & -1 & 0 & 0 & \tilde{x}_2^{(1)} X_2 & \tilde{x}_2^{(1)} \\ 0 & 0 & -X_2^\top & -1 & \tilde{x}_2^{(2)} X_2 & \tilde{x}_2^{(2)} \\ -X_3^\top & -1 & 0 & 0 & \tilde{x}_3^{(1)} X_3 & \tilde{x}_3^{(1)} \\ 0 & 0 & -X_3^\top & -1 & \tilde{x}_3^{(2)} X_3 & \tilde{x}_3^{(2)} \\ -X_4^\top & -1 & 0 & 0 & \tilde{x}_4^{(1)} X_4 & \tilde{x}_4^{(1)} \\ 0 & 0 & 0 & 0 & X_1^\top & 1 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.26}$$

$$Ap = b \tag{4.27}$$

with coefficient matrix $A \in \mathbb{R}^{8\times 12}$ and vector $b \in \mathbb{R}^8$. Then, we can parametrize the problem only by four unknowns using the vectors p_1, p_2, p_3, p_4 representing the nullspace

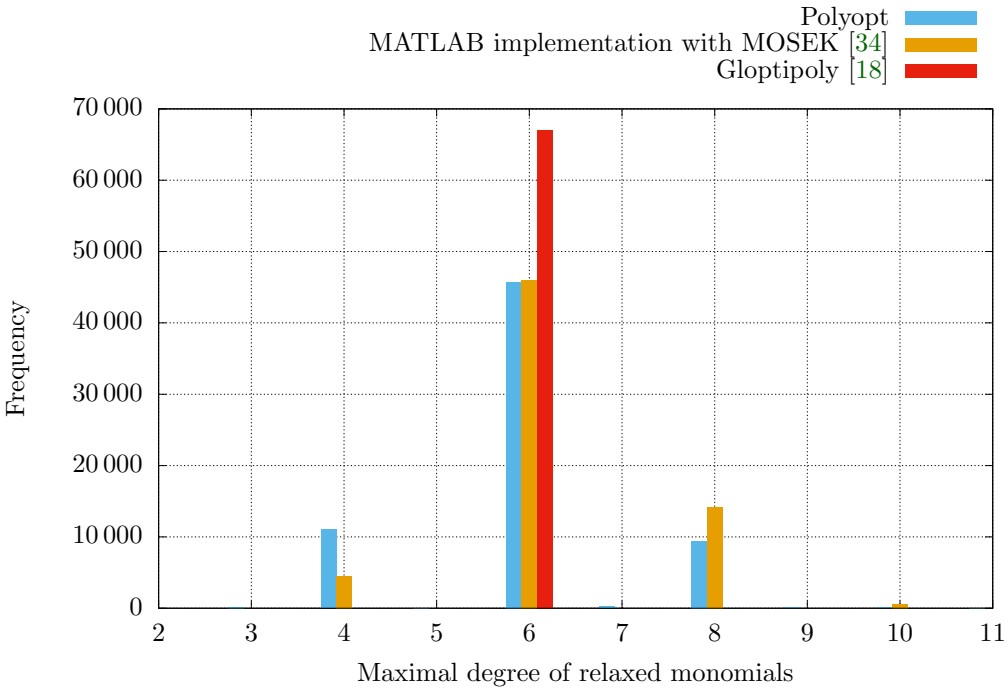

Figure 4.7. Histogram of maximal degrees of relaxed monomials of the P3P problem. It corresponds to the value of variable t in the last iteration of Algorithm 3.4 for the Polyopt package and the MATLAB with MOSEK implementation. For the Gloptipoly toolbox it corresponds to two times the given relaxation order.

of A and particular solution p_0 of the equation (4.27).

$$p = p_0 + \xi_1 p_1 + \xi_2 p_2 + \xi_3 p_3 + \xi_4 p_4 \tag{4.28}$$

To ensure that the camera projection matrix P can be decomposed to K_f, R and C as stated in the equation (4.21) we need to introduce next nine polynomial equations on P.

