Operator splitting methods for large convex conic programs

Michael Garstka

Trinity College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2021

Abstract

Convex optimisation is used to solve many problems of interest in optimal control, signal processing, finance, operations research, and machine learning. While medium-size problems can be solved efficiently by interior-point methods, modern applications often yield problems with very large dimensions. Compared to interior-point methods, operator splitting methods are computationally inexpensive, which allows them to solve much larger problems. However, they tend to converge slowly to high accuracy solutions and their performance is sensitive to algorithm parameters and problem scaling.

This thesis investigates different acceleration and scaling techniques to both improve the convergence of operator splitting methods to achieve higher accuracy solutions and to reduce the per-iteration computation time of the methods. We consider general convex conic problems and focus especially on the subset of semidefinite programs (SDPs), for which finding a solution is particularly challenging at large dimensions. The first part of the thesis introduces `COSMO`, a novel general purpose solver for convex conic programs based on the alternating direction method of multipliers (ADMM). We demonstrate that our splitting, which combines a quadratic objective function with convex conic constraints, leads to competitive performance both for problems in the conventional conic problem form but also for quadratic programs and SDPs with norm constraints. We further show how the user can use custom convex constraints to allow more natural problem formulations.

The complexity of solving SDPs can be greatly reduced if the coefficient matrices that constrain the matrix variable are sparse. Chordal decomposition techniques then allow an equivalent problem formulation with multiple constraints only on the nonzero blocks of the matrix variable which often leads to orders of magnitude speed-up. In the second part of the thesis we develop a novel clique merging algorithm that combines some of the blocks after the initial decomposition to speed up the projection step of ADMM. We demonstrate that the method leads to significant performance improvements and avoids disadvantageous decompositions.

In the third part we demonstrate how to use repeated restarts and safeguarding checks to globalize Anderson acceleration for an ADMM solver. Comprehensive benchmark results show a significant reduction in both iterations and solve time to higher accuracy solution and a more robust algorithm.

Operator splitting methods for large convex conic programs

Michael Garstka

Trinity College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2021

Für meine Eltern,
Beate und Klaus.

Abstract

Convex optimisation is used to solve many problems of interest in optimal control, signal processing, finance, operations research, and machine learning. While medium-size problems can be solved efficiently by interior-point methods, modern applications often yield problems with very large dimensions. Compared to interior-point methods, operator splitting methods are computationally inexpensive, which allows them to solve much larger problems. However, they tend to converge slowly to high accuracy solutions and their performance is sensitive to algorithm parameters and problem scaling.

This thesis investigates different acceleration and scaling techniques to both improve the convergence of operator splitting methods to achieve higher accuracy solutions and to reduce the per-iteration computation time of the methods. We consider general convex conic problems and focus especially on the subset of semidefinite programs (SDPs), for which finding a solution is particularly challenging at large dimensions. The first part of the thesis introduces `COSMO`, a novel general purpose solver for convex conic programs based on the alternating direction method of multipliers (ADMM). We demonstrate that our splitting, which combines a quadratic objective function with convex conic constraints, leads to competitive performance both for problems in the conventional conic problem form but also for quadratic programs and SDPs with norm constraints. We further show how the user can use custom convex constraints to allow more natural problem formulations.

The complexity of solving SDPs can be greatly reduced if the coefficient matrices that constrain the matrix variable are sparse. Chordal decomposition techniques then allow an equivalent problem formulation with multiple constraints only on the nonzero blocks of the matrix variable which often leads to orders of magnitude speed-up. In the second part of the thesis we develop a novel clique merging algorithm that combines some of the blocks after the initial decomposition to speed up the projection step of ADMM. We demonstrate that the method leads to significant performance improvements and avoids disadvantageous decompositions.

In the third part we demonstrate how to use repeated restarts and safeguarding checks to globalize Anderson acceleration for an ADMM solver. Comprehensive benchmark results show a significant reduction in both iterations and solve time to higher accuracy solution and a more robust algorithm.

Acknowledgements

The past 3.5 years were a great experience due to the support and encouragement of many exceptional people.

First and foremost I would like to thank my two supervisors, Paul Goulart and Mark Cannon. I consider myself very lucky to have been able to work with them. Both always encouraged me to follow my interests and helped me become an experienced researcher. I am thankful for their patience and support, especially in times of slow progress. I admire Paul's approach to generating new ideas, clarity of thought and communication, and I am grateful for his help with my code on many occasions. He also taught me the debugging-by-simplest-problem method. I am grateful for Mark's many useful suggestions to improve my projects, his ability to ask the right questions, and his wisdom on writing and editing papers and preparing presentations.

Moreover, I would like to acknowledge the generous financial support from the Clarendon Fund and Trinity College.

My academic experience in Oxford was also greatly shaped by the members of the Control group. Thanks to Bartolomeo Stellato, Goran Banjac, Ross Drummond, and Yang Zheng for all the helpful discussions during my first year. I also enjoyed the conversations during many lunches at St John's, Wadham, Mansfield and St Edmund Hall with my cohort, Licio Romao, Xiaonan Lu, Denis Lebedev, and Nikitas Rontsis. I especially realised the importance of working together, side-by-side with all of you once 'work from home' began in April 2020. I spent two intense, but rewarding, months with some members of the Control group, Paul, Sebastian East, Harrison Steel, Idris Kempf and others, to build the OxVent ventilator for Covid-19 patients. I have never learned so much in such a short space of time. I would also like to thank the developers of JuMP for their great work and in particular Juan Pablo Vielma for inviting me to the JuMP workshop in Santiago, Chile.

Before coming to Oxford, my academic path has been influenced by a number of people. I am thankful to Francesco Borrelli for hosting me at UC Berkeley and to Ugo Rosolia for his mentorship on my master's project. I would like to thank Herbert Werner (TUHH) and Andy Packard (UC Berkeley) for sparking my interest in control theory and optimisation. I am especially grateful for the help and advice

of Eugen Solowjow, Alyssa Novelia, and Andi Hansen during my time in the Bay Area. I would also like to thank the friends I made at the International House.

Oxford offers many distractions from work and I would like to thank the people that made my time here so memorable. These include Mark Verhagen, Julius Yam, Bart Rajappan, Leo Bevilacqua, Owain James, and many others from the Trinity College MCR and 18 Rawlinson Rd. Thank you especially for many foosball duels during lunch time and late into the night after Guest Night dinners in the Beer Cellar and at Magdalen College. Moreover, thank you to David Rindt for many runs, Trinity's M1 boat for numerous cold outings at dawn, and Xanita Saayman, Anis Nicholas, and Martin Wizard for the squash lessons.

Finally, I would like to thank Charlotte for her love, kindness, and encouragement during the past years. I am grateful for my parents, Beate and Klaus, and my sister, Katharina, for their constant unconditional love and support. I would not have accomplished this without all of them by my side.

Michael Garstka
Oxford,
June 2021

Contents

Notation

Sets

$\emptyset$	Empty set
$\mathbb{R}$	Real numbers
$\mathbb{R}_+$	Nonnegative real numbers
$\mathbb{R}^n$	Set of n-dimensional real vectors
$\mathbb{R}^{m\times n}$	Set of m-by-n real matrices
$\mathbb{S}^n$	Set of n-by-n real symmetric matrices
$\mathbb{S}^n_{++}$ ($\mathbb{S}^n_+$)	Set of n-by-n real symmetric positive (semi)definite matrices
$\mathcal{S}^n_+$	Set of vectorized n-by-n real symmetric positive semidefinite matrices
$\mathcal{H}$	Real Hilbert space

Vectors and Matrices

$\mathbf{0}$	Vector/Matrix of zeros in appropriate dimension
$\mathbf{1}$	Vector/Matrix of ones in appropriate dimension
$\langle x, y\rangle$	Inner product of vectors $\langle x, y\rangle = x^\top y$
$[x_1,\, x_2]$	Horizontal concatenation of vectors x_1 and x_2
$(x_1,\, x_2)$	Vertical concatenation of vectors x_1 and x_2
V_i	i-th column vector $v_i \in \mathbb{R}^n$ inside a matrix $V \in \mathbb{R}^{m\times n}$
$\mathrm{vec}(X)$	Vectorisation of matrix $X \in \mathbb{S}^n$ by stacking the columns
$\mathrm{mat}(x)$	Inverse operator of vec
$\mathrm{diag}(x)$	Operator that maps the vector $x \in \mathbb{R}^n$ to a diagonal matrix $X \in \mathbb{R}^{n\times n}$
$x \perp y$	Orthogonality of vectors x and y: $x^\top y = 0$
I	Identity matrix of appropriate dimension
$A^\top$	Transpose of matrix A
A^{-1}	Inverse of matrix A
$A^\dagger$	Moore-Penrose inverse of matrix A
$\mathrm{tr}(A)$	Trace of matrix A
$N(A)$	Nullspace of matrix A
$X \otimes Y$	Kronecker product of matrices X and Y
$\langle X, Y\rangle$	Inner product of matrices $\langle X, Y\rangle = \mathrm{tr}\left(X^\top Y\right)$
$A_{\alpha,\beta}$	Matrix created by slicing $A \in \mathbb{R}^{m\times n}$ according to a list of row indices α and a list of column indices β

Norms

$\lvert x \rvert$	Component-wise absolute value of vector x
$\lVert \cdot \rVert$	Vector norm
$\lVert x \rVert_0$	Cardinality of vector x
$\lVert x \rVert_1$	1-norm of vector x
$\lVert x \rVert_2$	Euclidean 2-norm of vector x
$\lVert x \rVert_\infty$	Infinity norm vector x
$\lVert X \rVert_F$	Frobenius norm of matrix X

Definitions and inequalities

$X \succeq Y$	$X - Y$ is positive semidefinite
$X \preceq Y$	$X - Y$ is negative semidefinite
$X \coloneqq Y$	X is defined by Y
$X =: Y$	Y is defined by X
$X \geq Y$	Element-wise inequality between X and Y
$X \geq_{\mathcal{K}} Y$	Conic inequality: $X - Y \in \mathcal{K}$

Set operations

$A \cap B$	Intersection of sets A and B
$A \cup B$	Union of sets A and B
$A \setminus B$	Relative complement of sets A and B
$A \times B$	Cartesian product of sets A and B: $A \times B = \{(a, b) \mid a \in A,\, b \in B\}$
$\sup A$	Supremum of set A
$\inf A$	Infimum of set A

Convex sets and functions

$\operatorname{argmin} f$	Set of minimizers of a function f
∇f	Gradient of a function f
∂f	Subdifferential of a function f
$\mathbf{dom}\, f$	Effective domain of the function f
$\mathbf{int}\, C$	Interior of the set C
$\mathbf{rint}\, C$	Relative interior of the set C
$\mathcal{I}_C$	Indicator function of a set C
$\mathcal{K}^\circ$	Polar cone of set $\mathcal{K}$
$\mathcal{K}^*$	Dual cone of set $\mathcal{K}$
$\mathcal{K}^\infty$	Recession cone of set $\mathcal{K}$
N_C	Normal cone of set C

Operators

Id	Identity operator
$\mathbf{Fix}\, T$	Set of fixed-points of operator T
Π_C	Projection onto a set C
$\mathbf{prox}\, f$	Proximal operator of a function f

Other notation

$\mathcal{O}(\cdot)$	Bachmann-Landau notation describing limiting behaviour of a function

Acronyms

AA	Anderson acceleration
AMD	Approximate minimum degree
ADMM	Alternating direction method of multipliers
CPU	Central processing unit
FOM	First-order method
GPU	Graphics processing unit
IPM	Interior-point method
KKT	Karush-Kuhn-Tucker (conditions)
LMI	Linear matrix inequality
LP	Linear program
MCS	Maximum cardinality search
MPC	Model predictive control
NP	Nondeterministic polynomial time
PSD	Positive semidefinite
QP	Quadratic program
SDP	Semidefinite program
SOCP	Second-order cone program
SVM	Support vector machine

1 Introduction

Mathematical optimisation or *mathematical programming* is a general framework to describe the problem

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && g_i(x) \leq 0, \quad i = 1, \ldots, m, \end{aligned} \tag{1.1}$$

where one attempts to find an optimal solution $x^* \in \mathbb{R}^n$ that minimizes an objective function $f\colon \mathbb{R}^n \to \mathbb{R}$ under a set of constraints $g_i\colon \mathbb{R}^n \to \mathbb{R}$. In practice the general non-convex mathematical program (1.1) is hard to solve efficiently and likely no polynomial-time algorithm exists, e.g. for programs with general quadratic objective and constraints unless it holds P=NP [117]. This is why the class of *convex optimisation* problems with convex objective and convex constraints has been the focus of development for solver algorithm design since at least the 1980s. Indeed, under mild assumptions convex programs are solvable in polynomial time [14] and *interior-point methods* (IPMs) for convex conic problems have shown robust practical performance for small and medium-sized problems ($n < 10\,000$). This has made convex optimisation a successful framework to solve problems in a wide range of domains, e.g. operations research, economics, control theory, machine learning, structural design, signal processing, and communications and networks [14, 25, 169].

A recent trend in many of these domains is to consider problems with a steadily increasing number of variables, e.g. by considering larger data sets with more features, larger networks, or more agents. However, the limiting factor of IPMs in practice is solving a Newton system at each iteration, which grows with the problem dimension n and has a computational cost of $\mathcal{O}(n^3)$. A particularly challenging convex problem class is *semidefinite programming*, which optimises a linear objective function over the intersection of the positive semidefinite (PSD) matrix cone and an affine space. Semidefinite programming is used to solve problems in combinatorial optimisation [65], robust control [26], stochastic optimisation [17, 18], polynomial optimisation [126], and attracts a growing interest in machine learning, e.g. in neural network verification [134], graph clustering [40], kernel matrix learning [91] and sparse principal component analysis [35]. The PSD condition on the matrix variable requires iterative solver algorithms to repeatedly compute either the eigenvalue decomposition of the iterate or perform a Cholesky decomposition. Both methods have an expensive computational cost of $\mathcal{O}(n^3)$ for a $n \times n$ matrix.

Two common approaches to solve large convex problems, and in particular semidefinite programs (SDPs), are the use of *first-order methods* (FOMs) and exploitation of sparsity in the problem. Compared to IPMs, FOMs such as the *alternating direction method of multipliers* (ADMM) have considerably lower per-iteration computation cost and can therefore handle much larger problems. For certain separable problems ADMM also decomposes into a decentralized and parallelizable algorithm. Unfortunately, FOMs typically converge slowly to high accuracy solutions and are therefore used in applications where solutions of only moderate accuracy are acceptable. Many FOMs are also called *operator splitting methods* as they arise from the more general problem of finding zeros of the sum of two *monotone operators*.

Exploiting sparsity is particularly effective for SDPs as it allows the problem to be decomposed into an equivalent problem with typically more PSD constraints of much smaller dimension. However, the decomposition is usually not unique and it remains a difficult question how to choose it for a given solver algorithm and computing hardware in order to achieve the best performance.

This thesis considers both of these approaches for improving scalability and as a result develops a user-friendly, customisable, and fast implementation of an ADMM-based solver for large convex conic programs with automatic decomposition of sparse SDPs. We further develop ways to address the challenges of FOMs and SDP decompositions. To achieve higher accuracy solutions we explore different acceleration schemes for FOMs. In particular we show how to apply a variant of *Anderson acceleration* (AA), a well-known extrapolation method, to our particular problem splitting. We show that repeated restarts and a simple nonexpansiveness condition on the residual norms can be used to globalize the method.

Moreover, we develop a novel *clique merging* method to generate SDP decompositions that reduce the per-iteration time of our ADMM algorithm compared to other commonly used techniques, avoiding unfavourable decompositions. Additionally, we exploit the parallelizability of ADMM to process PSD constraints in parallel on multiple CPU threads.

Outline

The main contributions of the thesis are organised into the following chapters:

Chapter 2 The development of the algorithms in this thesis relies on concepts from convex analysis and monotone operator theory. This chapter introduces the required definitions and properties of convex optimisation, convex sets, and convex functions. We further define a number of specific convex cones, their projection functions, and discuss cone-representable constraints. The chapter ends with a description of concepts from monotone operator theory that are used as building blocks for FOMs.

Chapter 3 We develop the Conic Operator Splitting Method (`COSMO`), a solver obtained by generalising the splitting used in the QP-solver `OSQP` [148] to general convex cones. The solver package solves problems with a convex quadratic objective function and any combination of proper convex cones such as the nonnegative

orthant, the second-order cone, the PSD cone, the exponential cone, and the power cone. We show that the splitting results in performance competitive with state-of-the-art commercial and open-source solvers on problems in standard "conic-LP" form, on quadratic problems (QPs), and SDPs with norm-constraints. This is in part due to not having to transform any quadratic objective terms into second-order cone constraints, which typically slows convergence for FOMs and does not preserve problem sparsity. We further show the advantages of the solver's model updates and warm starting features on parametric problems.

We implement the solver package in Julia, an expressive and fast compiled programming language. This allows us to combine the performance of a compiled language with the ability for the user to customise the solver. We show how to use a custom convex constraint to solve the closest doubly stochastic matrix problem orders of magnitude faster than with standard constraints. Our implementation further integrates many of Julia's features, such as support of arbitrary floating point precision and multithreading. This chapter is based on:

- M. Garstka et al. "COSMO: A conic operator splitting method for large convex problems". In: *18th European Control Conference (ECC)*. Naples, Italy, 2019, pp. 1951–1956
- M. Garstka et al. "COSMO: A Conic Operator Splitting Method for Convex Conic Problems (to appear)". In: *Journal of Optimization Theory and Applications* (2021)

Chapter 4 If an SDP has structured positive semidefinite constraints, one can use chordal decomposition techniques to decompose the problem into an equivalent problem with a number of PSD constraints involving smaller matrix variables. However, the initial decomposition into a large number of low-dimensional PSD constraints is not necessarily optimal and can in some cases even be disadvantageous. In this chapter we analyse and compare existing methods to merge the cliques of the sparsity pattern after an initial decomposition to reduce the per-iteration time of the solver algorithm. Existing methods were designed for IPMs and rely heavily

on heuristics. We present a novel clique merging algorithm that utilizes the reduced clique graph of the sparsity structure to make effective merge decisions, considering a large number of merge possibilities. We subsequently show that this algorithm can be used effectively in combination with FOMs due to their simple relationship between per-iteration computation time and clique dimension. We show that the approach speeds up the per-iteration projection time of `COSMO` for most real-world sparsity patterns in benchmark sets by a factor of up to 2 or 3. Moreover, it helps to avoid disadvantageous decompositions.

We further discuss how automatic decomposition affects the parallelization of a problem and implement multithreaded projections. Benchmark test comparisons with state-of-the art SDP solvers show that our approach yields orders of magnitude improvements for very large SDPs.

This chapter is based on:

- M. Garstka et al. "A clique graph based merging strategy for decomposable SDPs". In: *IFAC-PapersOnLine* 53.2 (2020). 21th IFAC World Congress, pp. 7355–7361
- M. Garstka et al. "COSMO: A Conic Operator Splitting Method for Convex Conic Problems (to appear)". In: *Journal of Optimization Theory and Applications* (2021)

Chapter 5 A drawback of first-order optimisation methods is their slow convergence to high accuracy solutions as well as their low robustness to varying algorithm parameters and problem scaling. In this chapter we first discuss FOMs from the perspective of nonexpansive operators. We then introduce AA as a particular type of generalized Broyden's method. We develop a safeguarded version of AA that allows the underlying ADMM algorithm to achieve higher accuracy solutions with significantly fewer iterations. We improve the robustness of AA through repeated restarts and a nonexpansiveness condition on the residual operator norm. Our numerical results on more than 500 problems show that, for large QPs and SDPs, the additional computational work of computing the accelerated point is typically less

than 15 % of the solve time while reductions in both mean iterations (between 42 % to 88 %) and mean solve time (between 25 % to 82 %) are achieved in comparison with the algorithm without acceleration.

Chapter 6 In this chapter we summarize the computational impact of the different techniques proposed in this thesis. We analyse the performance contributions of chordal decomposition, clique merging, multithreading, and acceleration for `COSMO` against `SCS` and `MOSEK` on large SDPs. In particular we demonstrate that the acceleration scheme used in `COSMO` integrates well with chordal decomposition. The chapter concludes with a discussion of the major contributions of this thesis and suggests directions for future research.

2

Preliminaries

Contents

This chapter introduces general definitions and concepts that are used throughout the thesis and are necessary to understand for the contents of Chapter 3, Chapter 4, and Chapter 5. Furthermore, each of these chapters begins with a separate background

section that introduces chapter-specific concepts.

This thesis is concerned with the design of algorithms to solve convex conic optimisation problems. Therefore, we start by defining the building blocks of convex optimisation problems: convex functions and convex sets. This allows us to introduce the general form of convex optimisation problems, Lagrangian duality, and optimality conditions. Then we define convex cones which allow us to elegantly introduce nonlinearities into a linear programming framework. For a number of common cones we show how to represent a wide range of nonlinear constraints and how to formulate common subclasses of convex optimisation problems such as linear programs, quadratic programs, and semidefinite programs.

The chapter closes with the definition of nonexpansive operators that are used as building blocks for many first-order optimisation methods. The concepts introduced are extensively covered in the standard literature [14, 16, 25, 137, 144].

2.1 Convex sets and cones

Convexity is an important property of sets and functions in the domain of optimisation. Many optimisation problems in real-world applications can be described with the help of *convex sets* and *convex functions* and are consequently easier to solve than non-convex problems.

