GpoSolver: A MATLAB/C++ Toolbox for Global Polynomial Optimization

version 1.2

2015, Jan Heller, `hellej1@cmp.felk.cvut.cz`

Contents

Abstract

Global polynomial optimization can be a powerful tool when applied to engineering problems. One of the most successful methods for solving such problems is based on convex linear matrix inequality (LMI) relaxations. Software implementations of this approach can be found for example in MATLAB toolboxes GloptiPoly and YALMIP. MATLAB language makes it very easy when it comes to modeling polynomial problems. However, when using these toolboxes, MATLAB is also required for the problem solving. GpoSolver aims at bridging this gap by providing a MATLAB-based problem modeling toolbox supplemented by a problem solving backend in a form of a C++ template library. Once a problem is conveniently modeled and parametrized in MATLAB, a C++ class is automatically generated by GpoSolver. This class can be easily included into an existing codebase and used to solve different instances of the problem based on the supplied parameters.

1 Introduction

Let $p(\mathbf{x})$, $g_i(\mathbf{x})$, $i = 1, \ldots, k$ be multivariate polynomials in $\mathbf{x} = (x_1, x_2, \ldots, x_m)^\top \in \mathbb{R}^m$, *i.e.*, $p(\mathbf{x}), g_i(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ [8]. The polynomial optimization problem (POP) can be stated as follows:

Problem 1. (Polynomial optimization problem)

$$\begin{array}{rl} \text{minimize} & p(\mathbf{x}) \\ \text{subject to} & g_i(\mathbf{x}) \geq 0,\ i = 1, \ldots, k, \\ \text{where} & \mathbf{x} = (x_1, x_2, \ldots, x_m)^\top \in \mathbb{R}^m, \\ & p(\mathbf{x}), g_i(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]. \end{array}$$

Since polynomial equalities can be expressed using pairs of opposite inequalities, *i.e.*,

$$g_i(\mathbf{x}) = 0 \;\Leftrightarrow\; g_i(\mathbf{x}) \geq 0 \wedge g_i(\mathbf{x}) \leq 0,$$

POP encompasses all polynomial optimization problems. In general, POP is an NP-hard problem [25]. In [20], Lasserre suggested to convexify POP using a hierarchy of semidefinite (SDP) relaxations $\mathcal{P}_\delta$, $\delta = 1, 2, \ldots$ Since the SDP problems $\mathcal{P}_\delta$ take the form of linear matrix inequalities (LMI), the hierarchy is sometimes called *Lasserre's LMI hierarchy* and $\mathcal{P}_\delta$ is called a *LMI relaxation of order* δ. The most agreeable property of Lasserre's hierarchy is the fact that the minima of $\mathcal{P}_\delta$ form a monotonically non-decreasing sequence of the lower bounds on the global minimum of Problem 1. Moreover, in most cases a relaxation $\mathcal{P}_{\delta'}$, $\delta' \in \mathbb{N}$ exists such that its minimum is equal to the global minimum of Problem 1.

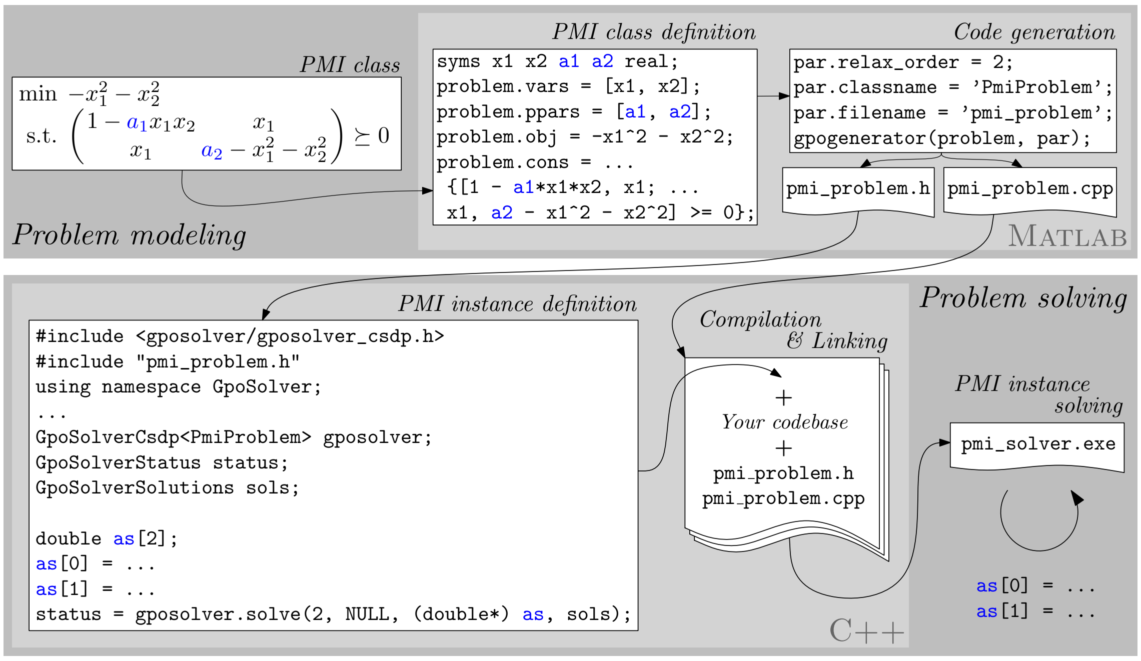
Another optimization problem, closely related to POP, is the polynomial matrix inequalities (PMI) optimization problem:

Problem 2. (Polynomial matrix inequalities optimization problem)

$$\begin{array}{rl} \text{minimize} & p(\mathbf{x}) \\ \text{subject to} & \mathtt{G}_i(\mathbf{x}) \succeq 0,\ i = 1, \ldots, k, \\ \text{where} & \mathbf{x} = (x_1, x_2, \ldots, x_m)^\top \in \mathbb{R}^m, \\ & p(\mathbf{x}) \in \mathbb{R}[\mathbf{x}], \mathtt{G}_i(\mathbf{x}) \in \mathbb{S}^{n_i}(\mathbb{R}[\mathbf{x}]). \end{array}$$

Here, $\mathbb{S}^{n_i}(\mathbb{R}[\mathbf{x}])$ stands for the set of $n_i \times n_i$ symmetric matrices with polynomial entries and the condition $\mathtt{G}_i(\mathbf{x}) \succeq 0$ means that the polynomial matrix $\mathtt{G}_i(\mathbf{x})$ is positive semidefinite. Note that if $n_i = 1$, $i = 1, ..., k$, Problem 2 becomes equivalent to Problem 1, *i.e.*, POP is a subset of PMI. Also note that if the polynomial degree of $p(\mathbf{x})$ and the maximal polynomial degree of all the entries of $\mathtt{G}_i(\mathbf{x})$, $i = 1, ..., k$, is one, PMI becomes LMI. Even though PMI is a strict superset of POP, it was shown in [17] that PMI is amendable to solution using an LMI hierarchy analogous to the LMI hierarchy for POP. In this work, we will be interested in implementing solutions for PMI, POP, and LMI problems. We will refer to these problems by the name of the largest problem set as PMI problems.

Many engineering problems can be formulated as POP or PMI and practically solved using an LMI relaxation of a sufficient order with certifiable global optima, *e.g.*, [19, 21, 29, 7, 11, 13, 12] to cite a few. It is thus of practical interest to implement such solutions in software. The theory of Lasserre's LMI hierarchy became a practical engineering tool after the introduction of the MATLAB toolbox GloptiPoly [15, 14, 18]. Using this toolbox, in the first phase, a user models POPs in a natural mathematical language. In a second phase, given a POP and a relaxation order δ, GloptiPoly automatically constructs the appropriate relaxation $\mathcal{P}_\delta$ and solves it using the SDP solver SeDuMi [31]. Further, it also tries to provide a certificate of global optimality and, if possible, it tries to extract the set of globally optimal solutions to the original POP from the solution of $\mathcal{P}_\delta$.



```
syms x1 x2 a1 a2 real;
problem.vars = [x1, x2];
problem.ppars = [a1, a2];
problem.obj = -x1^2 - x2^2;
problem.cons = ...
 {[1 - a1*x1*x2, x1; ...
 x1, a2 - x1^2 - x2^2] >= 0};
```

```
par.relax_order = 2;
par.classname = 'PmiProblem';
par.filename = 'pmi_problem';
gpogenerator(problem, par);
```

```
#include <gposolver/gposolver_csdp.h>
#include "pmi_problem.h"
using namespace GpoSolver;
...
GpoSolverCsdp<PmiProblem> gposolver;
GpoSolverStatus status;
GpoSolverSolutions sols;

double as[2];
as[0] = ...
as[1] = ...
status = gposolver.solve(2, NULL, (double*) as, sols);
```

```
as[0] = ...
as[1] = ...
```

Figure 1: *Example of GpoSolver workflow*. The workflow is divided into the problem modeling phase and the problem solving phase. Problem parameters of the PMI class are denoted in blue. The concrete values of these parameters are not determined until the problem solving phase. There, the parameters can be easily updated and different PMI instances conveniently solved.

Another MATLAB toolbox that can be used to solve POPs using the LMI relaxation hierarchy is YALMIP [23]. It provides the user with the same amount of modeling comfort as GloptiPoly and besides POP can solve PMI problems as well. Along with SeDuMi, YALMIP also interfaces many other SDP solvers and gives the user a very transparent way of switching between them.

SparsePOP [3] toolbox aims at solving larger scale POPs by exploiting the sparsity structure of the polynomials involved [33]. It can be used as a MATLAB toolbox with SeDuMi and SDPA [24] SDP solvers, or as a standalone executable linked against SDPA.

Dual to the Lasserre's LMI hierarchy is the polynomial sum of squares approach [28, 22]. This approach again leads to semidefinite programming problems. SOSTOOLS [5] is a MATLAB toolbox that implements this method. Among other things, it can provide lower bounds on POPs and, under certain circumstances, recover their global minimizers. However, in the case a POP has more than one global minimizer, SOSTOOLS is unable to recover them.

This work presents a MATLAB toolbox/C++ library for solving POP and PMI problems using the Lasserre's LMI hierarchy approach called GpoSolver. GpoSolver aims at situations where one wishes to repeatedly solve instances of a specific PMI problem with coefficients arising from concrete physical measurements. It combines several advantages. It is able to solve and to recover multiple global minima of both POP and PMI, it does not need the MATLAB environment in the problem solving phase, it interfaces several SDP solvers, and it is used as a C++ template library, *i.e.*, there is no need for additional standalone executable.

The rest of the paper is structured as follows. First, in Section 2 we give an overview of the GpoSolver toolbox and library. Next, we provide an introduction to the method of solving POP and PMI using the Lasserre's LMI hierarchy in Section 3. Further, we describe the problem modeling and solving in Sections 5, 5, and 6. Finally, we provide several examples of concrete engineering problems implemented and solved by GpoSolver.

2 GpoSolver

The purpose of GpoSolver is to provide a convenient way to repeatedly solve PMI problems with a common underlying structure. In the next, we will borrow from the nomenclature of the object oriented programming languages and call this underlying structure a *PMI class*. A PMI class is a PMI problem with unknowns of three kinds: *problem parameters*, *residual parameters*, and *problem variables*. A *PMI instance* is then a PMI class where numerical values have been substituted for the problem and residual parameters. Specifically, PMI classes addressed by GpoSolver are such that two different PMI instances of the same class are identical up the coefficients of $p(\mathbf{x})$, $g_i(\mathbf{x})$, and $\mathtt{G}_i(\mathbf{x})$, respectively. If a PMI class in fact describes a POP or an LMI, we might refer to it as a POP/LMI class. A detailed description of the problem and residual parameters as well detailed description of the PMI classes is given in Section 4.

