UCLouvain

École polytechnique de Louvain

Implementation of a semidefinite optimization solver in the Julia programming language

Auteur: **Tomasz Kwaśniewicz**
Promoteur: **François Glineur**
Lecteurs: **Julien Hendrickx, Benoît Legat**
Année académique 2020–2021
Master [120] : ingénieur civil en mathématiques appliquées

Abstract

Semidefinite programming has become a major research topic in optimization, finding applications in multiple domains. This surge of interest sparked the need for efficient solution methods and user-friendly software in order to deploy semidefinite programming in practice. At the same time, the `Julia` programming language introduced in 2012 became a major platform for developing optimization software. This report focuses on presenting the implementation of an interior-point solver for semidefinite programming written entirely in `Julia`.

The solver implements a Mehrotra-type predictor-corrector interior-point method. In order to assess its performance, several benchmarks are completed. These benchmarks are performed on a widely used problem set (SDPLIB) and several interior-point as well as first-order solvers are tested. The obtained results show that this newly implemented solver can achieve a decent performance and be even slightly faster than some well-established solvers for some problems.

Concepts related to this report are: semidefinite programming, conic optimization, Julia programming language.

Acknowledgement

I would like to especially thank my supervisor, Professor François Glineur for his implication during this master thesis. His expertise in the domain of optimization, his keen eye for detail and his thoughtfulness greatly helped me to finish this work.

I would like to thank Professor Julien Hendrickx and Dr Benoît Legat for accepting to read and assess this thesis.

Last but not least, I would like to thank my family and friends for their great support.

Contents

Chapter 1

Introduction

In this chapter, we will introduce the notion of semidefinite programming. It will be exemplified by diverse applications in several fields of engineering. A brief introduction to the `Julia` programming language will follow, emphasising its fitness for building optimization software. A short survey of available solvers and their characteristic will end this section.

1.1 Motivation

1.1.1 Introduction to semidefinite programming

A *semidefinite program* (SDP) is an optimization problem in the following form

$$\begin{aligned} &\min_{X \in \mathbb{S}^n} \operatorname{tr}(CX) \\ \text{s.t.} \quad &\operatorname{tr}(A_i X) = b_i,\ i = 1, \dots, m \\ &X \succcurlyeq 0 \end{aligned} \tag{1.1}$$

where the optimization variable X is a positive semidefinite matrix belonging to $\mathbb{S}^n$, $\mathbb{S}^n$ denotes the cone of real symmetric matrices of dimensions n-by-n and the $\operatorname{tr}(\cdot)$ operator stands for the trace of a matrix, i.e. the sum of its diagonal entries. $X \succcurlyeq 0$ (resp. $X \succ 0$) means that the matrix X is positive semidefinite (resp. positive-definite). The data of the problem consist of the matrix $C \in \mathbb{S}^n$ defining the objective function, m matrices $A_i \in \mathbb{S}^n$ and a vector $b \in \mathbb{R}^m$ defining linear equality constraints. It is worth noting that $\operatorname{tr}(XY)$ is a scalar product for symmetric matrices X, Y. The class of semidefinite programs includes the class of linear programs (LP). Indeed, in the special case when all the matrices are diagonal, the trace of a matrix product

becomes the dot product of the diagonals and the semidefinite constraint translates to all diagonal entries being nonnegative, hence (1.1) becomes

$$\begin{aligned} &\min c^T x \\ &\text{s.t} \quad Ax = b \\ &\quad x \geq 0 \end{aligned} \tag{1.2}$$

which is the familiar LP formulation. As SDP is in some sense a generalization of LP, many theorems and algorithms can be naturally extended to SDP. [WSV00]

The class of semidefinite programs includes several other important classes of optimization problems, such as convex quadratic problems or second-order programming. Such a general form ensures rich modelling capabilities and makes SDP suitable for many applications in engineering [VB96].

Besides the rich modelling capabilities, one of the main features of interest both theoretically and practically is the fact that efficient solution methods are available. Indeed, Nesterov and Nemirovski proposed a general framework of *interior-point methods* for conic optimization which includes semidefinite programming and proved that such problems can be solved in polynomial time. Since then, several specialized algorithms and solvers implementing them have been designed.

1.1.2 Some applications of SDP

Relaxation of combinatorial problems

Semidefinite programming found extensive use in the case of relaxations of combinatorial optimization problems [GR00]. Indeed, combinatorial problems are often found to be NP-hard, requiring a recourse to continuous relaxations. Notable examples include the max-cut problem (Example 1.1.1), the Lovász's θ-function or the quadratic assignment problem. The fact that many relaxations of combinatorial programs can be modelled as SDP stems from the link between combinatorial programs and quadratically constrained quadratic programs. Indeed, let us consider for instance the case of binary variables. This constraint is written as $x \in \{0, 1\}$. We can replace it by $x(x-1) = 0$. It turns out that dualizing such constraints gives an SDP [LO99].

Example 1.1.1. Let $G = (V, E)$ be a simple graph. We would like to partition the nodes of G into two complementary sets S and T such that the number of edges between S and T is maximized. This problem can be modelled as an LP with binary variables, however, the number of constraints is exponential. To avoid this, we will take the following formulation, where $x_i = 1$ if $i \in S$ and $x_i = -1$ if $i \in T$:

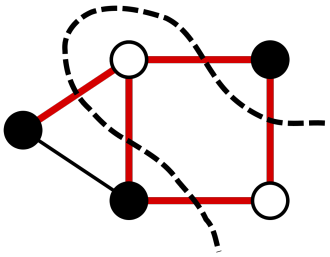

Figure 1.1: An example of a maximal cut

$$\max \sum_{(i,j)\in E} \frac{1 - x_i x_j}{2} \quad \text{s.t} \quad x_v \in \{-1, 1\},\ \forall v \in V$$

Using the degree sum formula on the first term and rearranging the second sum we can rewrite the objective as

$$\sum_{(i,j)\in E} \frac{1 - x_i x_j}{2} = \frac{1}{2}\Big(\sum_{(i,j)\in E} 1 - \sum_{(i,j)\in E} x_i x_j \Big) = \frac{1}{2}\Big(\frac{1}{2} \sum_{v\in V} \deg(v) - \frac{1}{2} \sum_{i\in V} \sum_{\substack{j\in V \\ (i,j)\in E}} x_i x_j \Big)$$

We can rewrite the second sum using the adjacency matrix cancelling the contribution of non-existing edges:

$$\sum_{\substack{j\in V \\ (i,j)\in E}} x_i x_j = \sum_{j\in V} x_i a_{ij} x_j$$

Finally, observing that $x_i x_i = 1$ and defining D as the diagonal degree matrix, we have

$$\max \sum_{(i,j)\in E} \frac{1 - x_i x_j}{2} = \max \frac{1}{4} x^T (D - A) x = \frac{1}{4} \max x^T L x, \text{ where } L = D - A$$

Using the fact that $x^T L x = x^T (Lx) = \operatorname{tr}(Lxx^t)$ we can define a new variable $X = xx^T$ and rewrite the linearized objective as $\operatorname{tr}(LX)$. In order to account for the binary constraints, we notice that they can be expressed as $\operatorname{diag}(X) = e, X \succcurlyeq 0, \operatorname{rank}(X) = 1$. Since the rank constraint is not convex, we drop it and obtain the following SDP relaxation:

$$\max \ \operatorname{tr}(LX) \text{ s.t } \operatorname{diag}(X) = e, X \succcurlyeq 0$$

It has been shown that algorithms based on such SDP relaxations and randomized rounding achieve an approximation ratio of 0.878, meaning that the expected size of the obtained cut is at least 0.878 of the optimal cut. [GW95].

Applications in control

The fact that SDP allows formulations with linear matrix inequalities leads to many applications in control theory, such as stability analysis (Example 1.1.2) or optimal control. Historically, the design of controllers was bound by the available computing power. If a control signal had to be computed nearly instantaneously on a embedded system, closed forms were often necessary, such as the LQ controller. However, with the advent of interior-point methods allowing for efficient solving of SDP problems, a tractable optimization problem now constitutes a viable option and allows for more complex control laws to be considered. These include allowing bounds on the command signal in the LQ controller. Control laws based on SDP were succesfully used in balancing humanoid robots.

Example 1.1.2. Let us consider a discrete-time linear system given by

$$x_{k+1} = Ax_k$$

Its stability is given by the eigenvalue test, or equivalently by finding a positive definite matrix P such that

$$APA^T - P \prec 0$$

If A varies with time, these results do not longer hold. However, we can try to find $P \succ 0$ such that

$$A_k P A_k^T - P \prec 0 \text{ for } k = 1, 2, 3, ...$$

which is a semidefinite feasibility problem.

Machine learning and statistics

Recent developments in statistics and machine learning resulted in formulations involving a solution of a SDP. This stems from the fact that these fields often use *correlation* matrices, which are symmetric and semidefinite positive. Examples include the nearest correlation matrix (Example 1.1.3) or the matrix completion problem [CD17]. The data fusion technique in the kernel methods also relies on SDP [De 07].

Example 1.1.3. Let us consider an approximate correlation matrix constructed from data which can be incomplete or not positive semidefinite. We would like to find the closest 'true' correlation matrix, in the sense of the Frobenius norm. This problem can be formulated as

$$\min \|X - A\|_F,\ diag(X) = e,\ X \in \mathbb{S}^n_+$$

where *diag* extracts the diagonal of a matrix as a vector. We can reformulate it [ApS21] as

$$\begin{aligned} &\min\ t \quad \text{s.t.} \\ &\|z\|_2 \leq t \\ &\text{svec}(A - X) = z \\ &diag(X) = e \\ &X \succcurlyeq 0 \end{aligned}$$

where the svec operator is a vectorization operator for symmetric matrices (Section 2.1.3). For the $\|z\|_2 \leq t$ constraint, refer to [KRT99] for casting second-order conic constraints as SDP.

For more real-world applications, please refer to Section 3. We will be concluding this section with a non-exhaustive list of more generic problems that could be cast as SDP [ApS21].

- ◇ Eigenvalue optimization
- ◇ Log-determinant optimization
- ◇ Singular value optimization

1.2 The Julia programming language and its applications to optimization

1.2.1 Brief presentation of the language

Julia [Bez+12] is a general-purpose programming language meant specifically for numerical and scientific computing. The main goal of its creators was to combine the ease of use and the readability of dynamic languages such as `Python` or `Matlab` with the high performance of compiled languages such as `C/C++` or `Fortran`. Indeed, Julia was proposed as a solution to the so-called *two language problem*. The *two language problem* refers to the observation that the prototyping phase of a software project is often done in a high-level dynamic language, while the actual implementation at the production stage is rewritten in one of the aforementioned

compiled languages. This can make the process more challenging. Indeed, it requires expertise in two distinct languages and the maintenance of two separate projects. This in turn can lead to the necessity of two separate teams, increasing costs and causing delays. One can thus sometimes choose to sacrifice performance for the ease of development. This problem also occurs in a slightly different form in optimisation, where the modelling phase is done in a very transparent modelling language, while the solver is coded in a very fast language for performance. It seems thus appropriate to investigate the benefits of having the modelling layer and the solver written in the same language.

Released in 2012 and thus being a relatively new language, Julia has been deployed in academia and in industry. Examples include pharmacy, macroeconomic modelling, machine learning, climatology and energy market modelling as well as optimization.[Com21]

Julia is a highly community-driven language and being meant specifically for scientific applications, it offers a wide selection of packages from different domains such as machine learning, finite element analysis and mathematical optimization. This creates a rich ecosystem, allowing the user to combine domain-specific functions, like solving optimization problems in subroutines.

Let us mention the Julia features making it appropriate for prototyping and testing mathematical software. Being intended for users accustomed to Matlab or Python, it aims to have a simple syntax inspired by imperative languages. A notable cosmetic feature of Julia is the support of Unicode characters in variable or function names, allowing programs to be visually closer to mathematical formulation in the spirit of modelling languages. It can be run in interactive mode (REPL) or in a Jupyter notebook, which makes it fit for quick prototyping, querying ad hoc solutions or visualizing results. The Julia standard library provides support for vectors and matrices as well as linear algebra routines, making it a fully functional linear algebra environment.

Julia is a multi-paradigm language and supports some useful constructs from other programming languages, such as functional programming features and object oriented programming. Functions in Julia are not pure mathematical functions since they can modify their input; however it is possible to write functional programs. Functions can be declared in a variety of ways, and operator overloading is supported. Moreover, Julia supports the use of macros. Since Julia code is itself a Julia object, we can write a function that takes some Julia code and modifies a certain part of its input (for example adding timing and printing the results). Such functions are called macros and can make the code more compact.

Objects in Julia are not objects in the sense of `C++` but rather bear a resemblance to those found in `Lisp`, meaning that methods are separated from the class definition. This implies that the methods can be specialized for any combination of types of its arguments, making the use of user-defined types easier. Indeed, if a particular method operating on a given type and a new user-defined type were to be added in `C++`, the user would need either to modify the

class, or to use inheritance features, both not ideal. In contrast, Julia uses *multiple dispatch* for this purpose. It means that the user has to write a function for a given combination of types, and the compiler will choose the right variant automatically. It also facilitates the interaction between different packages, since the compiler will figure out the corresponding types. This in turn allows for much more code combination and reuse. The power of *multiple dispatch* in the context of optimization is shown by the fact that implementing the `MOI` wrapper alone (Section 1.2.2) is enough for the solver to work automatically with both `JuMP` and `Convex.jl`.

The performance of Julia results from the use of the `LLVM` compiler. Indeed, this tool allows generating highly optimized machine code from a given Julia program. This contrasts with interpreted languages like Python where the interpreter has a large overhead. Moreover, Julia gives the user the possiblity to call many low-level libraries directly, making computationally-intensive operations as fast as they would be in the native language without overhead. For example, `BLAS` and `LAPACK` routines are readily available. Direct calls to `C` routines are also supported. This feature makes it very interesting from the *two languages problem* perspective. Julia also provides many benchmarking utilities as well as low-level code explorers. Indeed, using the `@code_lowered` family of macros, it makes it possible to see exactly the assembly code produced. Such features are absent from high-level languages such as Python or especially `Matlab`, where it is nearly impossible to know what exactly happens under the hood.

Despite it being branded as a language as easy as Python and as fast as C, writing efficient programs in Julia takes practice and many performance tips may seem non-trivial to the inexperienced user.

1.2.2 JuMP modelling language

`JuMP` [DHL17] is a mathematical optimization modelling language written and embedded in Julia. It currently supports many classes of optimization problems, allowing great flexibility in modelling. Modelling languages are a great tool for optimization practitioners because they allow to express optimization problems in a very concise way and experiment easily with other formulations or independent solvers.

One major advantage of JuMP over other modelling languages is the fact that it is embedded in a high-level language. It offers the users a wide spectrum of features, such as writing their own subroutines to generate or preprocess the data of the model, visualizing the results or embedding the optimization problem as a subroutine in a existing project. Classical modelling languages such as AMPL of GAMS are usually quite rigid, not allowing for much flexibility. Finally, as these languages are pretty old, some newer types of constraints are simply not implemented. For example, AMPL does not support semidefinite constraints.