Definition 2.1.1 (Convex set)**.** A set $C \subseteq \mathbb{R}^n$ is convex if the line segment between any two points in C is contained in C, i.e. $\forall x, y \in C$ and $0 \leq \alpha \leq 1$ it holds that

$$\alpha x + (1 - \alpha) y \in C.$$

Example 2.1.1. C is a convex set in $\mathbb{R}^n$ if

- C is an affine subspace
- C is a half-space
- C is a polyhedron $\{x \mid Ax \leq b,\, Fx = g\}$
- C is a sublevel set $\{x \mid f(x) \leq t\}$ of a convex function $f\colon \mathbb{R}^n \to \mathbb{R}$, c.f. Section 2.2

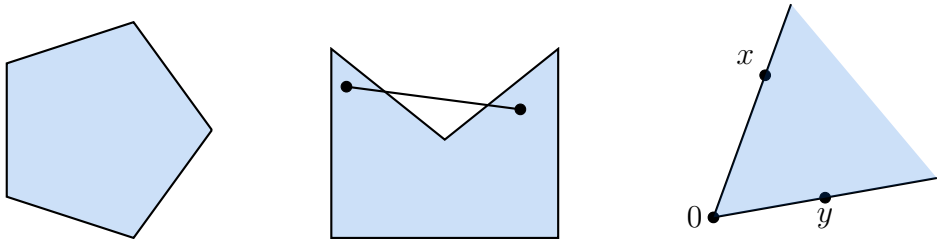
- C is the the Cartesian product of two convex sets C_1, C_2, i.e. $C = C_1 \times C_2 = \{(c_1, c_2) \mid c_1 \in C_1,\ c_2 \in C_2\}$
- $C = \bigcap_{i=1}^{m} C_i$ where C_i are convex subsets in $\mathbb{R}^n$.

Other examples of convex and non-convex sets are shown in Figure 2.1. An important subclass of convex sets are convex cones.

Definition 2.1.2 (Convex cone)**.** A set $\mathcal{K} \subseteq \mathbb{R}^n$ is a *convex cone* if it is convex and closed under nonnegative scaling with scalars $\alpha_1, \alpha_2 \geq 0$, i.e.

$$\alpha_1 x + \alpha_2 y \in \mathcal{K} \quad \forall x, y \in \mathcal{K}.$$

A convex cone in $\mathbb{R}^2$ is shown in Figure 2.1(c).



(a) A pentagon is a convex set. **(b)** Non-convex set. **(c)** Convex cone in $\mathbb{R}^2$.

Figure 2.1

Example 2.1.2. The following sets in $\mathbb{R}^n$ are examples of convex cones:

- the empty set $\emptyset$
- the norm cone $\{(x, t) \in \mathbb{R}^{n+1} \mid \|x\| \leq t\}$
- the set of real symmetric positive semidefinite matrices, i.e. the positive semidefinite cone $\mathbb{S}^n_+$

For multiple convex cones it holds that:

Theorem 1 ([137, Theorem 2.5])**.** *The intersection of convex cones is a convex cone.*

In Section 2.5 we will use convex cones to describe nonlinear variable constraints. To achieve this we extend the typical coordinate-wise partial ordering of vectors

in $\mathbb{R}^m$ $x \geq y \Leftrightarrow \{x_i \geq y_i,\, i = 1, \ldots, m\}$ to generalized vector/matrix inequalities. Specifically, we can write any inequality as:

$$x \geq y \Leftrightarrow x - y \geq 0 \Leftrightarrow x - y \in \mathbb{R}^m_+,$$

in which case the vector inequality is defined by the nonnegative orthant $\mathbb{R}^m_+$. However, we can replace $\mathbb{R}^m_+$ by another cone $\mathcal{K}$ and write

$$x \succeq_{\mathcal{K}} y \Leftrightarrow x - y \succeq_{\mathcal{K}} 0 \Leftrightarrow x - y \in \mathcal{K}, \tag{2.1}$$

as long as the following axioms of a partial ordering for vectors $x, y, u, v \in \mathbb{R}^m$ are obeyed [14], namely:

- reflexivity: $x \succeq_{\mathcal{K}} x$,
- antisymmetry: if $x \succeq_{\mathcal{K}} y$ and $y \succeq_{\mathcal{K}} x$, then $x = y$,
- transitivity: if $x \succeq_{\mathcal{K}} y$ and $y \succeq_{\mathcal{K}} z$, then $x \succeq_{\mathcal{K}} z$,
- compatibility with linear operations:
  - homogeneity: if $x \succeq_{\mathcal{K}} y$ and $a \in \mathbb{R}_+$, then $ax \succeq_{\mathcal{K}} ay$,
  - additivity: if $x \succeq_{\mathcal{K}} y$ and $u \succeq_{\mathcal{K}} v$, then $x + u \succeq_{\mathcal{K}} y + v$.

These axioms are satisfied if we require the cones used in generalized inequalities to be *proper* [14].

Definition 2.1.3 (Proper cone)**.** A (convex) cone $\mathcal{K}$ in $\mathbb{R}^m$ is proper if it

- is closed,
- is pointed: $\mathcal{K} \cap (-\mathcal{K}) = \{0\}$,
- has nonempty interior.

Given a convex set C we might want to group vectors y that satisfy certain conditions with all vectors in C. For that we define a number of special cones derived from C.

Definition 2.1.4 (Dual cone $\mathcal{K}^*$)**.** The *dual cone* $\mathcal{K}^*$ of the convex set $C \subseteq \mathbb{R}^n$ is defined as

$$\mathcal{K}^* \coloneqq \left\{ y \in \mathbb{R}^n \mid \inf_{x \in C} \langle x, y \rangle \geq 0 \right\}.$$

The dual cone is always closed and convex, even if C is not.

If a cone $\mathcal{K}$ is its own dual cone, i.e. $\mathcal{K} = \mathcal{K}^*$, the cone is called *self-dual*. Some example cones that possess this property are discussed in Section 2.4.

Definition 2.1.5 (Polar cone $\mathcal{K}^\circ$)**.** The polar cone $\mathcal{K}^\circ$ of a convex set $C \subseteq \mathbb{R}^n$ is defined as

$$\mathcal{K}^\circ := \left\{ y \in \mathbb{R}^n \mid \sup_{x \in \mathcal{C}} \langle x, y \rangle \leq 0 \right\}.$$

Notice that the polar cone is equal to the negative dual cone, i.e. it satisfies $\mathcal{K}^\circ = -\mathcal{K}^*$. Examples of both dual and polar cones are shown in Figure 2.2(a).

Definition 2.1.6 (Recession cone $\mathcal{K}^\infty$)**.** The *recession cone* $\mathcal{K}^\infty$ of a convex set $C \subseteq \mathbb{R}^n$ is defined as

$$\mathcal{K}^\infty := \{ y \in \mathbb{R}^n \mid x + \alpha y \in C,\, \forall x \in C,\, \alpha \geq 0 \}.$$

In other words, the recession cone contains all directions y of half-lines that start in $x \in C$ and remain inside C. An example of a recession cone is shown in Figure 2.2(b). It follows that the recession cone of a bounded set is the set that only contains the origin.

Definition 2.1.7 (Normal cone $N_C(x)$)**.** The *normal cone* $N_C(x)$ of a convex set $C \subseteq \mathbb{R}^n$ at a point $x \in C$ is defined as

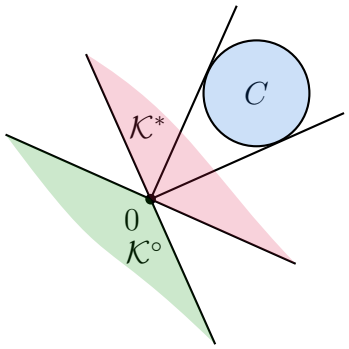
$$N_C(x) := \left\{ y \in \mathbb{R}^n \mid \sup_{\bar{x} \in C} \langle \bar{x} - x, y \rangle \leq 0 \right\}.$$

In other words the normal cone is the set of vectors that do not make an acute angle with any line segment in C with x as an endpoint. Examples of normal cones are shown in Figure 2.2(c)–2.2(d). The normal cone is always convex.

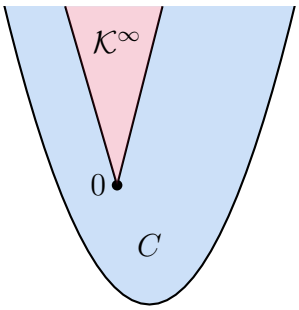
If the set C is itself a cone, then it is related to the polar cone by $N_{\mathcal{K}}(x) = \mathcal{K}^\circ \cap \{x\}^\perp$ where $\{x\}^\perp = \{y \in \mathbb{R}^n \colon y \perp x\}$.

Proof. To prove this relationship, assume $v \in \mathcal{K}^\circ \cap \{x\}^\perp$, then for any $\bar{x} \in \mathcal{K}$, we have $\langle v, \bar{x} - x \rangle = \langle v, \bar{x} \rangle - \langle v, x \rangle = \langle v, \bar{x} \rangle \leq 0$, which is the same as $v \in N_{\mathcal{K}}(x)$.

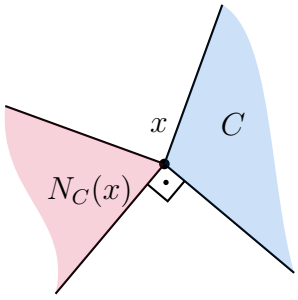
Conversely, assume $v \in N_{\mathcal{K}}(x)$, then we can choose $\bar{x} = x/2 \in \mathcal{K}$ and $\bar{x} = 2x \in \mathcal{K}$ and get $\langle v, x\rangle \leq 0 \leq \langle v, x\rangle$, which requires $v \perp x$. Thus, for any $\bar{x} \in \mathcal{K}$, $\langle v, \bar{x} - x\rangle = \langle v, \bar{x}\rangle \leq 0$, hence $v \in \mathcal{K}^\circ$. □



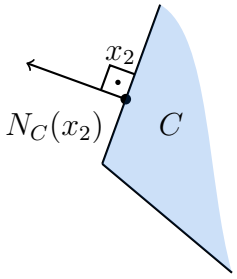
(a) Dual cone $\mathcal{K}^*$ and polar cone $\mathcal{K}^\circ$ of a set C.



(b) Recession cone $\mathcal{K}^\infty$ of a set C.



(c) Normal cone $N_C(x)$ of convex set C at x.



(d) Normal cone $N_C(x_2)$ of convex set C at x_2.

Figure 2.2

2.2 Convex functions

A function f is convex if, for two points x_1 and x_2, the line connecting the coordinates $(x_1, f(x_1))$ and $(x_2, f(x_2))$ lies above the graph of f.

Definition 2.2.1 (Convex function)**.** A function $f\colon \mathbb{R}^n \to \mathbb{R}$ is *convex* if the domain of f is a convex set and for all pairs of points x_1, x_2 and α with $0 \leq \alpha \leq 1$ it holds that

$$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha) f(x_2). \tag{2.2}$$

A function is *strictly convex* if the the inequality in (2.2) is strict. A function f is

strongly convex with parameter $m > 0$ if the augmented function

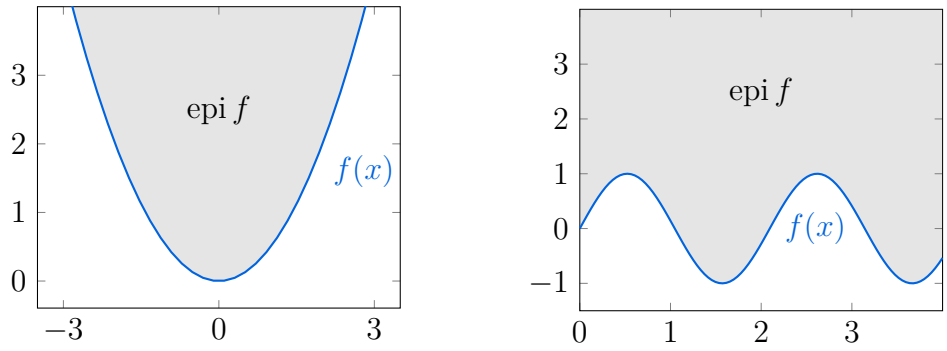
$$f_a(x) \coloneqq f(x) - \frac{m}{2} \|x\|_2^2$$

is convex. Clearly strong convexity $\Rightarrow$ strict convexity $\Rightarrow$ convexity.

A function is *concave* if $-f$ is convex. The convexity of a function $f\colon \mathbb{R}^n \to \mathbb{R}$ can also be established by looking at the *epigraph* of the function, which is the set defined as:

$$\mathbf{epi}\, f \coloneqq \{(x,t) \mid x \in \mathbf{dom}\, f,\ f(x) \leq t\}$$

and can be thought of as the region above the graph. It holds that a function is convex if and only if its epigraph is a convex set. Example epigraphs for two functions are shown in Figure 2.3.



(a) Epigraph of $f(x) = \frac{1}{2}x^2$ forming a convex set.

(b) Epigraph of $f(x) = \sin(3x)$ forming a non-convex set.

Figure 2.3

Example 2.2.1. Important convex functions are:

- affine functions: $f(x) = Ax + b$
- quadratic functions: $f(x) = x^\top Px + q^\top x$ with $P \in \mathbb{S}_+^n$
- norms
- max-function: $f(x) = \max\{x_1, \ldots, x_m\}$
- exponentials: $f(x) = \exp(\alpha x)$ for $\alpha \in \mathbb{R}$
- functions involving powers: $f(x) = x^\alpha$ on $\mathbb{R}_{++}$ for $\alpha \notin (0,1)$

Another important convex function that will be used extensively throughout later chapters is the *indicator function*. It associates the function

$$\mathcal{I}_C(x) \coloneqq \begin{cases} 0 & x \in C \\ +\infty & \text{otherwise} \end{cases} \tag{2.3}$$

to a convex set C. It can be used to convert an expression including a convex set into a convex function. Another function based on a set C is the *support function* at x

$$\sigma_C(x) \coloneqq \sup_{y \in C} \langle x, y \rangle,$$

which is always convex.

Given a closed nonempty convex set $C \subseteq \mathbb{R}^n$ we denote the *Euclidean projection* onto C

$$\Pi_C(y) = \operatorname*{argmin}_{x \in C} \|x - y\|_2 . \tag{2.4}$$

2.3 Convex optimisation

The general form of an optimisation problem is:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p \end{array} \tag{2.5}$$

where the goal is to find $x \in \mathbb{R}^n$ that minimizes the *objective function* $f(x) \colon \mathbb{R}^n \to \mathbb{R}$ while satisfying the *inequality constraints* $g_i(x) \leq 0$ with $f_i \colon \mathbb{R}^n \to \mathbb{R}$ and the *equality constraints* $h_i(x)$ with $h_i \colon \mathbb{R}^n \to \mathbb{R}$. A point x is *feasible* if it satisfies all the constraints of the problem, i.e. it is in the *domain* of the optimisation problem:

$$\mathcal{D} = \mathbf{dom}\, f \cap \bigcap_{i=1}^{m} \mathbf{dom}\, g_i \cap \bigcap_{i=1}^{p} \mathbf{dom}\, h_i.$$

The problem is *feasible* if at least one such point exists and *infeasible* otherwise. We denote the *optimal objective value* p^* as

$$p^* = \inf \{ f(x) \mid g_i(x) \leq 0,\, i = 1, \ldots, m,\, h_i(x) = 0,\, i = 1, \ldots, p \} .$$

An *optimal point*, or *optimal solution*, x^* is feasible and achieves $f(x^*) = p^*$.

Solving a general mathematical optimisation problem is NP-hard [115]. Therefore, for most applications the subclass of convex optimisation is preferred since polynomial-time algorithms to solve them generally exist [120]. The general problem form (2.5) describes a convex optimisation problem, if the objective function f is convex, the inequality constraints $g_i(x) \leq 0$ are convex, and the equality constraints $h_i(x) = a^\top x - b_i$ are affine [25].

With these restrictions the feasible domain of the problem becomes convex, since it is the intersection of p hyperplanes $\{x \mid Ax = b\}$ and the intersection of m convex sublevel sets $\{x \mid g_i(x) \leq 0\}$. An important property of convex optimisation problems is that locally optimal points are also globally optimal.

2.3.1 Lagrangian duality

Lagrangian duality allows the optimisation problem (2.5) to be studied from a different angle by moving the constraints into the objective. By introducing extra variables $\lambda \in \mathbb{R}^m$, $\nu \in \mathbb{R}^p$ that penalize constraint violations, we can define the *Lagrangian* $L\colon \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$:

$$L(x, \lambda, \nu) \coloneqq f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{i=1}^{p} \nu_i h_i(x).$$

The newly-introduced variables $\lambda \geq 0$ and ν are called *Lagrange multipliers* or *dual variables*.

We further define the *Lagrange dual function* $d\colon \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$ as the minimum of the Lagrangian w.r.t. x:

$$d(\lambda, \nu) \coloneqq \inf_{x \in \mathcal{D}} \left(f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \right).$$

$d(\lambda, \nu)$ is a pointwise infimum of a family of affine functions and therefore always concave. Moreover, the function bounds the optimal value p^* of the problem from

below:

$$\begin{aligned} d(\lambda,\nu) = \inf_{x\in\mathcal{D}} L(x,\lambda,\nu) &\leq L(x^*,\lambda,\nu) \\ &= f(x^*) + \sum_{i=1}^{m} \underbrace{\lambda_i g_i(x^*)}_{\leq 0} + \sum_{i=1}^{p} \underbrace{\nu_i h_i(x^*)}_{\leq 0} \\ &\leq f(x^*) = p^*, \end{aligned} \tag{2.6}$$

if $\lambda_i \geq 0$. However, this bound is only meaningful if $d(\lambda,\nu) > -\infty$ which defines the domain $\mathbf{dom}\, d$. All pairs $(\lambda,\nu) \in \mathbf{dom}\, d$ with $\lambda \geq 0$ are called *dual feasible*.

We can now look for the best lower bound for p^* by maximizing the dual function:

$$\begin{array}{ll} \text{maximize} & d(\lambda,\nu) \\ \text{subject to} & \lambda \geq 0 \end{array} \tag{2.7}$$

which is called the *dual problem* corresponding to (2.5). We denote the optimal value of the problem d^* and call the optimal pair (λ^*,ν^*) the *dual solution*. Notice, that since we are maximizing a concave objective function with a convex constraint, (2.7) is always convex. In practice the domain of the dual problem $\mathbf{dom}\, d = \{(\lambda,\nu) \mid d(\lambda,\nu) > -\infty\}$ is often made explicit by finding a number of linear constraints that describe the affine hull of $\mathbf{dom}\, d$. This will be shown for example in the derivation of the dual SDP in (2.20)–(2.21).

The lower-bound property $d^* \leq p^*$ of optimisation problems is called *weak duality*. If the primal problem is unbounded, i.e. $p^* = -\infty$, then we have $d^* = -\infty$ and we say the problem is *dual infeasible*. The opposite case, where the dual problem is unbounded, i.e. $d^* = \infty$, forces $p^* = \infty$ and the problem is called *primal infeasible*.

The difference in objective values $(p^* - d^*)$ is called the *duality gap*. If the objective values coincide $p^* = d^*$ we speak of *strong duality*. Strong duality does not hold in general, but convex problems usually have this property[1]. A sufficient condition to test for strong duality of a convex optimisation problem is *Slater's condition*, which

[1]For the remainder of the thesis we assume that strong duality holds for convex optimisation problems unless explicitly stated otherwise.

requires that a strictly feasible point $\tilde{x} \in \mathbf{rint}(\mathcal{D})$ exists:

$$\begin{aligned} g_i(\tilde{x}) &< 0 \quad i = 1, \ldots, m \\ A\tilde{x} &= b. \end{aligned} \tag{2.8}$$

2.3.2 KKT-optimality conditions

The necessary and sufficient optimality conditions for a convex optimisation problem with zero duality gap are derived by setting the gradient of the Lagrange function w.r.t. to x and evaluated at the optimal point (x^*, λ^*, ν^*) to zero:

$$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(x^*) + \sum_{i=1}^{p} \nu_i^* \nabla h_i(x^*) = 0. \tag{2.9}$$

Strong duality, $p^* = d^*$, which can be ensured through a variety of *constraint qualifications* such as Slater's constraint qualification (2.8), implies that at the optimal solution the *complementary slackness* condition

$$\sum_{i=1}^{m} \lambda_i^* g_i(x^*) = 0, \quad \Rightarrow \lambda_i^* g_i(x^*) = 0, \quad i = 1, \ldots, m. \tag{2.10}$$

holds; see (2.6). Adding the primal and dual domain conditions and the complementary slackness condition (2.10) to the optimality condition (2.9) yields the *Karush-Kuhn-Tucker* (KKT) conditions for an optimal triplet (x^*, λ^*, ν^*), which are shown in Table 2.1.

Table 2.1: Karush-Kuhn-Tucker (KKT) conditions for optimal solution (x^*, λ^*, ν^*).

Stationarity	$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(x^*) + \sum_{i=1}^{p} \nu_i^* \nabla h_i(x^*) = 0$
Primal feasibility	$g_i(x^*) \leq 0,\ i = 1, \ldots, m$ $h_i(x^*) = 0,\ i = 1, \ldots, p$
Dual feasibility	$\lambda_i^* \geq 0,\ i = 1, \ldots, m$
Complementary slackness	$\lambda_i^* g_i(x^*) = 0,\ i = 1, \ldots, m$

2.4 Specific convex cones and projections

A small number of proper convex cones are used in practice to model a large variety of convex optimisation problems arising from numerous applications. For example, Vielma et al. [164] showed that all 333 convex problem instances in the MINLPLIB2 benchmark library [165] are representable by four standard cones.