Conceptually, using GpoSolver is divided into two phases. The first phase is the *problem modeling* phase and it is implemented as a MATLAB toolbox and described in Section 5. The second phase is the *problem solving* phase and it is implemented as a C++ template library and described in Section 6. Figure 1 depicts the GpoSolver's workflow as well as the relationship between the two phases. First, a PMI class is defined as a MATLAB data structure. Next, based on this data structure, a C++ class describing an LMI relaxation of a given order is generated using `gpogenerator` function of the toolbox. In the problem solving phase, the generated C++ class, together with the GpoSolver C++ template library, is included into the user's code. Finally, by simply changing the values of the problem and residual parameters, different instances of the original PMI class can be solved. This two-phase approach has the advantage of convenient problem modeling in MATLAB, while the final executable can be deployed into a MATLAB-free environment where different instances of the original PMI class can be repeatedly solved.
The MATLAB toolbox will run on every MATLAB supported platform provided that Symbolic Math Toolbox with MuPAD engine is available. The C++ template library is also platform agnostic, however, there are several prerequisites as well. It is based on C++ linear algebra library Eigen [1] and it needs to be linked against at least one of these SDP solvers: CSDP [6], SDPA [24], or MOSEK [2]. Both parts of GpoSolver were tested on Ubuntu Linux 64-bit and Windows 7 64-bit. This paper describes GpoSolver version 1.2. The latest version of the software can be downloaded from the project's website:

```
http://cmp.felk.cvut.cz/gposolver
```

GpoSolver is distributed as an archive file containing the MATLAB toolbox, the C++ template library, and the third party DLL libraries facilitating the usage of GpoSolver under Windows 64-bit. In the next, we will denote the directory root of the GpoSolver distribution as `GPO_ROOT`.

3 Lasserre's LMI hierarchy

In this section, we will give a brief overview of Lasserre's LMI hierarchy. Also, in Algorithm 2 we will show how to solve PMI problems using this hierarchy. Technical details, as well as theoretical justification for the presented technique can be found in [20, 17]. An extensive survey on the topic of polynomial optimization and its connection to the problem of polynomial sum of squares can be found in [22].

Let $\mathbf{x} = (x_1, x_2, \ldots, x_m)^\top \in \mathbb{R}^m$ be a real vector and $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)^\top \in \mathbb{N}^m$ an integer vector. A *monomial* of total degree $n = \sum_{i=1}^m \alpha_i$ is defined as

$$\mathbf{x}^{\boldsymbol{\alpha}} = \prod_{i=1}^{n} x_i^{\alpha_i}.$$

A multivariate polynomial $p(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ of degree $n \in \mathbb{N}$ is a mapping from $\mathbb{R}^m$ to $\mathbb{R}$ defined as a linear combination of monomials of total degree up to n,

$$p(\mathbf{x}) = \sum_{|\boldsymbol{\alpha}| \leq n} p_{\boldsymbol{\alpha}} \mathbf{x}^{\boldsymbol{\alpha}} = \sum_{|\boldsymbol{\alpha}| \leq n} p_{\boldsymbol{\alpha}} x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_m^{\alpha_m} = \mathbf{p}^\top \boldsymbol{\psi}_n(\mathbf{x}),$$

where $\mathbf{p} \in \mathbb{R}^d$ is the vector of coefficients and $\boldsymbol{\psi}_n(\mathbf{x})$ is the canonical basis of $d = \binom{m+n}{m}$ monomials in m unknowns of total degree up to n

$$\boldsymbol{\psi}_n(\mathbf{x}) = (1, x_1, x_2, \ldots, x_m, x_1^2, x_1 x_2, \ldots, x_2^2, x_2 x_3, \ldots)^\top.$$

Analogously, the degree of $\mathtt{P} = (p_{i,j}(\mathbf{x})) \in \mathbb{S}^n(\mathbb{R}[\mathbf{x}])$ is the largest degree of all the polynomial elements of $\mathtt{P}$, $\deg \mathtt{P} = \max_{i,j} \deg p_{i,j}(\mathbf{x})$.

In order to construct Lasserre's LMI hierarchy, we will need to "linearize" the polynomials involved, *i.e.*, to substitute every monomial $\mathbf{x}^{\boldsymbol{\alpha}}$ with a new variable $y_{\boldsymbol{\alpha}} \in \mathbb{R}$. To do this, we define *Riesz functional* $\mathcal{L}_{\mathbf{y}} \colon \mathbb{R}_n[\mathbf{x}] \to \mathbb{R}[\mathbf{y}]$, a linear functional that for an m-variate polynomial of degree n, $p(\mathbf{x}) = \sum_{\boldsymbol{\alpha}} p_{\boldsymbol{\alpha}} \mathbf{x}^{\boldsymbol{\alpha}}$, returns a d-variate polynomial of degree one, $\mathcal{L}_{\mathbf{y}}(p(\mathbf{x})) = \sum_{\boldsymbol{\alpha}} p_{\boldsymbol{\alpha}} y_{\boldsymbol{\alpha}}$, $d = \binom{m+n}{m}$. We will also use $\mathcal{L}_{\mathbf{y}}$ as a matrix operator acting on $\mathbb{S}^n(\mathbb{R}[\mathbf{x}])$ as follows. If $\mathtt{P} \in \mathbb{S}^n(\mathbb{R}[\mathbf{x}])$, then $\mathtt{P}' = \mathcal{L}_{\mathbf{y}}(\mathtt{P})$ gives $\mathtt{P}' = (p'_{i,j}(\mathbf{y})) \in \mathbb{S}^n(\mathbb{R}[\mathbf{y}])$ such that $p'_{i,j}(\mathbf{y}) = \mathcal{L}_{\mathbf{y}}(p_{i,j}(\mathbf{x}))$.

The LMI hierarchy for a PMI Problem $\mathcal{P}$ starts with the relaxation of order $\delta_{\min}$,

$$\delta_{\min} = \max \left\{ 1, \left\lceil \frac{\deg p(\mathbf{x})}{2} \right\rceil, \left\lceil \frac{\deg \mathtt{G}_1(\mathbf{x})}{2} \right\rceil, \ldots, \left\lceil \frac{\deg \mathtt{G}_k(\mathbf{x})}{2} \right\rceil \right\}. \tag{1}$$

An LMI relaxation of any lower order is not possible, because not all of the monomials would be linearized. Next, we need to introduce the so-called *moment matrix* $\mathtt{M}_n(\mathbf{y})$ and *localizing matrices* $\mathtt{M}_n(\mathtt{G}_i, \mathbf{y})$ of $\mathtt{G}_i$:

$$\begin{aligned} \mathtt{M}_n(\mathbf{y}) &= \mathcal{L}_{\mathbf{y}}(\boldsymbol{\psi}_n(\mathbf{x}) \boldsymbol{\psi}_n(\mathbf{x})^\top), \\ \mathtt{M}_n(\mathtt{G}_i, \mathbf{y}) &= \mathcal{L}_{\mathbf{y}}(\boldsymbol{\psi}_n(\mathbf{x}) \boldsymbol{\psi}_n(\mathbf{x})^\top \otimes \mathtt{G}_i(\mathbf{x})), \end{aligned}$$

where the operator '$\otimes$' stands for the Kronecker matrix product. An LMI relaxation $\mathcal{P}_\delta$ of degree $\delta \geq \delta_{\min}$ of the PMI problem $\mathcal{P}$ is constructed as

Problem 3. (LMI relaxation $\mathcal{P}_\delta$ of degree δ)

$$\begin{array}{rl} \text{minimize} & \mathcal{L}_{\mathbf{y}}(p(\mathbf{x})) \\ \text{subject to} & \mathtt{M}_\delta(\mathbf{y}) \succeq 0, \\ & \mathtt{M}_{\delta - d_i}(\mathtt{G}_i, \mathbf{y}) \succeq 0, \ i = 1, \ldots, k, \\ \text{where} & \mathbf{y} = (y_1, y_2, \ldots, y_d)^\top \in \mathbb{R}^d, \\ & d_i = \left\lceil \frac{\deg \mathtt{G}_i(\mathbf{x})}{2} \right\rceil, d = \binom{m+2\delta}{m}. \end{array}$$

Note that since Riesz functional $\mathcal{L}_{\mathbf{y}}$ was used to linearize both the cost function and, via the moment and localizing matrices, the constraints, we can easily see that Problem 3 is an LMI problem. Problem 3 has $d = \binom{m+2\delta}{m}$ variables in vector $\mathbf{y} \in \mathbb{R}^d$.

Let $\mathbf{x}^* \in \mathbb{R}^m$ be a global minimizer of the PMI problem $\mathcal{P}$ and let us assume w.l.o.g. that $\delta_{\min} = 1$. Now, the *Lasserre's LMI hierarchy of problem* $\mathcal{P}$ is defined as $\mathcal{P}_\delta$, $\delta \in \mathbb{N}$. Next, let $\mathbf{y}_\delta^* \in \mathbb{R}^d$ denote the minimizer of $\mathcal{P}_\delta$ and let q_δ denote its cost function. The Lasserre's relaxation hierarchy provides a monotonically non-decreasing sequence $q_1(\mathbf{y}_1^*) \leq q_2(\mathbf{y}_2^*) \leq \cdots \leq p(\mathbf{x}^*)$ of the lower bounds that asymptotically converges to the global minimum of $\mathcal{P}$, $p(\mathbf{x}^*)$. It was shown in [26] that for practical POP problems this convergence happens in finitely many steps, *i.e.*, there exists $j \in \mathbb{N}$, such that $q_j(\mathbf{y}_j^*) = p(\mathbf{x}^*)$, and that the problems for which this convergence is

Algorithm 1 *FeasibilityCheck*: PMI instance $\mathcal{P}$ feasibility check

```
Input: PMI instance $\mathcal{P}$, LMI relaxation $\mathcal{P}_\delta$ minimum $q_{\min}$, tentative feasible point $\mathbf{x}$
Output: feasibility indicator feasible
  $feasible \leftarrow 0$
  // Cost fuction feasibility test
  if $|p(\mathbf{x}) - q_{\min}| >$ restol then // See parameter restol in Section 6.1
    return
  end if
  // Constraints feasibility test
  for all $\mathtt{G} \in \{\mathtt{G}\colon \mathtt{G} \succeq 0$ is a constraint in $\mathcal{P}\}$ do
    for all $e \in \{e\colon e$ is an eigenvalue of $\mathtt{G}(\mathbf{x})\}$ do
      if $-e >$ restol then return end if
    end for
  end for
  $feasible \leftarrow 1$
  return
```

asymptotic are in some sense very rare. Even though there is currently no general approach that could decide the relaxation order for which this finite convergence takes place, several approaches exist that can certify global optimality once a LMI relaxation has been solved [16, 27].

The following proposition gives a sufficient condition to certify the finite convergence of the LMI hierarchy:

Proposition 1. *(Certificate of finite convergence) Let $\mathcal{P}$ be the PMI problem from Problem 2 and let $\mathbf{x}^*$ be its optimal solution. Let $\mathbf{y}^*$ be the optimal solution of the LMI relaxation $\mathcal{P}_\delta$ of order $\delta \geq \delta_{\min}$ and $q(\mathbf{y}) = \mathcal{L}_{\mathbf{y}}(p(\mathbf{x}))$. Then*

$$\operatorname{rank}(\mathtt{M}_{\delta-\delta_{\min}}(\mathbf{y}^*)) = \operatorname{rank}(\mathtt{M}_\delta(\mathbf{y}^*)) \quad \Rightarrow \quad q(\mathbf{y}^*) = p(\mathbf{x}^*).$$

Proposition 1 is based on a result of Curto and Fialkow ([9], Theorem 1.1). Note that this a sufficient but not a necessary condition: global minimum may have been reached for some δ, yet still $\operatorname{rank}(\mathtt{M}_{\delta-\delta_{\min}}(\mathbf{y}^*)) < \operatorname{rank}(\mathtt{M}_\delta(\mathbf{y}^*))$. In Algorithm 2, we will refer to this proposition by the function call $\mathrm{RankCheck}(\mathcal{P}, \mathbf{y}^*)$, returning *true* if the presumptions are met. Extracting a global minimizer $\mathbf{x}^*$ or, in the case there is more than one, the set global minimizers $X^* = \{\mathbf{x}_1^*, \ldots, \mathbf{x}_n^*\} \subset \mathbb{R}^m$ from the solution of the LMI relaxation $\mathbf{y}_\delta^*$ is not as straightforward as certifying the finite convergence. In [16], Henrion and Lasserre described an algorithm that can be used to perform such an extraction. Since it is quite involved and beyond the scope of this introductory section, we will content ourselves with saying that this algorithm can extract up to $\operatorname{rank}(\mathtt{M}_\delta(\mathbf{y}^*))$ solutions based on Cholesky decomposition of $\mathtt{M}_\delta(\mathbf{y}^*)$. In Algorithm 2, we will refer to this method by the function call $\mathrm{ExtractMultipleMinima}(\mathcal{P}, \mathbf{y}^*)$, returning the set of global minimizers X^*.