One distinguishing feature of JuMP is the fact that it can interact with the solver during the

optimization process. This is different from other modelling languages where the generated data are passed to the solver and the solution is retrieved without other interaction. This in turn allows for many interesting constructs, for instance an easier implementation of branch-and-cut schemes in mixed-integer optimization, where new cuts are added successively to the model being solved. This traditionally required a custom implementation in a low-level language in order to be efficient, but can be easily expressed in JuMP.

```
using JuMP, Hypatia
model = Model(Hypatia.Optimizer)
# Some data in the standard form
A = [1.0 0.0
     0.0 0.0]
C = [1.0 2.0
     2.0 1.0]
b = 1.0
@variable(model, X[1:2,1:2], PSD)
@constraint(model, tr(A*X) == b)
@objective(model, Min, tr(C*X))
optimize!(model)
display(objective_value(model))
-3.0
display(value.(X))
  1.0 -2.0
 -2.0  4.0
```

Listing 1.1: A simple SDP `JuMP` model

When it comes to performance differences between modelling languages embedded in other high-level languages, JuMP takes advantage of some Julia constructs, notably macros. Indeed, as mentioned before, Julia code is itself a Julia object and can be passed to a modifying function altering its behaviour. This allows JuMP to handle formulations more efficiently than other languages such as CVX or Yalmip, which use operator overloading. It also allows JuMP to easily apply automatic differentiation routines.

```
using ProxSDP
set_optimizer(model, ProxSDP.Optimizer)
optimize!(model)
display(objective_value(model))
-3.0
display(value.(X))
  1.0 -2.0
 -2.0  4.0
```

Listing 1.2: A simple SDP `JuMP` model (continued)

JuMP is in fact a layer around MathOptInterface (`MOI`) [Leg+20]. `MOI` is a Julia API that tries to unify different solver interfaces. This in turn allows the users to use and test different solvers

without understanding each specific API and to implement their own solvers. MOI is designed to be very flexible and allow the user to experiment and implement new types of constraints. A notable feature of `MOI` is an algorithmic framework, called *bridges*, for automated reformulation of constraints.

Although JuMP is the main modelling language available in Julia, alternatives exist. One of them is `Convex.jl` which specializes in convex and conic optimization [Ude+14]. It differs from JuMP in that it requires the problem to be convex and issues a warning if this is not the case. It may seem more restrictive, but in the area of semidefinite programming and more generally convex optimization, many solver failures result from a seemingly convex problem not being convex. Hence such a check can help the practitioner. Another feature of `Convex.jl` absent from JuMP is the possibility of using complex variables. Indeed, such formulations combined with semidefinite programming often arise in electrical grid simulations.

```
using Convex, Hypatia
A = [1.0 0.0
     0.0 0.0]
C = [1.0 2.0
     2.0 1.0]
b = 1.0
X = Semidefinite(2)
model = minimize(tr(C*X), tr(A*X) == b)
solve!(model, Hypatia.Optimizer)
```

Listing 1.3: A simple SDP `Convex` model

1.2.3 A brief survey of available solvers and their characteristics

In this section, we will list some SDP solvers available through JuMP, with emphasis on solvers written purely in Julia.

Hypatia [CKV21] is a novel interior-point solver for generic conic optimization, including the semidefinite cone. It implements the Skaaja and Ye interior-point algorithm [SY15]. It is written purely in Julia. Its specificity consists in implementing many so-called 'exotic' cones. Indeed, many problems can be easily formulated using non-standard cones but need to be reformulated using standard cones supported by existing solvers, such as second-order cones, power cones or SDP cones. However, such an extended formulation often results in a much bigger problem, which becomes harder to solve. Hypatia provides many exotic cones along with a suitable barrier function. The solver also supports user-defined exotic cones, making it very flexible. Moreover, Hypatia is a highly modular solver, allowing the users to substitute certain parts of the code according to their needs. To our knowledge, it is the only conic interior-point solver entirely written in Julia. See section 3 for more details.

ProxSDP [SGV18] is another SDP solver written purely in Julia. However, the motivation for its creation stems from the fact that interior-point methods scale badly for large problems, making it difficult to deploy SDP for some real-world applications. Indeed, it was shown that problems with primal variables of the size of up to 5000×5000, which are generally too big for interior-point methods, could be solved in reasonable time using ProxSDP. ProxSDP implements a special case of the primal-dual hybrid gradient method, which is a first order method. This method is an operator-splitting-type method, consisting in rewriting an SDP program as a unconstrained convex non-smooth program using set indicator functions and observing that the subgradient optimality conditions can be expressed as finding a zero of a particular operator. This can be achieved with a fixed-point iteration. Contrary to interior-point methods, it does not involve storing and solving a large linear system. Instead, it requires matrix multiplications and eigendecomposition, which is the main computational bottleneck. It was observed that sometimes, only an approximate eigendecomposition can be used. As a result, it can often retrieve a low-rank solution, which is desired, while interior-point methods typically find a full-rank solution. This is because the algorithm first explores smaller truncations of the eigendecomposition and will output the first solution that converged, yielding a low-rank solution. This behaviour is very desirable in real-world applications.

COSMO [GCG20] is another solver implemented in pure Julia. It is based on the conic operator splitting method, also a first order method. It is in fact based on the ADMM algorithm [Boy+11], a variation of the augmented Lagrangian method specialized for problems with a separable objective function. One of foci of its creators was to exploit the chordal decomposition techniques which allow a significant speed-up by exploiting the structure of the problem. It was intended for problems arising in control theory.

A great number of non-native commercial or open-source solvers are available through JuMP. These include Mosek, SCS, CSDP, SDPT3, SeDuMi and SDPA. Indeed, the fact that different solvers can be accessed through MOI makes it easy to write an interface. Let us summarize all the available SDP solvers in the following table.

Solver name	Type of problems	Algorithm	Coded in	Particularities
COSMO	LP, QP, SOCP, SDP	ADMM (1^{st} order)	Julia	Routines for exploiting *chordal sparsity*
CSDP	LP, SDP	Predictor-Corrector (IPM)	C	-
Hypatia	General conic problems	Skaaja and Ye (IPM)	Julia	Support for *exotic* cones, modularity
MOSEK	LP, SOCP, SDP	IPM	C	Industrial-grade, proprietary
ProxSDP	LP, SOCP, SDP	Primal-dual hybrid gradient (1^{st} order)	Julia	Returns low-rank solutions
SCS	LP, SOCP, SDP	ADMM (1^{st} order)	C	-
SDPA	LP, SDP	Predictor-Corrector (IPM)	C++	-
SDPNAL	SDP	ADMM (1^{st} order)	Matlab/C	Meant for very large problems
SDPT3	LP, SOCP, SDP	Predictor-Corrector (IPM)	Matlab/C	-
SeDuMi	LP, SOCP, SDP	Predictor-Corrector (IPM)	Matlab/C	-

Table 1.1: Comparison of all SDP solvers supported in JuMP

Chapter 2

Presentation of the algorithm and implementation details

In this chapter, we will present the implemented interior-point method. After outlining the theory behind the algorithm, we will address practical and algorithmic details of its implementation.

2.1 Interior-point methods for SDP

2.1.1 Brief introduction to interior-point methods

In this section, we will recall the main ingredients needed for the interior-point method. It should give the reader an overall understanding without mentioning the details. The relevant papers will be referenced. We will start by recalling some results about duality in the SDP case. Then we will show how the solution to the original problem can be approximated by solving a sequence of barrier problems which will lead to the central path equations. Then the problem of requiring symmetric iterates will be addressed. We will then introduce the HKM search direction, and finally present the Mehrotra-type predictor-corrector algorithm.

Duality

Let us recall the primal formulation of a semidefinite program in standard form:

$$\begin{aligned} &\min_{X \in \mathbb{S}^n} \operatorname{tr}(CX) \\ \text{s.t.} \quad &\operatorname{tr}(A_i X) = b_i, \ i = 1, \dots, m \\ &X \succcurlyeq 0 \end{aligned} \tag{2.1}$$

The dual problem associated with (2.1) is given by

$$\begin{aligned} &\max_{y,S} \ b^T y \\ \text{s.t.} \quad &\sum_{i=1}^{m} y_i A_i + S = C \\ &S \succcurlyeq 0 \end{aligned} \tag{2.2}$$

In the rest of this section, $A \bullet B$ will denote $\operatorname{tr}(A^T B)$ in order to simplify the notation. Let us introduce the following sets.

- $\mathcal{P} = \{X \in \mathbb{S}^n : X \succcurlyeq 0\}, \quad \mathcal{P}_+ = \{X \in \mathbb{S}^n : X \succ 0\}$: the set of positive semidefinite matrices and its interior.
- $\mathcal{D} = \{(y, S) \in \mathbb{R}^m \times \mathcal{P}\}, \quad \mathcal{D}_+ = \{(y, S) \in \mathbb{R}^m \times \mathcal{P}_+\}$
- $F(P) = \{X \in \mathcal{P} : A_i \bullet X = b_i \ \forall i\}, \quad F^0(P) = \{X \in F(P) : X \in \mathcal{P}_+\}$
- $F(D) = \{(y, S) \in \mathcal{D} : \sum_{i=1}^{m} y_i A_i + S = C\}, \quad F^0(D) = \{(y, S) \in F(D) : S \in \mathcal{P}_+\}$

The duality theorems for SDP are not as strong as in the LP setting. For instance, *weak duality* still holds true, but unfortunately *strong duality* does not, unless the primal and the dual have a strictly feasible solution (i.e. $F^0(P)$ and $F^0(D)$ are non-empty). We shall assume that both the primal and the dual have strictly feasible solutions. In that case, the optimal solution is given by the following system of equations:

$$\begin{aligned} A_i \bullet X &= b_i, \ \text{for } i = 1, ..., m \\ \sum_{i=i}^{m} y_i A_i + S &= C \\ X \bullet S = 0, X &\succcurlyeq 0, S \succcurlyeq 0 \end{aligned} \tag{2.3}$$

Barrier problem and the central path

Dealing with the semidefinite constraints directly is difficult. As in the LP case, we would like

to find a barrier function for these constraints (a function whose limit tends to infinity when approaching the boundary of the feasible set). A natural choice is

$$f(X) = -\ln\left(\det\left(X\right)\right) \tag{2.4}$$

Indeed, we consider the semidefinite cone which is given by the set $\{X \in \mathbb{S}^n : X \succcurlyeq 0\}$. It can be equivalently written as $\{X \in \mathbb{S}^n : \lambda_i(X) \geq 0\}$. Applying the standard logarithmic barrier gives

$$f(X) = -\sum_{i=1}^{n} \ln\left(\lambda_i(X)\right) = -\ln\left(\prod_{i=1}^{n} \lambda_i(X)\right) = -\ln\left(\det(X)\right)$$

It is a n-self-concordant barrier for the semidefinite cone [Nes14]. It can be shown that $f'(X) = -X^{-1}$. We also associate a parameter ν with the barrier function, which will determine the importance of the barrier function in the objective.

The problem becomes

$$\begin{aligned} &\min_{X \in \mathbb{S}^n} C \bullet X + \nu f(X) \\ \text{s.t} \quad &A_i \bullet X = b_i,\ i = 1, \ldots, m \\ &X \succ 0 \end{aligned} \tag{2.5}$$

Writing the KKT conditions for (2.5), we obtain

$$\begin{aligned} A_i \bullet X &= b_i, \text{ for } i = 1, ..., m \\ C - \nu X^{-1} - \sum_{i=1}^{m} y_i A_i &= 0 \\ X &\succ 0 \end{aligned} \tag{2.6}$$

By putting $S = \nu X^{-1} \succ 0$, we can rewrite (2.6) as

$$\begin{aligned} A_i \bullet X &= b_i, \text{ for } i = 1, ..., m \\ \sum_{i=i}^{m} y_i A_i + S &= C \\ XS &= \nu I \end{aligned} \tag{2.7}$$

where the third equation was premultiplied by X. This is the *central-path* equation. At this point, we can outline the main idea of interior-point methods. We assume that if we solve a sequence of problems of type (2.5) with decreasing values of ν by solving the nonlinear set of equations given by (2.7), we will converge to the solution of the actual problem since the barrier term will become less and less significant. Solving this equation requires a suitable numerical method for nonlinear equations. In the context of interior-point methods, the Newton method is used. Moreover, applying multiple iterations is prohibitively expensive, hence we will limit the method to only one step, which will hopefully stay close to the central path.

Since we take only one step $(\Delta X, \Delta y, \Delta S)$ from the previous iterate (X, y, S), let us suppose

that the next iterate is given by $(X + \Delta X, y + \Delta y, S + \Delta S)$. Plugging it into (2.7), we obtain

$$\begin{aligned} A_i \bullet \Delta X &= b_i - A_i \bullet X, \text{ for } i = 1, ..., m \\ \sum_{i=1}^{m} \Delta y_i A_i + \Delta S &= C - \sum_{i=1}^{m} y_i A_i - S \\ \Delta X S + \Delta X \Delta S &= \nu I - XS \end{aligned} \tag{2.8}$$

To end this subsection, let us introduce the following operators for symmetric matrices. Given an $n \times n$ symmetric matrix U, we define the vectorization operator

$$\text{svec}(U) = \begin{bmatrix} u_{11} & \sqrt{2}u_{21} & \cdots & \sqrt{2}u_{n1} & u_{22} & \sqrt{2}u_{32} & \cdots & \sqrt{2}u_{n2} & \cdots & u_{nn} \end{bmatrix}^T \tag{2.9}$$

which stacks the scaled columns of the lower triangle of a symmetric matrix.
Similarly, given $u = \text{svec}(U)$ we define

$$\text{smat}(u) = U \tag{2.10}$$

This operator is useful for further developments. Indeed, we observe that

$$A \bullet B = \sum_{i=1}^{n} a_{ii} b_{ii} + 2 \sum_{i=1} A^T_{1:i-1,i} B_{1:i-1,i} = \text{svec}(A)^T \text{svec}(B) \tag{2.11}$$

hence the inner products are preserved.

Another useful operation is the symmetrized Kronecker product of two matrices G and K (not necessarly symmetric) denoted $G \otimes_S K$. Given a vector u such that $u = \text{svec}(U)$, we have

$$(G \otimes_S K)\text{svec}(U) = \frac{1}{2}\text{svec}(KUG^T + GUK^T) \tag{2.12}$$

These two operators will help us rewrite the system of equations in a more compact way.

Symmetrization

We would like to apply Newton's method to (2.7). However, the third equation is problematic. Indeed, the interior-point method needs the iterates X and S to be symmetric. We can observe that since symmetric matrices are closed under addition, S is guaranteed to be symmetric by the second equation. However this is not the case for X. One can observe that if (2.6) is used, X has to be symmetric by the same argument. However, this symmetrization behaves badly in practice. For this purpose, Zhang introduced the following linear transformation [Zha97]. This symmetrization operatior is given by

$$H_P(M) = \frac{1}{2}(PMP^{-1} + P^{-T}M^T P^T) \tag{2.13}$$

for a given invertible matrix P, introduced to make the iterate X symmetric. Its main property is that it does not alter (2.7) nor the proximity measure. It may seem not very intuitive at first, but one can observe that for the special case when P is the identity matrix, the symmetrization transformation becomes the projection on the set of symmetric matrices. It can be shown that (2.7) and (2.25) are invariant under transformation (2.13). The modified central path equation is given by

$$\begin{aligned} A_i \bullet X &= b_i, \text{ for } i = 1, ..., m \\ \sum_{i=i}^{m} y_i A_i + S &= C \\ H_P(XS) &= \nu I \end{aligned} \tag{2.14}$$

Again, supposing that the solution is given by $(X + \Delta X, y + \Delta y, S + \Delta S)$, this becomes

$$\begin{aligned} A_i \bullet \Delta X &= b_i - A_i \bullet X, \text{ for } i = 1, ..., m \\ \sum_{i=1}^{m} \Delta y_i A_i + \Delta S &= C - \sum_{i=1}^{m} y_i A_i - S \\ H_P(\Delta X S + \Delta X \Delta S) &= \nu I - H_P(XS) \end{aligned} \tag{2.15}$$

We can write the last equation in full detail, which gives

$$\begin{aligned} A_i \bullet \Delta X &= b_i - A_i \bullet X, \text{ for } i = 1, ..., m \\ \sum_{i=1}^{m} \Delta y_i A_i + \Delta S &= C - \sum_{i=1}^{m} y_i A_i - S \\ P(\Delta X S + \Delta X \Delta S)P^{-1} + P^{-T}(S\Delta X + \Delta S X)P^T &= 2\nu I - PXSP^{-1} - P^{-T}SXP^T \end{aligned} \tag{2.16}$$

Let us now use both the svec operator and the symmetrized Kronecker product to rewrite (2.14) in a more compact way. Recalling (2.11), let us define the following matrix

$$\mathcal{A} = \begin{bmatrix} \text{svec}(A_1)^T \\ \vdots \\ \text{svec}(A_m)^T \end{bmatrix}$$