This section concentrates on the zero cone, the nonnegative orthant, the second-order cone and the semidefinite cone, the exponential cone and its dual, and the power cone and its dual. For each cone we define the projection function which is used in Chapter 3 as an important component to design first-order algorithms. We also highlight some of the convex functions that are representable by each cone.

In some of the following chapters matrix data is considered in vectorized form. We denote the vectorization of a matrix X by stacking its columns as $x := \text{vec}(X)$ and the inverse operation as $\text{vec}^{-1}(X) = \text{mat}(x)$. For symmetric matrices it is often computationally beneficial to work only with the upper-triangular elements of the matrix. Denote the transformation of a symmetric matrix $V \in \mathbb{S}^n$ with i, j-th element V_{ij} into a vector of upper-triangular elements as

$$\text{svec}(V) := \left[V_{11}, \sqrt{2}V_{12}, V_{22}, \sqrt{2}V_{13}, \sqrt{2}V_{23}, V_{33}, \sqrt{2}V_{14}, \ldots, V_{nn}\right]^\top.$$

Here the scaling factor of $\sqrt{2}$ preserves the matrix inner product, i.e. $\text{tr}(AB) = \text{svec}(A)^\top \text{svec}(B)$ for symmetric matrices $A, B \in \mathbb{S}^n$. The inverse operation is denoted by $\text{svec}^{-1}(s) =: \text{smat}(s)$.

2.4.1 Zero cone

The *zero cone* of dimension n is denoted as $\{0\}^n$ and defined as the set of vectors with all elements equal to zero:

$$\{0\}^n := \{x \in \mathbb{R}^n \mid x_i = 0, \forall i = 1, \ldots, n\}.$$

The dual cone of the zero cone is the whole vector space $(\{0\}^n)^* = \mathbb{R}^n$. Given a vector $x \in \mathbb{R}^n$ the projection function (2.4) sets all the elements of the vector to

zero:

$$\Pi_{\{0\}^n}(x) = \mathbf{0}_n.$$

The zero cone is used to model convex equality constraints, e.g.

$$h(x) = b \Leftrightarrow h(x) - b \in \{0\}^n,$$

where $x, b \in \mathbb{R}^n$ and h is affine.

2.4.2 Nonnegative orthant

The *nonnegative orthant* $\mathbb{R}^n_+$ is defined as the set of vectors with all elements greater or equal zero:

$$\mathbb{R}^n_+ \coloneqq \{x \in \mathbb{R}^n \mid x_i \geq 0,\, \forall i = 1, \ldots, n\}.$$

The nonnegative orthant is a self-dual convex cone, i.e. $\mathbb{R}^{n\,*}_+ = \mathbb{R}^n_+$ with the following projection function:

$$\left(\Pi_{\mathbb{R}^n_+}(x)\right)_i = \begin{cases} x_i & x_i > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The projection sets each nonpositive element of the input vector to zero. The nonnegative orthant can be used to model inequality constraints, e.g.

$$g(x) \leq 0 \Leftrightarrow -g(x) \in \mathbb{R}^n_+,$$

where $x \in \mathbb{R}^n$ and g is convex.

2.4.3 Second-order cone

The second-order cone, also sometimes called the *quadratic* or *Lorentz cone*, denoted as $\mathcal{K}^n_{\text{soc}}$ (or simply $\mathcal{K}_{\text{soc}}$ if dimension is obvious from context), is defined as:

$$\mathcal{K}^n_{\text{soc}} \coloneqq \left\{(t, x) \mid x \in \mathbb{R}^{n-1}, t \in \mathbb{R}_+, \|x\|_2 \leq t\right\}.$$

The second-order cone is self-dual, i.e. $\mathcal{K}_{\text{soc}} = \mathcal{K}^*_{\text{soc}}$ and a vector (t, x) is projected onto it as follows:

$$\Pi_{\mathcal{K}_{\text{soc}}}((t, x)) = \begin{cases} 0 & \|x\|_2 \leq -t \\ (t, x) & \|x\|_2 \leq t \\ \left(\frac{1}{2} + \frac{1+t}{2\|x\|_2}\right)(t, x) & \|x\|_2 \geq |t|. \end{cases} \tag{2.11}$$

The second-order cone can be used to model convex constraints such as

$$\|A\hat{x} + b\|_2 \leq c^\top \hat{x} + d \Leftrightarrow \begin{bmatrix} c^\top \hat{x} + d \\ A\hat{x} + b \end{bmatrix} \in \mathcal{K}_{\text{soc}}, \tag{2.12}$$

where $\hat{x}, c \in \mathbb{R}^r$, $A \in \mathbb{R}^{n-1 \times r}$, $b \in \mathbb{R}^{n-1}$, $d \in \mathbb{R}$.

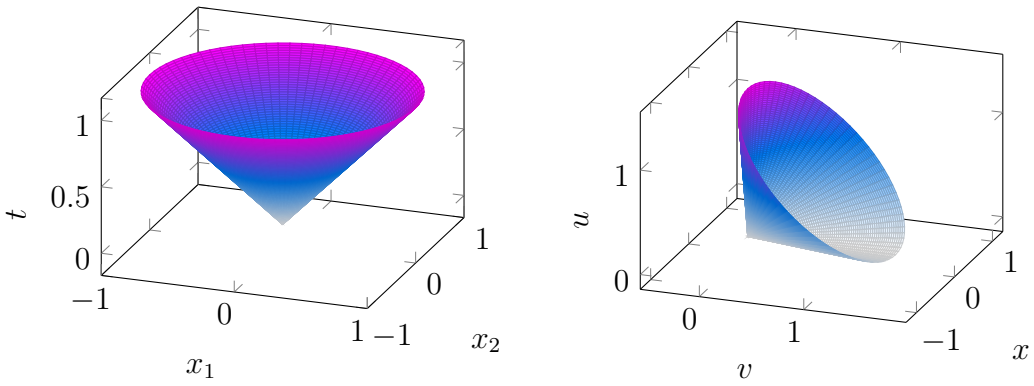
The rotated second-order cone $\mathcal{K}^n_{\text{rsoc}}$ is obtained by applying a transformation

$$T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & I_{n-2} \end{bmatrix}$$

that rotates the vector $(u, v, x) \in \mathcal{K}^n_{\text{soc}}$ by 45° in the (u, v)-plane. Thus, it is defined as

$$\mathcal{K}^n_{\text{rsoc}} := \left\{ (u, v, x) \mid x \in \mathbb{R}^{n-2}, u, v \geq 0, \|x\|_2^2 \leq 2uv \right\}.$$

To project onto $\mathcal{K}_{\text{rsoc}}$ one first undoes the rotation of the input vector, projects using (2.11), and then applies the rotation again. Both the second-order cone and the rotated second-order cone are shown in Figure 2.4.



(a) Second-order cone $\mathcal{K}^3_{\text{soc}}$. **(b)** Rotated second-order cone $\mathcal{K}^3_{\text{rsoc}}$.

Figure 2.4

The main advantage of $\mathcal{K}_{\text{rsoc}}$ is that certain constraints are more naturally represented with $\mathcal{K}_{\text{rsoc}}$ than with $\mathcal{K}_{\text{soc}}$. Some examples of constraints that can be represented using a (rotated) second-order cone are listed in Table 2.2.

Table 2.2: (Rotated) second-order cone representable constraints.

constraint	conic representation	
$\lvert x\rvert \leq t$	$(t, x) \in \mathcal{K}^2_{\text{soc}}$	(absolute value)
$\lVert x\rVert_1 = \sum \lvert x_i\rvert \leq t$	$(\xi_i, x_i) \in \mathcal{K}^2_{\text{soc}} \forall i,\ t = \sum \xi_i$	(1-norm)
$\sqrt{x} \geq \lvert t\rvert$	$(0.5, x, t) \in \mathcal{K}^3_{\text{rsoc}}$	
$t \geq \frac{1}{x},\ x > 0$	$(x, t, \sqrt{2}) \in \mathcal{K}^3_{\text{rsoc}}$	
$\sqrt{xy} \geq \lvert t\rvert,\ x, y \geq 0$	$(x, y, \sqrt{2}t) \in \mathcal{K}^3_{\text{rsoc}}$	
$t \geq (x^\top x)/y,\ y \geq 0$	$(0.5t, y, x) \in \mathcal{K}^{n+2}_{\text{rsoc}}$	
$0 \leq t \leq n\left(\sum \frac{1}{x_i}\right)^{-1}, x_i \geq 0$	$(\xi_i, x_i, t) \in \mathcal{K}^3_{\text{rsoc}} \forall i,\ \sum \xi_i = nt/2$	(harmonic mean)

2.4.4 Positive semidefinite matrix cone

The *positive semidefinite (matrix) cone* is denoted as $\mathbb{S}^n_+$ and defined as:

$$\mathbb{S}^n_+ := \left\{X \in \mathbb{S}^n \mid v^\top X v \geq 0 \quad \forall v \in \mathbb{R}^n\right\}.$$

Figure 2.5 shows the PSD cone with coordinates (x, y, z) such that

$$\begin{bmatrix} x & y \\ y & z \end{bmatrix} \in \mathbb{S}^2_+.$$

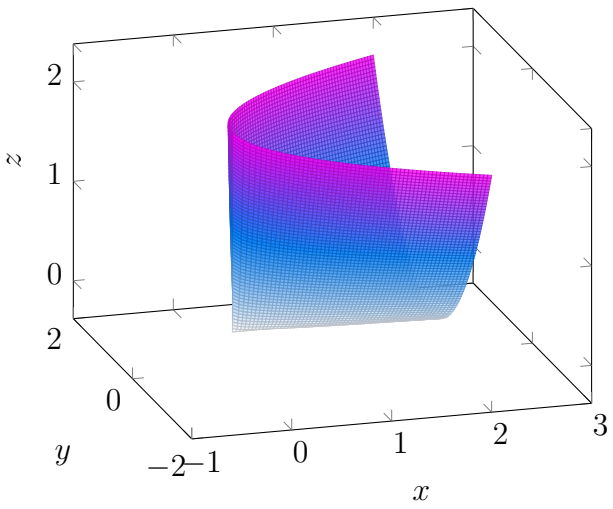


Figure 2.5: Positive semidefinite cone $\mathbb{S}^2_+$.

The positive semidefinite cone is self-dual, i.e. $\mathbb{S}^{n\,*}_+ = \mathbb{S}^n_+$. The projection of a matrix $X \in \mathbb{S}^n$ onto $\mathbb{S}^n_+$ makes use of the eigendecomposition, i.e. $X = V\Lambda V^\top$, where V's columns are formed by the eigenvectors $v_1, \ldots, v_n$ of X and Λ is a diagonal matrix

where the diagonal entries correspond to the eigenvalues $\lambda_1, \ldots, \lambda_n$. The projection is then given by removing the contribution of negative eigenvalues:

$$\Pi_{\mathbb{S}^n_+}(X) = \sum_{i=1}^{n} \max(\lambda_i, 0) v_i v_i^\top. \tag{2.13}$$

The positive semidefinite cone is used to model matrix constraints, i.e.

$$X \succeq 0 \Leftrightarrow X \in \mathbb{S}^n_+$$

as well as *linear matrix inequalities* (LMIs) involving a variable vector $x \in \mathbb{R}^m$ and known data matrices $A_0, \ldots, A_m \in \mathbb{S}^n$ of the form

$$A_0 + A_1 x_1 + A_2 x_2 + \cdots + A_m x_m \succeq 0.$$

Moreover, we can upper-bound the largest eigenvalue $\lambda_{\max}(X) \leq t$ of X using

$$tI - X \succeq 0$$

and the largest singular value $\sigma_1(X) \leq t$ (the *spectral norm* $\|X\|_2$) using

$$t^2 I - X^\top X \succeq 0.$$

The *nuclear norm* $\|X\|_* = \sum_{i=1}^n \sigma_i(X)$ can be found by solving

$$\begin{array}{ll} \text{maximize} & \operatorname{tr}(X^\top Y) \\ \text{subject to} & \begin{bmatrix} I & Y^\top \\ Y & I \end{bmatrix} \succeq 0. \end{array}$$

Other PSD cone representable functions, such as quadratic-over-linear LMIs, arise from *Schur's complement* [75]. Given a block matrix $M = \begin{bmatrix} A & B \\ B^\top & C \end{bmatrix}$ with matrices $A \in \mathbb{S}^p$, $B \in \mathbb{R}^{p \times m}$, and $C \in \mathbb{S}^m$ the Schur complement S of block A given M is defined as $S \coloneqq C - B^\top A^{-1} B$. Assuming A is invertible it holds that

- $M \succ 0$ if and only if $A \succ 0$ and $C - B^\top A^{-1} B \succ 0$
- if $A \succ 0$, then $M \succeq 0$ if and only if $C - B^\top A^{-1} B \succeq 0$.

Additionally, PSD cones are used to represent polynomial nonnegativity constraints in the context of sum-of-squares problems [126].

2.4.5 Exponential cone

The non-symmetric *exponential cone* $\mathcal{K}_{\text{exp}} \subset \mathbb{R}^3$ is defined as

$$\mathcal{K}_{\text{exp}} \coloneqq \left\{(x, y, z) \mid y > 0,\ y \exp\left(\frac{x}{y}\right) \leq z\right\} \cup \{(x, 0, z) \mid x \leq 0, z \geq 0\}$$

with its dual given by

$$\mathcal{K}_{\text{exp}}^* = \left\{(u, v, w) \mid u < 0, -u \exp\left(\frac{v}{u}\right) \leq \exp(1) w\right\} \cup \{(0, v, w) \mid v \geq 0,\ w \geq 0\}.$$

Here the second term in both unions makes the respective cone proper. The exponential cone is shown in Figure 2.6(a). The projection of a vector $v = (x, y, z)$ onto the exponential cone is described in [125] and given by

$$\Pi_{\mathcal{K}_{\text{exp}}}(v) = \begin{cases} v & v \in \mathcal{K}_{\text{exp}} \\ \mathbf{0}_3 & -v \in \mathcal{K}_{\text{exp}}^* \\ (x, \max(y, 0), \max(z, 0)) & x < 0,\ y < 0 \\ \hat{v}^* \text{ from (2.14)} & \text{otherwise,} \end{cases}$$

where in the last case $\hat{v}^*$ denotes the optimal solution of the subproblem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} \|\hat{v} - v\|_2^2 \\ \text{subject to} & y \exp\left(\frac{x}{y}\right) = z,\ y > 0. \end{array} \tag{2.14}$$

The subproblem can be solved efficiently for example by applying Newton's method to the dual problem. The exponential cone is mainly used to model constraints involving the exponential function, logarithm function, or negative entropy function $f(x) = x \log(x)$. Moreover, the "log-sum-exp trick" is a logarithmic change of variables to convert non-convex *geometric programs* into convex optimisation problems [27]. A non-exhaustive list of constraints that can be modelled using the exponential cone is shown in Table 2.3.

2.4.6 Power cone

The three-dimensional *power cone* with exponent $0 < \alpha < 1$ is denoted as $\mathcal{K}_{\text{pow},\alpha} \subset \mathbb{R}^3$ and defined as

$$\mathcal{K}_{\text{pow},\alpha} \coloneqq \left\{(x, y, z) \mid x^\alpha y^{1-\alpha} \geq |z|,\ x \geq 0,\ y \geq 0\right\}$$

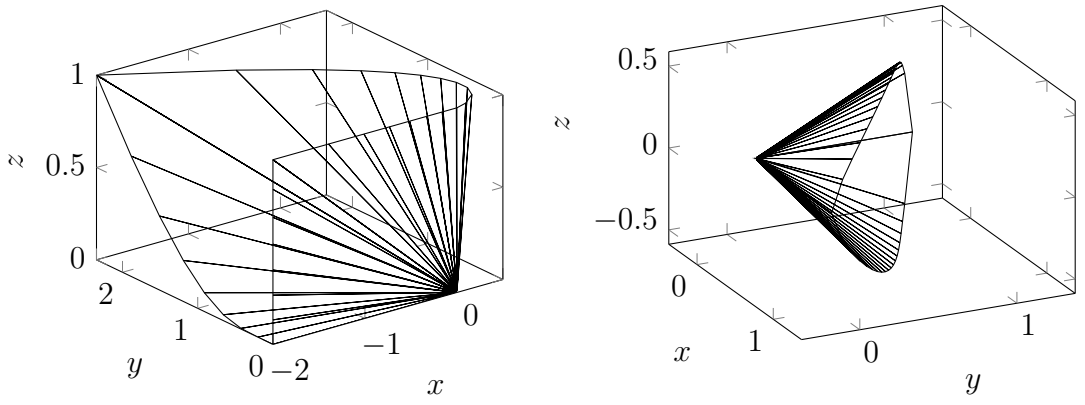
Table 2.3: Exponential cone representable constraints.

constraint	$\mathcal{K}_{\text{exp}}$ representation	
$t \geq \exp(x)$	$(x, 1, t) \in \mathcal{K}_{\text{exp}}$	
$t \leq \log(x)$	$(t, 1, x) \in \mathcal{K}_{\text{exp}}$	
$t \leq -x\log(x)$	$(t, x, 1) \in \mathcal{K}_{\text{exp}}$	(entropy)
$t \geq x\log(x/y)$	$(-t, x, y) \in \mathcal{K}_{\text{exp}}$	
$t \geq 1/\log(x),\ x > 1$	$(\xi, t, \sqrt{2}) \in \mathcal{K}^3_{\text{rsoc}},\ (\xi, 1, x) \in \mathcal{K}_{\text{exp}}$	
$t \geq x\exp(x),\ x \geq 0$	$(1/2, \xi, x) \in \mathcal{K}^3_{\text{rsoc}},\ (\xi, x, t) \in \mathcal{K}_{\text{exp}}$	
$t \geq \log(1 + \exp(x))$	$\xi + \nu \leq 1,$	
	$(x - t, 1, \xi) \in \mathcal{K}_{\text{exp}},\ (-t, 1, \nu) \in \mathcal{K}_{\text{exp}}$	
$t \geq \log(\sum \exp(x_i))$	$(x_i - t, 1, \xi_i) \in \mathcal{K}_{\text{exp}}\ \forall i,\ \sum \xi_i \leq 1$	(log-sum-exp)
$t \geq -\log(\frac{1}{1+\exp(-w^\top x)})$	$\xi + \nu \leq 1,$	(logistic cost)
	$(-w^\top x - t, 1, \xi) \in \mathcal{K}_{\text{exp}},\ (-t, 1, \nu) \in \mathcal{K}_{\text{exp}}$	

and its dual is given by

$$\mathcal{K}^*_{\text{pow},\alpha} = \left\{(u, v, w) \mid \left(\frac{u}{\alpha}\right)^{\alpha} \left(\frac{v}{1-\alpha}\right)^{1-\alpha} \geq |w|,\ u \geq 0,\ v \geq 0\right\}.$$

The power cone $\mathcal{K}_{\text{pow},\frac{1}{3}}$ is shown in Figure 2.6(b). The projection of a vector



(a) Exponential cone $\mathcal{K}_{\text{exp}}$ (from [160]) .

(b) Power cone $\mathcal{K}_{\text{pow},\frac{1}{3}}$.

Figure 2.6

$v = (x, y, z)$ onto $\mathcal{K}_{\text{pow},\alpha}$ is described by Hien [80] as

$$\Pi_{\mathcal{K}_{\text{pow},\alpha}}(v) = \begin{cases} v & v \in \mathcal{K}_{\text{pow},\alpha} \\ 0 & -v \in \mathcal{K}^*_{\text{pow},\alpha} \\ (\max(x, 0), \max(y, 0), z) & v \notin \mathcal{K}_{\text{pow},\alpha}, -v \notin \mathcal{K}^*_{\text{pow},\alpha}, z = 0 \\ \hat{v}^* \text{ from (2.15)} & \text{otherwise,} \end{cases}$$

where in the last case $\hat{v}^* = (\hat{x}, \hat{y}, \hat{z})$ is determined by solving the subproblem

$$\begin{array}{ll} \text{find} & r \\ \text{subject to} & \frac{1}{2}\left(x + \sqrt{x^2 + 4\alpha r(|z| - r)}\right)^{\alpha} \left(y + \sqrt{y^2 + 4(1-\alpha) r(|z| - r)}\right)^{1-\alpha} - r = 0 \\ & 0 < r < |z|\,. \end{array} \tag{2.15}$$

Once the unique solution r is found the components of the projected vector $\hat{v}$ are given by

$$\begin{aligned} \hat{x} &= \frac{1}{2}\left(x + \sqrt{x^2 + 4\alpha r(|z| - r)}\right) \\ \hat{y} &= \frac{1}{2}\left(y + \sqrt{y^2 + 4(1-\alpha) r(|z| - r)}\right) \\ \hat{z} &= r\frac{z}{|z|}. \end{aligned}$$

The power cone can be used to bound functions with powers and the p-norm of a vector $x \in \mathbb{R}^n$, i.e. $\|x\|_p \leq t$, see Table 2.4.