The method for solving Problem 2—or a PMI instance, in the parlance of Section 2—via the LMI hierarchy used in GloptiPoly as well as in GpoSolver is summed up by Algorithm 2. Given a PMI instance $\mathcal{P}$ and a relaxation order $\delta \geq \delta_{\min}$, the algorithm produces a—possibly empty—set of feasible solutions to $\mathcal{P}$. Using Proposition 1, this set can be certified as a set of globally optimal solutions. Note that since both the extraction algorithm and the PMI instance feasibility check in Algorithm 1 are numerical algorithms, GpoSolver may produce different solutions or may fail to certify global optimality based on the tolerance settings of the algorithm parameters. These parameters are described in Section 6.1. The return codes of Algorithm 2 are described in Section 6.

Algorithm 2 Solution to PMI instance $\mathcal{P}$ via Lasserre's hierarchy of LMI relaxations

```
Input: PMI instance $\mathcal{P}$, relaxation order $\delta \geq \delta_{\min}$
Output: solution status indicator $s$, lower bound $p_{\text{lb}}$, set of global minimizers $X^*$
  $X^* \leftarrow \emptyset$
  $p_{\text{lb}} \leftarrow -\infty$
  $\mathcal{P}_\delta \leftarrow$ LMI relaxation of $\mathcal{P}$ of degree $\delta$ (Problem 3)

  if $\mathcal{P}_\delta$ not primal-dual feasible then
    $s \leftarrow$ INFEASIBLE_SDP
    return
  end if

  $\mathbf{y}_\delta^* = (y_1, y_2, \ldots, y_d) \in \mathbb{R}^d \leftarrow$ minimizer of $\mathcal{P}_\delta$
  $p_{\text{lb}} \leftarrow q_\delta(\mathbf{y}_\delta^*)$ // minimum of $\mathcal{P}_\delta$

  if $\|\mathbf{y}_\delta^*\| >$ maxnorm then // See parameter maxnorm in Section 6.1
    $s \leftarrow$ UNBOUNDED_SOLUTION
    return
  end if

  $\mathbf{x} \leftarrow (y_1, y_2, \ldots, y_m)$ // y's corresponding to the first degree monomials $x_1, x_2, \ldots, x_m$
  feasible $\leftarrow$ FeasibilityCheck$(\mathcal{P}, p_{\text{lb}}, \mathbf{x})$ // See Algorithm 1

  if feasible then
    $X \leftarrow \{\mathbf{x}\}$
  else
    $X \leftarrow$ ExtractMultipleMinima$(\mathcal{P}_\delta, \mathbf{y}_\delta^*)$ // See [16, 27]
    if $X \equiv \emptyset$ then
      $s \leftarrow$ INCOMPLETE_BASIS
      return
    end if
  end if

  for all $\mathbf{x} \in X$ do
    feasible $\leftarrow$ FeasibilityCheck$(\mathcal{P}, p_{\text{lb}}, \mathbf{x})$ // See Algorithm 1
    if feasible then
      $X^* \leftarrow X^* \cup \{\mathbf{x}\}$
    end if
  end for

  if RankCheck$(\mathcal{P}_\delta)$ then // See Proposition 1
    $s \leftarrow$ SUCCESS_GLOPT
  else
    $s \leftarrow$ SUCCESS
  end if
  return
```

4 PMI class parametrization

Before we can start with the PMI class definitions in the MATLAB toolbox, we need to specify the general structure of these classes. Mathematically, these PMI classes lead to problems that are equivalent to Problem 2. However, not every aspect of Problem 2 can be parametrized in GpoSolver at the moment, *e.g.*, the number of constraints must be fixed. The precise formulation of PMI classes solvable by GpoSolver is stated by Problem 4.

Problem 4. (GpoSolver PMI class parametrization)

$$\begin{array}{rl} \text{minimize} & \sum_{j=1}^{\ell} r(\boldsymbol{\alpha}, \boldsymbol{\beta}_j, \mathbf{x}) \\ \text{subject to} & \mathtt{G}_i(\boldsymbol{\alpha}, \mathbf{x}) \succeq 0,\ i = 1, \ldots, k, \\ \text{where} & \boldsymbol{\beta}_j = (\beta_{j1}, \beta_{j2}, \ldots, \beta_{jm''})^\top \in \mathbb{R}^{m''},\ j = 1, \ldots, \ell, \\ & \boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_{m'})^\top \in \mathbb{R}^{m'}, \\ & \mathbf{x} = (x_1, x_2, \ldots, x_m)^\top \in \mathbb{R}^m, \\ & r(\boldsymbol{\alpha}, \boldsymbol{\beta}_j, \mathbf{x}) \in \mathbb{R}[\boldsymbol{\alpha}, \boldsymbol{\beta}_j, \mathbf{x}], \\ & \mathtt{G}_i(\boldsymbol{\alpha}, \mathbf{x}) \in \mathbb{S}^{n_i}(\mathbb{R}[\boldsymbol{\alpha}, \mathbf{x}]). \end{array}$$

Problem 4 allows for two sets of parameters $\boldsymbol{\alpha} \in \mathbb{R}^{m'}$, $\boldsymbol{\beta}_j \in \mathbb{R}^{m''}$, $j = 1, \ldots, \ell$ and for a summation-based cost function. We call the parameters $\boldsymbol{\alpha}$ *problem parameters* and they can appear in the cost function as well as in the constraints. The values of these parameters are to be specified during the problem solving phase, however, the number of problem parameters m' needs to be specified in the modeling phase. The second set of parameters $\boldsymbol{\beta}_j \in \mathbb{R}^{m''}$, $j = 1, \ldots, \ell$ we call *residual parameters*. Again, the number of parameters m'' must be specified in the modeling phase. The values of the residual parameters as well as the number of the parameter blocks ℓ are instance specific. The instance-specific parameter ℓ together with the summation-based cost function allow for PMI classes based on minimization of residual based cost functions. In the modeling phase, only the form of $r(\boldsymbol{\alpha}, \boldsymbol{\beta}_j, \mathbf{x})$ must be specified. If the problem contains no residual parameters, then $\ell = 1$ by default. To sum it up, the values of k, m, m', m'' and n_i need to be specified in the modeling phase. The parameter values as well as the number of residual blocks ℓ are instance dependent and are to be specified in the solving phase.

Another constraint on a PMI class is the requirement that both r and $\mathtt{G}_i$ must be polynomials in the problem variables $\mathbf{x}$ as well as in the parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}_j$. This means that the parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}_j$ can appear as monomials only. If the problem at hand calls for more complicated functions such as $\log$, $\exp$, $\sin$, etc., the user must substitute these using new variables in the modeling phase manually. In the solving phase, one must also take care of correctly “pre-computing” the values of these new variables. The reason behind this constraint is the fact that polynomial functions are easy to translate into C++ code and are defined for all real values.

In the next, we provide several examples of simple problems that are identified as PMI classes solvable by GpoSolver.

Example 1. Example 2.3 from [16].

$$\begin{array}{rl} \text{minimize} & -(x_1 - 1)^2 - (x_1 - x_2)^2 - (x_2 - \alpha_1)^2 \\ \text{subject to} & 2 - \alpha_2^3 - (x_1 - 1)^2 \geq 0, \\ & 1 - (x_1 - x_2)^2 \geq 0, \\ & 1 - (x_2 - \alpha_3)^2 \geq 0. \end{array}$$

This PMI class has $m = 2$ problem variables $\mathbf{x} = (x_1, x_2)^\top \in \mathbb{R}^2$ and $m' = 3$ problem parameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3)^\top \in \mathbb{R}^3$. There are $m'' = 0$ residual parameters. There are $k = 3$ constraints with $n_1 = n_2 = n_3 = 1$. Since all of the constraints have dimension one, this is a POP class. □

Example 2. Example 16 from [17].

$$\begin{array}{ll} \text{minimize} & -x_1^2 - x_2^2 \\ \text{subject to} & \begin{pmatrix} 1-\alpha_1 x_1 x_2 & x_1 \\ x_1 & \alpha_2 - x_1^2 - x_2^2 \end{pmatrix} \succeq 0. \end{array}$$

This PMI class has $m = 2$ problem variables $\mathbf{x} = (x_1, x_2)^\top \in \mathbb{R}^2$ and $m' = 2$ problem parameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)^\top \in \mathbb{R}^2$. We have no residual parameters, hence $m'' = 0$. Finally, we have $k = 1$ constraints with $n_1 = 2$ being the size of the single polynomial matrix constraint. □

Example 3.

$$\begin{array}{ll} \text{minimize} & x_2 \\ \text{subject to} & \alpha_1 \leq x_1 \leq \alpha_2, \\ & \alpha_3 \leq x_3 \leq \alpha_4, \\ & \begin{pmatrix} 1 & x_1 & \alpha_5^2 x_2 \\ x_1 & 1 & x_3 \\ \alpha_5^2 x_2 & x_3 & 1 \end{pmatrix} \succeq 0. \end{array}$$

In this example, we have $m = 3$ problem variables $x = (x_1, x_2, x_3)^\top \in \mathbb{R}^3$ and $m' = 5$ problem parameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5)^\top \in \mathbb{R}^5$. We have $k = 5$ constraints with $n_1 = n_2 = n_3 = n_4 = 1$ and $n_5 = 3$. Note that since the problem variables $\mathbf{x}$ appear as degree one monomials only, this problem is in fact an LMI class that does not have to be solved using an additional LMI relaxation. GpoSolver will recognize LMI classes and will solve them directly. □

Example 4.

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{\ell} (\alpha_1^2 x_1^2 - \beta_{j1} x_2 - \beta_{j2})^2 + (\alpha_2^2 x_1^2 - \beta_{j3} x_2 - \beta_{j4})^2 \\ \text{subject to} & \alpha_3^2 x_1^2 + \alpha_4^2 x_2^2 = 1. \end{array}$$

Here, we have $m = 2$ problem variables $x = (x_1, x_2)^\top \in \mathbb{R}^2$, $m' = 4$ problem parameters $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)^\top \in \mathbb{R}^4$, and, for the first time, also $m'' = 4$ residual parameters $\boldsymbol{\beta}_j = (\beta_{j1}, \beta_{j2}, \beta_{j3}, \beta_{j4})^\top \in \mathbb{R}^4$. We have $k = 1$ constraint. □

5 PMI class definition and code generation

Symbolic Math Toolbox for MATLAB with MuPAD engine is used to define GpoSolver PMI classes using a natural mathematical language. The toolbox is also used to define problem variables and parameters via symbolic variables, for polynomial expression manipulation and for the final code generation (`ccode` function).

5.1 PMI class definition

A PMI class is defined by constructing a MATLAB data structure with up to five fields:

`vars`	Vector of symbolic variables of size m enumerating the problem variables $\mathbf{x}$.
`ppars`	Vector of symbolic variables of size m' enumerating the problem parameters $\boldsymbol{\alpha}$.
`rpars`	Vector of symbolic variables of size m'' enumerating the residual parameters $\boldsymbol{\beta}_j$.

`obj`	Symbolic expression defining the polynomial residual $r(\boldsymbol{\alpha}, \boldsymbol{\beta}_j, \mathbf{x})$.
`cons`	Cell array of symbolic expressions of size k defining the polynomial constraints $\mathtt{G}_i(\boldsymbol{\alpha}, \mathbf{x})$.

Except for `vars`, none of the above fields are mandatory. However, it does not make much sense not to define both `obj` and `cons`. In such case, GpoSolver will produce an error message. If `obj` is not defined, instead of setting a constant cost function, GpoSolver will minimize the trace of the moment matrix, $\mathrm{trace}(\mathtt{M}_\delta(\mathbf{y}))$. It was observed in [15], that such a cost function leads to numerically more stable solutions. In the rest of the paper, we will denote the PMI class data structure as `problem`.

Contrary to the formalistic parametrization in Problem 4 where the constraints take the form $\mathtt{G}_i(\boldsymbol{\alpha}, \mathbf{x}) \succeq 0$, the constraints in the field `cons` can be specified using polynomial expressions on both sides of the inequality and, using the operators '>=' and '<=', one can express inequalities of both directions. Further, equalities are also supported using the operator '=='.

Next, we provide examples of PMI class definitions using the problem examples from the previous section.