We can rewrite the first equation as

$$\mathcal{A}\text{svec}(\Delta X) = b - \mathcal{A}\text{svec}(X) \tag{2.17}$$

Similarly, since symmetric matrices are closed under addition, the second equation can be rewritten as

$$\mathcal{A}^T \Delta y + \text{svec}(\Delta S) = \text{svec}(C) - \mathcal{A}^T y - \text{svec}(S) \tag{2.18}$$

Finally, identifying the symmetrized Kronecker products in (2.16), the third equation can be rewritten as

$$(P \otimes_S P^{-T} S)\text{svec}(\Delta X) + (PX \otimes_S P^{-T})\text{svec}(\Delta S) = \text{svec}(\nu I - PXSP^{-1} - P^{-T}SXP^T) \tag{2.19}$$

Putting

$$\begin{aligned} r_p &= b - \mathcal{A}\text{svec}(X) \\ r_d &= \text{svec}(C) - \mathcal{A}^T y - \text{svec}(S) \\ r_c &= \text{svec}(\nu I - PXSP^{-1} - P^{-T}SXP^T) \\ E &= P \otimes_S P^{-T}S \\ F &= PX \otimes_S P^{-T} \end{aligned} \tag{2.20}$$

we can rewrite (2.14) as a block linear system

$$\begin{bmatrix} 0 & \mathcal{A} & 0 \\ \mathcal{A}^T & 0 & I \\ 0 & E & F \end{bmatrix} \begin{bmatrix} \Delta y \\ \text{svec}(\Delta X) \\ \text{svec}(\Delta S) \end{bmatrix} = \begin{bmatrix} r_p \\ r_d \\ r_c \end{bmatrix} \tag{2.21}$$

Solving (2.21) is done using block Gaussian elimination. Without going into too much detail, the steps are the following:

$$(\mathcal{A}E^{-1}F\mathcal{A}^T)\Delta y = r_p + \mathcal{A}E^{-1}Fr_d - \mathcal{A}E^{-1}r_c \tag{2.22}$$

In the rest of this report we will refer to $\mathcal{A}E^{-1}F\mathcal{A}^T$ as $\mathcal{B}$. We can then compute ΔS as

$$\Delta S = \text{smat}(r_d) - \text{smat}(\mathcal{A}^T \Delta y) \tag{2.23}$$

And finally we compute ΔX as

$$\Delta X = \text{smat}(E^{-1}r_c - E^{-1}F\text{svec}(\Delta S)) \tag{2.24}$$

The current value of μ at each iteration will be approximated as

$$\mu = \frac{X \bullet S}{n} \tag{2.25}$$

Finally, we will put ν equal to $\sigma\mu$, with μ given by (2.25) and σ a carefully chosen parameter between 0 and 1. The choice of σ determines the efficiency of the algorithm. The two limiting cases are:

- $\sigma = 0$: the system becomes the original unperturbed KKT system. In this case we try to reach the optimal point directly in one step. That way we can make more progress towards the solution, but as a trade-off we get away from the central path. Moreover, since the iterates have to stay semidefinite positive, this step will often be damped to ensure that.
- $\sigma = 1$: we make one step towards the optimal point of the barrier problem for the given value of μ which will stay close to the central path, but will not progress as much to

the solution.

The HKM direction

The HKM direction was proposed by Kojima, Shindoh and Hara. It corresponds to the particular choice of P given by

$$P = S^{\frac{1}{2}} \tag{2.26}$$

This direction typically behaves very well on problems with several semidefinite blocks, which motivates this choice. It is also relatively easy to compute.

Solving the linear system

Let us present a pseudocode gathering all the steps presented in the subsection above. As it is often necessary to switch between a matrix X and its vectorization $\text{svec}(X)$, we will use the following convention: uppercase letters will denote matrices while their lowercase equivalents will denote their vectorizations (for instance x stands for $\text{svec}(X)$, Δs for $\text{svec}(\Delta S)$ and so on). The `Matlab` convention for adressing individual columns of matrices is used, i.e. $\mathcal{A}[:, i]$ stands for the i-th column of $\mathcal{A}$.

$L \leftarrow \text{cholesky}(X)$ ▷ Compute the lower triangular Cholesky factor L of X s.t. $X = LL^T$
$R \leftarrow \text{cholesky}(S)$ ▷ Cholesky factor of S s.t. $S = RR^T$

$r_p \leftarrow b - \mathcal{A}x$ ▷ Update the residuals
$r_d \leftarrow c - \mathcal{A}^T y - s$
$r_c \leftarrow \text{svec}(\mu\sigma\mathcal{I} - XS)$

for $i = 1 : m$ **do**
    $A_i \leftarrow \text{smat}(\mathcal{A}[i, :])$
    $\widetilde{\mathcal{B}}[:, i] \leftarrow \text{svec}(\frac{1}{2}(S^{-1}A_iX + (S^{-1}A_iX)^T))$ ▷ We do not compute S^{-1} explicitly
    ▷ but we use R and a triangular solver
end for

$\mathcal{B} \leftarrow \mathcal{A}\widetilde{\mathcal{B}}$
$\widetilde{H} \leftarrow S^{-1}(\text{smat}(R_d) - \text{smat}(R_c)X$
$h \leftarrow r_p - \mathcal{A}\text{svec}(\frac{1}{2}(\widetilde{H} + \widetilde{H}^T))$

$\Delta y = \mathcal{B}^{-1}h$ ▷ Compute the solution
$\Delta s = r_d - \mathcal{A}^T\Delta y$
$\Delta x = \text{svec}(\frac{1}{2}(S^{-1}\Delta SX + (S^{-1}\Delta SX)^T))$

Algorithm 1: Computation of a single step

Maintaining the iterates feasible

The computed solution $(\Delta X, \Delta y, \Delta S)$ will be used to update the current iterate (X, y, S) as $(X + \Delta X, y + \Delta y, S + \Delta S)$. However, such an updated iterate is not guaranteed to be positive

definite if the step is too long. For this purpose, we have to compute the maximum feasible step-lengths (α, β) such that $X + \alpha\Delta X$ and $S + \beta\Delta S$ fulfill this condition.

$$X + \alpha\Delta X \succ 0$$

Pre-multiplying by X^{-1} yields

$$X^{-1}X + \alpha X^{-1}\Delta X \succ 0$$

$$I + \alpha X^{-1}\Delta X \succ 0$$

This condition can be equivalently written as

$$\lambda_{\min}(I + \alpha X^{-1}\Delta X) > 0$$

This yields

$$1 + \alpha\lambda_{\min}(X^{-1}\Delta X > 0$$

We thus have to choose α according to

$$\alpha > \frac{-1}{\lambda_{\min}(X^{-1}\Delta X)}$$

Of course, if the updated iterate is already positive definite (i.e. $\lambda_{\min}(X^{-1}\Delta X) > 0$), the damping is not necessary. Moreover, a step-length parameter $\tau < 1$ is added, guaranteeing that the next iterate is strictly in the interior of the cone. Hence α will be computed as

$$\alpha = \min\left(1, \frac{-\tau}{\lambda_{\min}(X^{-1}\Delta X)}\right) \tag{2.27}$$

An analogous approach is followed for β, which is given by

$$\beta = \min\left(1, \frac{-\tau}{\lambda_{\min}(S^{-1}\Delta S)}\right) \tag{2.28}$$

2.1.2 The Predictor-Corrector algorithm

In this section we will finally present the predictor-corrector algorithm. It is a modification of the path-following method where in fact two steps are computed, a predictor step and a corrector step. A predictor step $(\delta X, \delta y, \delta S)$ is computed with the value $\sigma = 0$, which corresponds to trying to reach the optimal point directly. As mentioned in the previous subsection, the choice of σ is very important. Its value should be large if progress can be made easily, or small if we need to stay close to the central path. The predictor step serves exactly this purpose. Indeed, it helps determine how easy it is to make progress, but is not used to update the iterates afterwards. Then a corresponding value of σ is computed. Finally, a corrector step is computed and taken using the previously computed σ. As the predictor step can be seen as a first-order

approximation of the final step, it can be plugged into the system of equations in order to obtain a second-order correction. This second-order correction is given by

$$R_s = -H_p(\delta X \delta S) \tag{2.29}$$

It is worth noting that only the right-hand side of the linear system changes between the predictor and the corrector steps. Hence the same factorization can be reused. This makes the predictor-corrector a very clever modification.

The stopping criterion is taken from [TTT03]. We define the two following measures:

$$\texttt{rel\_gap} = \frac{X \bullet S}{\max(1, 0.5(|C \bullet X| + |b^T y|))} \tag{2.30}$$

$$\texttt{infeas\_meas} = \max\left(\frac{\|r_p\|}{\max(1, \|b\|)}, \frac{\|r_d\|}{\max(1, \|C\|)}\right) \tag{2.31}$$

and we stop the algorithm if both `rel_gap` and `infeas_meas` are sufficiently small.

1: Let X, y, S be initial iterates, $\tau = 0.98$
2:
3: **while** Stopping criterion false **do**
4: # Predictor step
5: Form the residuals ($\sigma = 0$ in r_c)

$$r_p = b - \mathcal{A}\text{svec}(X)$$
$$r_d = \text{svec}(C) - \mathcal{A}^T y - \text{svec}(S)$$
$$r_c = \text{svec}(\sigma\mu I - PXSP^{-1} - P^{-T}SXP^T)$$

6: Solve for $(\delta X, \delta y, \delta S)$ using Algorithm 20

$$\begin{bmatrix} 0 & \mathcal{A} & 0 \\ \mathcal{A}^T & 0 & I \\ 0 & E & F \end{bmatrix} \begin{bmatrix} \delta y \\ \text{svec}(\delta X) \\ \text{svec}(\delta S) \end{bmatrix} = \begin{bmatrix} r_p \\ r_d \\ r_c \end{bmatrix}$$

7: Compute the maximum feasible step-lengths

$$\alpha = \min\left(1, \frac{-\tau}{\lambda_{\min}(X^{-1}\delta X)}\right), \quad \beta = \min\left(1, \frac{-\tau}{\lambda_{\min}(S^{-1}\delta S)}\right)$$

8: Compute the parameter σ

$$\sigma = \min\left(1, \left[\frac{(X + \alpha\delta X) \bullet (S + \beta\delta S)}{X \bullet S}\right]^{exp}\right), \quad \text{where } exp \text{ is given by}$$
$$exp = \begin{cases} \max(1, 3\min(\alpha, \beta)^2) \text{ if } \mu(X, S) > 10^6 \\ 1 \text{ else} \end{cases}$$

9: # Corrector step
10: Compute the quadratic correction residual r_s and r_c with σ computed at line 8
11: Solve for $(\Delta X, \Delta y, \Delta S)$ using Algorithm 20

$$\begin{bmatrix} 0 & \mathcal{A} & 0 \\ \mathcal{A}^T & 0 & I \\ 0 & E & F \end{bmatrix} \begin{bmatrix} \Delta y \\ \text{svec}(\Delta X) \\ \text{svec}(\Delta S) \end{bmatrix} = \begin{bmatrix} r_p \\ r_d \\ r_c + r_s \end{bmatrix}$$

12: Compute the step-lengths using

$$\alpha = \min\left(1, \frac{-\tau}{\lambda_{\min}(X^{-1}\Delta X)}\right), \quad \beta = \min\left(1, \frac{-\tau}{\lambda_{\min}(S^{-1}\Delta S)}\right)$$

13: # Update the iterates

$$X \leftarrow X + \alpha\Delta X$$
$$y \leftarrow y + \beta\Delta y$$
$$S \leftarrow S + \beta\Delta S$$

14: **end while**

Algorithm 2: The Predictor-Corrector algorithm

2.2 Implementation details

2.2.1 Presentation

The solver is contained in a Julia package called `MySDPSolver`. It consists of three main parts: a structure to store the problem (`my_problem.jl`), an implementation of the predictor-corrector algorithm (`my_solver.jl`) and a wrapper around `MOI` (`MOI_wrapper.jl`).

```
using JuMP, MySDPSolver
model = Model(MySDPSolver.Optimizer)
# Some data in the standard form
A = [1.0 0.0
     0.0 0.0]
C = [1.0 2.0
     2.0 1.0]
b = 1.0
@variable(model, X[1:2,1:2], PSD)
@constraint(model, tr(A*X) == b)
@objective(model, Min, tr(C*X))
optimize!(model)
display(objective_value(model))
-2.999
display(value.(X))
  1.0 -1.999
 -1.999  4.0
```

Listing 2.1: A simple SDP `JuMP` model solved with `MySDPSolver`

The problem structure consists of the vectorized matrix C, the vector b and a sparse matrix $\mathcal{A}$ (since matrices in Julia are in column-major order, $\mathcal{A}^T$ is actually stored). Since the format was inspired by that of `SDPA` [Mit21], information about distinct blocks in the matrices is kept. Indeed, SDPs often exhibit a block structure, meaning that the variable X has the following form

$$X = \begin{bmatrix} X_1 & & \\ & \ddots & \\ & & X_n \end{bmatrix}$$

This in turn allows processing each block separately, greatly improving the performance since the individual operations are performed on smaller matrices.

The problem structure also stores the solution and some solver statistics. The solver is implemented roughly as in Algorithm (14). The wrapper constitutes the interface between the solver and `JuMP`.

2.2.2 Linear Algebra

In this section, we will discuss the choices regarding the numerical linear algebra. These will not affect the number of iterations of the algorithm, however they are crucial for practical efficiency of the code. Indeed, it was observed that most of the cpu time is spent on these computations. This also goes to show that improving numerical linear algebra is a good starting point for optimizing the solver.

In order to put some implementation choices into perspective, let us first partition the code into several distinctive parts and time each of them. This will give the reader a more clear image of the distinctive parts of the algorithm. It will also give an idea of their actual computational cost and pinpoint the bottlenecks.

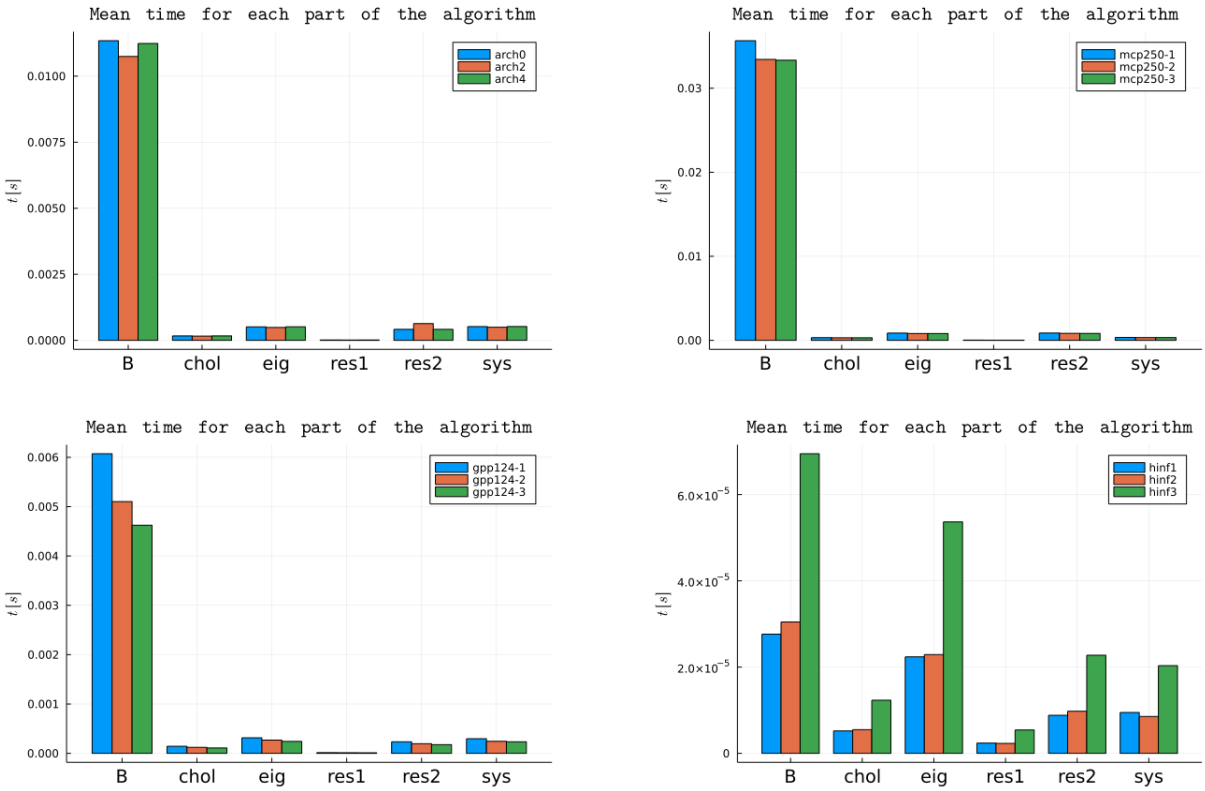

Figure 2.1: Identification of expensive parts of the algorithm for several test problems

These plots were generated by choosing three problems from four different families of problems found in the SDP-LIB [Bor99b]. The distinctive chosen parts of the algorithm are:

- ◇ Computation of the Cholesky factors (`chol`): lines 1 - 2
- ◇ Updating the residuals (`res1`): lines 4 - 6
- ◇ Computation of $\mathcal{B}$ (`B`): lines 8 - 14

- ◇ Solving the linear system (`sys`): line 18
- ◇ Computation of the minimum eigenvalues (`eig`): line 7
- ◇ Computation of residuals in the corrector step (`res2`): line 10

We only chose to measure the time for major matrix operations. Scalar operations and inexpensive vectorized operations were omitted. As we clearly see, the main computation time is spent forming $\mathcal{B}$. For the `hinf` problems, the difference is less noticeable. This is probably due to the fact that these problems are much smaller, with smaller variables and less constraints, hence the timings become much shorter and more even compared to the other families of problems.