$$p_{21}p_{31} + p_{22}p_{32} + p_{23}p_{33} = 0 \tag{4.29}$$
$$p_{11}p_{31} + p_{12}p_{32} + p_{13}p_{33} = 0 \tag{4.30}$$
$$p_{11}p_{21} + p_{12}p_{22} + p_{13}p_{23} = 0 \tag{4.31}$$
$$p_{11}^2 + p_{12}^2 + p_{13}^2 - p_{21}^2 - p_{22}^2 - p_{23} = 0 \tag{4.32}$$
$$p_{13}^2 p_{32} - p_{21}^2 p_{32} - p_{22}^2 p_{32} - p_{12}p_{13}p_{33} - p_{22}p_{23}p_{33} = 0 \tag{4.33}$$
$$p_{12}p_{13}p_{32} + p_{22}p_{23}p_{32} - p_{12}^2 p_{33} + p_{21}^2 p_{33} + p_{23}^2 p_{33} = 0 \tag{4.34}$$
$$p_{11}p_{13}p_{32} + p_{21}p_{23}p_{32} - p_{11}p_{12}p_{33} - p_{21}p_{22}p_{33} = 0 \tag{4.35}$$
$$p_{13}^2 p_{31} - p_{22}^2 p_{31} + p_{21}p_{22}p_{32} - p_{11}p_{13}p_{33} = 0 \tag{4.36}$$
$$p_{12}p_{13}p_{31} + p_{22}p_{23}p_{31} - p_{11}p_{21}p_{33} - p_{21}p_{22}p_{33} = 0 \tag{4.37}$$

By solving these equations we obtain the unknowns $\xi_1, \xi_2, \xi_3, \xi_4$, from which we recover the projection matrix P, which we decompose by standard methods to K_f, R and C. In general the system of equations has 10 complex solutions.

In the experiment, we have randomly selected 20 cameras, for each of them 100 quadruples of 2D–to–3D correspondences has been randomly chosen. For each quadruple, the coefficient vectors p_0, p_1, p_2, p_3, p_4 of the equation (4.28) has been precomputed.

Polynomial solver	Number of found real solutions	Percent of found real solutions
Automatic generator [22]	158 850	100.0 %
Polyopt	129 394	81.5 %
MATLAB implementation with MOSEK [34]	141 702	89.2 %
Gloptipoly [18]	71 934	45.3 %

Number of all complex solutions is 268 000.
Number of all real solutions is 158 850, which is 59.3 % of all complex solutions.

Table 4.1. Table of numbers of all real and complex solutions and of numbers of found real solutions by each of the selected polynomial solver for the P3P problem.

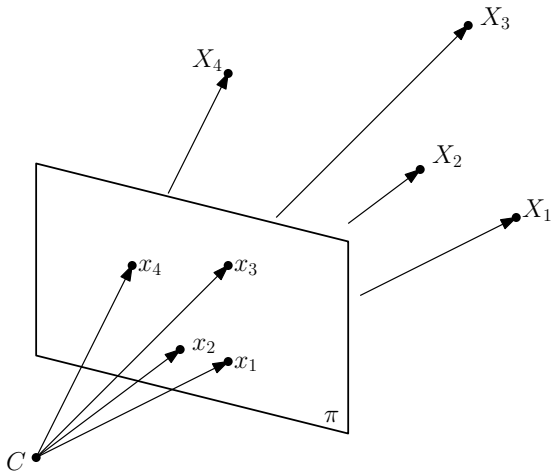

Figure 4.8. Scheme of the P3.5Pf problem. A pose of a calibrated camera with unknown focal length can be computed from four known 3D points X_1, X_2, X_3, X_4 and their projections x_1, x_2, x_3, x_4 into the image plane π. The camera projection center is denoted as C.

Then, the real roots $\xi_1, \xi_2, \xi_3, \xi_4$ of the equations (4.29) – (4.37) has been found by the selected polynomial solvers. The focal length f, the camera location C and the camera rotation R has been computed in a standard way from the solutions. Then, the best triple f, C and R minimizing the maximal reprojection error on all correspondences in the images for each camera and solver has been selected.

4.3.1. Performance of the polynomial solvers

We have tested our two implementations of the moment method, i.e. the Polyopt package and the MATLAB implementation with MOSEK semidefinite programming solver, on the previously described P3.5Pf problem. We compared them with the state of the art polynomial solver Gloptipoly with fixed relaxation order to three and with a pure algebraic solver generated by the automatic generator [22] as described in [24]. The generated solver consists of G-J elimination of a coefficient matrix of size 25×35 and of eigenvector computation of a multiplication matrix of size 10×10.

The histograms of the maximal reprojection errors can bee seen in Figure 4.9. To evaluate the estimated focal length, we computed the relative focal length error for each

of the estimated camera using the following formula:

$$e_f = \left| \frac{f - f_{GT}}{f_{GT}} \right|. \tag{4.38}$$

The histograms of these relative focal length errors are shown in Figure 4.10. We have computed the errors in camera positions with respect to the ground truth values as described by the equation (4.17). These errors are presented as histograms in Figure 4.11. Histograms of the residual rotation angle for each estimated camera computed by the equation (4.18) are in Figure 4.12. Execution times required to solve each instance of the equations (4.29) – (4.37) have been measured and presented in a form of histograms in Figure 4.13. Also the values of the variable t from the last iteration of Algorithm 3.4 and the relaxation orders given to the Gloptipoly toolbox are shown as histograms in Figure 4.14. The numbers of all complex and real solutions as well as the numbers of found real solutions by each of the polynomial solver are written in Table 4.2.