Example 5. MATLAB code snippet with the definition of the PMI class from Example 1.

```
syms x1 x2 a1 a2 a3 real;

problem.vars = [x1, x2];
problem.ppars = [a1, a2, a3];
problem.obj = -(x1 - 1)^2 - (x1 - x2)^2 - (x2 - a1)^2;
problem.cons = { 2 - a2^3 - (x1 - 1)^2 >= 0, ...
  1 - (x1 - x2)^2 >= 0, 1 - (x2 - a3)^2 >= 0 };
```

□

Example 6. MATLAB code snippet with the definition of the PMI class from Example 2.

```
syms x1 x2 a1 a2 real;

problem.vars = [x1, x2];
problem.ppars = [a1, a2];
problem.obj = -x1^2 - x2^2;
problem.cons = {[1 - a1*x1*x2, x1; x1, a2 - x1^2 - x2^2] >= 0};
```

□

Example 7. MATLAB code snippet with the definition of the PMI class from Example 3.

```
syms x1 x2 x3 a1 a2 a3 a4 a5 real;

problem.vars = [x1, x2];
problem.ppars = [a1, a2, a3, a4, a5];
problem.obj = x2;
problem.cons = { a1 <= x1, a2 >= x1, a3 <= x3, a4 >= x3, ...
  [1, x1, a5^2*x2; x1, 1, x3; a5^2*x2, x3, 1] >= 0 };
```

□

Example 8. MATLAB code snippet with the definition of the PMI class from Example 4.

```
syms x1 x2 a1 a2 a3 a4 b1 b2 b3 b4 real;

problem.vars = [x1 x2];
problem.ppars = [a1 a2 a3 a4];
problem.rpars = [b1 b2 b3 b4];
problem.obj = (a1^2*x1^2 - b1*x2 - b2)^2 + (a2^2*x1^2 - b3*x2 - b4)^2;
problem.cons = {a3^2 * x1^2 + a4^2 * x2^2 == 1};
```

□

5.2 Code generation

After a PMI class is defined via `problem` data structure, the C++ code with the LMI relaxation definition can be readily generated using function `gpogenerator`. The function will generate a header file and a source file with the same names to a directory of the user's choosing. The function resides in the `GPO_ROOT/matlab` directory of the GpoSolver distribution. It takes two mandatory parameters. The first parameter is the class definition `problem`, the second parameter is a data structure `params`. This data structure can have up to five optional fields:

`filename`	A string containing the full path to the resulting C++ code. For example, if set to `'problem'`, `gpogenerator` will generate two files `problem.cpp` and `problem.h` into the current working directory. If set to `'/home/user/p1.cpp'`, files `p1.cpp` and `p1.h` will be generated into the `/home/user/` directory. The default value is `'gpoproblem'`.
`relax_order`	Relaxation order of the generated LMI relaxation. If not set, `gpogenerator` will choose the minimal possible relaxation $\delta_{\min}$ (see Equation 1).
`classname`	A string containing the name of the generated C++ class. Default value is `'GpoProblem'`.
`cext`	A string containing the file extension of the generated source file. Default value is `'cpp'`.
`hext`	A string containing the file extension of the generated header file. Default value is `'h'`.

Example 9. Let us execute function `gpogenerator` to produce the second order LMI relaxation of the PMI class from Example 4:

```
>> cd GPO_ROOT/matlab;
>> params.relax_order = 2;
>> params.classname = 'PopProblem';
>> params.filename = 'pop_problem';
>> gpogenerator(problem, params);
Optimization variables ... [x1, x2]
Computing problem polynomial degree ... 4
Computing minimum relaxation degree ... 2
Computing moment matrix ... done
Computing localizing matrices ... done
Identifying monomials ... 15
Decomposing LMI constraints ... done
Decomposing objective function ... done
```

```
Generating pop_problem.h... done
Generating pop_problem.cpp... done
>>
```

We can see that the second order relaxation leads to a 15×15 moment matrix $\mathtt{M}_2(\mathbf{y}) \in \mathbb{S}^{15}$. Two files were generated into the current working directory `GPO_ROOT/matlab`: `pop_problem.h` and `pop_problem.cpp`. □

Note that Symbolic Math Toolbox for MATLAB is used extensively by `gpogenerator`. Some of the operations of the toolbox are quite time consuming, especially the code generation phase using the toolbox function `ccode`; `gpogenerator` thus may take quite some time to finish for more complicated PMI classes. The time spent generating the code by the `ccode` function is not easy to predict. However, Section 8 contains timings for several example problems, so that one can have a general idea about the time requirements.

6 PMI instance solving

The problem solving part of GpoSolver consists of a C++ template library located in the directory `GPO_ROOT/include`. GpoSolver can work with the following three SDP solvers: CSDP [6], SDPA [24], and MOSEK [2]. Both CSDP and SDPA are available under open source licenses. MOSEK is a commercial product, however, an academic license can be obtained free of charge. All of these solvers can be used and linked at the same time and a user can choose between them by simply instantiating objects of different names.

To use the library, one must first include at least one of the following header files:

```
gposolver/gposolver_csdp.h
gposolver/gposolver_sdpa.h
gposolver/gposolver_mosek.h
```

depending on the SDP solvers available. Also, the header file generated by `gpogenerator` must be included. Note that every class of the GpoSolver library lives in the namespace `GpoSolver`. Next, an object of `GpoSolverCsdp`, `GpoSolverSdpa`, or `GpoSolverMOSEK` class is instantiated. All of these classes are templated and take `gpogenerator` generated class as the only template parameter. Finally, to solve a PMI instance, one just needs to call a virtual method `solve` declared in the common ancestor class `GpoSolverBase`:

```
GpoSolverStatus GpoSolverBase::solve(
  const int num_res,
  const double *rvals,
  const double *pvals,
  GpoSolverSolutions &sols
);
```

The following table describes the method parameters:

`const int num_res`	The number of residual parameters blocks of the problem instance, *i.e.*, value of ℓ from Problem 4.
`const double *rvals`	Pointer to an array of length $\ell \cdot m''$ containing the values of the residual parameters $\boldsymbol{\beta}_j$ in the following order: $\beta_{11}, \beta_{12}, \dots, \beta_{1m''}, \beta_{21}, \beta_{22}, \dots, \beta_{\ell m''}$.
`const double *pvals`	Pointer to an array of length m' containing the values of the problem parameters $\boldsymbol{\alpha}$ in the following order: $\alpha_1, \alpha_2, \dots, \alpha_{m'}$.

`GpoSolverSolutions &sols`	A list of vectors containing the solutions, if any, to the original problem (`std::list<std::vector<double > >`).

As a return value, method `solve` returns an `enum` called `GpoSolverStatus`. The following table sums up the possible return values and their meaning:

`SUCCESS_GLOPT`	A solution set was successfully extracted and the global optimality was certified using Theorem 1.
`SUCCESS`	A solution set was successfully extracted, however, Theorem 1 could not be used to certify global optimality.
`UNBOUNDED_SOLUTION`	LMI relaxation $\mathcal{P}_\delta$ could be solved, but the l_2-norm of the solution was too large. This may indicate that additional bound constraints should be enforced. See parameter `maxnorm` in Section 6.1.
`INCOMPLETE_BASIS`	LMI relaxation $\mathcal{P}_\delta$ could be solved, but the solution extraction algorithm could not extract any solutions.
`INFEASIBLE_SDP`	LMI relaxation $\mathcal{P}_\delta$ is infeasible.

Example 10. Let's continue with the PMI class from Example 9: two files `pop_problem.cpp` and `pop_problem.h` were generated by `gpogenerator` into the `GPO_ROOT/matlab` directory; the C++ class name was chosen as '`PopProblem`'. The following shows an example of a PMI instance solving code based on the CSDP solver. A full working version of this solver can be found in `GPO_ROOT/examples/pop_solver.cpp`.

First, let's make the necessary includes.

```
#include <gposolver/gposolver_csdp.h>
#include "pop_problem.h"
```

Next, we will be using the GpoSolver namespace:

```
using namespace GpoSolver;
```

Let's say we want to solve a PMI instance with problem parameters $\boldsymbol{\alpha} = (1,2,3,4)^\top$ and $\ell = 2$ residual parameter blocks $\boldsymbol{\beta}_1 = (1,2,2,3)^\top$ and $\boldsymbol{\beta}_2 = (2,1,1,2)^\top$. To do that, we can define the following two arrays:

```
double pvals[] = {1, 2, 3, 4};
double rvals[][4] = {{1, 2, 2, 3}, {2, 1, 1, 2}};
```

Now, we define the main function, construct the solver object using the `PopProblem` class, and call `solve` method:

```
int main(int argc, const char* argv[]) {
  GpoSolverCsdp<PopProblem> gposolver;
  GpoSolverStatus status;
  GpoSolverSolutions sols;
  status = gposolver.solve(2, (double*) rvals, (double*) pvals, sols);
```

To interpret the solution, we first check the `status` variable:

```
  if (sols.size() == 0)
    return -1;
  else if (status == SUCCESS_GLOPT)
    cout << "Global optimality certified numerically" << endl;
  else if (status == SUCCESS)
    cout << "Alas, global optimality could not be certified" << endl;
```

Now, we just print out the solutions returned in the `sols` list:

```
  int c = 1;
  for (GpoSolverSolutions::iterator it = sols.begin();
         it != sols.end(); it++)
    {
      std::vector<double> &sol = *(it);
        cout << "Solution " << c++ << ": ";
        for (int i = 0; i < sol.size(); i++)
          cout << sol[i] << " ";
        cout << endl;
    }
```

Finally, we close the main function:

```
  return 0;
}
```

After compiling and linking, see Section 7, the resulting executable should give the following output:

```
GpoSolver: 2 solution(s) extracted
Global optimality certified numerically
Solution 1: -0.271539 -0.145
Solution 2: 0.271539 -0.145
```

□

6.1 Parameters

There are two sets of parameters that can be set within the GpoSolver library. The first set pertains to the GpoSolver library itself and there are a few justifiable situations where the user might want to tune these parameters. In the second set, there are the parameters that are exposed by the SDP solvers. You should not meddle with these parameters unless you really know what you are doing.

Parameters from both sets can be set using method `setParameter`:

```
void GpoSolverBase::setParameter(const std::string &pname, double val)
```

One can get the actual values of the parameters using method `getParameter`:

```
double GpoSolverBase::getParameter(const std::string &pname)
```

All parameters must be specified as `pname` string via their name and their value `val`. The following table lists the parameter names exported by GpoSolver.

`verbose`	GpoSolver does not output any textual information to the standard output. However, SDP solvers can provide status and iteration information in this way. To see this information, set `verbose` parameter to 1. This parameter works for all available SDP solvers. The default value is 0.
`sdptol`	SDP feasibility tolerance. Default value is 10^{-3}.
`maxnorm`	Maximum value of the l_2-norm of the solution of $\mathcal{P}_\delta$. If the l_2-norm of the solution is too large, it may indicate that additional bound constraints should be enforced or the problem/residual parameters scaled. Default value is 10^6.
`restol`	Numerical tolerance of the solution feasibility test (see steps 3 and 4 of Algorithm 2). Default value is 10^{-3}.

ranktol	Numerical tolerance of the matrix rank computation. Default value is 10^{-3}.
pivtol	Numerical tolerance of the Gaussian elimination pivoting of the solution extraction algorithm [16]. If set to a negative value, dynamic tolerance selection algorithm is used instead. Default value is -1.

The second set of parameters that can be set via `setParameter` method are the parameters exposed by the SDP solvers. Here, we will only list the parameter names. The parameter names copy the SDP solvers parameter names. The parameter values as well as their meaning can be found in the respective SDP solver manuals.

CSDP [6]	`axtol`, `atytol`, `objtol`, `pinftol`, `dinftol`, `maxiter`, `minstepfrac`, `maxstepfrac`, `minstepp`, `minstepd`, `usexzgap`, `tweakgap`, `affine`, `perturbobj`, `fastmode`.
SDPA [24]	`MaxIteration`, `EpsilonStar`, `LambdaStar`, `OmegaStar`, `LowerBound`, `UpperBound`, `BetaStar`, `BetaBar`, `GammaStar`, `EpsilonDash`.
MOSEK [2]	The list of MOSEK parameters is too long to be included verbatim into this table. Note that only real valued and integer valued MOSEK parameters can be set (*i.e.*, parameters with names starting with '`MSK_DPAR`' and '`MSK_IPAR`').

Note that method `setParameter` does not do any sanity checks on the parameter values. Also note that when using the CSDP solver, the current working directory must be writable for the executable. This is because the CSDP library uses a temporary file to pass the parameter values.

6.2 Error handling

GpoSolver library uses the C++ exceptions mechanism to signal runtime errors. If GpoSolver experiences an unrecoverable error, `std::runtime_error` exception is thrown. To intercept GpoSolver errors, simply insert the library calls into a `try/catch` block:

```
try
  {
    ...
    status = gposolver.solve(num_res, rvals, pvals, sols);
    ...
  }
catch (exception &e)
  {
    cerr << "GpoSolver exception: " << e.what() << endl;
  }
```

6.3 Output

GpoSolver does not output any textual information to the standard output. However, the SDP solvers can provide status and iteration information in this way. To see this information, set `verbose` parameter to 1, for example

```
solver.setParameter("verbose", 1);
```

This works for all available SDP solvers. Note however, that not all of the SDP solvers respect this setting entirely and some textual information may appear on the standard output even if `verbose` parameter is set to 0.