Computation of $\mathcal{B}$

This computation (line 8 to line 14) is the main bottleneck of the algorithm. This is hardly surprising. Indeed, if we consider all the matrices dense, m $n \times n$ matrix multiplications and backward substitutions have to be performed, which amounts to roughly $\mathcal{O}(m(n^3 + n^2))$ operations. As seen in Figure 2.1, computing $\mathcal{B}$ takes around 90% of the total computation time at each iteration. In the real setting, A_i are sparse. However, for memory efficiency reasons, we do not store all the matrices A_i but we use the vectorized form $\mathcal{A}$. This in turn requires applying the smat() function. This is tricky, since Julia does not have a special symmetric matrix type where only the upper triangle is stored, and custom algorithms recovering the sparsity pattern in the `CSC` format were not a big improvement. However, the way around this issue is to use the `MKLSparse.jl` package. It provides a way to call the functions from Intel's `MKLSparse` library, which provides `BLAS`-like routines for sparse matrices. Most importantly, it only needs the upper triangle of a symmetric sparse matrix, hence switching between the vectorized and matrix forms becomes nearly instantaneous.

Computation of the minimum eigenvalue

In order to satisfy the semidefinite constraint of the iterates, the minimum eigenvalue of $X^{-1}\delta X$ has to be computed (line 7). However, for the sake of efficiency, it is better to compute the minimum eigenvalue of $L^{-1}\delta X L^{-T}$, where $X = LL^T$. Indeed, using that and the fact that given two matrices A and B, AB and BA have the same spectrum, we have

$$\lambda_{\min}(X^{-1}\delta X) = \lambda_{\min}((LL^T)^{-1}\delta X) = \lambda_{\min}(L^{-T}L^{-1}\delta X) = \lambda_{\min}(L^{-1}\delta X L^{-T})$$

Since the Cholesky factors were already computed, forming $L^{-1}\delta X L^{-T}$ requires only two backward substitutions. Moreover, since δX is symmetric, it admits an eigendecomposition of the form $Q^T \Lambda Q$. Plugging it into the former expression, we have

$$L^{-1}\delta X L^{-T} = L^{-1}(Q^T\Lambda Q)L^{-T} = (L^{-1}Q^T)\Lambda(QL^{-T}) = (QL^{-T})^T\Lambda(QL^{-T})$$

which shows that this matrix is symmetric. This in turn allows us to use specific eigenvalue algorithms which are more efficient. The results are shown on Figure 2.2. The computation time for the smallest eigenvalue of $X^{-1}\delta X$ and that of $L^{-1}\delta X L^T$ for varying matrix sizes are compared. The first was computed using the standard `eigmin` Julia function which under the hood calls the `LAPACK` function `geevx`. The second was computed using the `syevr!` function from `LAPACK` directly.

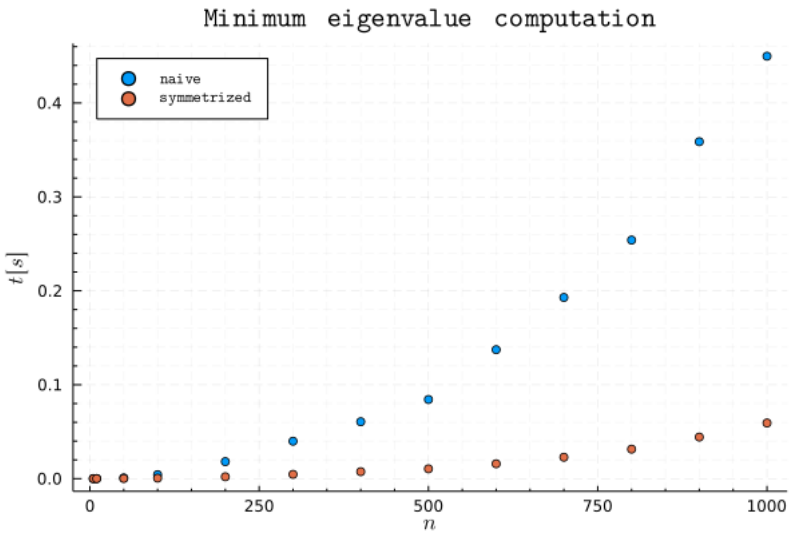

Figure 2.2: Mean times for computing the minimum eigenvalue

We observe a substantial difference between the two approaches.

Solving the linear system

Solving the linear system (line 18) is done using a QR factorization. As presented in [Tod98], it has better numerical stability than using a Cholesky factorization. Moreover, the Householder vectors are stored at the predictor step, and can be reused at the corrector step, saving computation time.

2.2.3 Variations of the algorithm

Besides the implementation of linear algebra, one very important aspect of the algorithm is the choice for the update rule for σ as well as the choice of τ. The existing rules were designed as heuristics working well in practice, however no theoretical justifications have been provided to account for their good behaviour. Therefore, it is worth investigating if other choices can lead to better results.

The choice of σ

Traditionally, Mehrotra proposed the following update strategy for the predictor-corrector

algorithm for linear programming:

$$\sigma = \left(\frac{(X + \alpha\Delta X) \bullet (\beta\Delta S)}{X \bullet S}\right)^3$$

In the SDP version, the exponent is not fixed but chosen according to the following update rule:

$$exp = \begin{cases} \max(1, 3\min(\alpha, \beta)^2) & \text{if } \mu(X, S) > 10^6 \\ 1 & \text{else} \end{cases}$$

We will thus test several strategies and report the results. We will test a strategy on one problem from each family of problems (see Section 3.1.1), so that hopefully the results are representative of the performance of a given strategy on most problems. Indeed, it can happen that a particular choice is very good for a certain problem, but performs extremely poorly on other problems.

We will first test the strategy with a fixed exponent.

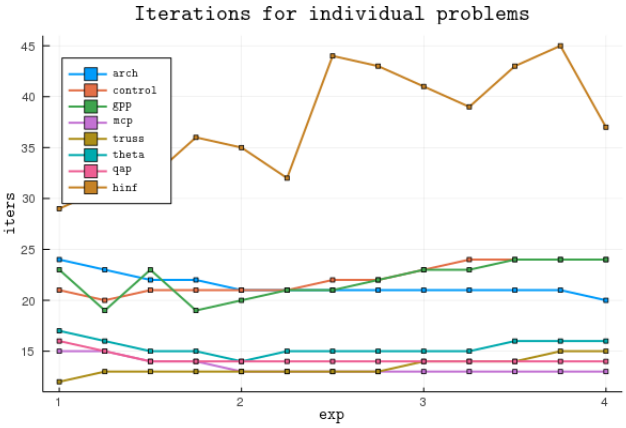

Figure 2.3: Evolution of the number of iterations for individual problems

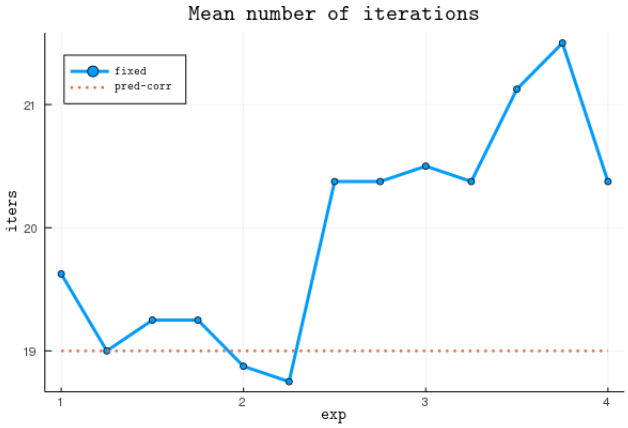

Figure 2.4: Mean number of iterations for the given set of problems

We do not observe the same behaviour for all problems. Indeed, for some problems (`arch`,`qap`) the number of iterations seems to decrease with the exponent. This suggests that the central path seems easier to follow and thus one might try to use smaller values of σ. Other problems exhibit a more chaotic behaviour. Finally, it seems that the initial choice of 3 in the LP setting is slightly too big for SDP, where a value around 2 (2.25) would seem more appropriate. In previous works, such tests were often performed on randomly generated problems, however, it is more likely that a real-world problem would be closer to one of those used, such a testing strategy seems more relevant. However, this strategy is not ideal too, since the sample is probably not big enough and the results can be sensitive to outliers.

The choice of τ

The algorithm is very sensitive to the choice of the step-length parameter τ. Several fixed values were proposed, as well as an adaptative updating strategy. In the case of LP, a fixed value of

$\tau = 0.99995$ is typically used. The adaptative strategy computes τ at iteration i as

$$\tau_i = 0.9 + 0.09 \min(\alpha_i, \beta_i)$$

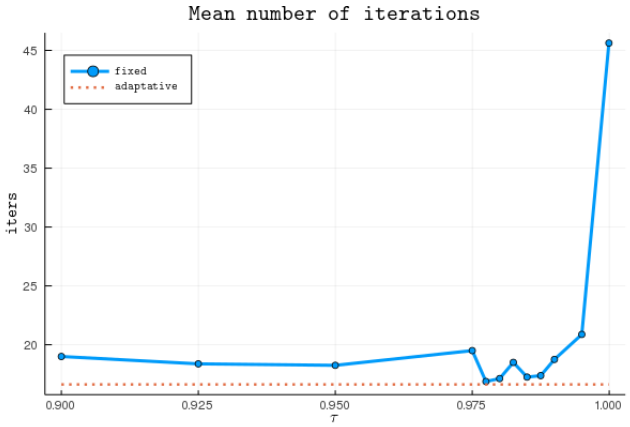

Figure 2.5: Mean number of iterations for the given set of problems. The fixed strategy is compared against the adaptative strategy.

We observe an interesting behaviour in the interval $[0.975, 0.995]$. The oscillations show that the choice of this value is not trivial. It seems that the best fixed value for τ is 0.98. SDP prefers thus a less aggressive choice than LP. However, we observe that the adaptative strategy yields better results overall. We conclude thus that the best strategy that should be implemented is the adaptative one, but if we were to use a fixed value for τ, 0.98 seems appropriate.

Initial iterate and scaling

The initial iterate and the scaling scheme presented in [TTT03]. It was observed that it has a tremendous impact on the convergence of the algorithm. Indeed, it was observed that without the appropriate scaling, the convergence became very slow and numerical problems could occur.

2.2.4 Julia-specific

As Julia is a relatively new language and some canonical approaches to code certain computations seem to not yet be established or communicated in the official documentation, some mechanics of the language can be puzzling for new users coming from `Matlab` or `Python` and impact the performance. Here we will list some of them.

Avoiding array allocation

Doing scientific computing often involves modifying certain parts or slices of an array. In `Matlab` or in `Python`, these operations are done in-place, hence no additional memory is allocated. However, in Julia such an operation creates an extra copy of the slice by allocating memory, performs the desired computation and then copies the result into the original array. Since

memory management is the main bottleneck in modern computing, it is crucial to avoid these unnecessary operations. Fortunately, Julia provides a way to change this behaviour, namely the `@views` macro. When placed before a block of code, all arrays will be modified in-place.

Bounds checking

By default, Julia always checks the bounds of an array when performing computations. This slows down the computation, since it involves an extra conditional evaluation and because it prevents the `SIMD` compiler to produce highly vectorized efficient code. Again, Julia provides a macro `@inbounds` which disables bound checking.

`BLAS` and `LAPACK` functions

Since Julia provides a direct way to call `BLAS` and `LAPACK` routines, it seems beneficial to use them in computationally intensive parts of the code. Indeed, using multiple dispatch, Julia should figure out the most appropriate routine and call it by itself, but it seems advantageous to make these calls directly. That way the user knows exactly what happens in the code. Moreover, it allows the user to always choose the in-place version of the routines and it was observed that when calling them directly, marginally less memory allocations are made. However, one issue needs to be addressed. The performance of `BLAS` and `LAPACK` depends on optimal cache and threads usage. More specifically, they should use the exact number of physical cores of a cpu to parallelize computations. However, in Julia, the number of threads is set to the number of logical cores by default. This has to be reset manually using `BLAS.set_num_threads()` function. It has tremendous impact on performance. Indeed, if set incorrectly, the computations could easily take 50% more time.

Writing compiler-friendly code

Since Julia uses the `LLVM` compiler, some coding styles promote the production of more optimized code by the compiler. This in turn may seem counter-intuitive for high-level language users. To illustrate this, let us present the difference in performance of two seemingly similar implementations of the smat! function. In the naive implementation, one loops through the upper triangle of the matrix and increments a counter in order to find the corresponding entry in the svec. The loops are thus dependent. However, one can rewrite it without a counter such that the loops are not dependent anymore. This allows the compiler to rearrange them in a optimal way, resulting in a speed-up.

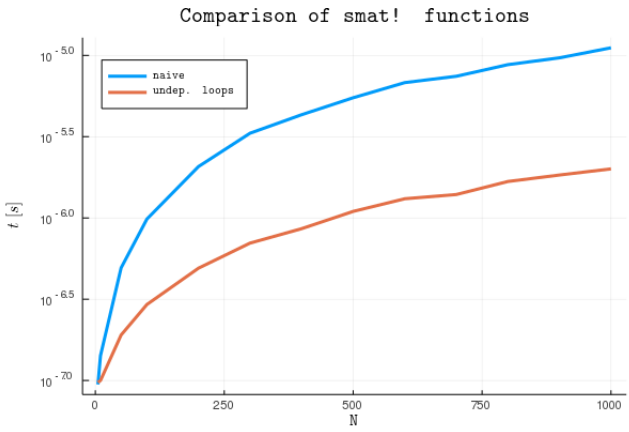

Figure 2.6: Comparison of both implementations with varying sizes of the vectorized matrix. For large matrices, the second implementation is one order of magnitude faster.

Obviously, this is not the most computationally demanding task and it only has a marginal effect on the final timing, but it goes to show that learning about compiler optimization techniques can be useful in Julia and sometimes can prevent the user from making performance-critical mistakes.

Chapter 3

Benchmark results

In this chapter, we will provide the numerical results obtained while testing the solver. More precisely, comparisons are drawn with other widely available solvers using the standard SDPLIB benchmark.