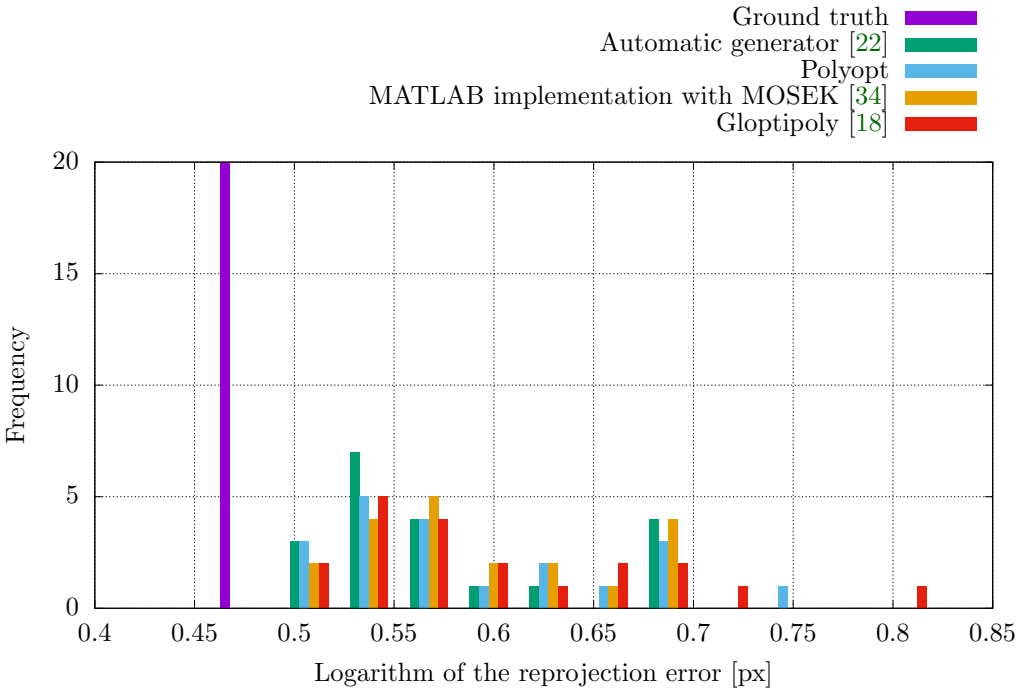

Figure 4.9. Histogram of the maximal reprojection errors of all correspondences in the image for the best camera positions and rotations estimated by the selected polynomial solvers for the P3.5Pf problem compared to the maximal reprojection errors computed for the ground truth camera positions and rotations.

From the results of the polynomial solvers applied on the P3.5Pf problem we can see that there is only 50 % of real solutions amongst all complex solutions. Given efficient polynomial solver computing only real solutions, in theory we would be able to reduce the computation time to half in contrary to the state of the art algebraic solvers, which compute all complex solutions. Although the moment based solvers did not recover all real solutions, the overall results are comparable to the algebraic solver as can be seen from the histograms. The only disadvantage of the moment method based solvers is the computation time, which is incomparable to the algebraic solver.