6.4 Solving PMI instances from MATLAB

Even though it may seem to go against the main idea behind GpoSolver, it is possible to solve PMI instances directly from MATLAB using function `gposolve`. This feature becomes useful when testing correctness of a PMI class definition. However, this approach cannot be recommended as a GloptiPoly substitution, since `gposolve` is much slower. This is because `gposolve` manipulates parametrized problems using Symbolic Math Toolbox whereas GloptiPoly can work with concrete problem instances from the beginning. The `gposolve` function requires SeDuMi [31] as its underlying SDP solver. The `gposolve` function again resides in the directory `GPO_ROOT/matlab`.

Before solving a PMI instance, we need to construct an LMI relaxation $\mathcal{P}_\delta$. This is done using function `gporelax`:

```
>> relax = gporelax(problem, order);
```

Given a PMI class definition `problem` (see Section 5.1) and a relaxation order `order` $= \delta$, `gporelax` will return data structure `relax`—the LMI relaxation $\mathcal{P}_\delta$. Relaxation `relax` is then an input parameter of function `gposolve`:

```
>> [status, solutions] = gposolve(relax, rvals, pvals, params);
```

The following table explains the function parameters:

`relax`	LMI relaxation returned by the function `gporelax`.
`rvals`	A $\ell \times m''$ matrix of values of residual parameters $(\boldsymbol{\beta}_1, \boldsymbol{\beta}_2, \ldots, \boldsymbol{\beta}_\ell)^\top$, where ℓ is the number of residual blocks and m'' is the number of residual parameters. The value order in the rows of `rvals` matrix must correspond to the order of variables in vector `rpars` from the problem class definition `problem`.
`pvals`	A vector of values of problem parameters $\boldsymbol{\alpha}$. The value order in `pvals` must correspond to the order of variables in the vector `ppars` from the problem class definition `problem`.
`params`	A data structure with fields and meaning equal to GpoSolver parameters from Section 6.1.

Function `gposolve` returns values `status` and `solutions`. The $s \times m$ matrix `solutions` contains s solution vectors of the original problem defined by `problem`, `solutions` $= (\mathbf{x}_1^*, \mathbf{x}_2^*, \ldots \mathbf{x}_s^*)^\top$. The following table lists the meaning of the values of parameter `status`:

`0`	A solution set was successfully extracted and the global optimality was certified.
`1`	A solution set was successfully extracted, however, global optimality could not be certified.
`2`	LMI relaxation `relax` could be solved, but the l_2-norm of the solution was too large. This may indicate that additional bound constraints should be enforced.
`3`	LMI relaxation `relax` could be solved, but the solution extraction algorithm could not extract any solutions.

4	LMI relaxation `relax` is infeasible.

Example 11. Let us once again use the PMI class from Example 9 and let us assume that the variable `problem` already contains the PMI class definition. As in Example 10, we would like to solve the problem instance with problem parameters $\boldsymbol{\alpha} = (1,2,3,4)^\top$ and $\ell = 2$ residual parameter blocks $\boldsymbol{\beta}_1 = (1,2,2,3)^\top$ and $\boldsymbol{\beta}_2 = (2,1,1,2)^\top$ using the second order relaxation:

```
>> relax = gporelax(problem, 2);
Optimization variables ... [x1, x2]
Computing problem polynomial degree ... 4
Computing minimum relaxation degree ... 2
Computing moment matrix ... done
Computing localizing matrices ... done
Identifying monomials ... 15
Decomposing LMI constraints ... done
Decomposing objective function ... done
>> rvals = [1, 2, 2, 3; 2, 1, 1, 2];
>> pvals = [1, 2, 3, 4];
>> [status, solutions] = gposolve(relax, rvals, pvals);
>> disp(solutions)
   -0.2715   -0.1450
    0.2715   -0.1450
>>
```

Notice that we did not use `gposolve` parameter `params`. As expected, we obtained two solutions $\mathbf{x}_1^* = (-0.2715, -0.145)^\top$ and $\mathbf{x}_2^* = (0.2715, -0.145)^\top$, as in Example 10. □

7 Building with GpoSolver C++ library

The GpoSolver C++ library has two dependencies, the linear algebra library Eigen [1] and one (or more) of the following SDP solvers: CSDP [6], SDPA [24], or MOSEK [2]. All of these solvers come as libraries that must be linked to the final executable.

Eigen. Eigen is a widely used C++ template library for linear algebra and GpoSolver requires at least version 3.2 of this library. The website of the Eigen project is

```
http://eigen.tuxfamily.org
```

Eigen is a part of most modern Linux distributions. On Ubuntu, Eigen can be easily installed by running the following BASH command:

```
$ sudo apt-get install libeigen3-dev
```

For the convenience of Windows users of GpoSolver, a copy of the library can be found inside the GpoSolver distribution in the directory `GPO_ROOT\win\include\Eigen`.

CSDP. The source code of the CSDP solver can be downloaded from the project's webpage:

```
http://projects.coin-or.org/Csdp
```

CSDP is not a part of Ubuntu repositories and must be compiled manually. The library itself has two more dependencies: LAPACK and BLAS. Luckily, these two libraries can installed via the `apt-get` utility:

```
$ sudo apt-get install libatlas-base-dev
```

In the next, we will assume that the environment variable `$CSDP_ROOT` points to the root directory of the CSDP distribution. After following the CSDP compilation instructions, a static library `libsdp.a` should appear in the `$CSDP_ROOT/lib` directory. For the convenience of Windows users of GpoSolver, a dynamic library `libsdp.dll` as well as its LAPACK and BLAS DLL dependencies can be found inside the GpoSolver distribution in the directory `GPO_ROOT\win\bin`.

SDPA. SDPA solver can be obtained from the project's webpage

```
http://sdpa.sourceforge.net
```

or via Ubuntu repository as

```
$ sudo apt-get install sdpa
```

GpoSolver requires version 7.3 of the SDPA library. Unfortunately, the SDPA solver does not provide dynamic or static libraries for Windows, thus using SDPA with GpoSolver is only possible under Linux. The SDPA library also depends on LAPACK and BLAS. Futher, it depends on the MUMPS solver. MUMPS can be easily installed on Ubuntu as:

```
$ sudo apt-get install libmumps-seq-dev
```

In the next, we will assume that the environment variable `$SDPA_ROOT` points to the root directory of the SDPA distribution.

MOSEK. MOSEK is a commercial solver that can be downloaded from the product's webpage

```
http://mosek.com/resources/downloads
```

MOSEK does not require any third party dependencies. In the next, we will assume that version 7 of the library is used and that the environment variable `$MOSEK_ROOT` points to the root directory of the MOSEK distribution.

7.1 Compiling

The recommended compilers for the GpoSolver library are g++ version 4.2 and newer and Microsoft Visual studio 2008 and newer. These are also the recommended compilers for Eigen Library, since these versions support instruction vectorization. However, because both GpoSolver and Eigen follow the c++98 standard, any compiler capable of this standard can be used. In order to get the best performance from Eigen in the release mode, `NDEBUG` macro should be defined. Also, speed optimizations `-O3` (g++) or `/O2` (MSVS) are recommended.

Example 12. Let us demonstrate how to compile a GpoSolver code based on different SDP solvers. We will do that using `pop_solver` example from directory `GPO_ROOT/examples`. The solver consist of the problem files generated by `gpogenerator`: `pop_problem.(cpp,h)` (Example 9) and the solver source code `solver.cpp` (Analogous to the source code from Example 10). The main difference between the code in `pop_solver.cpp` and the code in Example 10 is a macro enabling the selection of different SDP solvers based on different macro name definitions. If `CSDP_SOLVER` macro is defined, `GpoSolverCsdp` solver class will be used for the problem solving. Analogously, macros `SDPA_SOLVER` and `MOSEK_SOLVER` can be defined; see `pop_solver.cpp` for more details. We will use this mechanism to defer the solver selection to the time of compilation.

The following command can be used to compile `pop_solver` based on CSDP solver, assuming `GPO_ROOT/examples` is the current working directory:

```
$ g++ -DCSDP_SOLVER -DNDEBUG -O3 ‘pkg-config --cflags eigen3‘ \
      -I"$GPO_ROOT/include/" -I"$CSDP_ROOT/include/" \
      -c pop_solver.cpp pop_problem.cpp
```

To compile using the SDPA solver:

```
$ g++ -DSDPA_SOLVER -DNDEBUG -O3 ‘pkg-config --cflags eigen3‘ \
      -I"$GPO_ROOT/include/" -I"$SDPA_ROOT/" \
      -c pop_solver.cpp pop_problem.cpp
```

And finally, using the MOSEK solver:

```
$ g++ -DMOSEK_SOLVER -DNDEBUG -O3 -I"$GPO_ROOT/include/" \
      -I"$MOSEK_ROOT/7/tools/platform/linux64x86/h/" \
      ‘pkg-config --cflags eigen3‘ \
      -c pop_solver.cpp pop_problem.cpp
```

□

7.2 Linking

Since both GpoSolver and Eigen are template based libraries, the only thing left to link against are the SDP solvers and their respective dependencies. The following example demonstrates how to link an example code against CSDP, SDPA, and MOSEK solvers.

Example 13. Finally, let us link the executable of the pop_problem from Example 12. To link the solver against the CSDP library, the following command can be used:

```
g++ -fopenmp -o pop_solver_csdp pop_problem.o pop_solver.o -L"$CSDP_ROOT/lib/"
    -lsdp -llapack -lblas
```

Linking against the SDPA solver:

```
g++ -o pop_solver_sdpa pop_problem.o pop_solver.o -L"$SDPA_ROOT/" -lsdpa -
    ldmumps_seq -llapack -lblas -lpthread
```

And finally against the MOSEK solver:

```
g++ -o pop_solver_mosek pop_problem.o pop_solver.o \
    -Wl,-rpath,"$MOSEK_ROOT/7/tools/platform/linux64x86/bin/" \
    -L"$MOSEK_ROOT/7/tools/platform/linux64x86/bin/" -lmosek64
```

□

7.3 Windows

GpoSolver C++ library was tested on Windows 7 64-bit in MS Visual Studio 2008, 2010, and 2013. For the convenience of Windows users, directory GPO_ROOT\win contains all that is needed to build a working solver based on CSDP solver. The directory GPO_ROOT\win\ contain straightforward and self explanatory examples of building environments in the form MS Visual Studio projects. Note that when using these projects, the newly built executables will appear in the directory GPO_ROOT\win\bin, where all the necessary DLL files are also placed. Unfortunately, the SDPA solver does provide neither dynamic nor static libraries for Windows, thus using GpoSolver on Windows with SDPA is not possible. The MOSEK solver is very easy to use on Windows, however, it cannot be included in the GpoSolver distribution for license reasons.

8 Examples

GpoSolver distribution contains several examples of PMI class definitions as well as problem instance solvers in the directory GPO_ROOT/examples. The first example is the pop_problem from Example 4, used to demonstrate various features of GpoSolver throughout this document. The other examples are hec_problem, herwc_problem and pnp_problem. These examples are a bit more involved since they solve real engineering problems and are discussed in Sections 8.1, 8.2, and 8.3, respectively.

Besides the problems and the PMI class definitions, we also discuss time and memory requirements of the example problem solvers. Since most of the SDP solvers used by GpoSolver take advantage of multi-core CPUs, we cite both wall-clock and CPU time measurements. The presented memory measurements were taken using `massif` heap profiler (a part of `valgrind` analysis tools [4]) and account for all memory allocated by an executable and its dynamic libraries using functions `malloc`, `new`, and alike. Both time and memory measurements were taken on a i7-4770K 3.50GHz CPU based desktop running Ubuntu 64-bit.