3.1 Numerical results of several solvers using the SDP-Lib benchmark

In this section, we will test the solver on the problems found in the SDPLIB collection of semidefinite programs [Bor99b]. Furthermore, we will compare its performance with other available interior-point solvers, as well as two first-order solvers.

The problems in the SDPLIB library are in the following form:

$$\begin{aligned} &\min\ b^T y \\ &\text{st} \sum_{i=1}^{m} A_i y_i - C = S \\ &S \succcurlyeq 0 \end{aligned} \tag{3.1}$$

and its dual

$$\begin{aligned} &\max\ \text{tr}(CX) \\ &\text{st}\ \text{tr}(A_i X) = b_i,\ i = 1, ..., m \\ &X \succcurlyeq 0 \end{aligned} \tag{3.2}$$

This is nearly the standard form (1.1), except for the flipped objective sign. The matrices A_i, C, X and S are assumed to be symmetric. They exhibit a block-diagonal structure specified

in the file format. Diagonally constrained blocks are explicitly marked. Each block of the matrices A_i is specified in the sparse coordinate format.

The methodology consisted in reading each problem from a SDPA format file and converting it into a JuMP (resp. Yalmip) model using the function `JuMP.read_from_file` (resp. `loadsdpafile`). In the case of JuMP, the model is sometimes reformulated and does not correspond exactly to (3.1). In order to resolve this issue, the dualization package `Dualization.jl` [Bod+21] was used. Indeed, in some cases dualizing a model can make it closer to the form expected by the solver, avoiding additional constraints or slack variables.

Six solvers in total will be compared. The interior-point solvers comprise two native `Julia` solvers (`MySDPSolver` and `Hypatia`), `SeDuMi` [Stu99] (implemented and accessed through `Matlab` with some routines encoded as `.mex` files) and `Mosek` [ApS19], a `C` implementation accessed through its `Matlab` API. The solvers implementing first-order methods consist of two pieces of software written entirely in `Julia`, namingly `ProxSDP` and `COSMO`. Comparing interior-point methods with first-order methods will let us assess the differences between these two approaches in a practical setting. These will principally differ in the number of iterations and the precision of the obtained solution. Indeed, interior-point method usually take only a relatively small number iterations to converge, however the cost per iteration is very high. On the contrary, first-order methods exhibit a very cheap per iteration cost but need many more iterations to reach convergence (in fact the difference is often of several orders of magnitude). Moreover, interior-point methods are able to compute highly accurate solutions. This happens when the Newton method reaches the region of quadratic convergence, which means that only a few supplementary iterations are necessary to greatly improve precision. This is not the case of first-order methods, which are often able to reach a solution with modest accuracy in reasonable time, but show a very slow convergence rate when trying to get compute a more precise solution.

The stopping criteria for the interior-point methods are based on predefined small duality gap ε which the solver is supposed to attain. In order to have meaningful comparisons, this value is set the same for all the interior-point solvers. `ProxSDP` checks for convergence by monitoring the evolution of the norm of the difference between successive iterates, respectively for the primal (`tol_primal`) and the dual (`tol_dual`) [SGV18]. The algorithm stops when both these criteria are met. `COSMO` stop when the norms of the residual vectors become smaller than a fixed absolute tolerance ϵ_{abs} plus a relative tolerance ϵ_{rel} taking into account the size of the solution [GCG20].

Solver name	Tolerance	Iteration limit	Time limit [s]
COSMO	$\epsilon_{abs} = 10^{-5}$, $\epsilon_{rel} = 10^{-5}$	10^6 (default: 5000)	2400
Hypatia	$\varepsilon = 10^{-6}$	1000	-
Mosek	$\varepsilon = 10^{-6}$	50	-
MySDPSolver	$\varepsilon = 10^{-6}$	50	-
ProxSDP	tol_primal=10^{-3} tol_dual=10^{-3}	10^6	2400
SeDuMi	$\varepsilon = 10^{-6}$	150	-

Table 3.1: The settings used for different solvers. The meaning of Tolerance for each solver is explained above. For COSMO, the iteration limit was raised to allow for convergence. For solvers both COSMO and ProxSDP, a time limit of 40 minutes was set.

The benchmarks were performed on a desktop computer equipped with a Intel® Core™ i7-2600 CPU @ 3.40GHz × 8 processor with 11.7 GiB of RAM, running Fedora 27. The versions of the software used were:

- ◇ Julia 1.6.1, JuMP v0.21.4
- ◇ Hypatia v0.5.1
- ◇ Matlab 2018a, Yalmip
- ◇ SeDuMi 1.3.4
- ◇ Mosek 9.1.9

3.1.1 Results for SDPLIB

In this section, we will present the benchmarks for different families of problems found in the SDPLIB. For the comparison of solvers, the number of iterations is reported, as well as the total CPU time in seconds spent in the solver (time needed for loading the model is not taken into account). An averaged value avg for each column (computed using a geometric mean) is also provided. For MySDPSolver, an additional table containing more precise solver statistics is included. These are inspired by those found in [Stu98]:

- ◇ pobj: the value of the primal objective $C \bullet X$
- ◇ dobj: the value of the dual objective $b^T y$
- ◇ μ: the duality gap given by $\frac{X \bullet S}{n}$

- ◇ `relgap`: the ralative duality gap given by $\frac{X \bullet S}{\max(1,0.5(|C \bullet X| + |b^T y|))}$
- ◇ $\|Ax - b\|_2$: quantifies the constraint violations.
- ◇ $\|x_-\|_2$: defined as $\|[-\lambda_{\min}(X)]_+\|_2$, quantifies the semidefinite constraint violation for the primal.
- ◇ $\|s_-\|_2$: defined as $\|[-\lambda_{\min}(C - A^T y)]_+\|_2$, quantifies the semidefinite constraint violation for the dual.
- ◇ $\|x\|_2$: norm of the solution.
- ◇ $\|y\|_2$: norm of the solution.

The `arch` problems

These problems are originally from [NO92]. They arise from topology and geometry optimization of trusses and frames forming arches. In short, it consists in finding the optimal nodes and the connecting elements subject to some constraints and optimizing a physical objective, for instance the fundamental frequency.

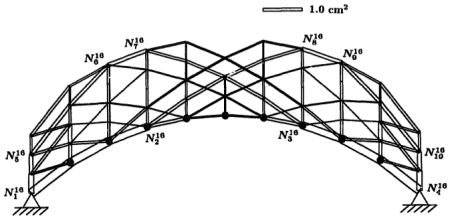

Figure 3.1: The truss with an optimal topology

These problems have a structure with one semidefinite block and one diagonally constrained block. The results for different solvers are available on Table (3.2).

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
arch0	174	335	23	1.157	40	7.633	29	2.33	21	0.743
arch2	174	335	22	1.033	34	4.621	28	2.311	21	0.689
arch4	174	335	23	1.078	36	4.754	30	2.078	21	0.704
arch8	174	335	23	1.078	50	6.634	34	2.754	23	0.745
avg	174	335	22.74	1.085	39.55	5.77	30.16	2.35	21.48	0.719

Table 3.2: Comparison of different solvers performance on the `arch` family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
arch0	-0.57	-0.57	$1 \cdot 10^{-7}$	$8.71 \cdot 10^{-7}$	$4.47 \cdot 10^{-10}$	0	0	10.85	$4.74 \cdot 10^{-2}$
arch2	-0.67	-0.67	$8.97 \cdot 10^{-8}$	$5.34 \cdot 10^{-7}$	$1.59 \cdot 10^{-10}$	0	0	9.99	$4.08 \cdot 10^{-2}$
arch4	-0.97	-0.97	$1.05 \cdot 10^{-7}$	$8.74 \cdot 10^{-7}$	$3.14 \cdot 10^{-10}$	0	0	11.18	$6.48 \cdot 10^{-2}$
arch8	-7.06	-7.06	$2.87 \cdot 10^{-8}$	$3.37 \cdot 10^{-7}$	$7.63 \cdot 10^{-9}$	0	0	41.21	0.4

Table 3.3: Statistics of `MySDPSolver` on the `arch` family

We observe a very good performance of `Mosek`. On the other hand `Hypatia` takes substantially more iterations to converge. `MySDPSolver` statistics look very decent. We also tested both `ProxSDP` and `COSMO`, but none managed to solve any of these problems. It seems that the convergence is too slow and the maximum number of iterations was reached for each instance, without getting close to the solution.

The `control` problems

These problems are found in [Fuj+99]. They come from the so-called $\mathcal{S}$-procedure in control theory where one tries to find an invariant ellipsoid for a linear system with uncertain time-varying feedback [VB96]. The structure of these problems consists of two semidefinite blocks, with the bigger being roughly twice the size of the other.

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
control1	21	15	20	0.191	26	2.144	30	0.156	24	0.283
control2	66	30	21	$2.6 \cdot 10^{-2}$	27	$3.6 \cdot 10^{-2}$	32	0.287	22	0.239
control3	136	45	23	$8.1 \cdot 10^{-2}$	36	0.146	34	0.659	25	0.303
control4	231	60	24	0.199	36	0.421	36	1.125	22	0.374
control5	351	75	25	0.442	44	1.53	34	1.775	22	0.588
control6	496	90	50	1.716	48	3.088	40	4.209	24	1.1
control7	666	105	28	1.753	51	6.755	40	7.107	27	2.11
control8	861	120	27	2.784	56	11.952	40	12.107	25	3.15
control9	1081	135	27	5.017	54	20.05	39	19.984	23	4.816
control10	1326	150	50	15.09	60	30.81	42	34.001	30	9.729
control11	1596	165	50	23.181	65	51.438	45	59.818	25	12.321
avg	350.94	73.63	29.51	0.94	43.86	2.708	37.19	3.46	24.35	1.306

Table 3.4: Comparison of different solvers performance on the `control` family

name	pobj	dobj	μ	relgap	$\|Ax - b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
control1	-17.78	-17.78	$7.23 \cdot 10^{-8}$	$4.44 \cdot 10^{-7}$	$1.55 \cdot 10^{-8}$	0	0	17.81	40.01
control2	-8.3	-8.3	$4.3 \cdot 10^{-8}$	$6.08 \cdot 10^{-7}$	$1.36 \cdot 10^{-8}$	0	0	8.33	196.65
control3	-13.63	-13.63	$1.04 \cdot 10^{-7}$	$8.41 \cdot 10^{-7}$	$1.92 \cdot 10^{-7}$	0	0	13.65	60.46
control4	-19.79	-19.79	$2.26 \cdot 10^{-8}$	$2.69 \cdot 10^{-7}$	$4.12 \cdot 10^{-7}$	0	0	15.25	157.94
control5	-16.88	-16.88	$9.17 \cdot 10^{-8}$	$3.29 \cdot 10^{-7}$	$1.43 \cdot 10^{-6}$	0	0	16.9	165.86
control6	-37.3	-37.3	$4.78 \cdot 10^{-7}$	$1.02 \cdot 10^{-6}$	$1.35 \cdot 10^{-5}$	0	0	37.31	268.12
control7	-20.63	-20.63	$1.15 \cdot 10^{-7}$	$5.08 \cdot 10^{-7}$	$4.18 \cdot 10^{-6}$	0	0	20.52	133.53
control8	-20.29	-20.29	$1.98 \cdot 10^{-7}$	$9.12 \cdot 10^{-7}$	$2.17 \cdot 10^{-6}$	0	0	18.86	155.41
control9	-14.68	-14.68	$1.09 \cdot 10^{-7}$	$7.11 \cdot 10^{-7}$	$7.3 \cdot 10^{-7}$	0	0	13.84	229.09
control10	-38.53	-38.53	$6.02 \cdot 10^{-7}$	$2 \cdot 10^{-6}$	$9.79 \cdot 10^{-6}$	0	0	38.54	490
control11	-31.96	-31.96	$3.98 \cdot 10^{-7}$	$1.67 \cdot 10^{-6}$	$8.8 \cdot 10^{-6}$	0	0	31.85	387.94

Table 3.5: Statistics of `MySDPSolver` on the `control` family

We observe that `MySDPSolver` is roughly two times faster than `SeDuMi`. This is quite surprising since they implement a very similar algorithm and `SeDuMi` should have a solid implementation. Since these problems consists of two semidefinite blocks of small to medium size, we conjecture that for such sizes, a simpler implementation has less overhead and is thus more efficient. `COSMO` and `ProxSDP` also exhibit a very slow convergence on these problems and fail to converge within the iteration limit.

The gpp problems

These problems are graph-partitioning problems originally also found in [FKN97]. The graph-partitioning problem is a problem in graph theory where one wants to partition the nodes of a weighted graph $G = (V, E)$ into k disjoint subsets such that the total weight of the edges connecting different subsets is minimized. This combinatorial problem is known to be NP-hard. Although based on similar ideas, the derivation of the SDP relaxation is less straightforward than in the case of the max-cut problem. A complete derivation can be found in [WZ99].

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
gpp100	101	100	21	0.269	189	6.993	27	0.735	32	0.679
gpp124-1	125	124	24	0.565	343	20.58	27	1.002	30	0.775
gpp124-2	125	124	20	0.482	236	13.146	26	0.943	33	0.984
gpp124-3	125	124	20	0.47	223	12.847	25	0.937	29	0.812
gpp124-4	125	124	22	0.527	220	14.271	28	1.016	33	0.68
gpp250-1	251	250	22	4.582	428	214.205	29	4.154	31	2.903
gpp250-2	251	250	20	4.228	166	86.063	28	4.027	32	3.241
gpp250-3	251	250	21	4.419	108	55.817	33	4.956	31	2.884
gpp250-4	251	250	22	4.642	168	87.751	29	4.091	30	2.827
gpp500-1	501	500	24	64.238	320	1239.659	28	21.652	30	14.913
gpp500-2	501	500	23	60.716	164	625.31	30	24.669	26	12.461
gpp500-3	501	500	23	61.292	287	1103.755	30	23.92	28	15.858
gpp500-4	501	500	20	52.538	183	706.157	32	25.506	28	16.7
avg	233.59	232.41	21.64	4.09	219.009	87.98	28.53	4.026	30.16	2.911

Table 3.6: Comparison of different solvers performance on the gpp family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
gpp100	44.94	44.94	$1.64 \cdot 10^{-7}$	$2.3 \cdot 10^{-7}$	$8.45 \cdot 10^{-6}$	0	0	47.61	2,388.43
gpp124-1	7.34	7.34	$2.99 \cdot 10^{-7}$	$4.89 \cdot 10^{-7}$	$4.99 \cdot 10^{-6}$	0	0	69.05	7,380.65
gpp124-2	46.86	46.86	$2.09 \cdot 10^{-7}$	$1.22 \cdot 10^{-7}$	$6.38 \cdot 10^{-6}$	0	0	66.2	4,386.83
gpp124-3	153.01	153.01	$1.68 \cdot 10^{-6}$	$4.5 \cdot 10^{-7}$	$2.43 \cdot 10^{-5}$	0	0	60.74	7,251.16
gpp124-4	418.99	418.99	$5.32 \cdot 10^{-6}$	$9.96 \cdot 10^{-7}$	$7.22 \cdot 10^{-6}$	0	0	63.08	5,991.88
gpp250-1	15.44	15.44	$9.87 \cdot 10^{-8}$	$1.73 \cdot 10^{-7}$	$2.81 \cdot 10^{-5}$	0	0	127.17	662.36
gpp250-2	81.87	81.87	$7.28 \cdot 10^{-7}$	$5.03 \cdot 10^{-7}$	$5.15 \cdot 10^{-6}$	0	0	111.49	6,086.27
gpp250-3	303.54	303.54	$3.18 \cdot 10^{-6}$	$9.65 \cdot 10^{-7}$	$1.94 \cdot 10^{-5}$	0	0	105.9	1,993.91
gpp250-4	747.32	747.33	$6.47 \cdot 10^{-6}$	$8.13 \cdot 10^{-7}$	$1.19 \cdot 10^{-4}$	0	0	103.85	1,691.97
gpp500-1	25.32	25.32	$7.69 \cdot 10^{-8}$	$8.3 \cdot 10^{-7}$	$1.22 \cdot 10^{-5}$	0	0	251.36	1,540.01
gpp500-2	156.06	156.06	$1.79 \cdot 10^{-7}$	$9.93 \cdot 10^{-7}$	$3.42 \cdot 10^{-6}$	0	0	199.86	1,419.28
gpp500-3	513.02	513.02	$1.34 \cdot 10^{-6}$	$5.7 \cdot 10^{-7}$	$3.12 \cdot 10^{-6}$	0	0	182.34	3,273.22
gpp500-4	1,567.02	1,567.02	$2.7 \cdot 10^{-6}$	$8.31 \cdot 10^{-7}$	$1.33 \cdot 10^{-5}$	0	0	180.27	771.13

Table 3.7: Statistics of MySDPSolver on the gpp family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
gpp100	275	0.46	4155	6.28	5
gpp124-1	716	2.11	3068	1.94	5
gpp124-2	308	0.76	1889	0.95	4
gpp124-3	375	0.96	2135	1.31	6
gpp124-4	350	0.96	3759	3.78	6
gpp250-1	903	11.84	4509	6.34	5
gpp250-2	411	5.12	4051	5.36	7
gpp250-3	496	6.79	3376	5.41	8
gpp250-4	400	4.93	3204	4.79	8
gpp500-1	2153	132.22	4328	23.86	8
gpp500-2	1154	61.48	4168	18.65	8
gpp500-3	803	42.82	3842	18.52	10
gpp500-4	624	36.93	3731	18.46	10

Table 3.8: Statistics of COSMO and ProxSDP on the gpp family

In this case, we observe a very good performance of ProxSDP. It exhibits solving times close to interior-point methods. We also observe that low-rank solutions are retrieved. We also observe a decent performance of COSMO. However, it appears that both these first-order methods are quite sensitive to the data. Indeed, interior-point methods show a very similar performance with little variance on problems coming from the same family, thus having a very similar structure, differing only by the size or the exact data of the problem. For instance, we observe a substantial difference in the number of iterations of COSMO between gpp500-1 and gpp500-2 while no such difference is to be seen for the interior-point methods. This is likely due to the sensitivity of first-order methods convergence rate to the scaling of the data, as reported in [GCG20]. Indeed, finding an optimal scaling is computationally often very hard, hence real-world algorithms rely on heuristics without any theoretical guarantee. This explains the observed difference.