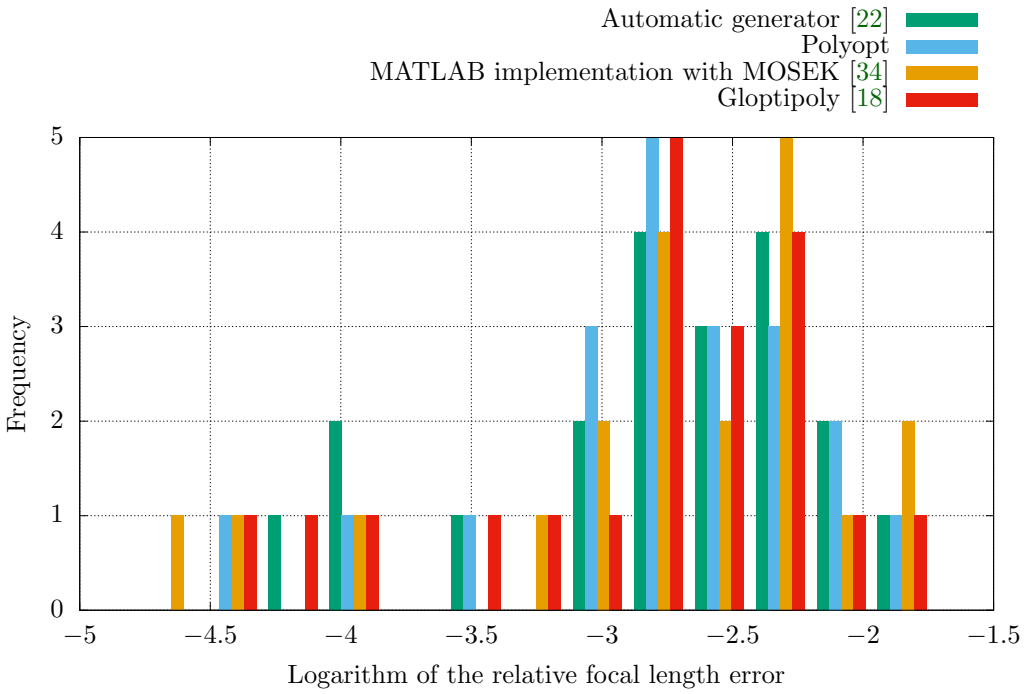

Figure 4.10. Histogram of the relative focal length errors computed by the selected polynomial solvers for the P3.5Pf problem with respect to the ground truth focal lengths.

4.4. Conclusions

We have described two minimal problems of computer vision: the calibrated camera pose problem and the calibrated camera pose with unknown focal length problem. We have solved the polynomial systems arisen from these problems by our implementation of moment method in Python and MATLAB and compared it to Gloptipoly [18] toolbox and to algebraic solvers generated by the automatic generator [22].

Our implementation succeeded in solving these problems, which one them is one polynomial of degree four in one variable and the second one is a system of nine polynomials of degrees two and three in four variables.

The reprojection errors, errors in camera positions and rotations and the focal length relative errors of cameras estimated by the moment method are comparable to the errors of cameras estimated by the algebraic solvers. On the other hand, the algebraic solvers are much faster than the implementations of the moment method.

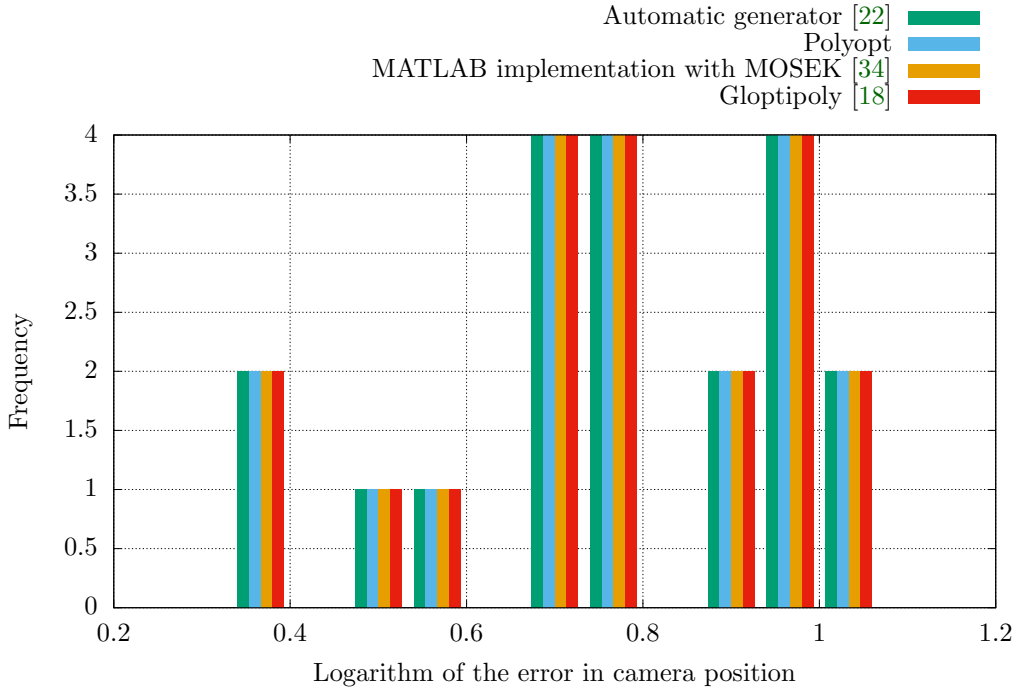

Figure 4.11. Histogram of the errors in estimated camera positions computed by the selected polynomial solvers for the P3.5Pf problem with respect to the ground truth camera positions.