Table 2.4: Power cone representable constraints.

constraint	$\mathcal{K}_{\text{pow},\alpha}$ representation	
$t \geq \lvert x\rvert^p\,,\ p > 1$	$(t, 1, x) \in \mathcal{K}_{\text{pow},1/p}$	(powers)
$\lvert t\rvert \leq x^p,\ x > 0,\ p \in (0,1)$	$(x, 1, t) \in \mathcal{K}_{\text{pow},p}$	
$t \geq 1/x^p,\ x > 0,\ p > 0$	$(t, x, 1) \in \mathcal{K}_{\text{pow},1/(1+p)}$	
$t \geq \|x\|_p\,, x \in \mathbb{R}^n,\ p > 1$	$(\xi_i, t, x_i) \in \mathcal{K}_{\text{pow},1/p}\,\forall i,\ \sum_i^n \xi_i = t$	(p-norm)
$\lvert t\rvert \leq \sqrt[n]{x_1 \cdots x_n},\ x_i > 0$	$(\xi_i, x_i, \xi_{i+1}) \in \mathcal{K}_{\text{pow},1-1/i}\, i = 2, \ldots, n$	(geometric mean)
	$\xi_2 = x_1, \xi_{n+1} = t$	

2.4.7 Conic constraints from convex sets and functions

The cones introduced in Section 2.4.1–Section 2.4.6 can be used to represent a wide variety of nonlinear convex constraints. More generally, conic constraints can be constructed from constraints involving convex sets and convex functions with the help of the *perspective function*; see Boyd and Vandenberghe [25]. The perspective function $p\colon \mathbb{R}^{n+1} \to \mathbb{R}^n$, for a vector (x, y) with $x \in \mathbb{R}^n$ and $y > 0$ is defined as

$$p(x, y) \coloneqq \frac{1}{y}x. \tag{2.16}$$

The perspective function normalizes the first n components of the vector with respect to the last component and then discards the last component. We can use the fact that the pre-image of any nonempty convex set C as viewed through the perspective function is a convex cone:

$$\mathcal{K} = \{(x, y) \in \mathbb{R}^n \times \mathbb{R} \mid \frac{1}{y}x \in C,\, y > 0\} \cup \{(0, 0)\}.$$

Notice that the union with the origin makes the cone proper. This identity allows us to turn any convex set constraint into a convex conic constraint, i.e.

$$x \in C \Leftrightarrow (x, 1) \in \mathcal{K}.$$

Similar to (2.16) the perspective of a function $f\colon \mathbb{R}^n \to \mathbb{R}$ is defined as the function $p_f\colon \mathbb{R}^{n+1} \to \mathbb{R}$:

$$p_f(x, y) \coloneqq yf\left(\frac{x}{y}\right)$$

with $\mathbf{dom}\, p_f = \{(x, y) \mid \frac{1}{y}x \in \mathbf{dom}\, f,\, y > 0\}$ which preserves the convexity of f. Moreover, the epigraphs of p_f and f are related:

$$(x, y, t) \in \mathbf{epi}\, p_f \Leftrightarrow f(x/y) \leq t/y \Leftrightarrow (x/y, t/y) \in \mathbf{epi}\, f.$$

Therefore, the set given by the epigraph of the perspective of a convex function f is a convex cone

$$\mathbf{epi}\, p_f = \{(x, y, t) \in \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \mid yf(x/y) \leq t,\, y > 0\},$$

which we can turn into a proper cone by adding the origin

$$\mathcal{K} = \mathbf{epi}\, p_f \cup \{(0, 0, 0)\}.$$

This identity can be used to turn a convex epigraph-type constraint $f(x) \leq t$ into a conic constraint, i.e.

$$f(x) \leq t \Leftrightarrow (x, 1, t) \in \mathcal{K}.$$

An example for this is the exponential cone discussed in Section 2.4.5, which is the epigraph of the perspective applied to $f(x) = \exp(x)$, made proper by the application of the closure operator.

2.4.8 Box

Although not a cone, the *box* $\mathcal{B}$ defined as

$$\mathcal{B} \coloneqq \{x \mid l \leq x \leq u\}$$

with x, l and $u \geq l \in \mathbb{R}^n$ is a convex compact set and can be used to model two-sided constraints. The projection onto $\mathcal{B}$, also called the *saturation-operator*, is applied element-wise

$$\Pi_{\mathcal{B}}(x_i) = \max(\min(x_i, u_i), l_i).$$

Notice that any constraints involving $\mathcal{B}$ are representable by an intersection of one-sided conic constraints using the nonnegative orthant $\mathbb{R}^n_+$. The support function of the box with boundaries l and u is given by

$$\sigma_{\mathcal{B}}(x) = \langle l, \min(x, 0)\rangle + \langle u, \max(x, 0)\rangle.$$

2.5 Conic optimisation

A subset of convex optimisation problems are (convex) *conic problems* or *cone programs*. In *standard form*, they have a linear objective function in $x \in \mathbb{R}^n$, some affine equality constraints and a constraint that requires x to be inside a proper convex cone $\mathcal{K}$:

$$\begin{array}{ll} \text{minimize} & q^\top x \\ \text{subject to} & Ax = b \\ & x \in \mathcal{K}, \end{array} \tag{2.17}$$

where $q \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Alternatively, cone programs can be expressed in *inequality form*:

$$\begin{array}{ll} \text{minimize} & q^\top x \\ \text{subject to} & Fx + g \succeq_{\mathcal{K}} 0, \end{array}$$

where $F \in \mathbb{R}^{m \times n}$, $g \in \mathbb{R}^m$ and we make use of the generalized vector inequality notation introduced in (2.1). The advantage of the conic problem format is that a large number of convex optimisation problem classes can be expressed in this form by choosing a small number of proper convex cones. Moreover, this format allowed IPMs for linear programs to be extended to convex conic programs. In the following

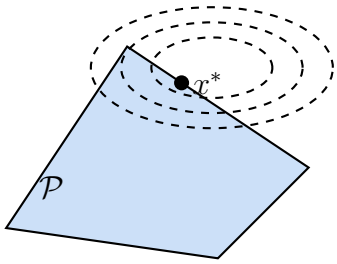
sections we introduce some of the most common conic classes; linear programs, quadratic programs, second-order-cone programs, and semidefinite programs.

2.5.1 Linear and quadratic programs

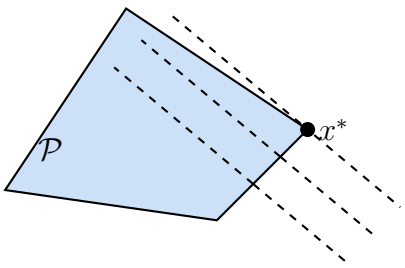
A QP consists of a quadratic objective function with equality and inequality constraints:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^\top Px + q^\top x \\ \text{subject to} & Ax \leq b \\ & Cx = d, \end{array} \tag{2.18}$$

where $P \in \mathbb{S}^n_+$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $C \in \mathbb{R}^{p\times n}$, $d \in \mathbb{R}^p$. The equality and inequality constraints form a polytope and for a feasible problem with some active constraint(s) the solution will be on the boundary of the polytope $\mathcal{P}$, see Figure 2.7(a). The constraints can be modelled using the zero cone $\{0\}^p$ and the nonnegative orthant $\mathbb{R}^m_+$. It is possible to transform (2.18) into the conic standard form (2.17) by modelling the quadratic part of the objective function using an additional second-order cone constraint. If $P = 0$ the objective function becomes linear and the problem is called a *linear program* (LP). If the optimal solution for an LP is unique it is found on one of the vertices of the constraint polytope $\mathcal{P}$, see Figure 2.7(b).



(a) Optimal point x^* on the boundary of the constraint set (QP).



(b) Optimal point x^* on the vertex of the constraint set (LP).

Figure 2.7

2.5.2 Second-order cone programs

A second-order cone program (SOCP) has a linear cost function and second-order cone constraints:

$$\begin{array}{ll} \text{minimize} & q^\top x \\ \text{subject to} & \|A_i x + b_i\|_2 \leq c_i^\top x + d_i, \quad i = 1, \ldots, p \end{array}$$

where $A_i \in \mathbb{R}^{m \times n}$, $b_i \in \mathbb{R}^m$, $c_i \in \mathbb{R}^n$ and $d \in \mathbb{R}$. Using the transformation of (2.12) the constraint can be written as:

$$\begin{bmatrix} c^\top x + d \\ Ax + b \end{bmatrix} \in \mathcal{K}_{\text{soc}}^{m+1}.$$

Furthermore, notice that any strictly convex quadratic constraint:

$$\hat{x} P \hat{x} + 2q^\top \hat{x} + r \leq 0$$

with $P \in \mathbb{S}_{++}^n$, $\hat{x}, q \in \mathbb{R}^{n-1}$, $r \in \mathbb{R}$ and Cholesky factorisation $P = LL^\top$ can be written in the form of (2.12):

$$xPx + 2q^\top \hat{x} + r \leq 0 \Leftrightarrow \left\| Lx + L^{-1}q \right\|_2 \leq \sqrt{-r + q^\top P^{-1} q}.$$

This shows how QPs and quadratically constrained QPs can be solved with the help of second-order cones. More examples of applications involving using second-order-cones are described in [99].

2.5.3 Semidefinite programs

The optimized variable in a *standard form* SDP is a symmetric matrix $X \in \mathbb{S}^n$. The objective is linear in X and X is constrained to be positive semidefinite:

$$\begin{array}{ll} \text{minimize} & \langle C, X \rangle \\ \text{subject to} & \langle A_i, X \rangle = b_i \quad i = 1, \ldots, m \\ & X \in \mathbb{S}_+^n, \end{array}$$

where $A_i, C \in \mathbb{S}^n$ and $b_i \in \mathbb{R}$. The *inequality form* of an SDP is written in form of a variable $x \in \mathbb{R}^m$ and has LMI constraints:

$$\begin{array}{ll} \text{minimize} & q^\top x \\ \text{subject to} & A_0 + A_1 x_1 + A_2 x_2 + \cdots + A_m x_m \in \mathbb{S}_+^n, \end{array} \tag{2.19}$$

with problem matrices $A_0, A_1, \ldots, A_m \in \mathbb{S}^n$.

It can be useful to consider the dual of an SDP, as in some cases it might be easier to solve or may possess a certain favourable structure. The dual can be derived by following the steps discussed in Section 2.3.1 and then making the constraints that describe the domain of the dual function explicit. The Lagrangian of a standard form SDP is given by

$$L(X, Y, \nu) = \langle C, X \rangle + \sum_{i=1}^{m} \nu_i \left(\langle A_i, X \rangle - b_i \right) - \langle X, Y \rangle,$$

where we introduce the dual variables $\nu \in \mathbb{R}^m$ for the equality constraints and a positive semidefinite dual matrix variable $Y \in \mathbb{S}^n_+$ for the conic inequality constraint.[2] The dual function is given by:

$$\begin{aligned} d(Y, \nu) &= \inf_X L(X, Y, \nu) \\ &= \inf_X \left(\langle C, X \rangle + \sum_{i=1}^{m} \nu_i \left(b_i - \langle A_i, X \rangle \right) - \langle X, Y \rangle \right). \end{aligned}$$

Next, we use the derivative of the terms w.r.t. to X to express the condition that we want $d(Y, \nu) > -\infty$:

$$\begin{aligned} d(Y, \nu) &= b^\top \nu + \inf_X \left(\langle C, X \rangle - \sum_{i=1}^{m} (\nu_i \langle A_i, X \rangle) - \langle X, Y \rangle \right) \\ &= \begin{cases} b^\top \nu & \text{if } C - A^\top \nu - Y = 0, \\ -\infty & \text{otherwise,} \end{cases} \end{aligned} \tag{2.20}$$

where $A = [\mathrm{vec}(A_1), \ldots, \mathrm{vec}(A_m)]$ contains the vectorised coefficient matrices as columns. Next, we maximize the dual function to find the best lower-bound on p^* to obtain the dual form SDP:

$$\begin{array}{ll} \text{maximize} & b^\top \nu \\ \text{subject to} & A^\top \nu + Y = C \\ & Y \in \mathbb{S}^n_+. \end{array} \tag{2.21}$$

Notice that this form is essentially the inequality form SDP in (2.19).

[2] The positive semidefiniteness of Y follows from the dual feasibility condition, see Table 2.1.

2.6 Nonexpansive operators

There exist many approaches to solve convex optimisation problems. Monotone operator theory offers a unified framework to derive and analyse properties, such as convergence, of many first-order algorithms. The general idea is to transform the original problem of finding the zero of a monotone operator into a problem of finding a fixed-point of a related *nonexpansive operator* and then performing a fixed-point iteration. While initially developed for functional analysis and partial differential equations, the theory was applied to design iterative algorithms for convex optimisation problems in the 1970s [94, 144]. The literature on this topic is extensive, and we focus here on the tools necessary to provide a link to the ADMM algorithm in Chapter 3 and the acceleration techniques in Chapter 5. Good overviews on monotone operator theory and its relationship with convex optimisation are provided by Bauschke and Combettes [12] and Ryu and Boyd [144].

We consider in the following *multivalued* operators or *relations*, i.e. $A\colon \mathcal{H} \to 2^{\mathcal{H}}$ on a Hilbert space $\mathcal{H}$. This means that for $x \in \mathcal{H}$, $A(x)$ is a (possibly empty) subset of $\mathcal{H}$, notated as $A(x) = \{y \mid (x, y) \in A\}$. The identity operator is denoted as $\mathrm{Id} = \{(x, x) \mid x \in \mathcal{H}\}$ and the inverse operator as $A^{-1} = \{(y, x) \mid (x, y) \in A\}$.

In this section we will discuss operators which are monotone.

Definition 2.6.1 (Monotone operator)**.** Let $A\colon \mathcal{H} \to 2^{\mathcal{H}}$. Then A is called *monotone* if

$$(v - w)^\top (x - y) \geq 0$$

for all $(x, v) \in A$, $(y, w) \in A$. $A(x)$ is *maximally monotone* if there exists no monotone operator $B(y)$ that properly contains it, i.e. for every $(x, v) \in A$

$$(x, v) \in A \Leftrightarrow (\forall (y, w) \in A)\, \langle x - y, v - w \rangle \geq 0.$$

Important properties of monotone operators are:

- If F and G are monotone, $F + G$ is monotone.

- If F is (maximally) monotone, then F^{-1} is (maximally) monotone.

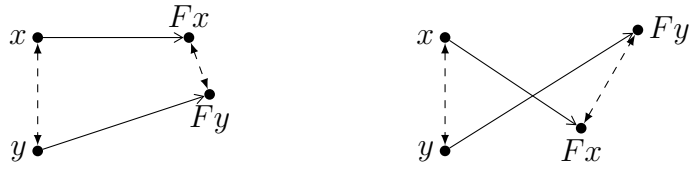
If $F(x)$ is a singleton, F is a function. For operators (or mappings) that map onto $\mathbb{R}^n$ we define the following useful properties.

Definition 2.6.2 (Nonexpansive operator)**.** An operator $F\colon \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^n$ has a *Lipschitz constant* L if it satisfies

$$\|Fx - Fy\| \leq L \|x - y\| \quad \forall x, y \in \mathcal{D}.$$

When $L = 1$, F is called *nonexpansive* and when $L < 1$, F is called *contractive*.

Examples of contractive and nonexpansive operators are shown in Figure 2.8.



(a) Contractive operator. **(b)** Nonexpansive operator.

Figure 2.8

An important property of operators for the design of converging algorithms is *firmly nonexpansiveness*.

Definition 2.6.3 (Firmly nonexpansive operator)**.** An operator $F\colon \mathcal{D} \to \mathbb{R}^n$ is said to be *firmly nonexpansive* if, for all $x, y \in \mathcal{D}$ it holds:

$$\langle Fx - Fy, x - y \rangle \geq \|Fx - Fy\|^2 .$$

Note that firmly nonexpansive $\Rightarrow$ nonexpansive.

For nonexpansive operators we are often interested in finding the fixed-points.

Definition 2.6.4 (Set of fixed-points)**.** For an operator $F\colon \mathbb{R}^n \to \mathbb{R}^n$ the *set of fixed-points* is defined as:

$$\mathbf{Fix}\, F \coloneqq \{x \in \mathbf{dom}\, F \mid x = Fx\} .$$

If F is nonexpansive and $\mathbf{dom}\, F = \mathbb{R}^n$, then $\mathbf{Fix}\, F$ is closed and convex [144]. Given a nonexpansive operator F we can define modified operators that allow algorithms with more favourable convergence properties, such as the α-averaged operator.

Definition 2.6.5 (α-averaged operator)**.** Consider the operator $F\colon \mathcal{D} \to \mathbb{R}^n$ and let $\alpha \in (0, 1)$. Then F is called *α-averaged* if there exists a nonexpansive operator $R\colon \mathcal{D} \to \mathbb{R}^n$ such that

$$F = (1 - \alpha)\mathrm{Id} + \alpha R.$$

If R is nonexpansive, then F is nonexpansive and for $\alpha \in]0, 1/2]$ firmly nonexpansive [12, Remark 4.37]. α-averaged operators have the property that the fixed-point iteration $x^{k+1} = Fx^k$ converges to a fixed point if one exists [144]. This is not generally the case for merely nonexpansive operators.

An important class of algorithms to solve constrained, non-smooth and large-scale optimisation problems are called *proximal algorithms*. A comprehensive overview of the topic is given by Parikh and Boyd [125]. The main building block for this family of algorithms is the proximal operator of a function.

Definition 2.6.6 (Proximal operator of f)**.** The *proximal operator* $\mathbf{prox}_f\colon \mathbb{R}^n \to \mathbb{R}^n$ of a closed proper convex function $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ is defined as

$$\mathbf{prox}_f(y) \coloneqq \operatorname*{argmin}_x \left(f(x) + \frac{1}{2} \|x - y\|^2 \right). \tag{2.22}$$

Sometimes the scaled version of the operator is considered and denoted as

$$\mathbf{prox}_{\rho f}(y) \coloneqq \operatorname*{argmin}_x \left(f(x) + \frac{1}{2\rho} \|x - y\|^2 \right), \tag{2.23}$$

with parameter $\rho > 0$.

Evaluating the proximal operator requires itself solving a small convex optimisation problem. Since the function is strongly convex, a unique minimizer exists for every $y \in \mathbb{R}^n$. In the context of convex optimisation algorithms the proximal operator is often evaluated for the indicator function (2.3) of a closed nonempty convex set C.

In such cases the operator becomes the (Euclidean) projection onto C (2.4), which is firmly nonexpansive.

We introduce three more operators that appear as building blocks in many proximal point algorithms and in Chapter 3 for ADMM.

Definition 2.6.7 (Resolvent operator)**.** The *resolvent operator* of an operator F is defined as

$$R_F \coloneqq (\mathrm{Id} + \alpha F)^{-1},$$

where $\alpha > 0$. The resolvent operator is firmly nonexpansive for maximally monotone F [12, Corollary 23.11].

Definition 2.6.8 (Cayley operator)**.** The *Cayley operator* of an operator F is defined in terms of the resolvent operator

$$C_F \coloneqq 2R_F - \mathrm{Id}.$$

It is also called *reflected resolvent operator* of F.

For a maximal monotone operator F, C_F is nonexpansive [12, Corollary 23.11]. Both R_F and C_F are used to transform the problem of finding a zero of the operator $0 \in F(x)$ into the problem of finding the fixed points of R_F or C_F. We further define the subdifferential $\partial f(x)$.

Definition 2.6.9 (Subdifferential)**.** The *subdifferential* $\partial f(x)$ of $f\colon \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ at x is given by

$$\partial f(x) = \{g \mid g^\top (y - x) \leq f(y) - f(x),\ \forall y \in \mathbf{dom}\, f\},$$

i.e. a closed convex set of all subgradients. If f is convex, closed, and proper, then ∂f is a maximal monotone operator [144].

We note the following relationship: The resolvent of the subdifferential operator ∂f equals the proximal operator associated with f, i.e.

$$\begin{aligned} x = R_{\partial f}(y) = (\mathrm{Id} + \alpha \partial f)^{-1}(y) &\Leftrightarrow x + \alpha \partial f(x) \ni y \\ &\Leftrightarrow 0 \in \partial_x \left(\alpha f(x) + \frac{1}{2} \|x - y\|_2^2 \right) \\ &\Leftrightarrow x = \operatorname*{argmin}_x \left(f(x) + \frac{1}{2\alpha} \|x - y\|_2^2 \right). \end{aligned} \tag{2.24}$$

3

A conic operator splitting method for convex conic problems

Contents

3.1 Introduction

In this chapter we develop a first-order algorithm for large convex conic optimisation problems. The algorithm extends the standard conic form (2.17) to support quadratic objective functions. As shown in Section 2.5 the conic problem format provides a unifying framework to model the standard problem classes encountered in convex optimisation: linear programs, quadratic programs, second-order cone programs and semidefinite programs. Moreover, many non-standard convex optimisation problems can be represented using some combination of the cones introduced in Section 2.4. We will demonstrate the flexibility of the format in modelling a variety of problems involving standard and non-standard cones in Section 3.9.

Methods to solve each of the above mentioned standard problem classes in polynomial time are well known and a number of open- and closed-source solvers are widely available. However, the trend for data and training sets of increasing size in control and decision making problems as well as machine learning poses a challenge for state-of-the-art software. Algorithms for LPs were first used to solve military planning and allocation problems in the 1940s [36]. In 1947 Dantzig developed the simplex method that solves LPs by searching for the optimal solution among the vertices of the feasible set defined by the linear inequality constraints. Extensions to the method led to the general field of *active-set methods* [168] that are able to solve both LPs and QPs, and which search for an optimal point by iteratively constructing a set of active constraints. Although often efficient in practice, a major theoretical drawback is that the worst-case complexity increases exponentially with the problem size [173].