The `hinf` problems

These linear matrix inequalities were generated as test problems for the `SDPpack` software [Ali+97].

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
hinf1	13	14	43	$1 \cdot 10^{-2}$	27	$1.4 \cdot 10^{-2}$	21	0.166	34	0.242
hinf2	13	16	21	$5 \cdot 10^{-3}$	27	$2.2 \cdot 10^{-2}$	20	0.14	48	0.237
hinf3	13	16	23	$6 \cdot 10^{-3}$	37	$3.4 \cdot 10^{-2}$	19	0.105	37	0.226
hinf4	13	16	24	$6 \cdot 10^{-3}$	70	$4.7 \cdot 10^{-2}$	27	0.2	31	0.285
hinf5	13	16	23	$6 \cdot 10^{-3}$	44	$3.6 \cdot 10^{-2}$	17	$9.936 \cdot 10^{-2}$	24	0.217
hinf6	13	16	36	$9 \cdot 10^{-3}$	43	$3.1 \cdot 10^{-2}$	21	0.139	32	0.222
hinf7	13	16	21	$5 \cdot 10^{-3}$	34	$2.5 \cdot 10^{-2}$	18	$8.8 \cdot 10^{-2}$	23	0.199
hinf8	13	16	25	$7 \cdot 10^{-3}$	39	$3.8 \cdot 10^{-2}$	21	0.177	30	0.225
hinf10	21	18	50	$1.5 \cdot 10^{-2}$	49	$3.5 \cdot 10^{-2}$	24	0.223	36	0.212
hinf11	31	22	50	$2 \cdot 10^{-2}$	39	$3 \cdot 10^{-2}$	25	0.199	38	0.214
hinf12	43	24	50	$2.6 \cdot 10^{-2}$	37	$3.3 \cdot 10^{-2}$	32	0.14	59	0.307
hinf13	57	30	50	$4.3 \cdot 10^{-2}$	46	$6.6 \cdot 10^{-2}$	14	0.157	34	0.254
hinf14	73	34	50	$5.7 \cdot 10^{-2}$	72	0.126	23	0.31	33	0.216
hinf15	91	37	50	$8.3 \cdot 10^{-2}$	66	0.141	16	0.183	35	0.23
avg	22.52	19.72	34.5	$1.3 \cdot 10^{-2}$	42.97	$3.9 \cdot 10^{-2}$	20.82	0.15	34.31	0.23

Table 3.9: Comparison of different solvers performance on the `hinf` family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
hinf1	-2.03	-2.03	$5.39 \cdot 10^{-8}$	$7.11 \cdot 10^{-7}$	$1.92 \cdot 10^{-7}$	0	0	3.5	18,667.85
hinf2	-10.97	-10.97	$9.58 \cdot 10^{-8}$	$7.96 \cdot 10^{-7}$	$1.06 \cdot 10^{-6}$	0	0	325.33	1,042.24
hinf3	-56.95	-56.95	$3.24 \cdot 10^{-8}$	$6.03 \cdot 10^{-9}$	$1.05 \cdot 10^{-5}$	0	0	636.89	4,851.61
hinf4	-274.76	-274.76	$2.53 \cdot 10^{-6}$	$6.09 \cdot 10^{-7}$	$3.36 \cdot 10^{-7}$	0	0	43.17	$1.22 \cdot 10^{5}$
hinf5	-362.42	-362.32	$2.72 \cdot 10^{-6}$	$1.07 \cdot 10^{-7}$	$3.27 \cdot 10^{-4}$	0	0	2,046.2	12,901.98
hinf6	-448.95	-448.94	$1.18 \cdot 10^{-6}$	$3.72 \cdot 10^{-8}$	$3.2 \cdot 10^{-4}$	0	0	4,054.59	$1.99 \cdot 10^{5}$
hinf7	-390.83	-390.82	$2.58 \cdot 10^{-6}$	$1.67 \cdot 10^{-7}$	$1.69 \cdot 10^{-4}$	0	0	26,640.69	1,868.57
hinf8	-116.17	-116.16	$1.35 \cdot 10^{-6}$	$4.84 \cdot 10^{-7}$	$1.38 \cdot 10^{-4}$	0	0	741.85	13,788.4
hinf10	-108.82	-108.77	$2.32 \cdot 10^{-6}$	$4.26 \cdot 10^{-7}$	$3.9 \cdot 10^{-7}$	$5.69 \cdot 10^{-16}$	0	15.27	$1.41 \cdot 10^{9}$
hinf11	-65.93	-65.89	$1.99 \cdot 10^{-6}$	$5.66 \cdot 10^{-7}$	$1.34 \cdot 10^{-6}$	0	0	8.98	$1.47 \cdot 10^{7}$
hinf12	$-8.49 \cdot 10^{-3}$	$-6.96 \cdot 10^{-3}$	$1.82 \cdot 10^{-4}$	$1.26 \cdot 10^{-4}$	$2.06 \cdot 10^{-10}$	0	0	0.84	$2.36 \cdot 10^{9}$
hinf13	-44.51	-44.58	$6.05 \cdot 10^{-2}$	$3.58 \cdot 10^{-2}$	$1.89 \cdot 10^{-4}$	0	0	3,982.86	$1.49 \cdot 10^{6}$
hinf14	-12.99	-12.99	$7.33 \cdot 10^{-4}$	$5.85 \cdot 10^{-3}$	$2.36 \cdot 10^{-6}$	0	0	304.26	$8.28 \cdot 10^{8}$
hinf15	-24.13	-24.07	$5.72 \cdot 10^{-3}$	$6.4 \cdot 10^{-3}$	$1.02 \cdot 10^{-4}$	0	0	6,809.38	$1.29 \cdot 10^{6}$

Table 3.10: Statistics of `MySDPSolver` on the `hinf` family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
hinf1	621	$3.8 \cdot 10^{-2}$	1000000	91.05	5
hinf2	7639	0.49	32820	3.35	5
hinf3	2356	0.15	443986	45.88	5
hinf4	886	$5.7 \cdot 10^{-2}$	640862	65.57	5
hinf5	12187	0.75	-	-	-
hinf6	175321	10.51	-	-	-
hinf7	7834	0.47	-	-	-
hinf8	2525	0.16	682802	71.04	5
hinf10	2800	0.2	152391	17.15	7
hinf11	17728	1.55	119542	14.92	9
hinf12	89204	8.61	86066	11.5	10
hinf13	114525	14.95	-	-	-
hinf14	5850	0.85	67076	12.01	15
hinf15	-	-	-	-	-

Table 3.11: Statistics of COSMO and ProxSDP on the hinf family

These problem are found to be very badly conditioned. Indeed, Mosek and SeDuMi often issue warnings about running into numerical errors. The particular problem hinf9 had to be removed. We observe that although COSMO managed to solve all but one problems, ProxSDP fails to converge for five. The interior-point methods manage to solve all of them, however we observe much worse statistics for MySDPSolver than for other families of problems, hinting at the fact that these problems cause numerical errors. We also observe, as for some other families of problems, that Mosek seems slower for very small problems. It can be due to some preprocessing routines absent from other solvers, which cause a certain overhead for smaller problems, but give Mosek a substantial advantage on bigger problems.

The `truss` problems

These optimal truss topology design are also taken from [Ali+97].

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
truss1	6	13	13	$3 \cdot 10^{-3}$	14	$1.1 \cdot 10^{-2}$	14	0.105	8	0.197
truss2	58	133	19	$5.4 \cdot 10^{-2}$	33	0.208	22	0.517	16	0.198
truss3	27	31	15	$8 \cdot 10^{-3}$	16	$1.9 \cdot 10^{-2}$	17	0.117	12	0.167
truss4	12	19	13	$4 \cdot 10^{-3}$	14	$1.1 \cdot 10^{-2}$	16	$9.19 \cdot 10^{-2}$	10	0.126
truss5	208	331	22	0.399	38	1.015	25	0.716	14	0.233
truss6	172	451	30	0.894	52	2.157	27	4.667	47	0.366
truss7	86	301	29	0.421	33	0.786	25	5.824	23	0.179
truss8	496	628	24	2.442	46	9.37	26	2.411	15	0.712
avg	60.18	114.11	19.63	$8 \cdot 10^{-2}$	27.32	0.22	20.92	0.63	15.58	0.23

Table 3.12: Comparison of different solvers performance on the `truss` family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
truss1	9	9	$2.89 \cdot 10^{-8}$	$3.05 \cdot 10^{-7}$	$5.81 \cdot 10^{-10}$	0	0	11.69	14.94
truss2	123.38	123.38	$2.96 \cdot 10^{-7}$	$7.2 \cdot 10^{-7}$	$1.47 \cdot 10^{-8}$	0	0	179.26	154.43
truss3	9.11	9.11	$2.11 \cdot 10^{-8}$	$5.37 \cdot 10^{-7}$	$7.35 \cdot 10^{-13}$	0	0	15	16.06
truss4	9.01	9.01	$2.77 \cdot 10^{-8}$	$3.66 \cdot 10^{-7}$	$9.18 \cdot 10^{-10}$	0	0	12.27	14.97
truss5	132.64	132.64	$2.62 \cdot 10^{-8}$	$5.54 \cdot 10^{-7}$	$1.29 \cdot 10^{-10}$	0	0	219.89	181.01
truss6	901	901	$2.38 \cdot 10^{-7}$	$3.69 \cdot 10^{-7}$	$9.5 \cdot 10^{-7}$	0	0	944.85	3,252.28
truss7	900	900	$5.07 \cdot 10^{-7}$	$6.35 \cdot 10^{-7}$	$4.89 \cdot 10^{-7}$	0	0	940.83	3,238.34
truss8	133.11	133.11	$4.89 \cdot 10^{-9}$	$2.14 \cdot 10^{-7}$	$4.03 \cdot 10^{-10}$	0	0	257.38	181.61

Table 3.13: Statistics of `MySDPSolver` on the `truss` family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
truss1	123	$6 \cdot 10^{-3}$	418	$3 \cdot 10^{-2}$	1
truss2	6932	2.64	193230	124.93	4
truss3	1419	0.14	5027	0.77	3
truss4	267	$2.1 \cdot 10^{-2}$	1501	0.18	2
truss5	37076	39.7	599281	728.51	68
truss6	91325	135	-	-	-
truss7	153335	34.4	-	-	-
truss8	39625	110.56	-	-	-

Table 3.14: Statistics of `COSMO` and `ProxSDP` on the `truss` family

We observe that `COSMO` manages to solve all the problems, although the timings for the bigger problems are quite slow compared to interior-point methods. We also observe that `ProxSDP` solves the first four problems rapidly, is very slow on `truss5` and fails to converge for the last ones. This can be accounted for by noticing that while the rank of the first four solutions is

low (meaning that only a truncated eigendecomposition needed to be computed), the latter problems exhibit a much higher rank of the solution hence are much harder for `ProxSDP`.

The `theta` problems

These problems come from [Bor99a]. They consist in computing the *Lovász number* of a graph (also called the *Lovász θ* function). The Lovász number $\theta(G)$ is a real number associated with a given graph G such that

$$\omega(G) \leq \theta(G) \leq \chi(G)$$

where $\omega(G)$ is the clique number of G and $\chi(G$ is its chromatic number. While both of these are NP-complete to compute, $\theta(G)$ is formulated as a SDP and can be solved in polynomial time providing useful respectively upper and lower bounds. In the case of *perfect graphs*, $\omega(G)$ and $\xi(G)$ are equal, they can be computed by solving the SDP for $\theta(G)$ in polynomial time.

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
theta1	104	50	14	$5.6 \cdot 10^{-2}$	16	0.813	21	0.385	12	0.191
theta2	498	100	15	0.729	17	2.285	22	1.052	13	0.319
theta3	1106	150	16	4.066	17	15.054	22	4.449	11	0.827
theta4	1949	200	16	13.481	17	55.745	22	23.529	11	2.277
theta5	3028	250	16	40.041	18	192.496	25	111.295	10	4.984
avg	805	130.25	15.37	2.45	16.98	12.45	22.36	5.42	11.35	0.89

Table 3.15: Comparison of different solvers performance on the `theta` family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
theta1	-23	-23	$1.35 \cdot 10^{-7}$	$8.29 \cdot 10^{-7}$	$1.89 \cdot 10^{-15}$	0	0	0.88	202.89
theta2	-32.88	-32.88	$1.46 \cdot 10^{-7}$	$8.43 \cdot 10^{-7}$	$5.83 \cdot 10^{-15}$	0	0	0.46	410.07
theta3	-42.17	-42.17	$2.47 \cdot 10^{-8}$	$1.84 \cdot 10^{-7}$	$1.85 \cdot 10^{-14}$	0	0	0.38	613.26
theta4	-50.32	-50.32	$6.68 \cdot 10^{-8}$	$4.92 \cdot 10^{-7}$	$1.51 \cdot 10^{-15}$	0	0	0.34	846.33
theta5	-57.23	-57.23	$6.77 \cdot 10^{-8}$	$4.77 \cdot 10^{-7}$	$3.16 \cdot 10^{-15}$	0	0	0.3	1,068.33

Table 3.16: Statistics of `MySDPSolver` on the `theta` family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
theta1	227	0.11	12261	5.58	2
theta2	604	0.88	7720	13.56	2
theta3	887	2.87	29534	98.83	129
theta4	561	3.84	-	-	-
theta5	1203	11.18	-	-	-

Table 3.17: Statistics of `COSMO` and `ProxSDP` on the `theta` family

We can draw similar conclusions as before concerning `ProxSDP`. We also observe a very good performance of `COSMO`.

The qap problems

These SDP relaxations of *quadratic assignment problems* come from [FKN97]. Given a set of locations and facilities of equal size and a cost function and a distance function, one wants to find the optimal assignement such that the total weighted distance is minimized:

$$\min \sum_{a,b \in P} c(a,b) \cdot d(f(a), f(b))$$

It is a fundamental problem in combinatorial optimization (the *travelling salesman problem* can be seen as a special case) but is known to be NP-hard, hence SDP relaxations are used.