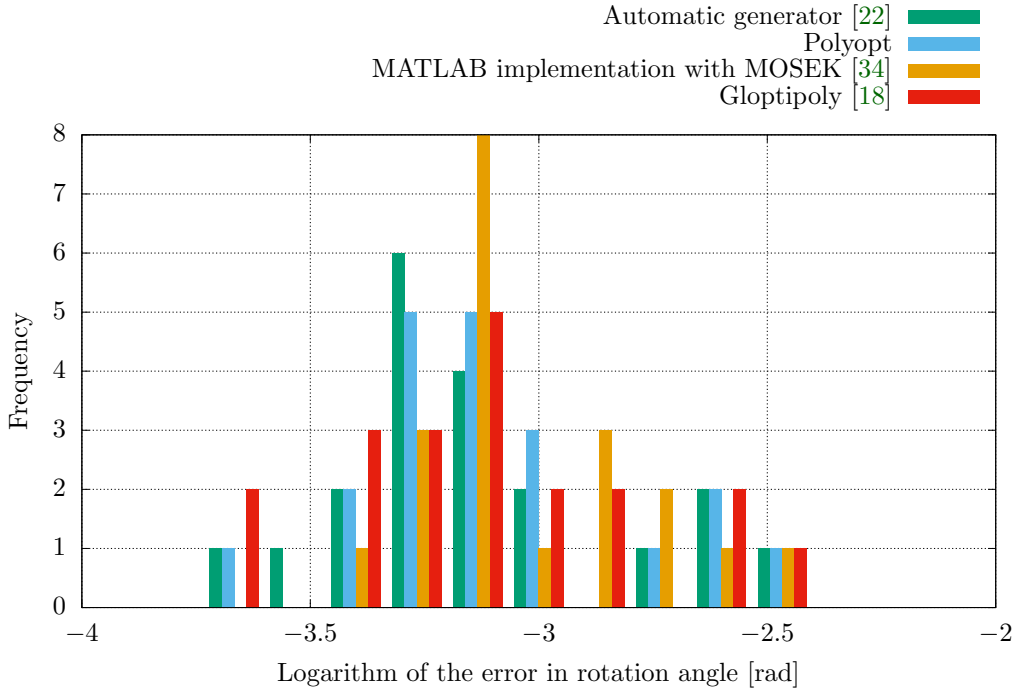

Figure 4.12. Histogram of the errors in rotation angles computed by the selected polynomial solvers for the P3.5Pf problem with respect to the ground truth camera rotations.

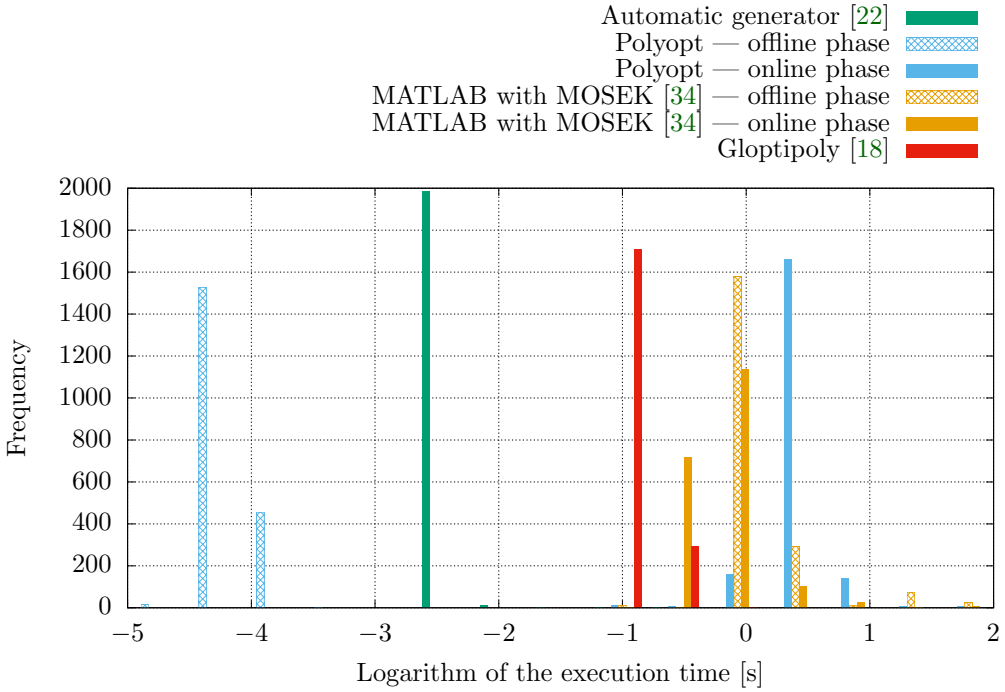

Figure 4.13. Histogram of the execution times required to compute the P3.5Pf problem by the selected polynomial solvers.