The most common approach taken by modern convex solvers is the *interior-point method* [173], which stems from Karmarkar's original projective algorithm [87],

and is able to solve both LPs and QPs in polynomial time. IPMs have since been extended to problems with PSD constraints in [4] and [120]. The primal-dual interior point methods apply variants of Newton's method to iteratively find a solution to a set of optimality KKT conditions [76]. At each iteration the algorithm alternates between a Newton step that involves factoring a Jacobian matrix and a line search to determine the magnitude of the step to ensure a feasible iterate. Most notably, the Mehrotra predictor-corrector method in [107] forms the basis of several implementations because of its strong practical performance [172]. However, interior-point methods typically do not scale well for very large problems since the Jacobian matrix must be calculated and factored at each step.

Different approaches to overcome this limitation are active research areas. Firstly, a renewed focus on first-order methods with computationally cheaper per-iteration-cost. First-order methods are known to handle larger problems well at the expense of reduced accuracy compared to interior-point methods. In the 1960s Everett [45] proposed a dual decomposition method that allows one to decompose a separable objective function, making each iteration cheaper to compute. Augmented Lagrangian methods by Miele ([108–110]), Hestenes [78], and Powell [130] are more robust and helped to remove the strict convexity conditions on problems, while losing the decomposition property. By splitting the objective function, the alternating direction method of multipliers, first described in [55, 64], allowed the advantages of dual decomposition to be combined with the superior convergence and robustness of augmented Lagrangian methods. Subsequently, it was shown that ADMM can be analysed from the perspective of monotone operators and that it is a special case of Douglas-Rachford splitting [44] as well as of the proximal point algorithm in [138], which allowed further insight into the method.

ADMM methods are simple to implement and computationally cheap, even for large problems. However, they tend to converge slowly to a high accuracy solution and the detection of infeasibility is more involved compared to interior-point methods. They are therefore most often used in applications where a modestly accurate solution is sufficient [125]. Most of the early advances in first-order methods such

as ADMM happened in the 1970s/80s long before the demand for large scale optimisation, which may explain why they stayed less well-known and have only recently resurfaced.

Another approach to improve the solution times for very large problems is to exploit sparsity in the problem data. This approach will be discussed in more detail in Chapter 4.

3.1.1 Related Work

Widely used solvers for conic problems, especially SDPs, include `SeDuMi` [149], `SDPT3` [154] (both open source, MATLAB), and `MOSEK` [114] (commercial, C) among others. All of these solvers implement primal-dual interior-point methods.

The SDP-solver `SDPNAL` [178] (open source, MATLAB) is based on the augmented Lagrangian method. Since the augmented Lagrangian resulting from their problem definition contains a projection operator, `SDPNAL` is using an inexact semismooth Newton method to solve the inner minimization problem. They further use the conjugate gradient method to solve the linear system at each Newton step.

Several other first-order solvers, based on ADMM, have been released recently. The solver `OSQP` [148] is implemented in C and detects infeasibility based on the differences of the iterates [10] but only solves LPs and QPs. The C-based `SCS` [122] implements an operator splitting method that solves the primal-dual pair of conic programs in order to provide infeasibility certificates. The underlying homogeneous self-dual embedding method has been extended by [181] to exploit sparsity and implemented in the MATLAB solver `CDCS`. The conic solvers `SCS` and `CDCS` are unable to handle quadratic cost functions directly. Instead they are forced to reformulate problem with quadratic objective functions by adding a second-order cone constraint, which increases the problem size. Moreover, they rely on primal-dual formulations to detect infeasibility.

3.1.2 Outline

Section 3.2 introduces the ADMM algorithm as an extension of two precursors, the dual ascent method and the Augmented Lagrangian method of multipliers. We show how ADMM can also be viewed as a proximal point algorithm. In Section 3.3 we define the general conic problem format, its dual problem, as well as optimality and infeasibility conditions. Section 3.4 gives the details of the ADMM algorithm that is used by `COSMO`. Problem scaling to improve the convergence and the termination criteria of the algorithm are described in Section 3.5 and Section 3.6. Details on the implementation of the algorithm in the programming language Julia are discussed in Section 3.8. Section 3.9 demonstrates the performance of `COSMO` vs. other state-of-the art solvers on a number of benchmark problems. Section 3.10 concludes the chapter.

3.1.3 Contributions

With the solver package described in this chapter we make the following contributions:

1. We implement a first-order method for large conic problems that is able to detect infeasibility without the need of a homogeneous self-dual embedding.
2. `COSMO` supports any combination of conic constraints with a quadratic objective function, thus reducing the overhead for applications with both quadratic objective function and PSD constraints. This also avoids a major disadvantage of existing conic solvers compared to native QP solvers, i.e. no additional matrix factorisation for the conversion into a second-order cone constraint is needed and favourable sparsity in the objective can be maintained.
3. The open-source solver is written in a modular way in the fast and flexible programming language Julia. The design allows users to extend the solver by specifying a specific linear system solver and by defining their own convex cones or custom projection methods. We show how this allows more natural problem formulations and that it can lead to significant performance gains.

4. The core solver serves as a platform to integrate more advanced features such as chordal decomposition and clique merging (Chapter 4) as well as acceleration techniques (Chapter 5).

3.2 Background

In this chapter we develop an ADMM based algorithm to solve general convex conic problems. Before describing the specific algorithm and design decisions we derive the general form of ADMM and discuss method properties such as convergence, decomposability, and the range of problems that can be modelled. ADMM can be discussed (and derived) from at least two different view points. In the 1970s Gabay and Mercier [55] developed ADMM to solve nonlinear variational problems. It combined the advantages of two methods that were previously developed, the decomposability of *dual ascent method* and the robustness of *Augmented Lagrangian methods*.

In the early 1980s Gabay [54] showed that ADMM is equivalent to the Douglas-Rachford splitting. The Douglas-Rachford splitting in turn was shown to be a specific case of the proximal point algorithm [43], which completed the link between ADMM and proximal point algorithms. This means that ADMM can be expressed using the proximal operator (2.22) and many powerful convergence results from monotone operator theory can be applied. In this section we will briefly show how to derive the general form of the ADMM algorithm and how it can be interpreted from these two different viewpoints. A good overview on ADMM is provided in the survey paper by Boyd et al. [28]

We now turn to the general form of an equality-constrained convex optimisation problem:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & Ax = b, \end{array} \tag{3.1}$$

with convex objective $f\colon \mathbb{R}^n \to \mathbb{R}$, decision variable $x \in \mathbb{R}^n$ and constraint matrix $A \in \mathbb{R}^{m\times n}$. While the constraint of (3.1) looks like it would restrict the type of problems that can be represented, one has to remember that other convex

constraints, e.g. conic constraints, can be modelled by adding them to the objective function f using the indicator function (2.3).

3.2.1 Dual ascent method

A natural way to solve (3.1) is to transform it into an unconstrained problem and use a (sub)gradient method to find a solution. To achieve this we follow the steps discussed in Section 2.3.1 to derive the dual problem. The Lagrangian for (3.1) is given by

$$L(x, y) = f(x) + y^\top (Ax - b),$$

with dual variable y. Consequently, the dual problem is derived by maximizing the dual function $g(y) = \inf_x L(x, y)$. We can find the optimal solution y^* by using gradient ascent:

$$y^{k+1} := y^k + \eta^k \nabla g$$

with step size (or learning rate) $\eta^k > 0$ and where ∇g is the gradient of $g(y) = Ax^{k+1} - b$ which we iteratively evaluate at x^{k+1}. This yields the *dual ascent method*

$$\begin{aligned} x^{k+1} &:= \operatorname*{argmin}_x L(x, y^k) \\ y^{k+1} &:= y^k + \eta^k (Ax^{k+1} - b). \end{aligned}$$

Assuming strong duality, an appropriate choice for η^k, and a strongly convex f, the method converges linearly to the optimal solution. The strong convexity requirement rules out a lot of potential problems, e.g. with affine objectives. One advantage of the dual ascent method is that it leads to decentralised algorithms if f is separable w.r.t. x, i.e.

$$f(x) = \sum_{i=1}^{N} f_i(x_i)$$

where the decision variable $x = (x_1, \ldots, x_N)$ has been partitioned into subvectors x_i. Assuming we partition the constraint matrix along the appropriate dimensions $A = [A_1 \cdots A_N]$ the Lagrangian will also be separable in x

$$L(x, y) = \sum_{i=1}^{N} L_i(x_i, y)$$

which means that the first step in (3.2) can be carried out in parallel in N separate steps

$$x_i^{k+1} \coloneqq \underset{x_i}{\operatorname{argmin}}\ L_i(x_i, y^k)$$
$$y^{k+1} \coloneqq y^k + \eta^k(Ax^{k+1} - b).$$

This decomposition property has been discussed in the 1960s by Everett [45] and actively researched in the 1980s [156, 157] and in the 1990s [32]. The algorithm design can be viewed as N individual agents receiving the latest global variable y^k from a coordinator, then calculating their individual contribution x_i^{k+1} to x^{k+1}, which is then send back to the coordinator.

3.2.2 Augmented Lagrangian methods

Augmented Lagrangian methods were first discussed in the late 1960s by Hestenes [78] and Powell [130]. Their idea is to relax the restrictions on f in the dual ascent method to handle more general problems. Similarly to penalty methods, a penalty term with penalty parameter $\rho > 0$ is added to the Lagrangian

$$L_\rho(x, y) = f(x) + y^\top(Ax - b) + \frac{\rho}{2}\|Ax - b\|_2^2.$$

While this does not change the optimal solution of the problem, it makes the new objective strongly convex, provided A has full rank. Applying dual ascent yields the *method of multipliers*

$$x^{k+1} \coloneqq \underset{x}{\operatorname{argmin}}\ L_\rho(x, y^k)$$
$$y^{k+1} \coloneqq y^k + \eta^k(Ax^{k+1} - b).$$

This method converges for convex f and can also handle cases where f becomes $+\infty$, e.g. through the application of the indicator function. The disadvantage of this method is that the penalty term removes the separability of the Lagrangian when f is separable, removing an important computational advantage of the dual ascent approach, and making this method unusable for decentralised architectures.

3.2.3 Alternating direction method of multipliers

The alternating direction method of multipliers was developed by Gabay and Mercier [55] by combining a decomposition of the augmented Lagrangian functional with a dual ascent algorithm. It turns out that ADMM inherits the robustness properties of the augmented Lagrangian method and the decomposability of the dual ascent method. A general problem format for ADMM is obtained by splitting the objective into two separable parts, $f(x)$ and $g(z)$, and then linking the variables $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$ by a set of equality constraints

$$\begin{array}{ll} \text{minimize} & f(x) + g(z) \\ \text{subject to} & Ax + Bz = c, \end{array} \tag{3.5}$$

with $A \in \mathbb{R}^{p \times n}$ and $B \in \mathbb{R}^{p \times m}$. The ADMM algorithm is obtained by first constructing the augmented Lagrangian for the problem

$$L_\rho(x, z) = f(x) + g(z) + y^\top (Ax + Bz - c) + \frac{\rho}{2} \|Ax + Bz - c\|_2^2$$

and then minimizing with respect to the two variables followed by a dual variable update:

$$\begin{aligned} x^{k+1} &:= \operatorname*{argmin}_x L_\rho(x, z^k, y^k) = \operatorname*{argmin}_x \left\{ f(x) + \frac{\rho}{2} \left\| Ax + Bz^k - c + \frac{1}{\rho} y^k \right\|_2^2 \right\} \\ z^{k+1} &:= \operatorname*{argmin}_z L_\rho(x^k, z, y^k) = \operatorname*{argmin}_z \left\{ g(x) + \frac{\rho}{2} \left\| Ax^{k+1} + Bz - c + \frac{1}{\rho} y^k \right\|_2^2 \right\} \\ y^{k+1} &:= y^k + \rho(Ax^{k+1} + Bz^{k+1} - c). \end{aligned} \tag{3.6}$$

This leads to an alternating minimization scheme that gives the method its name. From the algorithm one can see that the splitting of the function allows us to handle problems with separable cost functions f or g. It was shown that ADMM converges for proper closed convex functions $f \colon \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$ and $g \colon \mathbb{R}^m \to \mathbb{R} \cup \{+\infty\}$ if primal and dual optimal points (x^*, z^*, y^*) exist and strong duality holds [28]. Moreover, both functions are allowed to be non-smooth. Notice that the extended domains of f and g allow us to use indicator functions of nonempty convex sets to model problems with conic constraints.

As noted at the beginning of Section 3.2, ADMM can also be derived as a proximal point algorithm and is identical to Douglas-Rachford splitting [94]. We will briefly demonstrate this as it can be helpful to interpret ADMM from the angle of an averaged nonexpansive fixed-point operator. The idea is to represent the problem form (3.5) as a *stationary problem*, i.e. the problem of finding the zero of the split operator

$$0 \in F(z) + G(z) \tag{3.7}$$

with maximal monotone operators F and G.

If the problem has a solution, the Douglas-Rachford method was shown to find the zeros for this problem by applying the iteration

$$v^{k+1} := (\frac{1}{2}\mathrm{Id} + \frac{1}{2}C_F C_G)(v^k), \quad z^{k+1} := R_G(v^{k+1}) \tag{3.8}$$

where C_F and C_G are the Cayley operators of F and G, and R_G is the resolvent operator of G. To transform ADMM into the problem form (3.7) notice that for an unconstrained split problem it holds:

$$\text{minimize} \quad d_1(u) + d_2(u) \Leftrightarrow 0 \in \partial d_1(u) + \partial d_2(u).$$

This problem form can be achieved by writing down the Fenchel dual of (3.5) which we will do using image functions as outlined in [63]:

$$(D \triangleright \psi)(y) := \inf\{\psi(x) \mid Dx = y\}, \tag{3.9}$$

where $D \in \mathbb{R}^{n\times m}$ and $\psi\colon \mathbb{R}^n \to \mathbb{R} \cup \{+\infty\}$. Using (3.9) the ADMM problem format (3.5) can be written in the unconstrained Douglas-Rachford compatible form

$$\text{minimize} \quad \underbrace{(-A \triangleright f)(-u - c)}_{d_1(u)} + \underbrace{(-B \triangleright g)(u)}_{d_2(u)}. \tag{3.10}$$

To write down the Douglas-Rachford algorithm for (3.10) we have to evaluate the Cayley operators C_F, C_G applied to $F(u) = \partial d_1(u)$ and $G(u) = \partial d_2(u)$. The link to proximal point algorithms is established by the fact that the Cayley operator is

the reflected resolvent operator: $C_F = 2R_F - \mathrm{Id}$, and the resolvent operator of the subdifferential mapping ∂d is precisely the proximal operator (2.22) applied to d, i.e

$$x = R_{\partial d}(y) \Leftrightarrow x = \mathbf{prox}_{\alpha d}(y),$$

which we established in (2.24). Consequently, the Douglas-Rachford splitting (3.8) can then be written in terms of proximal operators whose outcomes are stored in intermediate variables x, z:

$$z^k := \mathbf{prox}_{\gamma d_2}(v^k) \tag{3.11a}$$

$$x^k := \mathbf{prox}_{\gamma d_1}(2z^k - v^k) \tag{3.11b}$$

$$v^{k+1} := v^k + (x^k - z^k). \tag{3.11c}$$

The relaxed or averaged form of Douglas-Rachford splitting is written as

$$v^{k+1} := v^k + 2\alpha(x^k - z^k), \text{ with } \alpha \in (0, 1).$$

For Douglas-Rachford splitting we consider the following convergence result:

Theorem 2 ([12, Theorem 26.11])**.** *Assume $F(z)$ and $G(z)$ are maximal monotone operators and a solution to the stationary problem*

$$\textit{find } z \textit{ s.t. } 0 \in F(z) + G(z) \tag{3.12}$$

exists. Then there exists a fixed point $v^* \in \mathbf{Fix}\, C_F C_G$ and the Douglas-Rachford iteration (3.11a)–(3.11c) produces converging iterates $v^k \to v^*$. Moreover, $z^k = R_G(v^k)$ converges to the solution z^* of (3.12).

Proof. To prove convergence of v^k we define the Douglas-Rachford operator $D = R_F C_G + \mathrm{Id} - R_G$. Since R_F, R_G and $\mathrm{Id} - R_G$ are firmly nonexpansive, it holds that D is firmly nonexpansive with fixed points $\mathbf{Fix}\, D = \mathbf{Fix}\, C_F C_G$ [12, Proposition 4.31]. By substituting $D(v^k) = v^k + x^k - z^k$ into (3.11c) we get $v^{k+1} = v^k + (Dv^k - v^k)$. Together with the assumption that $\mathbf{Fix}\, D \neq \emptyset$ and the firm nonexpansiveness of D it follows that the sequence $(v^k)_{k\in\mathbb{N}}$ is bounded and converges to a point $v \in \mathbf{Fix}\, D$. [12, Corollary 5.17]. The convergence of z^k follows from the continuity of R_G. □

Evaluating the proximal operators of the Douglas-Rachford iteration (3.11a) – (3.11c) yields ADMM in α-averaged operator form, i.e. the algorithm is written as an operator acting on a single vector v^k:

$$\begin{aligned} z^k &:= \operatorname*{argmin}_z \left\{ g(z) + \frac{\rho}{2} \left\| Bz + v^k \right\|_2^2 \right\} \\ x^k &:= \operatorname*{argmin}_x \left\{ f(x) + \frac{\rho}{2} \left\| Ax + 2Bz^k + v^k - c \right\|_2^2 \right\} \\ v^{k+1} &:= v^k + 2\alpha(Ax^k + Bz^k - c), \end{aligned}$$

with $\alpha > 0$ and $\rho = \frac{1}{\gamma}$. This alternative form of ADMM differs from the standard algorithm form in (3.6) in the order of the minimization steps and the operator variable v. Giselsson et al. [63] show how each form can be transformed into the other.

3.3 Conic problem format

In the following we develop the ADMM-based algorithm from Section 3.2.3 for a more specific problem format and problem splitting. In particular, we will address the convex optimisation problem with quadratic objective function and a number of conic constraints

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^\top P x + q^\top x \\ \text{subject to} & Ax + s = b \\ & s \in \mathcal{K}, \end{array} \tag{3.14}$$

where $x \in \mathbb{R}^n$ is the primal *decision variable* and $s \in \mathbb{R}^m$ is the primal *slack variable*. The objective function is defined by positive semidefinite matrix $P \in \mathbb{S}_+^n$ and vector $q \in \mathbb{R}^n$. The constraints are defined by matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and a proper convex cone $\mathcal{K}$ which itself can be a Cartesian product of cones

$$\mathcal{K} = \mathcal{K}_1^{m_1} \times \mathcal{K}_2^{m_2} \times \cdots \times \mathcal{K}_N^{m_N}, \tag{3.15}$$

with cone dimensions $\sum_{i=1}^N m_i = m$. As shown in Section 2.4 any LP, QP, SOCP, SDP, and generally problems with convex constraints can be written in the form (3.14) using an appropriate choice of cones.

3.3.1 Dual problem and optimality conditions

Before stating the optimality conditions for the quadratic conic problem format (3.14) we derive the corresponding dual problem. The Lagrangian for (3.14) is given by

$$L(x, s, y) = \tfrac{1}{2}x^\top Px + q^\top x + y^\top(b - Ax - s),$$

with dual variable y and with the domain of L $\mathbf{dom}\, L = \mathbb{R}^n \times \mathcal{K} \times \mathbb{R}^m$. The dual function is derived as follows

$$\begin{aligned} d(y) &:= \inf_{x,s} L(x, s, y) \\ &= \inf_x \left\{ \tfrac{1}{2}x^\top Px + (-A^\top y + q)^\top x \right\} + b^\top y + \inf_{s\in\mathcal{K}} \left\{ -y^\top s \right\} \\ &= \inf_x \left\{ \tfrac{1}{2}x^\top Px + (-A^\top y + q)^\top x \right\} + b^\top y - \sup_{s\in\mathcal{K}} \left\{ y^\top s \right\}. \end{aligned}$$

The infimum of the terms that depend on x is obtained at $Px - A^\top y + q = 0$. Notice that the last term in the sum is simply the indicator function of the polar cone $\mathcal{K}^\circ$. The dual problem associated with (3.14) is then given by maximizing the dual function

$$\begin{aligned} \text{maximize} \quad & -\tfrac{1}{2}x^\top Px + b^\top y \\ \text{subject to} \quad & Px - A^\top y = -q \\ & y \in \mathcal{K}^\circ. \end{aligned} \tag{3.16}$$

The first-order conditions for optimality (assuming linear independence constraint qualification) follow from the KKT conditions (Table 2.1):

$$Ax + s = b, \tag{3.17a}$$

$$Px + q - A^\top y = 0, \tag{3.17b}$$

$$s \in \mathcal{K}, \quad y \in N_{\mathcal{K}}(s), \tag{3.17c}$$

where the normal cone ensures the complementary slackness condition. Assuming strong duality, if there exists a $x^* \in \mathbb{R}^n$, $s^* \in \mathbb{R}^m$, and $y^* \in \mathbb{R}^m$ that fulfil (3.17a)–(3.17c) then the pair (x^*, s^*) is called the primal solution and y^* is called the dual solution of problem (3.14).

If instead of a Cartesian product of cones $\mathcal{K}$ we allow a product $\mathcal{K}_\mathcal{B} = \mathcal{K} \times \mathcal{B}$ of cones and a compact set $\mathcal{B}$ in the constraints, we have to use the modified dual problem definition

$$\begin{aligned} \text{maximize} \quad & -\tfrac{1}{2}x^\top Px + b^\top y - \sigma_{\mathcal{K}_\mathcal{B}}(y) \\ \text{subject to} \quad & Px - A^\top y = -q \\ & y \in (\mathcal{K}_\mathcal{B}^\infty)^\circ. \end{aligned}$$

This allows for example a constraint involving the box discussed in Section 2.4.8.