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
qap5	136	26	14	$4.3 \cdot 10^{-2}$	17	1.103	14	0.177	8	0.219
qap6	229	37	18	$8.1 \cdot 10^{-2}$	43	0.452	24	0.421	23	0.327
qap7	358	50	19	0.23	176	5.163	25	0.736	27	0.426
qap8	529	65	19	0.504	167	11.315	25	1.235	26	0.631
qap9	748	82	19	1.056	237	41.154	28	2.822	22	0.81
qap10	1021	101	19	2.241	180	60.56	24	4.359	22	1.462
avg	406.38	54.39	17.89	0.31	98.55	6.45	22.817	0.97	19.92	0.53

Table 3.18: Comparison of different solvers performance on the qap family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
qap5	436	436	$1.78 \cdot 10^{-4}$	$7.52 \cdot 10^{-7}$	$4.82 \cdot 10^{-6}$	0	0	6	3,532.47
qap6	381.44	381.43	$7.85 \cdot 10^{-4}$	$7.69 \cdot 10^{-7}$	$2.09 \cdot 10^{-6}$	0	0	4.39	$1.81 \cdot 10^{5}$
qap7	424.84	424.81	$8.05 \cdot 10^{-4}$	$7.13 \cdot 10^{-7}$	$3.24 \cdot 10^{-6}$	0	0	5.1	$1.93 \cdot 10^{5}$
qap8	756.99	756.93	$1.16 \cdot 10^{-3}$	$9.05 \cdot 10^{-7}$	$2.14 \cdot 10^{-6}$	0	0	3.79	$4.21 \cdot 10^{5}$
qap9	1,410.05	1,409.89	$2.24 \cdot 10^{-3}$	$6.48 \cdot 10^{-7}$	$1.25 \cdot 10^{-5}$	0	0	4.15	$2.47 \cdot 10^{5}$
qap10	1,092.69	1,092.58	$1.28 \cdot 10^{-3}$	$5.79 \cdot 10^{-7}$	$4.22 \cdot 10^{-6}$	0	0	3.49	$5.23 \cdot 10^{5}$

Table 3.19: Statistics of MySDPSolver on the qap family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
qap5	210	$3.2 \cdot 10^{-2}$	20513	3.4	2
qap6	275	$7.9 \cdot 10^{-2}$	1000000	263.57	2
qap7	300	0.13	42892	18.88	2
qap8	351	0.25	42892	19.55	2
qap9	228	0.24	142803	165.7	65
qap10	228	0.37	-	-	-

Table 3.20: Statistics of COSMO and ProxSDP on the qap family

We observe a very good performance of COSMO. We also observe a similar phenomenon as for the previous family for ProxSDP, when the solving seems to be conditioned by the rank of the solution.

The mcp problems

These max-cut problems are from [Fuj+99].

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
mcp100	100	100	14	0.189	13	0.971	19	0.432	11	0.312
mcp124-1	124	124	15	0.335	14	0.692	20	0.54	12	0.357
mcp124-2	124	124	14	0.301	15	0.729	20	0.537	12	0.374
mcp124-3	124	124	15	0.313	14	0.715	20	0.594	11	0.334
mcp124-4	124	124	15	0.313	14	0.721	21	0.578	12	0.322
mcp250-1	250	250	16	3.135	16	8.323	22	2.544	12	0.657
mcp250-2	250	250	15	2.902	17	9.092	21	2.398	12	0.696
mcp250-3	250	250	15	2.921	16	8.693	21	2.407	11	0.625
mcp250-4	250	250	16	3.126	16	8.828	22	2.541	12	0.615
mcp500-1	500	500	17	41.078	19	78.998	23	13.868	13	2.842
mcp500-2	500	500	17	41.409	17	71.009	23	13.984	13	2.984
mcp500-3	500	500	16	38.196	20	83.288	23	14.15	12	2.713
mcp500-4	500	500	16	38.144	19	78.025	22	13.371	13	2.95
avg	232.41	232.41	15.43	2.68	16.01	6.68	21.26	2.32	11.98	0.79

Table 3.21: Comparison of different solvers performance on the mcp family

name	pobj	dobj	μ	relgap	$\|Ax-b\|_2$	$\|x_-\|_2$	$\|s_-\|_2$	$\|x\|_2$	$\|y\|_2$
mcp100	−226.16	−226.16	$1.12 \cdot 10^{-7}$	$2.75 \cdot 10^{-7}$	$3.74 \cdot 10^{-14}$	0	0	49.89	24.25
mcp124-1	−141.99	−141.99	$5.77 \cdot 10^{-8}$	$2.8 \cdot 10^{-7}$	$2.8 \cdot 10^{-14}$	0	0	64.95	15.13
mcp124-2	−269.88	−269.88	$4.18 \cdot 10^{-7}$	$9.58 \cdot 10^{-7}$	$1.1 \cdot 10^{-14}$	0	0	63.24	26.12
mcp124-3	−467.75	−467.75	$2.21 \cdot 10^{-7}$	$3.21 \cdot 10^{-7}$	$2.88 \cdot 10^{-14}$	0	0	60.51	43.93
mcp124-4	−864.41	−864.41	$2.15 \cdot 10^{-7}$	$1.62 \cdot 10^{-7}$	$2.48 \cdot 10^{-14}$	0	0	65.34	78.88
mcp250-1	−317.26	−317.26	$1.21 \cdot 10^{-7}$	$4.93 \cdot 10^{-7}$	$3.2 \cdot 10^{-14}$	0	0	132.08	23.21
mcp250-2	−531.93	−531.93	$3.41 \cdot 10^{-7}$	$9.17 \cdot 10^{-7}$	$2.33 \cdot 10^{-14}$	0	0	108.75	36.06
mcp250-3	−981.17	−981.17	$4.99 \cdot 10^{-7}$	$7.06 \cdot 10^{-7}$	$3.26 \cdot 10^{-14}$	0	0	103.98	64.39
mcp250-4	−1,681.96	−1,681.96	$2.18 \cdot 10^{-7}$	$1.78 \cdot 10^{-7}$	$1.74 \cdot 10^{-13}$	0	0	110.12	108.43
mcp500-1	−598.15	−598.15	$1.31 \cdot 10^{-7}$	$5.88 \cdot 10^{-7}$	$8.23 \cdot 10^{-14}$	0	0	225.43	31.74
mcp500-2	−1,070.06	−1,070.06	$1.97 \cdot 10^{-7}$	$4.21 \cdot 10^{-7}$	$2.03 \cdot 10^{-13}$	0	0	209.89	52.18
mcp500-3	−1,847.97	−1,847.97	$5.83 \cdot 10^{-7}$	$8.66 \cdot 10^{-7}$	$1.15 \cdot 10^{-13}$	0	0	191.19	86.15
mcp500-4	−3,566.74	−3,566.74	$1.33 \cdot 10^{-6}$	$9.31 \cdot 10^{-7}$	$3.85 \cdot 10^{-13}$	0	0	185.39	162.61

Table 3.22: Statistics of MySDPSolver on the mcp family

	COSMO		ProxSDP		
name	iters	t	iters	t	rank
mcp100	180	0.26	1368	2.17	94
mcp124-1	1079	2.71	8010	27.85	107
mcp124-2	252	0.55	8615	30.59	118
mcp124-3	125	0.26	6522	25.16	118
mcp124-4	150	0.34	5650	22.41	119
mcp250-1	1013	11.15	28768	310.89	225
mcp250-2	431	4.08	8179	85.68	241
mcp250-3	301	2.92	7576	80.83	242
mcp250-4	267	2.58	6490	73.58	241
mcp500-1	1921	109.59	-	-	-
mcp500-2	828	42.39	-	-	-
mcp500-3	678	34.29	-	-	-
mcp500-4	400	22.18	-	-	-

Table 3.23: Statistics of COSMO and ProxSDP on the mcp family

Once again, COSMO manages to solve all the problems in reasonable time. We also observe much worse behaviour of ProxSDP. We see that it fails in recovering low-rank solutions, which is quite interesting giving the formulation of the problem.

3.2 Additional tests

In this subsection, some more specific tests will be performed.

3.2.1 The influence of precision on the number of iterations for first-order methods

Obtaining a high degree of precision using interior-point methods is quite straightforward: once the quadratic convergence region is reached, only a few iterations are needed to increase the number of significant digits. This is not the case for first-order methods, where reaching a solution with allow level of precision might be quite easy, but improving precision is extremely hard and subject some undesired phenomena, such as oscillations. More specifically, we will compare how increasing the desired precision for respectively COSMO and ProxSDP influences the number of iterations. We shall pick families of problems where these solvers performed well already, namely the gpp family for ProxSDP and the qap family for COSMO.

	$1e-5$		$1e-7$		$1e-9$		$1e-12$	
name	iters	t	iters	t	iters	iters	iters	t
qap5	210	$3.5 \cdot 10^{-2}$	289	$4.5 \cdot 10^{-2}$	-	-	-	-
qap6	275	$1.53 \cdot 10^{-1}$	-	-	-	-	-	-
qap7	300	$2.02 \cdot 10^{-1}$	-	-	-	-	-	-
qap8	351	$2.78 \cdot 10^{-1}$	-	-	-	-	-	-
qap9	228	$2.46 \cdot 10^{-1}$	$9.58 \cdot 10^{5}$	$1.15 \cdot 10^{3}$	-	-	-	-
qap10	228	$3.53 \cdot 10^{-1}$	-	-	-	-	-	-

Table 3.24: Comparison of the number of iterations changing with precision for COSMO

We observe that COSMO really struggles when higher precision is needed. Indeed, when the wanted precision was increased, only two problems could be solved within the given iteration limit (10^6). We also observe a very big difference in the number of iterations and the timings. This is probably due to the sensitivity to the scaling of the data. This seems to be a common behaviour for first-order methods, as it reaches a solution with lower precision very quickly, but obtaining further digits is subject to oscillations and very slow convergence.

	$1e-3$		$1e-5$		$1e-7$		$1e-9$	
name	iters	t	iters	t	iters	t	iters	iters
gpp100	4155	$6.28 \cdot 10^{0}$	95,186	$1.55 \cdot 10^{2}$	33,351	$4.95 \cdot 10^{1}$	-	-
gpp124-1	3068	$1.94 \cdot 10^{0}$	5,062	$1.33 \cdot 10^{1}$	16,023	$5.81 \cdot 10^{1}$	-	-
gpp250-1	4509	$6.34 \cdot 10^{0}$	7,810	$4.09 \cdot 10^{1}$	117.66	3,965	-	-
gpp500-4	3731	$1.85 \cdot 10^{1}$	4,745	$2.3 \cdot 10^{1}$	30,134	$3.6 \cdot 10^{2}$	-	-

Table 3.25: Comparison of the number of iterations changing with precision for ProxSDP

These results seem to corroborate the expected evolution of the number of iterations with increasing precision. They also to justify the use of first-order methods in SDP in applications when an approximate solution is needed.

3.2.2 Comparison of the performance of SeDuMi and MySDPSolver

The relative performance of MySDPSolver and SeDuMi seems to depend on the size of the biggest semidefinite block. While SeDuMi seems to be slower on problems with small to medium-sized blocks, it tends to outperform MySDPSolver instances with larger blocks.

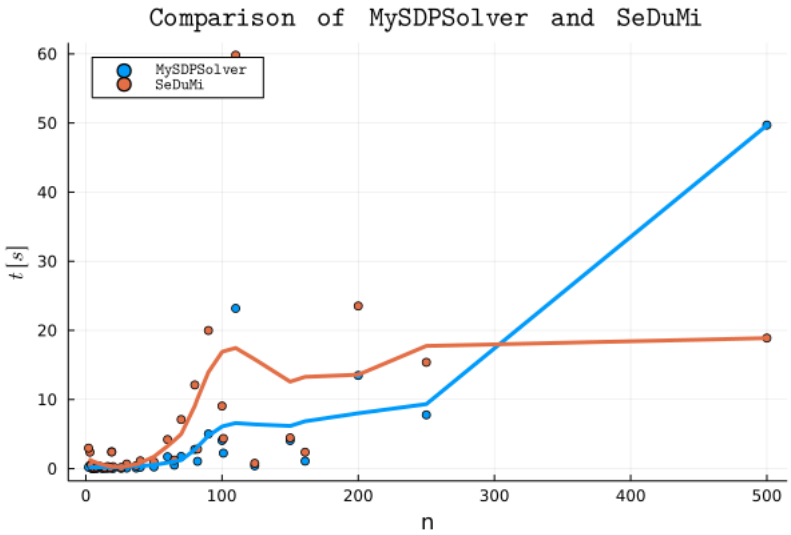

Figure 3.2: Mean solution time as a function of the biggest semidefinite block. The smoothed curves were obtained using local regression.

We observe that `MySDPSolver` is noticeably faster on problems with the biggest SDP block being of the size up to 100-by-100. We then observe a slight advantage of `MySDPSolver`, however these results are more prone to error due to insufficient number of data points. We finally see that `SeDuMi` overtakes `MySDPSolver` for n around 300. This is probably due to a better exploitation of sparsity. However, this fact suggest that such an implementation has a certain overhead for smaller or less sparse matrices. `MySDPSolver` can thus be useful for problems with such a structure.

3.3 Remarks on the results

Let us discuss these results more globally. First, we observe a very solid performance of `Mosek` clearly winning these benchmarks. This is hardly surprising since it is a highly optimized code meant specifically for industrial applications and is often considered state-of-the-art in the domain of SDP solvers. Moreover, it is the only existing commercial code for semidefinite programming. However, we observe that `MySDPSolver` and `SeDuMi` do approach its performance for a lot of problems and are not orders of magnitude slower, which goes to show that, unless the speed and reliability are crucial for their application, the users might consider using open source solvers as a first approach, especially considering the high price of a `Mosek` licence.

Secondly, let us discuss the performance of `Hypatia`. Globally, we observed much slower timings and higher iterations counts than in the case of the other SDP solvers. This could be due to a couple of reasons. One of them is the fact that `Hypatia` is a new project, meaning that it does not have a huge user base yet, hence some bugs are yet to be caught. In particular, minor problems with the integration with `JuMP` seem to persist. Since the test programs are loaded through `JuMP`, it can causes the observed worse performance. In most cases, the high iteration

count is due to the solver not stopping, although it seems that a solution has been found. This might suggest some problems with the stopping criterion. Finally, as `Hypatia` is meant specifically for exotic cones, comparing it with specialized SDP solvers could be penalizing.

Finally, the relative performance of `MySDPSolver` and `SeDuMi` seems to depend on the size of the problem. While `SeDuMi` seems to be slower on small to medium-sized problems, it outperforms `MySDPSolver` on larger instances.

Chapter 4

Application to the PEP problem

In this chapter, a short introduction to the *performance estimation problem* will be followed by some tests on problems of this class.

4.1 Introduction to the PEP problem

The *performance estimation problem* (PEP) arises in the performance estimation framework [THG17]. It consists in numerically assessing the worst-case performance of a given first-order optimization method. Indeed, the standard theoretical bounds for these methods are often pessimistic, hence the user might want to compute tight non-improvable guarantees. The formulation of the PEP problem stems from the observation that finding the worst behaviour is itself an optimization problem. After some reformulation it can be cast as SDP.