A `cmake` script in the directory `GPO_ROOT/examples` can be used to build the examples on Linux. Depending on the available SDP solvers, the script will build up to three versions of each solver: `*_solver_csdp`, `*_solver_sdpa`, and `*_solver_mosek`. Version 2.6 or newer of `cmake` is required. To build the examples, execute the following BASH commands:

```
$ mkdir gpo_examples && cd gpo_examples
$ cmake -DCSDP_INCLUDE_DIR="$CSDP_ROOT/include/" \
   -DCSDP_LIBRARY_DIR="$CSDP_ROOT/lib/" \
   -DSDPA_INCLUDE_DIR="$SDPA_ROOT" -DSDPA_LIBRARY_DIR="$SDPA_ROOT/" \
   -DMOSEK_INCLUDE_DIR="$MOSEK_ROOT/7/tools/platform/linux64x86/h/" \
   -DMOSEK_LIBRARY_DIR="$MOSEK_ROOT/7/tools/platform/linux64x86/bin/" \
   "$GPO_ROOT/examples/"
$ make
```

To build the example executables on Windows, MS Visual Studio projects from directory `GPO_ROOT\win` can be used. There are three subdirectories `vs2008`, `vs2010`, and `vs2013`, containing solutions for the respective Visual Studio versions. When using these projects, the newly built executables will appear in the directory `GPO_ROOT\win\bin`, where all the necessary DLL files are also placed. The example solvers build using the provided solutions will not contain the SDP solver specific suffix, since only CSDP solver is available.

Note that all examples can be compiled without generating the problem source codes via `gpogenerator`—the source codes ale already included in the directory.

8.1 Hand-Eye calibration

The problem of hand-eye calibration [30, 32] appeared for the first time in the connection with cameras mounted on robotic systems, however, it arises also in seemingly unrelated fields ranging from medical applications such as ultrasound and endoscopy to automotive industry. Algebraically, it can be formulated as a set of matrix equations

$$\mathtt{A}_i\mathtt{X} = \mathtt{X}\mathtt{B}_i,\ i = 1, \ldots, n \tag{2}$$

where $\mathtt{A}_i, \mathtt{B}_i \in SE(3)$ are matrices that come from the robot and camera measurements and $\mathtt{X} \in SE(3)$ is the unknown rigid transformation. By $SE(3)$ we understand the set of rigid transformations

$$SE(3) = \left\{ \begin{pmatrix} \mathtt{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix} \in \mathbb{R}^{4\times4} \middle| \mathtt{R} \in \mathbb{R}^{3\times3} \text{ is rotation}, \mathbf{t} \in \mathbb{R}^3 \right\}.$$

Let us denote the rotational and translational parts of $\mathtt{X}$ as $\mathtt{R}_\mathtt{X} \in \mathbb{R}^{3\times3}$, $\mathbf{t}_\mathtt{X} \in \mathbb{R}^3$, *i.e.*,

$$\mathtt{X} = \begin{pmatrix} \mathtt{R}_\mathtt{X} & \mathbf{t}_\mathtt{X} \\ \mathbf{0}^\top & 1 \end{pmatrix} \in SE(3).$$

If not for the constraint that $\mathtt{R}_\mathtt{X}$ must be a rotation matrix, System 2 would be simply an overdetermined linear system. However, by using the quaternion parametrization of rotations we can express this constraint as a polynomial constraint and the hand-eye calibration problem can then be formulated as POP [11]. First, let us introduce the quaternion parametrization of the rotation matrix $\mathtt{R}$. Let $\mathbf{q} = (q_1, q_2, q_3, q_4) \in \mathbb{R}^4$ be a unit quaternion, $\mathbf{q}^\top \mathbf{q} = 1$. The rotation matrix $\mathtt{R}(\mathbf{q})$ expressing the same rotation as $\mathbf{q}$ is parametrized as follows

$$\mathtt{R}(\mathbf{q}) = \begin{pmatrix} q_1^2+q_2^2-q_3^2-2q_4^2 & 2q_2q_3-2q_4q_1 & 2q_2q_4+2q_3q_1 \\ 2q_2q_3+2q_4q_1 & q_1^2-q_2^2+q_3^2-q_4^2 & 2q_3q_4-2q_2q_1 \\ 2q_2q_4-2q_3q_1 & 2q_3q_4+2q_2q_1 & q_1^2-q_2^2-q_3^2+2q_4^2 \end{pmatrix}. \tag{3}$$

This parametrization can be easily plugged into a parametrization of the whole transformation $\mathtt{X}$

$$\mathtt{X}(\mathbf{q}, \mathbf{t}) = \begin{pmatrix} \mathtt{R}_\mathtt{X}(\mathbf{q}) & \mathbf{t}_\mathtt{X} \\ \mathbf{0}^\top & 1 \end{pmatrix}. \tag{4}$$

To express the hand-eye calibration problem as POP, we will simply minimize the Frobenius norm of the residuals of System 2 under the constraint $\mathbf{q}^\top \mathbf{q} = 1$:

Problem 5.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^{n} \|\mathtt{A}_i\mathtt{X}(\mathbf{q}, \mathbf{t}_\mathtt{X}) - \mathtt{X}(\mathbf{q}, \mathbf{t}_\mathtt{X})\mathtt{B}_i\|^2 \\ \text{subject to} & \mathbf{q}^\top \mathbf{q} = 1, \\ \text{where} & \mathbf{q} \in \mathbb{R}^4, \mathbf{t}_\mathtt{X} \in \mathbb{R}^3. \end{array}$$

Problem 5 is a POP class in 7 variables $\mathbf{q} \in \mathbb{R}^4, \mathbf{t} \in \mathbb{R}^3$ and 24 residual parameters $\mathtt{A}_i$ and $\mathtt{B}_i$. The cost function is a polynomial of degree 4 and it is composed of 85 monomials. The POP class definition of Problem 5 can be found in the directory `GPO_ROOT/examples/` in the file `hec_problem1.m`:

```
function prob = hec_problem1

syms qx1 qx2 qx3 qx4 tx1 tx2 tx3 real;
syms ra1 ra2 ra3 ra4 ra5 ra6 ra7 ra8 ra9 ta1 ta2 ta3 real;
syms rb1 rb2 rb3 rb4 rb5 rb6 rb7 rb8 rb9 tb1 tb2 tb3 real;
syms n real

X = getSE(qx1, qx2, qx3, qx4, tx1, tx2, tx3);

A = sym(eye(4,4));
A(1:3, 1:3) = [ra1, ra2, ra3; ra4, ra5, ra6; ra7, ra8, ra9];
A(1:3, 4) = [ta1; ta2; ta3];

B = sym(eye(4,4));
B(1:3,1:3) = [rb1, rb2, rb3; rb4, rb5, rb6; rb7, rb8, rb9];
B(1:3, 4) = [tb1; tb2; tb3];

E = A*X - X*B;

prob.obj = dot(E(:), E(:));
prob.vars = [qx1 qx2 qx3 qx4 tx1 tx2 tx3];
prob.cons = [qx1^2 + qx2^2 + qx3^2 + qx4^2 == 1];

prob.rpars = [ra1 ra4 ra7 ra2 ra5 ra8 ra3 ra6 ra9 ta1 ta2 ta3 ...
              rb1 rb4 rb7 rb2 rb5 rb8 rb3 rb6 rb9 tb1 tb2 tb3 ];
prob.ppars = [];
end

function X = getSE(a, b, c, d, t1, t2, t3)
  X = [a^2 + b^2 - c^2 - d^2, 2*b*c - 2*a*d, 2*b*d + 2*a*c, t1;
       2*b*c + 2*a*d, a^2 - b^2 + c^2 - d^2, 2*c*d - 2*a*b, t2;
```

```
        2*b*d - 2*a*c, 2*c*d + 2*a*b, a^2 - b^2 - c^2 + d^2, t3;
        0, 0, 0, 1];
end
```

The files `hec_problem1.cpp` and `hec_problem1.h` were generated from `hec_problem1.m` by `gpogenerator` as the second order relaxation. The hand-eye calibration solver is implemented by `hec_solver.cpp`. The final executable is called `hec_solver1_(csdp|sdpa|mosek)`. In the rest of the paper, we will drop the SDP solver specific suffix. The executable takes care of the parameter scaling and contains the possibility to specify a parameter file as a command line parameter. The file `GPO_ROOT/examples/data/hec_problem_data1.txt` is an example of the problem instance parameter file. Let us try to solve this instance using `hec_solver1`:

```
$ ./hec_solver1 $GPO_ROOT/examples/data/hec_problem_data1.txt
...
GpoSolver status: INCOMPLETE_BASIS
GpoSolver: 0 solution(s) extracted
```

Even though Problem 5 is a theoretically sound POP formulation of the hand-eye calibration problem, GpoSolver was not able to solve this PMI class instance. In this case, it was caused by the fact that the unknown translation $\mathbf{t}_{\mathtt{X}}$ was unbounded and GpoSolver was unable to extract any solution due to numerical inaccuracies. However, since from the physical setup of the system we can estimate an approximate bound on the length of $\mathbf{t}_{\mathtt{X}}$, we can add this bound as a new polynomial constraint. Problem 6 shows the updated variant of the hand-eye calibration problem:

Problem 6.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^{n} \|\mathtt{A}_i\mathtt{X}(\mathbf{q},\mathbf{t}_{\mathtt{X}}) - \mathtt{X}(\mathbf{q},\mathbf{t}_{\mathtt{X}})\mathtt{B}_i\|^2 \\ \text{subject to} & \mathbf{q}^\top\mathbf{q} = 1, \\ & \mathbf{t}_{\mathtt{X}}^\top\mathbf{t}_{\mathtt{X}} \leq n, \\ \text{where} & \mathbf{q} \in \mathbb{R}^4, \mathbf{t} \in \mathbb{R}^3. \end{array}$$

Again, problem 6 is a POP class in 7 variables $\mathbf{q} \in \mathbb{R}^4, \mathbf{t} \in \mathbb{R}^3$ and 24 residual parameters $\mathtt{A}_i$ and $\mathtt{B}_i$. This time, we also have one problem parameter $n \in \mathbb{R}$. The POP class definition of Problem 6 can be found in `hec_problem2.m`. It is equivalent to the definition in `hec_problem1.m`, with the exception of parameters `cons` and `ppars`:

```
prob.cons = [qx1^2 + qx2^2 + qx3^2 + qx4^2 == 1, tx1^2 + tx2^2 + tx3^2 <= n];
prob.ppars = n;
```

The solver executable is called `hec_solver2`. Let us now use the updated solver `hec_solver2` to try to solve the problem instance in `hec_problem_data1.txt`:

```
$ ./hec_solver2 $GPO_ROOT/examples/data/hec_problem_data1.txt
...
GpoSolver status: SUCCESS_GLOPT
GpoSolver: 2 solution(s) extracted
Global optimality certified numerically

q(1) =   -0.999549  -0.0256171  -0.0146832 -0.00554959
t(1) =  -0.122605   0.157114 0.00850124
X(1) =
  0.999507 -0.0103419  0.0296374  -0.122605
 0.0118464   0.998626 -0.0510482   0.157114
-0.0290687  0.0513741   0.998256 0.00850124
         0          0          0          1

q(2) =   0.999549  0.0256171  0.0146832 0.00554959
t(2) =  -0.122605   0.157114 0.00850124
X(2) =
  0.999507 -0.0103419  0.0296374  -0.122605
 0.0118464   0.998626 -0.0510482   0.157114
-0.0290687  0.0513741   0.998256 0.00850124
         0          0          0          1
```

This time, GpoSolver was able to recover two solutions and to certify their global optimality. Now, let us notice that even though the GpoSolver returned two *algebraic* solutions, this problem instance has only one *geometrical* solution: $\mathtt{X}_1$ is equal to $\mathtt{X}_2$. This is caused by the fact that we used quaternion parametrization and because quaternions are a double cover of the group of rotations, it holds that if $\mathbf{q}^*$ is the solution to Problem 6, so is $-\mathbf{q}^*$. Since in hand-eye calibration we are generally interested in geometrical solutions, we can eliminate one of the mirror solutions in most of the cases by adding a constraint $q_1 \geq 0$:

Problem 7.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^{n} \|\mathtt{A}_i\mathtt{X}(\mathbf{q},\mathbf{t}_\mathtt{X}) - \mathtt{X}(\mathbf{q},\mathbf{t}_\mathtt{X})\mathtt{B}_i\|^2 \\ \text{subject to} & \mathbf{q}^\top\mathbf{q} = 1, \\ & q_1 \geq 0, \\ & \mathbf{t}_\mathtt{X}^\top\mathbf{t}_\mathtt{X} \leq n, \\ \text{where} & \mathbf{q} \in \mathbb{R}^4, \mathbf{t} \in \mathbb{R}^3. \end{array}$$

Problem 7 POP class definition can be found in `hec_problem3.m`.This time only `cons` parameter needs to be updated:

```
prob.cons = [qx1^2 + qx2^2 + qx3^2 + qx4^2 == 1, tx1^2 + tx2^2 + tx3^2 <= n...
             qx1 >= 0];
```

The final solver executable is called `hec_solver3`. Let us once again try solve the problem instance in `hec_problem_data1.txt`:

```
$ ./hec_solver3 $GPO_ROOT/examples/data/hec_problem_data1.txt
...
GpoSolver status: SUCCESS_GLOPT
GpoSolver: 1 solution(s) extracted
Global optimality certified numerically

q(1) =   0.999549  0.0256171  0.0146832 0.00554959
t(1) =  -0.122605   0.157114 0.00850124
X(1) =
  0.999507 -0.0103419  0.0296374  -0.122605
 0.0118464   0.998626 -0.0510482   0.157114
-0.0290687  0.0513741   0.998256 0.00850124
         0          0          0          1
```

As a rule of thumb, adding supporting constraints such as in Problems 6 and 7 may help GpoSolver to recover solutions and, in some cases, even speed up the convergence of the underlying SDP solver.