3.3.2 Infeasibility certificates

Primal and dual infeasibility conditions based on successive iterates were developed for Douglas-Rachford splitting / ADMM by [10, 98]. We are applying the results of Banjac et al. [10] to the problem form (3.14). These conditions are based on separating hyperplanes that separate the constraints in the primal problem (3.14) and dual problem (3.16) respectively. To simplify the notation of the conditions, define the cone $\bar{\mathcal{K}} := -\mathcal{K} + \{b\}$. Then, the sets

$$\mathcal{P} = \left\{ x \in \mathbb{R}^n \mid Px = 0,\ Ax \in \bar{\mathcal{K}}^\infty,\ \langle q, x\rangle < 0 \right\}, \tag{3.18a}$$

$$\mathcal{D} = \left\{ y \in \mathbb{R}^m \mid A^\top y = 0,\ \sigma_{\bar{\mathcal{K}}}(y) < 0 \right\} \tag{3.18b}$$

provide certificates for primal and dual infeasibility. The existence of some $y \in \mathcal{D}$ certifies that problem (3.14) is primal infeasible, while the existence of some $x \in \mathcal{P}$ certifies dual infeasibility.

3.4 ADMM algorithm

We use the same splitting as in [148] to transform problem (3.14) into the standard ADMM format (3.5). First we introduce the dummy variables $\tilde{x} = x$ and $\tilde{s} = s$ and then stack them appropriately as the ADMM variables $\boldsymbol{x} = (\tilde{x}, \tilde{s})$ and $\boldsymbol{z} = (x, s)$. We then choose the objective splitting $f(\boldsymbol{x}) = f(\tilde{x}, \tilde{s}) = \frac{1}{2}\tilde{x}^\top P\tilde{x} + q^\top\tilde{x} + \mathcal{I}_{Ax+s=b}(\tilde{x}, \tilde{s})$

and $g(\boldsymbol{z}) = g(x,s) = \mathcal{I}_{\mathbb{R}^n(x)} \times \mathcal{I}_{\mathcal{K}}(s)$ which leads to the optimisation problem

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2}\tilde{x}^\top P\tilde{x} + q^\top \tilde{x} + \mathcal{I}_{Ax+s=b}(\tilde{x},\tilde{s}) + \mathcal{I}_{\mathcal{K}}(s) \\ &\text{subject to } (\tilde{x},\tilde{s}) = (x,s), \end{aligned} \tag{3.19}$$

where the indicator functions of the sets $\{(x,s) \in \mathbb{R}^n \times \mathbb{R}^m \mid Ax+s=b\}$ and $\mathcal{K}$ were used to move the constraints of (3.14) into the objective function. To simplify the notation we have omitted the indicator function $\mathcal{I}_{\mathbb{R}^n}(x)$ in (3.19). The augmented Lagrangian of the split optimisation problem (3.19) is given by

$$\begin{aligned} L(x,s,\tilde{x},\tilde{s},\lambda,y) = \tfrac{1}{2}\tilde{x}^\top P\tilde{x} + q^\top\tilde{x} + \mathcal{I}_{Ax+s=b}(\tilde{x},\tilde{s}) + \mathcal{I}_{\mathcal{K}}(s) \\ + \frac{\sigma}{2}\left\|\tilde{x} - x + \tfrac{1}{\sigma}\lambda\right\|_2^2 + \frac{\rho}{2}\left\|\tilde{s} - s + \tfrac{1}{\rho}y\right\|_2^2, \end{aligned}$$

with step size parameters $\rho > 0$ and $\sigma > 0$ and dual variables $\lambda \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. Applying the alternating minimization scheme (3.6) on $\mathbf{x}$ and $\mathbf{z}$ yields the ADMM iteration

$$(\tilde{x}^{k+1},\tilde{s}^{k+1}) = \operatorname*{argmin}_{\tilde{x},\tilde{s}} L\left(\tilde{x},\tilde{s},x^k,s^k,\lambda^k,y^k\right), \tag{3.20a}$$

$$x^{k+1} = \alpha\tilde{x}^{k+1} + (1-\alpha)x^k + \frac{1}{\sigma}\lambda^k, \tag{3.20b}$$

$$s^{k+1} = \operatorname*{argmin}_{s} \frac{\rho}{2}\left\|\alpha\tilde{s}^{k+1} + (1-\alpha)s^k - s + \tfrac{1}{\rho}y^k\right\|_2^2 + I_{\mathcal{K}}(s), \tag{3.20c}$$

$$\lambda^{k+1} = \lambda^k + \sigma\left(\alpha\tilde{x}^{k+1} + (1-\alpha)x^k - x^{k+1}\right), \tag{3.20d}$$

$$y^{k+1} = y^k + \rho\left(\alpha\tilde{s}^{k+1} + (1-\alpha)s^k - s^{k+1}\right), \tag{3.20e}$$

where we relaxed the z-update and the dual variable update with relaxation parameter $\alpha \in (0,2)$ according to [43]. Notice from (3.20b) and (3.20d) that the dual variable corresponding to the constraint $x = \tilde{x}$ satisfies $\lambda^k = 0$ for all k. Alternative splittings for (3.14) are discussed by Parikh and Boyd [125]. The advantage of our splitting is that the constraint $x = \tilde{x}$ leads to an advantageously structured KKT coefficient matrix when (3.20a) is evaluated. This is discussed in more detail in Section 3.4.1.

Alternatively one could also apply ADMM to the dual problem. However, eliminating x from (3.16) leads to the problem

$$\begin{aligned} \text{maximize} \quad & -\tfrac{1}{2}(A^\top y - q)^\top P^\dagger (A^\top y - q) + b^\top y \\ \text{subject to} \quad & A^\top y - q \perp N(P) \\ & y \in \mathcal{K}^\circ, \end{aligned} \tag{3.21}$$

where $N(P)$ denotes the nullspace and $P^\dagger$ the generalized inverse of P. Applying ADMM to (3.21) for problems with general quadratic objective matrices $P \in \mathbb{S}^n_+$ would therefore require an additional step to compute $P^\dagger$.

In the following two sections we describe how to evaluate the proximal operators in (3.20a) and (3.20c).

3.4.1 Solution of the equality-constrained LS problem

The minimization problem in (3.20a) has the form of an equality-constrained least squares problem:

$$\begin{aligned} \text{minimize} \quad & \tfrac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \tfrac{\sigma}{2}\left\|\tilde{x} - x^k\right\|_2^2 + \tfrac{\rho}{2}\left\|\tilde{s} - s^k + \tfrac{1}{\rho}y^k\right\|_2^2 \\ \text{subject to} \quad & A\tilde{x} + \tilde{s} = b. \end{aligned} \tag{3.22}$$

The solution of (3.22) can be obtained by solving a single linear system. The corresponding Lagrangian is given by:

$$\mathcal{L}(\tilde{x}, \tilde{s}, \nu) = \tfrac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \tfrac{\sigma}{2}\left\|\tilde{x} - x^k\right\|_2^2 + \tfrac{\rho}{2}\left\|\tilde{s} - s^k + \tfrac{1}{\rho}y^k\right\|_2^2 + \nu^\top \left(A\tilde{x} + \tilde{s} - b\right),$$

where the Lagrangian multiplier $\nu \in \mathbb{R}^m$ corresponds to the equality-constraint $Ax + s = b$. Thus, the KKT optimality conditions for this equality constrained QP are given by

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \tilde{x}} = P\tilde{x}^{k+1} + q + \sigma\left(\tilde{x}^{k+1} - x^k\right) + A^\top \nu^{k+1} = 0, \\ \frac{\partial \mathcal{L}}{\partial \tilde{s}} = \rho\left(\tilde{s}^{k+1} - s^k + \frac{1}{\rho}y^k\right) + \nu^{k+1} = 0, \\ A\tilde{x}^{k+1} + \tilde{s}^{k+1} - b = 0. \end{aligned}$$

Elimination of $\tilde{s}^{k+1}$ from these equations leads to the linear system

$$\begin{bmatrix} P + \sigma I & A^\top \\ A & -\frac{1}{\rho} I \end{bmatrix} \begin{bmatrix} \tilde{x}^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} -q + \sigma x^k \\ b - s^k + \frac{1}{\rho} y^k \end{bmatrix} \tag{3.23}$$

with

$$\tilde{s}^{k+1} = s^k - \frac{1}{\rho} \left(\nu^{k+1} + y^k \right).$$

The linear system in (3.23) can be solved using direct or indirect methods.

Direct solution method

Note that the introduction of the dummy variable $\tilde{x}$ led to the term σI in the upper-left corner of the coefficient matrix in (3.23). Consequently, the coefficient matrix is always quasi-definite [162], i.e. it always has a positive definite upper-left block and a negative definite lower-right block, and is therefore full rank even when $P = 0$ or A is rank deficient. This allows the solution of problems without strong convexity in the objective function or with linearly dependent constraints. Following Vanderbei [162] the left hand side of (3.23) always has a well-defined $LDL^\top$ factorization with lower triangular matrix L and diagonal matrix D. Since P and A represent constant problem data, this means that we can factor the linear system once at the beginning of the algorithm and then use forward- and back-substitution to compute $\tilde{x}, \nu$ at each ADMM iteration. If the step sizes ρ or σ are changed during an iteration the KKT system has to be factored again. Assuming a problem with variable dimension n and constraint dimension m, i.e. $A \in \mathbb{R}^{m \times n}$, the factorisation can be computed using at most $\frac{1}{3}N^3$ flops and $2N^2$ flops for the forward- and back-substitution where $N = m + n$. However, in practice the coefficient matrix is very sparse so a permutation allows us to achieve much sparser factors and therefore cheaper substitution steps. Computation of $\tilde{s}^{k+1}$ can be done in $\mathcal{O}(m)$ steps.

Indirect solution method

When the number of variables and constraints is large and the number of nonzeros in the KKT matrix reaches the range 10^5–10^6 indirect solution methods, like the conjugate gradient method or the minimum residual method tend to become more efficient at solving the linear system. Instead of a single factorisation step these methods use an iterative procedure to find a solution of (3.23). In order to apply the conjugate gradient method [79] we need a positive definite coefficient matrix. This can be done by eliminating ν^{k+1} from the equations, i.e.

$$\left(P + \sigma I + \rho A^\top A\right) \tilde{x}^{k+1} = -q + \sigma x^k - \rho A^\top (b - s^k + \frac{1}{\rho} y^k) \tag{3.24}$$

and then recovering $\tilde{s}^{k+1} = b - A\tilde{x}^{k+1}$. The left-hand side of (3.24) is always positive definite. At each ADMM-iteration this reduced system, with some optional preconditioning, will be solved up to a specified accuracy. Notice that changing the algorithm parameters σ and ρ does not lead to any extra computational work.

3.4.2 Projection step

The minimization problem in (3.20c) can be interpreted as the ρ-weighted proximal operator (2.23) of the indicator function $\mathcal{I}_\mathcal{K}$. It is therefore equivalent to the Euclidean projection $\Pi_\mathcal{K}$ onto the cone $\mathcal{K}$, i.e.

$$s^{k+1} = \Pi_\mathcal{K} \left(\alpha \tilde{s}^{k+1} + (1 - \alpha) s^k + \frac{1}{\rho} y^k \right). \tag{3.25}$$

If $\mathcal{K}$ is a Cartesian product of cones as in (3.15) this projection is separable and given by the projection of the relevant components of the argument of $\Pi_\mathcal{K}$ onto each cone $\mathcal{K}_i$. For example, a problem with N SDP constraints requires N projections, but, since each of these operates on an independent segment of the input vector, they can be performed in parallel. A number of projection functions for common cones were discussed in Section 2.4.

Algorithm 1: ADMM iteration

Input : initial values x^0, s^0, y^0, problem data P, q, A, b, and parameters $\sigma > 0$, $\rho > 0$, $\alpha \in (0, 2)$

1 **Do**
2 $\quad (\tilde{x}^{k+1}, \nu^{k+1}) \leftarrow$ solve linear system (3.23);
3 $\quad \tilde{s}^{k+1} \leftarrow s^k - \frac{1}{\rho}(\nu^{k+1} + y^k)$;
4 $\quad x^{k+1} \leftarrow \alpha\tilde{x}^{k+1} + (1-\alpha)x^k$;
5 $\quad s^{k+1} \leftarrow \Pi_{\mathcal{K}}\left(\alpha\tilde{s}^{k+1} + (1-\alpha)s^k + \frac{1}{\rho}y^k\right)$;
6 $\quad y^{k+1} \leftarrow y^k + \rho(\alpha\tilde{s}^{k+1} + (1-\alpha)s^k - s^{k+1})$;
7 **while** *termination criteria not satisfied*;

3.4.3 Algorithm steps

The calculations performed at each iteration are summarized in Algorithm 1. Line 2 involves solving the linear system, as discussed in the previous section, by a direct or indirect method. This requires one factorisation bounded by, but due to sparsity usually much cheaper than, $\frac{1}{3}N^3$ flops. In all subsequent iterations the substitution step is bounded by, but usually much cheaper than, $2N^2$ flops. Lines 3, 4, and 6 are computationally inexpensive (requiring $\mathcal{O}(m)$, $\mathcal{O}(n)$, and $\mathcal{O}(m)$ respectively) since they involve only vector addition and scalar-vector multiplication.

The projection in line 5 is crucial to the performance of the algorithm depending on the particular cones employed in the model: assuming a cone dimension of m, projections onto the zero-cone or the nonnegative orthant are inexpensive at $\mathcal{O}(m)$ compared to the factorisation step, while a projection onto the positive-semidefinite cone with side dimension m involves an eigenvalue decomposition. Since direct methods for eigendecompositions have a complexity of approximately $\mathcal{O}(m^3)$ with generally high constant terms, this turns line 5 into the most computationally expensive operation of the algorithm for large SDPs, and improving the efficiency of this step will be the objective of much of Chapter 4.

3.4.4 Algorithm convergence

For feasible problems, Algorithm 1 produces a sequence of iterates (x^k, s^k, y^k) that converges to a limit satisfying the optimality conditions in (3.17) as $k \to \infty$. This

also means that the primal and dual residuals converge to zero. Similarly to Stellato et al. [148] we can demonstrate the convergence of Algorithm 1 by reformulating it in Douglas-Rachford form and applying the convergence results from Theorem 2. Using the Douglas-Rachford iterates from (3.13) and choosing $\mathbf{z} = (x, s)$, $\mathbf{x} = (\tilde{x}, \tilde{s})$, $\mathbf{v} = (v_x, v_s)$ as well as

$$\begin{aligned} g(\mathbf{z}) &= \mathcal{I}_{\mathbb{R}^n \times \mathcal{K}}, \\ f(\mathbf{x}) &= \frac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \mathcal{I}_{A\tilde{x}+\tilde{s}=b} \end{aligned}$$

we get the equivalent algorithm

$$x^k := \Pi_{\mathbb{R}^n}(v_x^k), \quad s^k := \Pi_{\mathcal{K}}(v_s^k), \quad y^k := \rho(\Pi_{\mathcal{K}}(v_s^k) - v_s^k), \tag{3.26a}$$

$$(\tilde{x}, \tilde{s}) := \underset{A\tilde{x}+\tilde{s}=b}{\operatorname{argmin}} \frac{1}{2}\tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + \frac{\sigma}{2}\left\|\tilde{x} - x^k\right\|_2^2 + \frac{\rho}{2}\left\|\tilde{s} - (2s^k - v_s^k)\right\|_2^2, \tag{3.26b}$$

$$v_x^{k+1} := v_x^k + \alpha(\tilde{x}^k - \Pi_{\mathbb{R}^n}(v_x^k)), \quad v_s^{k+1} := v_s^k + \alpha(\tilde{s}^k - \Pi_{\mathcal{K}}(v_s^k)), \tag{3.26c}$$

with state-carrying vector $\mathbf{v}$. Note that $v_x^k = x^k$ for all k from (3.26a). From (3.26a) we also see that s^k and y^k fulfil the conic optimality conditions (3.17c) by construction.

Furthermore, convergence of the residual iterates in (3.17a)–(3.17b) can be concluded from the convergence of the splitting variables

$$x^k - \tilde{x}^k \to 0, \quad s^k - \tilde{s}^k \to 0, \tag{3.27}$$

which holds for the Douglas-Rachford iterates [12, Theorem 26.11(ii)]. The optimality conditions for the subproblem (3.26b) are

$$\begin{aligned} (P + \sigma I)\,\tilde{x} + q - \sigma x^k + \rho A^\top \left(\tilde{s} - 2s^k + v_s^k\right) &= 0 \\ A\tilde{x} + \tilde{s} &= b. \end{aligned} \tag{3.28}$$

Using (3.28) and (3.27), we can show that the primal residual

$$Ax^k + s^k - b = \underbrace{A\tilde{x}^k + \tilde{s}^k - b}_{\to 0} + A\underbrace{(x^k - \tilde{x}^k)}_{\to 0} + \underbrace{s^k - \tilde{s}^k}_{\to 0} + (b - b) \to 0$$

converges to zero. The same can be shown for the dual residual

$$Px^k + q - A^\top y^k = \underbrace{(P+\sigma I)\tilde{x}^k + q - \sigma x^k + \rho A^\top(\tilde{s}^k - 2s^k + v_s^k)}_{\to 0} + (P+\sigma I)\underbrace{(x^k - \tilde{x}^k)}_{\to 0} - \rho A^\top \underbrace{(s^k - \tilde{s}^k)}_{\to 0} \to 0.$$

For infeasible problems, Banjac et al. [10] showed that Algorithm 1 leads to convergence of the successive differences between iterates

$$\delta x^k = x^k - x^{k-1},\ \delta s^k = s^k - s^{k-1},\ \delta y^k = y^k - y^{k-1}.$$

For primal infeasible problems $\delta y = \lim_{k\to\infty} \delta y^k$ will satisfy condition (3.18b), whereas for dual infeasible problems $\delta x = \lim_{k\to\infty} \delta x^k$ is a certificate of (3.18a).

3.5 Problem scaling

The rate of convergence of ADMM and other first-order methods depends in practice on the scaling of the problem data; see [61]. Particularly for badly conditioned problems, this suggests a preprocessing step where the problem data is scaled in order to improve convergence. For certain problem classes an optimal scaling has been found, see [61, 62, 68]. However, the computation of the optimal scaling is often more complicated than solving the original problem. Consequently, most algorithms rely on heuristic methods such as matrix equilibration.

We scale the equality constraints by diagonal positive definite matrices D and U. The scaled form of (3.14) is given by:

$$\begin{aligned} \text{minimize} \quad & \tfrac{1}{2}\hat{x}^\top \hat{P}\hat{x} + \hat{q}^\top \hat{x} \\ \text{subject to} \quad & \hat{A}\hat{x} + \hat{s} = \hat{b}, \\ & \hat{s} \in U\mathcal{K}, \end{aligned} \tag{3.29}$$

with scaled problem data

$$\hat{P} = DPD, \quad \hat{q} = Dq, \quad \hat{A} = UAD, \quad \hat{b} = Ub, \tag{3.30}$$

and the scaled convex cone $U\mathcal{K} \coloneqq \{Uv \in \mathbb{R}^m \mid v \in \mathcal{K}\}$. After solving (3.29) the original solution is obtained by reversing the scaling:

$$x = D\hat{x}, \quad s = U^{-1}\hat{s}, \quad y = U\hat{y}. \tag{3.31}$$

One heuristic strategy that has been shown to work well in practice is to choose the scaling matrices D and U to equilibrate, i.e. reduce the condition number of, the problem data. The Ruiz equilibration technique described in Ruiz [143] iteratively scales the rows and columns of a matrix to have an infinity-norm of 1 and converges linearly. We apply the modified Ruiz algorithm shown in Algorithm 2 to reduce the condition number of the symmetric matrix

$$R = \begin{bmatrix} P & A^\top \\ A & 0 \end{bmatrix}$$

which represents the problem data. Since R is symmetric it suffices to consider the

Algorithm 2: Modified Ruiz equilibration

```
set D = I_n, U = I_m, c = 1_{m+n};
Do
    for i = 1, ..., n + m do
        if ||R_{c,i}||_∞ > τ then
            c_i ← ||R_{c,i}||_∞^{-1/2};
    D̂ = diag(c_{1:n}),   Û = diag(c_{n+1:n+m});
    D = D̂ · D,   U = Û · U;
    P = D̂ P D̂,   A = Û A D̂;
    assemble R;
while ||1 − c||_∞ > tol;
return D, U;
```

columns $R_{c,i}$ of R. At each iteration the scaling routine calculates the norm of each column. For the columns with norms higher than the tolerance τ the scaling vector c is updated with the inverse square root of the norm[1]. If the norm is below the tolerance, the corresponding column will be scaled by 1.

[1] For the presented results a value of $\tau = 10^{-6}$ was chosen.

Since the matrix U scales the (possibly composite) cone constraint, the scaling must ensure that if $s \in \mathcal{K}$ then $U^{-1}s \in \mathcal{K}$. Let $\mathcal{K}$ be a Cartesian product of N cones as in (3.15) and partition U into blocks

$$U = \operatorname{diag}(U_1, \ldots, U_N),$$

with block $U_i \in \mathbb{R}^{m_i \times m_i}$ which scales the constraint corresponding to $\mathcal{K}_i$. For each cone $\mathcal{K}_i \in \mathbb{R}^{m_i}$ that requires a scalar or symmetric scaling, *e.g.* a second-order cone or positive semidefinite cone, the corresponding block U_i is replaced with

$$U_i^* := u_i I_{m_i}, \quad \text{for } i = 1, \ldots, N$$

where the mean value of the diagonal entries of the original block in U, $u_i = \operatorname{tr}(U_i)/m_i$, was chosen as a heuristic scaling factor.