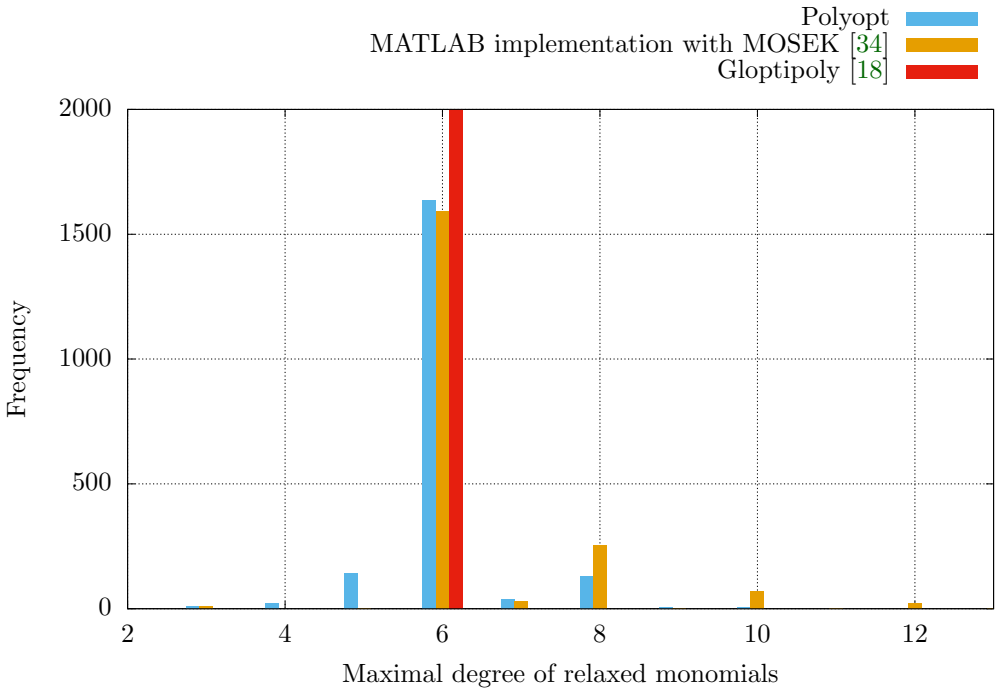

Figure 4.14. Histogram of maximal degrees of relaxed monomials of the P3.5Pf problem. It corresponds to the value of variable t in the last iteration of Algorithm 3.4 for the Polyopt package and the MATLAB with MOSEK implementation. For the Gloptipoly toolbox it corresponds to two times the given relaxation order.

Polynomial solver	Number of found real solutions	Percent of found real solutions
Automatic generator [22]	9608	100.0 %
Polyopt	8110	84.4 %
MATLAB implementation with MOSEK [34]	8698	90.5 %
Gloptipoly [18]	5907	61.5 %

Number of all complex solutions is 20 000.
Number of all real solutions is 9608, which is 48.0 % of all complex solutions.

Table 4.2. Table of numbers of all real and complex solutions and of numbers of found real solutions by each of the selected polynomial solver for the P3.5Pf problem.

5. Conclusions

In this work, we have reviewed and implemented interior-point method for semidefinite programs solving. In the experiments on synthetic semidefinite programs we have verified the correctness of the implementation and we have compared it to the state of the art methods. The results showed that our implementation is significantly slower than the state of the art methods, but an efficient semidefinite programs solver was not a goal of this work. However, the goal was to understand the implemented method, so we can now exploit its advantages and minimize the effects of its disadvantages when applied in polynomial optimization methods. In Section 2.6 we have shown that typically only few iterations of Algorithm 2.3 are enough to get a good approximation of the optimal point.

Furthermore, we have focused on polynomial optimization. We have implemented a method, which uses hierarchies of semidefinite programs to solve the original non-convex problem. The new implementation has been compared to the state of the art methods on synthetically generated polynomial optimization problems. From the results can be seen that our implementation is slower than the state of the art methods, which is mainly because of the inefficient SDP solver, which is the most time consuming part of the algorithm.

The main contribution of this work is the review and implementation of the moment method for polynomial systems solving. The advantage of this method is that it allows us to find only real solutions of polynomial systems, which can save some computation time. We have successfully applied this method to some minimal problems from geometry of computer vision, which is a novel idea in the field of computer vision.