Now, let us present some performance statistics. The first interesting figure is the time taken by `gpogenerator` to produce the C++ code. In this case, it took 24.6 s, 27.8 s, and 28.1 s to generate the C++ classes for Problems 5, 6, and 7, respectively. The following table sums up the runtime statistics for the final Problem 7:

	CSDP	SDPA	MOSEK	SeDuMi
Wall-clock time (s)	0.16	0.08	0.08	0.39
CPU time (s)	1.17	0.08	0.08	0.53
Memory (MB)	1.41	1.49	5.33	—

The first three columns show statistics for `hec_solver3` executable on the problem instance from `hec_problem_data1.txt` when linked to CSDP, SDPA, and MOSEK, respectively. The last column shows the timings then this problem instance was solved in MATLAB using function `gposolve`. Note that the figures in the first three columns show timings for the complete execution of `hec_solver3`, whereas the final column shows timings for the `sedumi` function call only. The memory statistics show the peak value for the execution of the solver. Also note that the memory statistics include memory allocated by the respective SDP solver library as well as by GpoSolver and `hec_solver3` themselves.

8.2 Hand-Eye and Robot-World calibration

In this section, we will present a computationally more demanding example of a POP class. We will extend the problem of hand-eye calibration to include also calibration of the transformation from the robot's base to the coordinate system connected with the world. This problem is sometimes referred to as the hand-eye and robot-world calibration problem [11] and it can be formulated algebraically as

$$\mathtt{A}'_i\mathtt{X} = \mathtt{Z}\mathtt{B}'_i,\ i = 1, \dots, n. \tag{5}$$

Here, $\mathtt{X}, \mathtt{Z} \in SE(3)$ are the unknown hand-eye and robot-world rigid transformations. As we can see, the problem formulation is analogous to Equation 2. This time however, the measurement matrices $\mathtt{A}'_i, \mathtt{B}'_i \in SE(3)$ encode absolute instead of relative robot and camera poses. Let us once again use the quaternion parametrization of rotations from Equation 4: $\mathtt{X}(\mathbf{q}_\mathtt{X}, \mathbf{t}_\mathtt{X})$, $\mathtt{Z}(\mathbf{q}_\mathtt{Z}, \mathbf{t}_\mathtt{Z})$, where $\mathbf{q}_\mathtt{X}, \mathbf{q}_\mathtt{Z} \in \mathbb{R}^4$ and $\mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z} \in \mathbb{R}^3$ are the respective quaternions and translations. Through the rest of this example, we will again construct three iterations of the problem by adding supporting constraints to observe the effect of these constraints on the practical solution of a problem instance. Let us start with the following POP instance:

Problem 8.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^n \|\mathtt{A}'_i\mathtt{X}(\mathbf{q}_\mathtt{X}, \mathbf{t}_\mathtt{X}) - \mathtt{Z}(\mathbf{q}_\mathtt{Z}, \mathbf{t}_\mathtt{Z})\mathtt{B}'_i\|^2 \\ \text{subject to} & \mathbf{q}_\mathtt{X}^\top \mathbf{q}_\mathtt{X} = 1, \mathbf{q}_\mathtt{Z}^\top \mathbf{q}_\mathtt{Z} = 1, \\ \text{where} & \mathbf{q}_\mathtt{X}, \mathbf{q}_\mathtt{Z} \in \mathbb{R}^4, \mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z} \in \mathbb{R}^3. \end{array}$$

Problem 8 is a POP class in 14 variables $\mathbf{q}_\mathtt{X}, \mathbf{q}_\mathtt{Z} \in \mathbb{R}^4$, $\mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z} \in \mathbb{R}^3$ and 24 residual parameters $\mathtt{A}'_i, \mathtt{B}'_i$. The cost function is a polynomial of degree 4 and it is composed of 209 monomials. The POP instance definition of Problem 8 can be found in `GPO_ROOT/examples/herwc_problem1.m`. The solver executable is `herwc_solver1`. Let us try to solve a problem instance in the file `GPO_ROOT/examples/data/herwc_problem_data1.txt`:

```
$ ./herwc_solver1 $GPO_ROOT/examples/data/herwc_problem_data1.txt
...
GpoSolver status: INCOMPLETE_BASIS
GpoSolver: 0 solution(s) extracted
```

As in the case of Problem 5, GpoSolver was not able to recover any solution. Let us try to bound the lengths of the unknown translations $\mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z}$ by adding two problem parameters $n, m \in \mathbb{R}$ and two constraints $\mathbf{t}_\mathtt{X}^\top \mathbf{t}_\mathtt{X} \le n, \mathbf{t}_\mathtt{Z}^\top \mathbf{t}_\mathtt{Z} \le m$.

Problem 9.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^n \|\mathtt{A}'_i\mathtt{X}(\mathbf{q}_\mathtt{X}, \mathbf{t}_\mathtt{X}) - \mathtt{Z}(\mathbf{q}_\mathtt{Z}, \mathbf{t}_\mathtt{Z})\mathtt{B}'_i\|^2 \\ \text{subject to} & \mathbf{q}_\mathtt{X}^\top \mathbf{q}_\mathtt{X} = 1, \mathbf{q}_\mathtt{Z}^\top \mathbf{q}_\mathtt{Z} = 1, \\ & \mathbf{t}_\mathtt{X}^\top \mathbf{t}_\mathtt{X} \le n, \mathbf{t}_\mathtt{Z}^\top \mathbf{t}_\mathtt{Z} \le m, \\ \text{where} & \mathbf{q}_\mathtt{X}, \mathbf{q}_\mathtt{Z} \in \mathbb{R}^4, \mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z} \in \mathbb{R}^3. \end{array}$$

The POP class definition of Problem 9 can be found in herwc_problem2.m. The solver executable for this iteration is herwc_solver2:

```
$ ./herwc_solver2 $GPO_ROOT/examples/data/herwc_problem_data1.txt
...
GpoSolver status: INCOMPLETE_BASIS
GpoSolver: 0 solution(s) extracted
```

Unfortunately, whereas bounding the translation in Problem 6 helped GpoSolver to recover two global minima, in this case it was still unable to found any. Finally, let us add constraints $q_{\mathtt{X}1} \geq 0, q_{\mathtt{Z}1} \geq 0$ to help GpoSolver to disregard one of the global minima.

Problem 10.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^{n} \|\mathtt{A}_i'\mathtt{X}(\mathbf{q}_\mathtt{X}, \mathbf{t}_\mathtt{X}) - \mathtt{Z}(\mathbf{q}_\mathtt{Z}, \mathbf{t}_\mathtt{Z})\mathtt{B}_i'\|^2 \\ \text{subject to} & \mathbf{q}_\mathtt{X}^\top \mathbf{q}_\mathtt{X} = 1, \mathbf{q}_\mathtt{Z}^\top \mathbf{q}_\mathtt{Z} = 1, \\ & \mathbf{t}_\mathtt{X}^\top \mathbf{t}_\mathtt{X} \leq n, \mathbf{t}_\mathtt{Z}^\top \mathbf{t}_\mathtt{Z} \leq m, \\ & q_{\mathtt{X}1} \geq 0, q_{\mathtt{Z}1} \geq 0, \\ \text{where} & \mathbf{q}_\mathtt{X}, \mathbf{q}_\mathtt{Z} \in \mathbb{R}^4, \mathbf{t}_\mathtt{X}, \mathbf{t}_\mathtt{Z} \in \mathbb{R}^3. \end{array}$$

The POP class definition of Problem 10 can be found in herwc_problem3.m. The solver executable for this iteration is herwc_solver3:

```
$ ./herwc_solver3 $GPO_ROOT/examples/data/herwc_problem_data1.txt
...
GpoSolver status: SUCCESS_GLOPT
GpoSolver: 1 solution(s) extracted
Global optimality certified numerically

qx(1) =  0.999549   0.02561 0.0146864 0.0055494
tx(1) =  -0.122703   0.157134 0.00853214
X(1) =
  0.999507 -0.0103416  0.0296438  -0.122703
  0.011846   0.998627  -0.051034   0.157134
-0.0290754    0.05136   0.998257 0.00853214
         0          0          0          1

qz(1) =  0.750224    0.2759 -0.582159 -0.148776
tz(1) =  -0.258406  -0.246466 -0.0362008
Z(1) =
  0.277914 -0.0980045  -0.955593  -0.258406
 -0.544465    0.80349  -0.240751  -0.246466
  0.791405   0.587195   0.169941 -0.0362008
         0          0          0          1
```

Only after adding the final nonnegativity constraints was GpoSolver able to obtain and certify the global minimum.

It took 133 s, 156 s, and 163 s for gpogenerator to generate the C++ classes for problems herwc_problem1, herwc_problem2, and herwc_problem3, respectively. The following table sums up the runtime statistics for the problem instance herwc_problem_data1.txt. The semantics is analogous to the statistics from the hand-eye calibration example in Section 8.1.

	CSDP	SDPA	MOSEK	SeDuMi
Wall-clock time (s)	29.10	21.10	8.23	147.87
CPU time (s)	49.93	21.10	13.51	242.18
Memory (MB)	76.54	88.25	121.11	—

As we can see, this problem is computationally significantly more demanding than the hand-eye calibration problem. Also, the differences between the SDP solvers are more pronounced. In this case, the MOSEK solver delivers the fastest solution. On the other hand, it is also the solver with the highest memory demands.

8.3 Absolute camera pose problem

The third example comes from the field of computer vision where it is known as the absolute camera pose problem, sometimes also as the perspective *n*-point (P*n*P) problem [10]. Here, we solve a variation of this problem: given a set of 3D points $\mathbf{X}_i \in \mathbb{R}^3$, $j = 1, \dots, n$, in the world coordinate frame and the corresponding rays in the camera coordinate frames $\mathbf{c}_i, \mathbf{v}_i \in \mathbb{R}^3$, $j = 1, \dots, n$, where $\mathbf{c}_i$ denotes the origin of the ray and $\mathbf{v}_i$ its directions, the problem is to recover the transformation

$$\mathtt{A} = \begin{pmatrix} \mathtt{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix} \in SE(3)$$

from the world coordinate frame to the coordinate frame of the camera, *i.e.*, the camera pose in the world coordinate frame. As shown in [29], the problem can be formulated and effectively solved as POP. First, let us define the problem's cost function. We minimize the sum of distances of the points $\mathbf{X}_i$ from the corresponding rays. This cost function is also known as *object space error*:

$$\sum_{i=1}^{n} \|(\mathtt{I} - \mathtt{V}_i)(\mathtt{R}\mathbf{X}_i + \mathbf{t} - \mathbf{c}_i)\|^2, \text{ where } \mathtt{V}_i = \frac{\mathbf{v}_i \mathbf{v}_i^\top}{\mathbf{v}_i^\top \mathbf{v}_i}. \tag{6}$$

It was shown in [29] that this expression can be simplified by differentiating w.r.t. the translation $\mathbf{t}$ and setting the result equal to zero:

$$\mathbf{t}^* = \left(\sum_{i=1}^{n} \mathtt{Q}_i\right)^{-1} \sum_{i=1}^{n} \mathtt{Q}_i(\mathtt{R}\mathbf{X}_i - \mathbf{c}_i) \text{ where } \mathtt{Q}_i = (\mathtt{I} - \mathtt{V}_i).$$

Further, let us define the operator $\mathtt{C}(\mathbf{X})$ as

$$\mathtt{C}(\mathbf{X}) = \begin{pmatrix} \mathbf{X}^\top & 0_{1\times3} & 0_{1\times3} \\ 0_{1\times3} & \mathbf{X}^\top & 0_{1\times3} \\ 0_{1\times3} & 0_{1\times3} & \mathbf{X}^\top \end{pmatrix}.$$