3.6 Termination criteria

The termination criteria discussed in this section are based on the unscaled problem data and iterates. Thus, before checking for termination the solver first reverses the scaling according to equations (3.30)–(3.31). To measure the progress of the algorithm, we define the primal and dual residuals

$$r_p := Ax + s - b, \tag{3.32a}$$

$$r_d := Px + q - A^\top y \tag{3.32b}$$

of the problem. According to [28, Section 3.3] a valid termination criterion is that the size of the norms of the residual iterates in (3.32) are small. Our algorithm terminates if the residual norms are below the sum of an absolute and a relative tolerance term:

$$\left\|r_p^k\right\|_\infty \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\left\{\left\|Ax^k\right\|_\infty, \left\|s^k\right\|_\infty, \|b\|_\infty\right\}, \tag{3.33a}$$

$$\left\|r_d^k\right\|_\infty \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \max\left\{\left\|Px^k\right\|_\infty, \|q\|_\infty, \left\|A^\top y^k\right\|_\infty\right\}, \tag{3.33b}$$

where ϵ_{abs} and ϵ_{rel} are user defined tolerances.

Following the infeasibility detection scheme in [10], the algorithm periodically checks if the one-step differences δx^k and δy^k of the primal and dual variable fulfil the normalized infeasibility conditions (3.18a)–(3.18b) up to certain tolerances $\epsilon_{\text{p,inf}}$ and $\epsilon_{\text{d,inf}}$. The solver returns a primal infeasibility certificate if

$$\begin{aligned} \left\|A^\top \delta y^k\right\|_\infty / \left\|\delta y^k\right\|_\infty &\leq \epsilon_{\text{p,inf}}, \\ \sigma_{\bar{\mathcal{K}}}\left(\delta y^k\right) &\leq \epsilon_{\text{p,inf}}, \end{aligned}$$

holds and a dual infeasibility certificate if

$$\begin{aligned} \left\|P\delta x^k\right\|_\infty / \left\|\delta x^k\right\|_\infty &\leq \epsilon_{\text{d,inf}}, \\ q^\top \delta x^k / \left\|\delta x^k\right\|_\infty &\leq \epsilon_{\text{d,inf}}, \\ A\delta x^k + v &\in \bar{\mathcal{K}}^\infty, \\ \text{with } \|v\|_\infty &\leq \epsilon_{\text{d,inf}} \left\|\delta x^k\right\|_\infty \end{aligned}$$

holds.

3.7 Step size adaptation

Another measure to reduce the dependence of the problem scaling on the convergence of ADMM is an update procedure for the step size, or penalty parameter, ρ in Algorithm 1. Looking at the ADMM update equations (3.6) one can see that the parameter penalizes violation of primal feasibility. This suggests that large values of ρ tend to proportionally reduce the primal residual whereas smaller values tend to reduce the dual residual. In practice a good step size adaptation strategy would update ρ_k to drive down the primal and dual residual norms (3.32a)–(3.32b) simultaneously to achieve the convergence criteria (3.33a)–(3.33b). A popular approach aims to keep the residual norms within a certain factor μ of each other. Whenever $\left\|r_p^k\right\|_2 > \mu \left\|r_d^k\right\|_2$ or $\left\|r_d^k\right\|_2 > \mu \left\|r_p^k\right\|_2$ occurs, the scheme multiplies or divides ρ^k by preselected factors to prevent the residual norms from diverging too far [28].

Using the same reasoning, we adopt the slightly different scheme proposed in [148]. We initialise $\rho = 0.1$. The step size is then adapted in certain intervals, e.g. every 25 iterations of Algorithm 1. The next ρ is chosen by scaling the previous value by the normalized ratio of residual norms

$$\rho \leftarrow \rho \sqrt{\frac{\left\|r_p^k\right\|_\infty / \max\left\{\|Ax^k\|_\infty, \|s^k\|_\infty, \|b\|_\infty\right\}}{\left\|r_d^k\right\|_\infty / \max\left\{\|Px^k\|_\infty, \|q\|_\infty, \|A^\top y^k\|_\infty\right\}}} \tag{3.36}$$

Similar to the previous scheme ρ is increased if the normalized dual residual norm is smaller than the normalized primal residual norm and vice versa.

Since ρ appears in the lower-right block of the KKT system (3.23), any update of it requires a new numerical factorization. However, the symbolic factorisation can be reused as the locations of the nonzeros in the coefficient matrix do not change. Especially for problems where the factorisation step dominates the algorithm in terms of solve time, it is important not to update ρ too frequently. Otherwise, the time spent on refactoring the KKT matrix will exceed the time saved by the reduction in iterations. If instead of a direct factorisation an indirect method, e.g. conjugate gradient method, is used to evaluate line 2 in Algorithm 1, then the ρ-updates can be done without extra cost. Notice that updating ρ does not change the convergence guarantees of the method if one assumes that after an initial period ρ remains fixed for all subsequent iterations [28].

3.8 Implementation in COSMO

We have implemented our algorithm in the Conic Operator Splitting Method (`COSMO`), an open-source package written in Julia [19]. Julia allows the solver to be written in a flexible, modular and extensible way, while still maintaining the benefits of a fast compiled language. The source code and documentation are available at

https://github.com/oxfordcontrol/COSMO.jl.

`COSMO` offers the user two interfaces to describe the constraints of the optimisation problem: a direct interface, and an interface to the modelling languages `JuMP` [42] and `Convex.jl` [158]. These interfaces connect the solver to the Julia optimisation

ecosystem, which provide flexible problem description and automatic problem reformulation. We further provide an interface to the Python language.

A high-level overview of the order and interaction of problem scaling, ρ-adaptation, termination conditions, and the ADMM steps is given in Algorithm 3. The algorithm

Algorithm 3: High-level solver algorithm (direct approach).

Input : initial values x^0, s^0, y^0, problem data P, q, A, b, $\sigma > 0$, $\rho > 0$, $\alpha \in (0, 2)$ and solver settings n_{adapt}, n_{term}, n_{infeas}, $k_{\max}$

status $\leftarrow$ `unsolved`;
$(\hat{P}, \hat{q}, \hat{A}, \hat{b}, \hat{x}^0, \hat{s}^0, \hat{y}^0) \leftarrow$ Ruiz scaling of problem data and initial values, see Section 3.5;
F $\leftarrow$ factor and cache coefficient matrix (3.23);
for $k = 1, \ldots, k_{\max}$ **do**
  $(\hat{x}^k, \hat{s}^k, \hat{y}^k) \leftarrow$ Algorithm 1$(F, \hat{x}^{k-1}, \hat{s}^{k-1}, \hat{y}^{k-1}, \hat{P}, \hat{q}, \hat{A}, \hat{b}, \sigma, \rho, \alpha)$;
  `// check termination`
  if $k \mod n_{term} = 0$ *and conditions* (3.33a)(3.33b) *hold* **then**
    status $\leftarrow$ `solved`;
    break;
  `// check infeasibility`
  if $k \mod n_{infeas} = 0$ **then**
    $\delta x^k = \hat{x}^k - \hat{x}^{k-1}, \delta y^k = \hat{y}^k - \hat{y}^{k-1}$;
    if *condition* (3.34) *holds for* δy^k **then**
      status $\leftarrow$ `primal_infeasible`;
      break;
    if *condition* (3.35) *holds for* δx^k **then**
      status $\leftarrow$ `dual_infeasible`;
      break;
  `// ρ-adaptation`
  if $k \mod n_{adapt} = 0$ **then**
    update $\rho \leftarrow$ (3.36);
    $F \leftarrow$ refactor coefficient matrix (3.23);
if *status* = ***primal_infeasible*** *or status* = ***dual_infeasible*** **then**
  return status, $(\delta x^k, \delta y^k)$;
else
  $(x, s, y) \leftarrow$ reverse scaling of $(\hat{x}, \hat{s}, \hat{y})$;
  return status, (x, s, y);

starts by scaling the problem data and any provided solution estimates. If a direct method to solve the linear system is chosen, the solver will factor the coefficient

matrix and store the factors to allow a fast back-substitution step in subsequent iterations. In most iterations we simply perform the ADMM steps in Algorithm 1. To save unnecessary calculations at each iteration, checking the termination and infeasibility conditions and updating the ρ-parameter is done at user-specified intervals, e.g. every 25 iterations. Moreover, every ρ-adaptation leads to another factorisation of the coefficient matrix. Especially for LPs and QPs, this can slow the solver down if it happens more than 2-3 times. Thus, after a new ρ parameter is determined we can add additional conditions, e.g. requiring the new ρ to be at least one order of magnitude different than the previous value.

As shown in Algorithm 1 the two main steps of the ADMM iteration require solving a linear system and projecting onto a Cartesian product of cones. The implementation of these two parts allows customisation by the user. For the solution of the linear system in (3.23) the user can either use the `QDLDL` [148] solver provided with `COSMO`, or the standard sparse solver from `SuiteSparse` [38], or choose one of the provided interfaces to direct and indirect solvers, *e.g.* `PARDISO` [86, 145], conjugate gradient, minimal residual method, or else link their own implementation.

The second important part of the algorithm is the projection step onto a Cartesian product of convex sets. By default `COSMO` supports the zero cone, the nonnegative orthant, the hyperbox, the second-order cone, the PSD cone, the exponential cone and its dual, and the power cone and its dual. Our Julia implementation also allows the user to define their own convex cones[2] and custom projection functions. To implement a custom cone $\mathcal{K}_c$ the user has to provide:

- a projection function that projects an input vector onto the cone,
- a function that determines if a vector is inside the dual cone $\mathcal{K}_c^*$,
- a function that determines if a vector is inside the recession cone of $-\mathcal{K}_c$.

The latter two functions are required for our solver to implement checks for infeasibility as described in Section 3.3.2 and Section 3.4.4. An example that shows the advantages of defining a custom cone is provided in Section 3.9.2.

[2]To allow infeasibility detection the user has to either define a convex cone, a convex compact set or a composition of the two.

Rontsis et al. [141] used COSMO's algorithm with a specialized implementation of the projection function for positive semidefinite constraints. The projection method used approximate matrix eigendecompositions to significantly reduce the projection time, while maintaining all the features of COSMO such as scaling, infeasibility detection and interfaces to linear system solvers. It was demonstrated that this can provide a significant (up to 20x) reduction in solve time.

Moreover, Julia's type abstraction features are used to enable the solver to solve problems of arbitrary floating-point precision. This allows for example the memory usage of the solver for very large problems to be reduced by switching to 32-bit single-precision floating-point format.

We further allow efficient warm starting of the variables x, s, y, which can significantly improve the convergence if good initial guesses are provided. This is often the case in applications where x carries the state of some system that gets optimized in rapid succession, e.g. model predictive control [24].

For large LPs and QPs, the factorisation step in Algorithm 1 is the computational bottleneck. In some applications we are interested in solving very similar versions of the same problem, e.g. a model where just a hyperparameter changes. As one can see in (3.23), the KKT matrix is only dependent on A and P. Consequently, if only the problem vectors b or q change we can reuse the factorisation for the following optimisation attempt to significantly speed up the algorithm for a range of similar problems. The following example illustrates this.

Example 3.8.1 (Pareto-optimal front in portfolio optimisation)**.** We want to rebalance a Markowitz portfolio of n assets with asset allocation vector $x \in \mathbb{R}^n$ to achieve a good balance between expected returns $\mu^\top x$ and expected risk (variance) $x^\top \Sigma x$. Here $\mu \in \mathbb{R}^n$ are the forecasted returns and $\Sigma \in \mathbb{S}^n_+$ represents the covariance matrix of the risk model. The covariance matrix is commonly given in factor risk form, i.e. it can be written as $\Sigma = D + FF^\top$, where D is diagonal and $F \in \mathbb{R}^{n \times k}$ is the factor matrix with $k \ll n$. The initial asset allocation is denoted as x_0. To find

an optimal rebalancing for a provided risk-return parameter γ without short-selling, one can solve the quadratic program

$$\begin{array}{ll} \text{minimize} & x^\top D x + y^\top y - \gamma^{-1}\mu^\top x \\ \text{subject to} & y = F^\top x \\ & \mathbf{1}_n^\top x = \mathbf{1}_n^\top x_0 \\ & x \geq 0. \end{array} \tag{3.37}$$

To find a good risk-return balance this problem is typically solved for many different values of γ. By comparing problem (3.37) to our conic problem format (3.14) one can see that μ, the parameter that is varied, only changes the linear cost term, i.e. the problem data vector q. Consequently, if this problem is solved many times the factorisation can be reused each time.

3.9 Benchmark results

This section presents benchmark results of `COSMO` against the interior-point solver `MOSEK` v9.0 and the accelerated first-order ADMM solver `SCS` v2.1.1. When applied to a quadratic program, `COSMO`'s main algorithm becomes very similar to the first-order QP solver `OSQP`. To test the performance penalty of using a pure Julia implementation against a similar C implementation we also compare our solver against `OSQP` v0.6.0 on QP problems.

We consider a number of problem sets to test different aspects of `COSMO`. The advantage of supporting a quadratic cost function in a conic solver is shown by solving QPs from the Maros and Mészáros QP repository [105] in Section 3.9.1 and SDPs with quadratic objectives in the form of nearest correlation matrix problems in Section 3.9.3. To highlight the advantages of implementing custom constraints, we consider a problem set with doubly-stochastic matrices in Section 3.9.2. The impact of warm starting and factorisation caching in a portfolio backtest scenario is shown in Section 3.9.4.

All the experiments were carried out on a computing node of the University of Oxford ARC-HTC cluster with 16 logical Intel Xeon E5-2560 cores and 64 GB of

DDR3 RAM. All the problems were run using Julia v1.3 and the problems were passed to the solvers via MathOptInterface [92].

To evaluate the accuracy of the returned solution we compute three errors adapted from the DIMACS error measures for SDPs [84]:

$$\epsilon_1 = \frac{\|A_a x - b_a\|_2}{1 + \|b_a\|_2}, \quad \epsilon_2 = \frac{\left\|Px + q - A_a^\top y_a\right\|_2}{1 + \|q\|_2}, \quad \epsilon_3 = \frac{\left|x^\top P x + q^\top x - b_a^\top y_a\right|}{1 + |q^\top x| + |b_a^\top y_a|},$$

where A_a, b_a and y_a correspond to the rows of A, b and y that represent active constraints. This is to ensure meaningful values even if the problem contains inactive constraints with very large, or possibly infinite, values b_i. The maximum of the three errors for each problem and solver is reported in the results below.

We configured `COSMO`, `MOSEK`, `SCS` and `OSQP` to achieve an accuracy of $\epsilon = 10^{-3}$. We remark that higher accuracies $\epsilon \leq 10^{-5}$ would benefit `MOSEK`, the only interior-point solver, more than the other solvers as it typically requires only a few additional iterations to improve the accuracy from $\epsilon = 10^{-3}$. We set the maximum allowable solve time for the Maros and Mészáros problems to 5 min and to 30 min for the other problem sets. All other solver parameters were set to the solvers' standard configurations. `COSMO` uses a Julia implementation of the `QDLDL` solver to factor the quasi-definite linear system. Similarly, we configured `SCS` to use its default direct solver `QDLDL` for the linear system.

3.9.1 Maros and Mészáros QP test set

The Maros and Mészáros test problem set [105] is a repository of challenging convex QP problems that is widely used to compare the performance of QP solvers. Table 5.2 shows the ranges of values for the number of variables, number of constraints, and nonzeros in the constraint matrix among the problems. As comparison metrics we compute the failure rate, the number of fastest solve times and the normalized shifted geometric mean for each solver. The shifted geometric mean is more robust against large outliers (compared to the arithmetic mean) and against small

outliers (compared to the geometric mean) and is commonly used in optimisation benchmarks; see [112, 148]. The shifted geometric mean $\mu_{g,s}$ is defined as:

$$\mu_{g,s} \coloneqq \sqrt[n]{\prod_p (t_{p,s} + \text{sh})} - \text{sh} \tag{3.38}$$

with total solver time $t_{p,s}$ of solver s and problem p, shifting factor sh and size of the problem set n. In the reported results a shifting factor of $sh = 10$ was chosen and the maximum allowable time $t_{p,s} = 300\,\text{s}$ was used, after which solver s was deemed to have failed on problem p. Lastly, we normalize the shifted geometric mean for solver s by dividing by the geometric mean of the fastest solver. The failure rate $f_{r,s}$ is given by the number of unsolved problems compared to the total number of problems in the problem set. As unsolved problems we count instances where the algorithm does not converge within the allowable time or fails during the setup or solve phase. Table 3.1 shows the normalized shifted geometric mean and the failure rate for each solver. Also shown is the number of cases where solver s was the fastest solver.

Table 3.1: Normalized shifted geometric mean, number of fastest solve time, and failure rates for solvers tested on the Maros and Mészáros QP test set.

	OSQP	COSMO	MOSEK	SCS
Normalized shifted geometric mean	1.000	1.169	1.897	75.015
Number of fastest solve times	75	27	33	0
Failure rates $f_{r,s}$ [%]	4.444	5.185	10.370	83.704

`OSQP` shows the best performance in terms of lowest failure rates, number of fastest solves and in the shifted geometric mean of solve times. `COSMO` follows very closely. The shifted geometric mean of `MOSEK` seems to suffer from a higher failure rate compared to `OSQP`/`COSMO`, and `SCS` fails on a large number of problems. The higher failure rate could be due to the necessary transformation into a second-order-cone problem.

For this problem set of QPs `COSMO`'s algorithm reduces, with some minor differences, to the algorithm of `OSQP`. Consequently, this benchmark is useful to evaluate the

performance penalty that COSMO pays due to its implementation in the higher-order language Julia. The results in Table 3.1 show that the performance difference is very small. This can also be seen by looking at the solve times of each solver for increasing problem dimension, as shown in Figure 3.1.

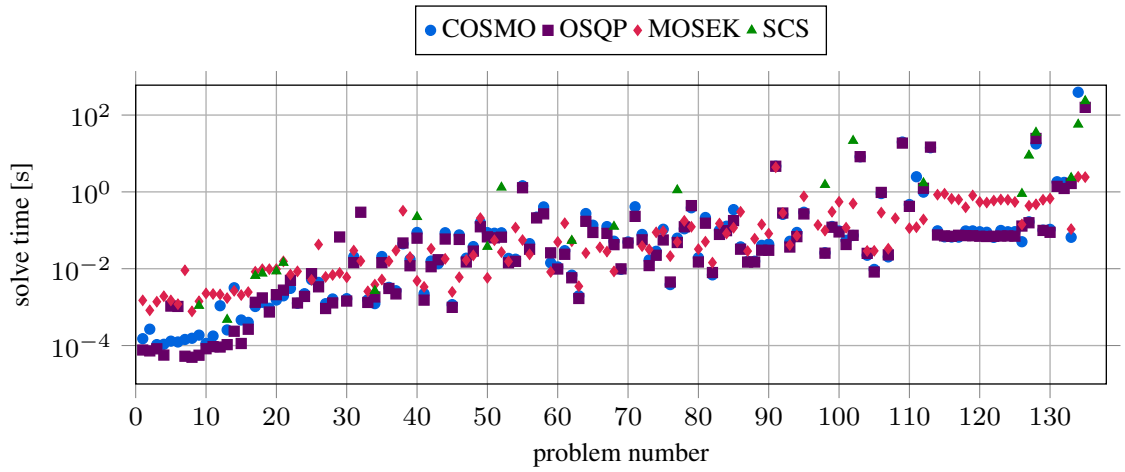


Figure 3.1: Solve time of benchmarked solvers for problems of the Maros and Mészáros QP problem set. Only problem results classified as solved are shown. The problems are ordered by increasing number of non-zeros in the constraint matrix.

COSMO and OSQP have very similar solve times, aside from very small problems that are solved in under 10^{-5} s to 10^{-4} s. This difference is primarily due to overheads incurred from features in our Julia implementation that support more than one constraint type during problem setup. The marginally better resulting performance of OSQP for the smallest problems in the test set is the reason that OSQP is the fastest solver in a larger number of cases in Table 3.1.

3.9.2 Custom convex cones

In many cases writing a custom solver algorithm for a particular problem can be faster than using available solver packages if a particular aspect of the problem structure can be exploited to speed up parts of the computations. As mentioned earlier, COSMO supports user customisation by allowing the definition of new convex cones. This is useful if constraints of the problem can be more naturally expressed using this new cone and a fast projection method for the cone exists. A fast

specialized projection method in an ADMM framework has for example been used by Barman et al. [11] to solve the error-correcting code decoding problem.