To see the performance of the implementation of the moment method on some real problems, we have described two simple minimal problems (namely P3P and P3.5Pf problems) from geometry of computer vision, which we have tested on real 3D scene. We have found out that for the P3P problem there is about 40 % of non-real solutions, which need not be computed. For the P3.5Pf problem it is about 50 % of all solutions. The comparison with the algebraic solvers showed that the moment method is applicable on the minimal problems, i.e. that the estimation errors of the camera poses and the focal lengths are comparable to the results of the state of the art methods. The main drawback of the moment method is the computation time, which is significantly higher compared to the algebraic solvers generated by the automatic generator [22].

5.1. Future work

We have seen that the described moment method can be applied on minimal problems from computer vision, but due to its slow computation time it is not comparable to the algebraic methods. The performance can be improved by many ways.

Firstly, it showed up that the semidefinite programs constructed inside the moment method have typically no feasible strictly interior point. Therefore, the interior-point algorithm we have implemented in Chapter 2 can not be used to find a feasible point from the relative interior. In this work, we have solved this issue by solving another semidefinite program (3.127). Different approach can be to implement another SDP

solver, which would use an infeasible interior-point method. Another solution to this issue may bring a method called facial reduction [41]. This method is able to find a reduced version of the original problem by solving a sequence of easier semidefinite programs. Using this method we would be able to remove the superfluous dimensions of the SDP problem, which causes that then the problem is solvable by an interior-point method and moreover the size of the problem is reduced, and therefore some computation time can be saved.

Secondly, it is possible that the idea of automatic generators [22, 23] could be transformed to the optimization world, i.e. that for a given problem we would be able to generate a parametrized solver that would solve the problem efficiently for a given value of parameters of the problem. This idea is proposed in [8]. The authors suggest that if we found out that the SDP relaxation is tight for a given value of parameters, and therefore solves the problem correctly, then under some sufficient conditions the relaxation is tight even for small perturbation of the parameters. Moreover, from histograms in Figure 4.7 and Figure 4.14 we can see that typically one relaxation order prevails over the others. Therefore, it would make sense not to start from the minimal possible relaxation order in Algorithm 3.4, but to solve the problem only for one given relaxation order that would be known in advance.

Although we were unable to show that the moment method can beat the algebraic methods in computation time, there might be a problem on which the moment method will be faster. Such a problem would have probably a lot of non-real solutions and only few real solutions, so the moment method could benefit from its advantages. If such a problem would be found, the moment method could be included amongst the state of the art methods for polynomial systems solving in computer vision.

Another advantage of usage optimization methods for polynomial systems solving, which we have not shown in this work, is that overconstrained systems can be solved by them. An overconstrained system can be solved in precise arithmetic, but typically it has no solution when solved on real noisy data in floating-point arithmetic. But the constraints of such problems can be relaxed and the errors of them minimized, and then the optimization techniques as described in Section 3.3 can be applied. We have not provided such an experiment in this work, but it would be nice to find a problem, on which this approach would be applicable and to see, how this approach is performing compared to the state of the art approaches.

In this work, we have provided one implementation for polynomial optimization problems solving (Section 3.3) and a different one for solving polynomial systems (Section 3.4). This is because of the evolution of this work. But since both these methods are based on hierarchies of semidefinite programs, it makes sense to implement one universal algorithm, which would be able to solve both tasks. The advantages are obvious. We would be able to add constraints in form of polynomial equations to the polynomial optimization problems (3.58), which currently allows only polynomial inequality constraints. On the other hand, we would be able to introduce polynomial inequalities into the systems of polynomial equations and eliminate some solutions by this approach. This would lead to smaller multiplication matrices, and therefore to faster eigenvector computations. A typical example from minimal problems from computer vision may be to impose positivity on focal lengths.

A. Contents of the enclosed CD

- `/`
  - `thesis/` ..... folder with files related to the thesis
    - `data/` ..... dataset for the experiments
    - `sources/`
      - `scripts/` ..... scripts performing the experiments
    - `thesis.pdf` ..... digital copy of the thesis
  - `polyopt/` ..... the polyopt package
    - `polyopt/` ..... sources of the polyopt package
    - `tests/` ..... unit tests
    - `demoPOPSolver.py` ..... polynomial optimization demo
    - `demoPSSolver.py` ..... polynomial systems solving demo
    - `demoSDPSolver.py` ..... semidefinite programming demo
    - `setup.py` ..... install script
  - `momentMethod/` ..... MATLAB implementation of the moment method
    - `solve.m` ..... MATLAB function implementing the moment method

Bibliography

[1] Alice vision: Photogrammetric computer vision framework. https://alicevision.github.io/ [Online; accessed 2017-11-09]. 64

[2] LADIO: Live action data input / output project. https://griwodz.github.io/ladioproject/ [Online; accessed 2017-11-09]. 63