Now, we can factor out the rotation $\mathtt{R} = (\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$ from $\mathbf{t}^*$:

$$\mathbf{t}^* = \mathtt{T}_{3\times10} \begin{pmatrix} \mathbf{r} \\ 1 \end{pmatrix}, \text{ where } \mathtt{T}_{3\times10} = -\left(\sum_{i=1}^{n} \mathtt{Q}_i\right)^{-1} \sum_{i=1}^{n} \mathtt{Q}_i \begin{pmatrix} \mathtt{C}(\mathbf{X}_i) & \mathbf{c}_i \end{pmatrix}, \mathbf{r} = \begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \mathbf{r}_3 \end{pmatrix}. \tag{7}$$

By plugging $\mathbf{t}^*$ into Equation 6, we get the following expression

$$\sum_{i=1}^{n} \mathbf{r} \left(\begin{pmatrix} \mathtt{C}(\mathbf{X}_i) & \mathbf{c}_i \end{pmatrix} + \mathtt{T}_{3\times10} \right)^\top \mathtt{Q}_i \left(\begin{pmatrix} \mathtt{C}(\mathbf{X}_i) & \mathbf{c}_i \end{pmatrix} + \mathtt{T}_{3\times10} \right) \mathbf{r} = \sum_{i=1}^{n} \mathbf{r}\mathtt{M}\mathbf{r}.$$

As in the case of `hec_problem`, the non-convexity of the problem lies in the fact that $\mathtt{R}$ is a rotation. Thus, we can again use the quaternion representation from Equation 3 to parametrize $\mathbf{r}$ as $\mathbf{r}(\mathbf{q})$ and to formulate the problem as POP:

Problem 11.

$$\begin{array}{rl} \text{minimize} & \sum_{i=1}^{n} \mathbf{r}(\mathbf{q})\mathtt{M}\mathbf{r}(\mathbf{q}) \\ \text{subject to} & \mathbf{q}^\top \mathbf{q} = 1, \\ & q_1 \geq 0, \\ \text{where} & \mathbf{q} \in \mathbb{R}^4. \end{array}$$

Problem 11 is a POP class in 4 variables $\mathbf{q} \in \mathbb{R}^4$, 30 problem parameter $\mathtt{T}_{3\times10}$ and 9 residual parameters $\mathbf{X}_i$, $\mathbf{v}_i$, and $\mathbf{c}_i$. Once the optimal rotation $\mathbf{r}(\mathbf{q}^*)$ is recovered, the optimal translation can be simply computed using Equation 7. The cost function is a polynomial of degree 4 and it is composed of 46 monomials.

The class definition of Problem 11 can be found in the directory `GPO_ROOT/examples` as `pnp_problem.m`:

```
function prob = pnp_problem
q = sym('q', [4, 1]); assume(q, 'real');
X = sym('X', [3, 1]); assume(X, 'real');
v = sym('v', [3, 1]); assume(v, 'real');
c = sym('c', [3, 1]); assume(c, 'real');
T = sym('T', [3, 10]); assume(T, 'real');

z13 = zeros(1, 3);
CX = [X', z13, z13; z13, X', z13; z13, z13, X'];
Q = eye(3,3) - v * v';
R = q2rot(q);
r = R(:);

M = ([CX c] + T)' * Q * ([CX c] + T);
Mr = M(1:(end-1), 1:(end-1));
Mc = 2*M(1:(end-1), end);
Mcc = M(end, end);

prob.obj = r' * Mr * r + Mc' * r + Mcc;
prob.vars = q;
prob.cons = [q(1)^2 + q(2)^2 + q(3)^2 + q(4)^2 == 1, q(1) >= 0];

prob.rpars = [v' c' X'];
prob.ppars = T(:)';

end

function R = q2rot(q)
  R = [q(1)^2 + q(2)^2 - q(3)^2 - q(4)^2, 2*q(2)*q(3) - 2*q(1)*q(4),
              2*q(2)*q(4) + 2*q(1)*q(3); 2*q(2)*q(3) + 2*q(1)*q(4),
       q(1)^2 - q(2)^2 + q(3)^2 - q(4)^2, 2*q(3)*q(4) - 2*q(1)*q(2);
              2*q(2)*q(4) - 2*q(1)*q(3), 2*q(3)*q(4) + 2*q(1)*q(2),
      q(1)^2 - q(2)^2 - q(3)^2 + q(4)^2];
end
```

The files `pnp_problem.cpp` and `pnp_problem.h` were generated from `pnp_problem.m` by `gpogenerator` as the second order relaxation. The file `pnp_solver.cpp` implements the P*n*P solver itself. The solver takes care of the computation of the matrix $\mathtt{T}_{3\times10}$ and contains the possibility to specify a parameter file as a command line parameter. There are four parameter files in the `GPO_ROOT/examples/data` directory: `pnp_problem_data(1,2,3,4).txt`. Let us try to solve the problem instance `pnp_problem_data1.txt`:

```
$ ./pnp_solver $GPO_ROOT/examples/data/pnp_problem_data1.txt
...
GpoSolver status: SUCCESS_GLOPT
GpoSolver: 1 solution(s) extracted
Global optimality certified numerically

A(1) =
    -0.85334    -0.521346   0.00300841    -0.331193
   -0.521347     0.853345  0.000673816    0.0514275
  -0.0029185 -0.000993432    -0.999995      1.96502
           0            0            0            1
```

Each of the P*n*P problem instances from the previous paragraph has only one globally optimal solution. However, there are geometrically valid configurations for which the P*n*P problem has more that one solution. For example, if a camera with an absolute pose $\mathtt{R} = (\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)$, $\mathbf{t}$ observes a planar scene, *i.e.*, points that lie in a common plane, a camera pose $\mathtt{R}' = (-\mathbf{r}_1, -\mathbf{r}_2, \mathbf{r}_3)$, $-\mathbf{t}$ explains this configuration equally well. The difference between these configurations is that in the first case the points lie in front of the camera, whereas in the second one the points lie behind it (or vice versa). The second configuration might not be a physically valid one for cameras with a narrow field of view, however, it is definitely plausible for omnidirectional cameras. The file `pnp_problem_data_planar.txt` contains a P*n*P problem instance representing such a configuration:

```
$ ./pnp_solver $GPO_ROOT/examples/data/pnp_problem_data_planar.txt
...
GpoSolver status: SUCCESS_GLOPT
GpoSolver: 2 solution(s) extracted
Global optimality certified numerically

A(1) =
          -1  8.39766e-06 -6.75131e-06      1.00011
-8.39745e-06           -1 -3.05957e-05  8.17055e-05
-6.75157e-06 -3.05956e-05            1 -0.000172918
           0            0            0            1

A(2) =
           1 -8.39339e-06  6.75131e-06     -1.00011
 8.39319e-06            1  3.05957e-05 -8.17012e-05
-6.75157e-06 -3.05956e-05            1  0.000172918
           0            0            0            1
```

As we can see, GpoSolver was able to recover both global minima of this configuration.

It took 50.1 s for `gpogenerator` to generate the files `pnp_problem.(cpp,h)`. The following table sums up the runtime statistics for the instance `pnp_problem_data1.txt`. Again, the semantics is analogous to the statistics from the previous examples.

	CSDP	SDPA	MOSEK	SeDuMi
Wall-clock time (s)	0.05	0.01	0.01	0.24
CPU time (s)	0.33	0.01	0.02	0.34
Memory (MB)	0.16	0.18	2.75	—

Acknowledgements

The authors were supported by the EC under project FP7-SPACE-2012-312377 PRoViDE.

References

[1] Eigen: C++ template library for linear algebra.
`eigen.tuxfamily.org`.

[2] Mosek: large scale optimization software.
`www.mosek.com`.

[3] SparsePOP: Sparse SDP relaxation of polynomial optimization problems.
`http://www.is.titech.ac.jp/~kojima/SparsePOP/`.

[4] Valgrind: instrumentation framework for building dynamic analysis tools.
`www.valgrind.org`.

[5] G. Valmorbida S. Prajna P. Seiler A. Papachristodoulou, J. Anderson and P. A. Parrilo. SOSTOOLS: Sum of squares optimization toolbox for MATLAB, 2013.
`http://www.cds.caltech.edu/sostools`.

[6] Brian Borchers. CSDP, a library for semidefinite programming. *Optimization methods and Software*, 11(1-4):613–623, 1999.

[7] G. Chesi. Lmi techniques for optimization over polynomials in control: A survey. *Automatic Control, IEEE Transactions on*, 55(11):2500–2510, Nov 2010.

[8] D.A. Cox, J.B. Little, and D. O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, 2005.

[9] Raúl Curto and Lawrence Fialkow. The truncated complex k-moment problem. *Transactions of the American mathematical society*, 352(6):2825–2855, 2000.

[10] R.I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2003.

[11] Jan Heller, Didier Henrion, and Tomas Pajdla. Hand-eye and robot-world calibration by global polynomial optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[12] Jan Heller and Tomas Pajdla. World-base calibration by global polynomial optimization. In *The International Conference on 3D Vision (3DV)*, 2014.

[13] Jan Heller, Tomas Pajdla, and Didier Henrion. Stable radial distortion calibration by polynomial matrix inequalities programming. In *Asian Conference on Computer Vision (ACCV)*, 2014.

[14] Didier Henrion and Jean-Bernard Lasserre. GloptiPoly: Global optimization over polynomials with Matlab and SeDuMi. *ACM Transactions on Mathematical Software (TOMS)*, 29(2):165–194, 2003.

[15] Didier Henrion and Jean-Bernard Lasserre. Solving global optimization problems over polynomials with GloptiPoly 2.1. In *Global Optimization and Constraint Satisfaction*, pages 43–58. Springer, 2003.

[16] Didier Henrion and Jean-Bernard Lasserre. Detecting global optimality and extracting solutions in GloptiPoly. In *Positive polynomials in control*, pages 293–310. Springer, 2005.

[17] Didier Henrion and Jean-Bernard Lasserre. Convergent relaxations of polynomial matrix inequalities and static output feedback. *Automatic Control, IEEE Transactions on*, 51(2):192–202, 2006.

[18] Didier Henrion, Jean-Bernard Lasserre, and Johan Löfberg. GloptiPoly 3: moments, optimization and semidefinite programming. *Optimization Methods & Software*, 24(4-5):761–779, 2009.

[19] Fredrik Kahl and Didier Henrion. Globally optimal estimates for geometric reconstruction problems. volume 2, pages 978 –985 Vol. 2, oct. 2005.

[20] Jean-Bernard Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal on Optimization*, 11:796–817, 2001.

[21] Jean-Bernard Lasserre, Tomas Prieto-Rumeau, and Mihail Zervos. Pricing a class of exotic options via moments and SDP relaxations. *Mathematical Finance*, 16(3):469–494, 2006.

[22] Monique Laurent. Sums of squares, moment matrices and optimization over polynomials, to appear in emerging applications of algebraic geometry. In *of IMA Volumes in Mathematics and its Applications*. Springer, 2009.

[23] Johan Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Computer Aided Control Systems Design, 2004 IEEE International Symposium on*, pages 284–289. IEEE, 2004.

[24] Kazuhide Nakata, Maho Nakata, Mituhiro Fukuda, Kazuhiro Kobayashi, Kazushige Goto Makoto Yamashita, Katsuki Fujisawa. A high-performance software package for semidefinite programs: Sdpa 7. Technical report, Dept. of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan, September 2010.

[25] Katta G Murty and Santosh N Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical programming*, 39(2):117–129, 1987.

[26] Jiawang Nie. Optimality conditions and finite convergence of Lasserre's hierarchy. *Mathematical Programming*, pages 1–25, 2012.

[27] Jiawang Nie. Certifying convergence of Lasserre's hierarchy via flat truncation. *Mathematical Programming*, 142(1-2):485–510, 2013.

[28] Pablo A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96:293–320, 2003.

[29] G. Schweighofer and A. Pinz. Globally optimal O(n) solution to the PnP problem for general camera models. In *BMVC 2008*, 2008.

[30] Y.C. Shiu and S. Ahmad. Calibration of wrist-mounted robotic sensors by solving homogeneous transform equations of the form AX=XB. *Robotics and Automation, IEEE Transactions on*, 5(1):16 –29, feb 1989.

[31] J.F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999.

[32] R.Y. Tsai and R.K. Lenz. A new technique for fully autonomous and efficient 3D robotics hand/eye calibration. *Robotics and Automation, IEEE Transactions on*, 5(3):345 –358, June 1989.

[33] Hayato Waki, Sunyoung Kim, Masakazu Kojima, and Masakazu Muramatsu. Sums of squares and semidefinite program relaxations for polynomial optimization problems with structured sparsity. *SIAM Journal on Optimization*, 17(1):218–242, 2006.