Let us illustrate the methodology on a simple gradient method with fixed step-size on the class of L-smooth convex functions. A convex function is L-smooth if

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \quad \forall\, x, y$$

Let us remind that the gradient method with step-size $h \in]0, 2[$ is given by

Given $f \in \mathcal{F}_{0,L}(\mathbb{R}^d)$, $x_0 \in \mathbb{R}^d$

for $i = 0 : N - 1$ **do**
$x_{i+1} = x_i - \frac{h}{L}\nabla f(x_i)$
end for

Algorithm 3: Gradient method with fixed step-size

We then choose a performance measure such as the objective function accuracy $f(x_N) - f(x_*)$, the norm of the gradient $\|\nabla f(x_N)\|$ or the distance to the optimum $\|x_N - x_*\|$. We would like to find the worst-case performance, hence find a function $f \in \mathcal{F}_{0,L}(\mathbb{R}^d)$ maximizing the performance criterion. Obviously, this is an infinite-dimensional problem. However, using the fact that the method will evaluate f and its gradient at a finite number of iterates, we can reformulate the problem using the values of the function and the gradient at the iterates, making the problem finite-dimensional. This is possible using the theory of *convex interpolation* [Tay17]. Adding these interpolations constraints yields a quadratically constrained quadratic program, which is NP-hard to solve in general. In order to tackle this issue, a reformulation using a suitably defined Gram matrix similar to that in Example 1.1.1 is performed. After dropping a rank constraint, we obtain a semidefinite program (SDP-PEP).

The SDP-PEP has a particular structure with one main semidefinite block called G belonging to $\mathbb{R}^{(N+2)\times(N+2)}$ and smaller blocks of size one. Let $g_i \ \forall \ i = 0...N$ denote the gradient of f evaluated at the iterates and x_0 the initial iterates. G is precisely the Gram matrix given by $\begin{pmatrix} x_0 & g_1 \dots g_N \end{pmatrix} \begin{pmatrix} x_0 & g_1 \dots g_N \end{pmatrix}^T$.

4.2 Numerical results

Let us now test the same solvers on some generated PEP problem instances. For each number of iterates N, three different values of h will be chosen, namely $h_1 = 0.5$, $h_2 = 1$ and $h_3 = 1.5$. Since the size of this problem increases with N, we expect to see a big difference in solving times. The choices for N are: 20, 50, 75, 100.

			MySDPSolver		Hypatia		SeDuMi		Mosek	
name	m	n	iters	t	iters	t	iters	t	iters	t
gradLS20-1	299	23	32	0.184	26	3.129	23	0.781	7	0.17
gradLS20-2	299	23	34	0.198	30	3.604	29	0.6	7	0.16
gradLS20-3	299	23	35	0.2	33	3.921	30	0.6	9	$8 \cdot 10^{-2}$
gradLS50-1	1484	53	55	14.367	37	152.469	30	15.4	9	0.96
gradLS50-2	1484	53	50	10.213	43	174.586	36	17.9	12	1.1
gradLS50-3	1484	53	54	10.891	46	184.562	40	20.4	12	1.08
gradLS75-1	3159	78	61	96.523	45	974.717	37	220	11	5.43
gradLS75-2	3159	78	57	86.856	54	1092.252	43	245.2	12	6.05
gradLS75-3	3159	78	62	93.406	57	1148.358	47	268.6	13	6.23
gradLS100-1	5459	103	77	528.577	-	-	43	1614.9	12	22.31
gradLS100-2	5459	103	67	449.707	-	-	45	1663	14	24.08
gradLS100-3	5459	103	68	458.153	-	-	52	2177.6	17	29.58

Table 4.1: Comparison of different solvers performance on the PEP problem

Several observations can be made. First, we observe that the number of iterations N has

a great impact on the solving time. This is due to the size of the PEP problem growing with N. Then, we observe a interesting behaviour with varying h. Indeed, `MySDPSolver` seems to be the fastest for $h = 1$ and the slowest for $h = 0.5$. This is different from `SeDuMi`, `Hypatia` (for the problems it managed to solve) and `Mosek`, which all seem to be the fastest for $h = 0.5$ and the slowest for $h = 1.5$, hence the solving time seems to increase with h.

We can compare the obtained results with the worst-case bound given by the theory as

$$\frac{1}{2}\max\left(\frac{1}{2Nh+1}, (1-h)^{2N}\right) \tag{4.1}$$

name	Mosek	MySDPSolver	theoretical value	absolute error
gradLS20-1	0.02381	0.02381	0.0238	$9.5 \cdot 10^{-6}$
gradLS20-2	0.012195	0.012195	0.0122	$4.9 \cdot 10^{-6}$
gradLS20-3	0.008197	0.008197	0.0082	$3.28 \cdot 10^{-6}$
gradLS50-1	0.009804	0.009808	0.0098	$3.92 \cdot 10^{-6}$
gradLS50-2	0.00495	0.00495	0.005	$4.95 \cdot 10^{-5}$
gradLS50-3	0.003311	0.003311	0.0033	$1.13 \cdot 10^{-5}$
gradLS75-1	0.006579	0.006579	0.0066	$2.11 \cdot 10^{-5}$
gradLS75-2	0.003311	0.003311	0.0033	$1.13 \cdot 10^{-5}$
gradLS75-3	0.002212	0.002212	0.0022	$1.24 \cdot 10^{-5}$
gradLS100-1	0.004951	0.004951	0.005	$4.95 \cdot 10^{-5}$
gradLS100-2	0.002488	0.002488	0.0025	$1.25 \cdot 10^{-5}$
gradLS100-3	0.001661	0.001661	0.0017	$3.89 \cdot 10^{-5}$

Table 4.2: Comparison of the obtained bounds with the theory

`Mosek` and `SeDuMi` were found to give the same first five digits for all problems, hence only `Mosek` is featured. Very close solutions were also obtained using `MySDPSolver`. We observe a certain discrepancy between the theoretical results and the computed values.

As expected, `Mosek` exhibits a very good performance. We observe a big performance difference between `SeDuMi` and `MySDPSolver`. Indeed, `MySDPSolver` is much faster on these problems. It is not strictly due to a major algorithmic difference but rather a very penalizing problem structure for `SeDuMi`. Indeed, `SeDuMi` has distinctive routines coded in `C` when dealing with SDP blocks, opposed to linear blocks which are processed in `Matlab` directly. This means that since there are many semidefinite blocks of size one, `SeDuMi` will call the .mex files each times. If the blocks are big, this is advantageous since the `BLAS` and `LAPACK` routines will be called directly and make use of the multicore processor. The same goes for linear blocks, whose processing will also be multithreaded natively in `Matlab`. However, calling the .mex files for a lot of blocks of size one will result in the loss of the multithreading, since each block has one element and will be processed on exactly one core sequentially. This means that all the computations are done using only one core. `MySDPSolver` has a simple routine which detects such a scenario and reformulates the problem. Finally, we observe that `Hypatia` hits the time-limit (one hour) set

for the last four problems. It seems that in this case, the time it takes `Hypatia` to perform one iteration is much bigger than for the other solvers.

Chapter 5

Conclusion

5.1 Reflections

The major goal of this work was to develop a fully-functioning interior-point semidefinite solver in the `Julia` programming language and compare it with other available options. In particular, the benchmarks were performed against (i) well-established `Matlab`-oriented solvers (`SeDuMi` and `Mosek`), (ii) a novel interior-point solver `Hypatia` written purely in `Julia` and (iii) two first-order solvers also native to `Julia` (`ProxSPD` and `COSMO`). Both the implementation phase and the benchmarks allow to assess the fitness of `Julia` for writing optimization solvers as well as the current state of `Julia` software meant for semidefinite programming.

The implementation phase seems to suggest that `Julia` has great potential for developing new solvers. Indeed, it provides the access to high-performance libraries critical for scientific applications, as well as convenient tools to structurize a solver project (`MathOptInterface`). The emergence of new research solvers seems to support this thesis. However, some questionable choices in the `Julia` philosophy, such as a certain lack of documentation, might hinder its potential by making the learning curve unnecessarily steep.

The benchmark results seem to indicate that even a relatively simple implementation can have a decent performance, in some cases similar or better than existing solvers.

5.2 Improvements

Some improvements can of course be made. One major direction of improvement seems to be an implementation which takes sparsity patterns of the variables into account. Indeed, it

was observed that `MySDPSolver` seems to be much slower on problems with such a structure compared to solvers exploiting it. This also can make the solver more memory-efficient, which seems to be a limitation: indeed, some encountered problems could not be solved because the computations required too much memory, in contrast with solvers like `SeDuMi` and especially `Mosek`, where the memory management is very efficient. However, no major robustness problems were encountered, which seems to validate the algorithm. As the solver will be made public, making it more fit for big problems seems to be a logical focus.

Bibliography

Introduction

[WSV00] Henry Wolkowicz, Romesh Seigal, and Lieven Vandenberghe. "Introduction". In: *Handbook of Semidefinite Programming*. Springer, 2000. Chap. 1, pp. 1–7.

Applications of SDP

[ApS21] MOSEK ApS. *The MOSEK Modeling Cookbook*. 2021. URL: https://docs.mosek.com/MOSEKModelingCookbook-letter.pdf.

[CD17] Augustin Cosse and Laurent Demanet. *Stable rank one matrix completion is solved by two rounds of semidefinite programming relaxation*. 2017. arXiv: 1801.00368 [math.NA].

[De 07] Tijl De Bie. "Deploying SDP for machine learning". eng. In: *European Symposium on Artificial Neural Networks, Proceedings*. Bruges, Belgium: d-side publishing, 2007, pp. 205–210. ISBN: 2-930307-07-2.

[GR00] Michel Goemans and Franz Rendl. "Combinatorial Optimization". In: *Handbook of Semidefinite Programming*. Springer, 2000. Chap. 12, pp. 343–359.

[GW95] Michel X. Goemans and David P. Williamson. "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *J. ACM* 42.6 (Nov. 1995), pp. 1115–1145. ISSN: 0004-5411. DOI: 10.1145/227683.227684. URL: https://doi.org/10.1145/227683.227684.

[KRT99] Etienne Klerk, Cornelis Roos, and Tamás Terlaky. "A Short Survey on Semidefinite Programming". In: (Sept. 1999).

[LO99] Claude Lemaréchal and François Oustry. *Semidefinite Relaxations and Lagrangian Duality with Application to Combinatorial Optimization*. Research Report RR-3710. INRIA, 1999. URL: https://hal.inria.fr/inria-00072958.

Brief presentation of the Julia programming language

[Bez+12] Jeff Bezanson et al. "Julia: A Fast Dynamic Language for Technical Computing". In: *MIT* (Sept. 2012).

[Com21] Julia Computing. *Julia Computing Case study*. https://juliacomputing.com/case-studies/. Accessed: 2021-07-05. 2021.

[Mit21] Hans Mittelmann. *The SDP problem File format*. http://plato.asu.edu/ftp/sdpa_format.txt. Accessed: 2021-07-05. 2021.

JuMP modelling language

[DHL17] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (Jan. 2017), pp. 295–320. ISSN: 1095-7200. DOI: 10.1137/15m1020575. URL: http://dx.doi.org/10.1137/15M1020575.

[Leg+20] Benoit Legat et al. *MathOptInterface: a data structure for mathematical optimization problems*. 2020. arXiv: 2002.03447 [math.OC].

[Ude+14] Madeleine Udell et al. *Convex Optimization in Julia*. 2014. arXiv: 1410.4821 [math.OC].

Brief survey of available solvers

[Boy+11] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Found. Trends Mach. Learn.* 3.1 (Jan. 2011), pp. 1–122. ISSN: 1935-8237. DOI: 10.1561/2200000016. URL: https://doi.org/10.1561/2200000016.

[CKV21] Chris Coey, Lea Kapelevich, and Juan Pablo Vielma. *Solving natural conic formulations with Hypatia.jl*. 2021. arXiv: 2005.01136 [math.OC].

[GCG20] Michael Garstka, Mark Cannon, and Paul Goulart. *COSMO: A conic operator splitting method for convex conic problems*. 2020. arXiv: 1901.10887 [math.OC].

[SY15] Anders Skaaja and Yinyu Ye. *A Homogeneous Interior-Point Algorithm for Non-symmetric Convex Conic Optimization*. 2015. URL: https://web.stanford.edu/~yyye/nonsymmhsdimp.pdf.

[SGV18] Mario Souto, Joaquim D. Garcia, and Alvaro Veiga. *Exploiting Low-Rank Structure in Semidefinite Programming by Approximate Operator Splitting*. 2018. arXiv: 1810.05231 [math.OC].

Brief introduction to interior-point methods

[Nes14] Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*. 1st ed. Springer Publishing Company, Incorporated, 2014. ISBN: 1461346916.

[Tod98] Michael Todd. "On The Nesterov-Todd Direction In Semidefinite Programming". In: (Nov. 1998).

[TTT03] R. H. Tütüncü, K. C. Toh, and M. J. Todd. "Solving semidefinite-quadratic-linear programs using SDPT3". In: *MATHEMATICAL PROGRAMMING* 95 (2003), pp. 189–217.

[Zha97] Yin Zhang. "On Extending Some Primal-Dual Interior-Point Algorithms From Linear Programming to Semidefinite Programming". In: *SIAM Journal on Optimization* 8 (May 1997). DOI: 10.1137/S1052623495296115.

Implementation details

[Bor99b] Brian Borchers. "SDPLIB 1.2, a library of semidefinite programming test problems". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 683–690. DOI: 10.1080/10556789908805769. eprint: https://doi.org/10.1080/10556789908805769. URL: https://doi.org/10.1080/10556789908805769.

Numerical results

[Ali+97] F. Alizadeh et al. *SDPPACK User"s Guide – Version 0.9 Beta for Matlab 5.0*. Tech. rep. USA, 1997.

[ApS19] MOSEK ApS. *The MOSEK optimization toolbox for MATLAB manual. Version 9.0*. 2019. URL: http://docs.mosek.com/9.0/toolbox/index.html.

[Bod+21] Guilherme Bodin et al. *jump-dev/Dualization.jl: v0.3.4*. Version v0.3.4. Apr. 2021. DOI: 10.5281/zenodo.4718987. URL: https://doi.org/10.5281/zenodo.4718987.

[Bor99a] Brian Borchers. "CSDP, A C library for semidefinite programming". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 613–623. DOI: 10.1080/10556789908805765. eprint: https://doi.org/10.1080/10556789908805765. URL: https://doi.org/10.1080/10556789908805765.

[FKN97] Katsuki Fujisawa, Masakazu Kojima, and Kazuhide Nakata. "Exploiting Sparsity in Primal-Dual Interior-Point Methods for Semidefinite Programming". In: *MATHEMATICAL PROGRAMMING* 79 (1997), pp. 235–253.

[Fuj+99] Katsuki Fujisawa et al. "Numerical Evaluation of SDPA". In: *High Performance Optimization*. Applied Optimization 33. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1999. Chap. 11.

[NO92] Tsuneyoshi Nakamura and Makoto Ohsaki. "A natural generator of optimum topology of plane trusses for specified fundamental frequency". In: *Computer Methods in Applied Mechanics and Engineering* 94.1 (1992), pp. 113–129. ISSN: 0045-7825. DOI: https://doi.org/10.1016/0045-7825(92)90159-H. URL: https://www.sciencedirect.com/science/article/pii/004578259290159H.

[Stu99] Jos F Sturm. "Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones". In: *Optimization methods and software* 11.1-4 (1999), pp. 625–653.

[Stu98] Jos F. Sturm. *SeDuMi Benchmark*. 1998.

[VB96] Lieven Vandenberghe and Stephen Boyd. "Semidefinite Programming". In: *SIAM Rev.* 38.1 (Mar. 1996), pp. 49–95. ISSN: 0036-1445. DOI: 10.1137/1038003. URL: https://doi.org/10.1137/1038003.

[WZ99] Henry Wolkowicz and Qing Zhao. "Semidefinite programming relaxations for the graph partitioning problem". In: *Discrete Applied Mathematics* 96-97 (1999), pp. 461–479. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(99)00102-X. URL: https://www.sciencedirect.com/science/article/pii/S0166218X9900102X.

The PEP problem

[Tay17] Adrien B. Taylor. "Convex interpolation and performance estimation of first-order methods for convex optimization". In: (2017). URL: https://dial.uclouvain.be/pr/boreal/object/boreal:182881.

[THG17] Adrien B. Taylor, Julien M. Hendrickx, and François Glineur. “Performance estimation toolbox (PESTO): Automated worst-case analysis of first-order optimization methods”. In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. 2017, pp. 1278–1283. DOI: 10.1109/CDC.2017.8263832.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**