[3] 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. 24, 30

[4] 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. 38, 51

[5] 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. 12

[6] Jacek Bochnak, Michel Coste, and Marie-Françoise Roy. *Real Algebraic Geometry*, volume 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer, Berlin, Germany, 1998. 36

[7] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. 51

[8] Diego Cifuentes, Sameer Agarwal, Pablo A. Parrilo, and Rekha R. Thomas. On the local stability of semidefinite relaxations. *ArXiv e-prints*, October 2017. 79

[9] 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. 35, 36

[10] 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. 12

[11] 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. 38, 51, 62

[12] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, June 1981. 63, 64, 67

[13] Xiao-Shan Gao, Xiao-Rong Hou, Jianliang Tang, and Hang-Fei Cheng. Complete solution classification for the perspective-three-point problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(8):930–943, August 2003. 64

[14] 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. 10

[15] Johann August Grunert. Das pothenotische problem in erweiterter gestalt nebst über seine anwendungen in der geodäsie. *Grunerts Archiv für Mathematik und Physik*, 1:238–248, 1841. 64

[16] Jan Heller and Tomas Pajdla. Gposolver: A matlab/c++ toolbox for global polynomial optimization. *Optimization Methods Software*, 31(2):405–434, March 2016. 42

[17] Didier Henrion. Optimization on linear matrix inequalities for polynomial systems control, September 2014. 43, 44

[18] 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. 42, 48, 50, 51, 52, 53, 61, 62, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77

[19] Heiko Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):328–341, February 2008. 64

[20] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984. 12

[21] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer US, Boston, MA, 1972. 10

[22] 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. 8, 51, 62, 63, 66, 67, 68, 69, 70, 72, 73, 74, 75, 76, 77, 78, 79

[23] Viktor Larsson, Kalle Astrom, and Magnus Oskarsson. Efficient solvers for minimal problems by syzygy-based reduction. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017. 8, 51, 62, 63, 79

[24] Viktor Larsson, Zuzana Kukelova, and Yinqiang Zheng. Making minimal solvers for absolute pose estimation compact and robust. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2017. 68, 72

[25] 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. 42, 43

[26] Jean B. Lasserre. *An Introduction to Polynomial and Semi-Algebraic Optimization*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2015. 40

[27] 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. 52

[28] 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. 52

[29] 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]. 40, 41

[30] 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. 5, 52, 53, 54, 55, 56, 57, 58, 61

[31] 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. 24, 30

[32] Johan Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. 31, 59

[33] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004. 64

[34] 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]. 13, 24, 27, 30, 31, 33, 59, 62, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77

[35] Yurii Nesterov. Squared functional systems and optimization problems. In *High Performance Optimization*, pages 405–440. Springer US, 2000. 42

[36] Yurii Nesterov. *Introductory lectures on convex optimization : A basic course*. Springer, 2004. 5, 13, 14, 15, 16, 17, 18, 19, 20, 21, 25

[37] 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. 12

[38] 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. 13

[39] Michael Overton. On minimizing the maximum eigenvalue of a symmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 9:256–268, April 1988. 12

[40] Gabor Pataki. *On the multiplicity of optimal eigenvalues*. University of Michigan, Ann Arbor, MI (United States), December 1994. 12

[41] Frank Permenter and Pablo A. Parrilo. Partial facial reduction: simplified, equivalent SDPs via approximations of the PSD cone. *ArXiv e-prints*, August 2014. 60, 79

[42] Stephen Prajna, Antonis Papachristodoulou, Peter Seiler, and Pablo A. Parrilo. Sostools: Sum of squares optimization toolbox for matlab, 2004. 42

[43] Mihai Putinar. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal*, 42(3):969–984, 1993. 42

[44] Naum Z. Shor. Class of global minimum bounds of polynomial functions. *Cybernetics*, 23(6):731–734, November 1987. 42

[45] Henrik Stewenius, Christopher Engels, and David Nister. Recent developments on direct relative orientation. *ISPRS Journal of Photogrammetry and Remote Sensing*, 60(4):284–294, May 2006. 63

[46] Jos F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999. 13, 24, 27, 30, 31, 33, 48

[47] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, 2011. 24

[48] 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]. 24, 30

[49] Changchang Wu. P3.5p: Pose estimation with unknown focal length. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 68

[50] 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. 13

[51] Yinqiang Zheng, Yubin Kuang, Shigeki Sugimoto, Kalle Åström, and Masatoshi Okutomi. Revisiting the pnp problem: A fast, general and optimal solution. In *2013 IEEE International Conference on Computer Vision*, pages 2344–2351, December 2013. 64