To demonstrate the advantage of custom convex cones, consider the problem of finding the doubly stochastic matrix that is nearest, in the Frobenius norm, to a given symmetric matrix $C \in \mathbb{S}^n$. Doubly stochastic matrices are used for instance in spectral clustering [176] and matrix balancing [135]. A specialized algorithm for this problem type has been recently discussed by [140]. Doubly stochastic matrices have the property that all rows and columns each sum to one and all entries are nonnegative. The nearest doubly stochastic matrix X can be found by solving the optimisation problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|X - C\|_F^2 \\ \text{subject to} & X_{ij} \geq 0 \\ & X\mathbf{1} = \mathbf{1} \\ & X^\top \mathbf{1} = \mathbf{1}, \end{array} \tag{3.39}$$

with symmetric real matrix $C \in \mathbb{S}^n$ and decision variable $X \in \mathbb{R}^{n \times n}$. This problem can be solved as a QP in the following form using equality and inequality constraints:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}(x^\top x - 2c^\top x + c^\top c) \\ \text{subject to} & \begin{bmatrix} \mathbf{1}_n^\top \otimes I_n \\ I_n \otimes \mathbf{1}_n^\top \\ -I_{n^2} \end{bmatrix} x + s = \begin{bmatrix} \mathbf{1}_{2n} \\ \mathbf{1}_{2n} \\ \mathbf{0}_{n^2} \end{bmatrix} \\ & s \in \{0\}^{4n} \times \mathbb{R}_+^{n^2}, \end{array}$$

with $x = \text{vec}(X)$ and $c = \text{vec}(C)$ However, the problem can be written in a more compact form by using a custom projection function to project the matrix iterate onto the affine set of matrices $\mathcal{C}_{\sum}$, whose rows and columns each sum to one. In general the projection of vector $s \in \mathbb{R}^n$ onto the affine set $\mathcal{C}_a = \{s \in \mathbb{R}^n \mid As = b\}$ is given by

$$\Pi_{\mathcal{C}_a}(s) = s - A^\top \left(AA^\top\right)^{-1} (As - b),$$

where A is assumed to have full rank. In the case of $\mathcal{C}_a = \mathcal{C}_{\sum}$ we can exploit the fact that the inverse of $AA^\top$ can be efficiently computed.

The projection of a symmetric matrix $S \in \mathbb{S}^n$ onto the set of matrices where the sum of rows and columns each equal to one can be written as the optimisation

problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\left\|s_p - s\right\|_F^2 \\ \text{subject to} & As_p = \mathbf{1}_{2n-1}, \end{array} \tag{3.40}$$

where $s = \text{vec}(S)$ is the vectorized matrix, s_p is the projected vector and A is partitioned such that

$$A = \begin{bmatrix} A_r & A_c \end{bmatrix}^\top,$$

where A_r is used to constrain the rows of S and A_c is used to constrain the columns of S:

$$A_r = \mathbf{1}_n^\top \otimes I_n \quad A_c = \begin{bmatrix} I_{n-1} \otimes \mathbf{1}_n^\top & \mathbf{0}_{n-1\times n} \end{bmatrix}.$$

Notice that A_c is a $(n-1)\times n$ matrix because the redundant constraint on the last column was removed. The KKT optimality conditions for problem (3.40) are

$$\begin{aligned} As_p &= \mathbf{1}_{2n-1}, \\ s_p - s + A^\top \eta &= 0, \end{aligned} \tag{3.41}$$

with dual variable η. Eliminating s_p and solving for η yields

$$\eta = (AA^\top)^{-1}(As - \mathbf{1}_{2n-1}). \tag{3.42}$$

The projected vector s_p can be recovered from (3.41):

$$s_p = s - A^\top \eta. \tag{3.43}$$

It turns out that the inverse of $AA^\top$ in (3.42) can be efficiently computed without a factorisation. To see this, form $AA^\top$ and write (3.42) as:

$$\begin{bmatrix} nI_n & \mathbf{1}_n\mathbf{1}_{n-1}^\top \\ \mathbf{1}_{n-1}\mathbf{1}_n^\top & nI_{n-1} \end{bmatrix} \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \text{ with } \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = As - \mathbf{1}_{2n-1}, \tag{3.44}$$

where η and the right-hand side were partitioned in such a way that $\eta_1, r_1 \in \mathbb{R}^n$ and $\eta_2, r_2 \in \mathbb{R}^{n-1}$. Eliminating η_1 from (3.44), yields

$$\left(I_{n-1} - \tfrac{1}{n}\mathbf{1}_{n-1}\mathbf{1}_{n-1}^\top\right)\eta_2 = \tfrac{1}{n}\left(r_2 - \tfrac{1}{n}\mathbf{1}_{n-1}\mathbf{1}_n^\top r_1\right).$$

The vector η_2 is determined by applying the Sherman-Morrison formula to compute the inverse of the matrix on the left:

$$\eta_2 = \tfrac{1}{n}\left(I_{n-1} + \mathbf{1}_{n-1}\mathbf{1}_{n-1}^\top\right)\left(r_2 - \tfrac{1}{n}\mathbf{1}_{n-1}\mathbf{1}_n^\top r_1\right).$$

The vector η_1 is then computed by substituting η_2 into the upper equation in (3.44):

$$\eta_1 = \tfrac{1}{n}\left(r_1 - \mathbf{1}_n \mathbf{1}_{n-1}^\top \eta_2\right).$$

Having computed the elements of the dual variable η, the projected vector s_p is obtained by solving (3.43).

The necessary steps to carry out the projection are summarized in Algorithm 4.

Algorithm 4: Projection of $s \in \mathbb{R}^n$ onto $C_\sum$

$A_r = \mathbf{1}_n^\top \otimes I_n \quad A_c = \begin{bmatrix} I_{n-1} \otimes \mathbf{1}_n^\top & \mathbf{0}_{n-1\times n} \end{bmatrix}$;
$A = \begin{bmatrix} A_r & A_c \end{bmatrix}^\top$;
$r = [r_1, r_2]^\top = As - \mathbf{1}_{2n-1}$;
$\eta_2 = \tfrac{1}{n}\left(I_{n-1} + \mathbf{1}_{n-1}\mathbf{1}_{n-1}^\top\right) \cdot \left(r_2 - \tfrac{1}{n}\mathbf{1}_{n-1}\mathbf{1}_n^\top r_1\right)$;
$\eta_1 = \tfrac{1}{n}\left(r_1 - \mathbf{1}_n \mathbf{1}_{n-1}^\top \eta_2\right)$;
$\eta = [\eta_1, \eta_2]^\top$;
$\Pi_{\mathcal{C}_\sum}(s) = s - A^\top \eta$;

Notice that Algorithm 4 can be implemented efficiently without ever assembling and storing A and $\mathbf{1}\mathbf{1}^\top$. By using the custom convex set $\mathcal{C}_\sum$ and the corresponding projection function, problem (3.39) can now be rewritten as

$$\begin{array}{ll} \text{minimize} & (1/2)(x^\top x - 2c^\top x + c^\top c) \\ \text{subject to} & \begin{bmatrix} -I_{n^2} \\ -I_{n^2} \end{bmatrix} x + s = \mathbf{0}_{2n^2} \\ & s \in \mathcal{C}_\sum \times \mathbb{R}_+^{n^2}. \end{array} \tag{3.45}$$

The sparsity pattern of the new constraint matrix A consists of two diagonals and the number of non-zeros reduces from $3n^2$ to $2n^2$. We expect this to reduce the initial factorisation time of the linear system in (3.23) as well as the forward- and back-substitution steps.

Figure 3.2 shows the total solve time of all the solvers for problem (3.39) with randomly generated dense matrix C with C_{ij} uniformly distributed on the interval $[0, 1]$, i.e. $C_{ij} \sim \mathcal{U}(0, 1)$, and increasing matrix dimension. Additionally, we show the solve time for `COSMO` in the problem form (3.39) and with a specialized custom set and projection function as in (3.45). It is not surprising that `COSMO` and `OSQP`

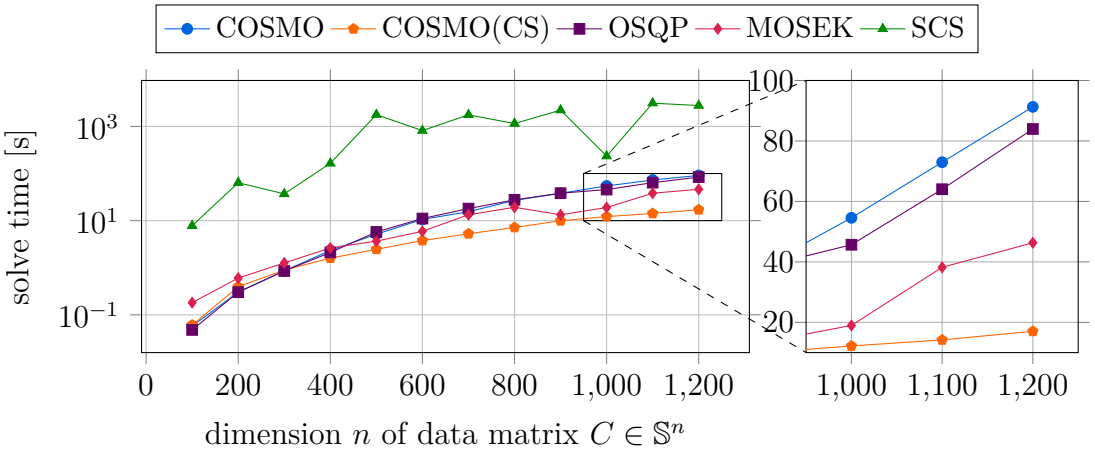


Figure 3.2: Solve time of benchmarked solvers for increasing problem size of doubly stochastic matrix problems. The orange line shows the solve time of `COSMO(CS)` with a custom convex set and projection function.

scale in the same way for this problem type. `MOSEK` is slightly slower for smaller problem dimensions and overtakes `COSMO`/`OSQP` for problems of dimensions $n \geq 500$. This might be due to fact that `MOSEK` uses a faster multithreaded linear system solver while `OSQP`/`COSMO` relied in these tests on the single-threaded solver `QDLDL`. The longer solve time of `SCS` is due to slow convergence of the algorithm for this problem type. Furthermore, when the problem is solved with a custom convex set as in (3.45) `COSMO`(CS) is able to outperform all other solvers. Table 3.2 shows the total solve time and the factorisation time of the two versions of `COSMO` for small, medium and large problems. As predicted the lower solve time can be mainly attributed to the faster factorisation time.

Table 3.2: Solve times and factorisation times of `COSMO` and `COSMO(CS)` for small, medium and large doubly stochastic matrix problems.

	Factorisation time (s)		Solve time (s)	
n	COSMO	COSMO(CS)[1]	COSMO	COSMO(CS)[1]
100	0.018	0.006	0.058	0.061
400	0.995	0.138	2.329	1.596
800	13.551	0.552	26.932	7.163
1200	52.717	1.322	91.278	17.032

[1] solving (3.45) with a custom convex set $\mathcal{C}_{\sum}$ and projection function

3.9.3 Nearest correlation matrix

Consider the problem of projecting a matrix C onto the set of correlation matrices, i.e. real symmetric positive semidefinite matrices with diagonal elements equal to 1. This problem is for example relevant in portfolio optimisation [81]. The correlation matrix of a stock portfolio might lose its positive semidefiniteness due to noise and rounding errors of previous data manipulations. Consequently, it is of interest to find the nearest correlation matrix X to a given data matrix $C \in \mathbb{R}^{n \times n}$. The problem is given by:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|X - C\|_F^2 \\ \text{subject to} & X_{ii} = 1, \quad i = 1, \dots, n \\ & X \in \mathbb{S}_+^n. \end{array}$$

In order to transform the problem into the standard form (3.14) used by `COSMO`, C and X are vectorized and the objective function is expanded:

$$\begin{array}{ll} \text{minimize} & (1/2)(x^\top x - 2c^\top x + c^\top c) \\ \text{subject to} & \begin{bmatrix} E \\ -I \end{bmatrix} x + s = \begin{bmatrix} \mathbf{1}_n \\ \mathbf{0}_{n^2} \end{bmatrix} \\ & s \in \{0\}^n \times \mathcal{S}_+^n, \end{array}$$

with $c = \text{vec}(C) \in \mathbb{R}^{n^2}$ and $x = \text{vec}(X) \in \mathbb{R}^{n^2}$. Here $E \in \mathbb{R}^{n \times n^2}$ is a matrix that extracts the n diagonal entries X_{ii} from its vectorized form x.

For the benchmark problems we randomly sample the data matrix C with entries $C_{i,j} \sim \mathcal{U}(-1, 1)$ from a uniform distribution. Figure 3.3 and Table 3.3 show the benchmark results for increasing matrix dimension n. Unsurprisingly, the first-order methods `SCS` and `COSMO` outperform the interior-point solver `MOSEK` for these large SDPs. Furthermore, for larger problems the solve times of `COSMO` and `SCS` scale in a similar way. `COSMO` seems to benefit from directly supporting the quadratic objective term in the problem while `SCS` has to transform it into an additional second-order cone constraint. This increases the factorisation time and the projection time.

3.9.4 Portfolio backtest

An advantage of ADMM is that it can be easily warm started. Moreover, if many similar parametric problems are solved and the changing parameters do not

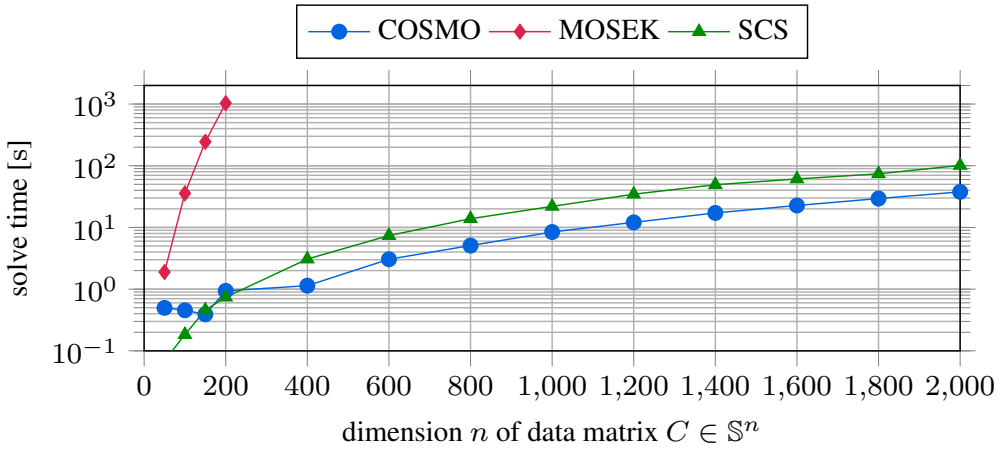


Figure 3.3: Solve time of benchmarked solvers for increasing problem size of nearest correlation matrix problems. The results for `MOSEK` are shown until they exceeded the time limit of 30 min.

Table 3.3: Benchmark results for nearest correlation matrix problems.

	Solve time (s)			Iterations			Max error[2]		
n	COSMO	MOSEK	SCS	COSMO	MOSEK	SCS	COSMO	MOSEK	SCS
50	0.5	1.89	**0.07**	25	4	20	2.05×10^{-5}	1.35×10^{-3}	6.18×10^{-7}
100	0.45	35.68	**0.18**	25	4	20	2.02×10^{-5}	6.26×10^{-3}	4.24×10^{-6}
150	**0.39**	244.35	0.46	25	4	20	5.15×10^{-5}	8.61×10^{-3}	9.00×10^{-6}
200	0.94	1032.25	**0.74**	25	4	20	7.59×10^{-5}	3.64×10^{-3}	8.41×10^{-5}
400	**1.14**	$***^{\dagger}$	3.07	25	***	20	2.62×10^{-4}	***	8.52×10^{-5}
600	**3.05**	$***^{\dagger}$	7.38	25	***	20	1.53×10^{-4}	***	1.01×10^{-4}
800	**5.08**	$***^{\dagger}$	13.85	25	***	20	4.03×10^{-4}	***	1.07×10^{-4}
1000	**8.43**	$***^{\dagger}$	21.94	25	***	20	8.11×10^{-4}	***	1.40×10^{-4}
1200	**12.1**	$***^{\dagger}$	34.58	25	***	20	1.02×10^{-3}	***	1.73×10^{-4}
1400	**17.19**	$***^{\dagger}$	49.23	25	***	20	1.02×10^{-3}	***	1.93×10^{-4}
1600	**22.69**	$***^{\dagger}$	61.07	25	***	20	8.00×10^{-4}	***	2.21×10^{-4}
1800	**29.41**	$***^{\dagger}$	74.0	25	***	20	5.14×10^{-4}	***	2.43×10^{-4}
2000	**37.74**	$***^{\dagger}$	101.18	25	***	20	3.31×10^{-4}	***	2.57×10^{-4}

[2] $\max\{\epsilon_1, \epsilon_2, \epsilon_3\}$; † time limit reached

affect the matrices A or P in problem format (3.14), the factorisation of the KKT matrix (3.23) can be reused. To demonstrate the computational benefits of these two properties we simulate a portfolio backtest. We use the Markowitz portfolio model described in (3.37) to model the allocation of assets over a number of trading days. We further model transaction costs incurred through buying and selling of the assets, see the survey paper [30]. The following model for the transaction costs $c_i(x_i - x_i^0)$ is used:

$$c_i(x_i - x_i^0) = a_i \left| x_i - x_i^0 \right| + b_i \left| x_i - x_i^0 \right|^{\frac{3}{2}},$$

where the cost parameters $a_i, b_i > 0$ are set by the trader. For each asset i the first term represents the bid-ask spread and broker fees and the second term models the impact that the trade has on the market. To represent the function we define slack variables $s_i = |x_i - x_i^0|$ to model the volume change and t_i to represent the market impact cost. Assuming that one of the assets is risk-free, we can use a power cone to represent the market impact cost term for each asset:

$$t_i^{\frac{2}{3}} 1^{\frac{1}{3}} \geq \left| x_i - x_i^0 \right| \Leftrightarrow (t_i, 1, x_i - x_i^0) \in \mathcal{K}_{\text{pow},\frac{2}{3}}.$$

The final problem is given by

$$\begin{array}{ll} \text{minimize} & x^\top D x + y^\top y - \gamma^{-1} \mu^\top x \\ \text{subject to} & y = F^\top x \\ & \mathbf{1}_n^\top x + a \mathbf{1}_n^\top s + b \mathbf{1}_n^\top t = \mathbf{1}_n^\top x_0 \\ & x \geq 0,\ s \geq x_0 - x,\ s \geq x - x_0 \\ & (t_i, 1, x_i - x_i^0) \in \mathcal{K}_{\text{pow},\frac{2}{3}}, \quad i = 1, \ldots, n. \end{array}$$

For the backtest we generate problems using $k = 100, 200, \ldots, 500$ factors and $n = 100k$ assets. The risk matrices $F \in \mathbb{R}^{n \times k}$ and $D \in \mathbb{R}^{n \times n}$ are randomly generated with $F \sim \mathcal{N}(0, 1)$ and 50 % nonzero elements as well as D with diagonal elements $D_{ii} \sim \mathcal{U}[0, \sqrt{k}]$. The initial mean return vector was generated with $\mu_i \sim \mathcal{N}(0, 1)$. We choose $\gamma = 1$ for the risk-return parameter and $a = 10^{-3}$ and $b = 0.05$ for the transaction cost parameters of each asset. For the backtest we consider 60 consecutive trading days where we assume a constant risk model but daily changing expected returns according to

$$\mu_i^k \sim 0.9 \mu_i^{k-1} + \mathcal{N}(0, 0.1).$$

Table 3.4 shows the total solve time, the total factorisation time for the KKT matrix, and the mean number of iterations for all 60 problems for both the solver with and without warm starting and factorisation caching.

The warm started method with factorisation caching achieves an increasing improvement factor of 5 to 8. The mean number of the iterations for each problem is 4 to 5x lower than for the method without warm starting.

Table 3.4: Accumulated solve and factorisation time (s), and mean iterations of COSMO with and without warm starting and factorisation caching for Portfolio backtests.

factors k	solve time[1]	solve time (ws)[2]	improve-ment	factor time[3]	factor time (ws)	mean iter	mean iter (ws)
100	240.20	45.90	5.23	18.50	0.31	110.00	27.08
200	777.02	128.78	6.03	111.63	1.74	120.00	30.00
300	1517.90	231.14	6.57	259.90	5.16	127.92	27.92
400	2749.47	351.60	7.82	614.33	8.67	123.75	28.75
500	4236.87	490.54	8.64	1029.78	17.52	132.08	27.92

[1] total solve time (s);
[2] (ws): COSMO with solution warm starting and factorisation caching;
[3] total factorisation time (s);

3.10 Conclusions

In this chapter we described the first-order solver COSMO and the ADMM algorithm on which it is based. The solver combines direct support of quadratic objectives with the ability to handle a large number of conic constraints. This allows the algorithm to perform well on large standard quadratic programs as well as large conic problems such as semidefinite programs. Moreover, we showed the advantage of the modularity of the ADMM-algorithm and our implementation by solving a problem that benefitted from custom convex constraints and projections. The performance of the solver was illustrated on a number of benchmark problems that challenge different aspects of modern solvers. Our implementation in the Julia programming language allows us rapid development and testing of ideas. Further performance gains are achievable for example by using Julia's first-class support for GPU programming to allow the solver to run on GPUs.

4

Chordal decomposition of sparse semidefinite programs

Contents

4.1 Introduction

In this chapter we revisit and further explore the solution of large semidefinite programs that were introduced in Section 2.5.3. For the rest of the chapter we refer to the *primal form SDP*

$$\begin{array}{ll} \text{minimize} & \langle C, X \rangle \\ \text{subject to} & \langle A_k, X \rangle = b_k, \quad k = 1, \ldots, m \\ & X \in \mathbb{S}^n_+, \end{array} \tag{4.1}$$

with variable X and coefficient matrices $A_i, C \in \mathbb{S}^n$. The corresponding *dual form SDP* is given by

$$\begin{array}{ll} \text{maximize} & b^\top y \\ \text{subject to} & \sum_{k=1}^{m} A_k y_k + S = C \\ & S \in \mathbb{S}^n_+, \end{array} \tag{4.2}$$

with dual variable $y \in \mathbb{R}^m$ and slack variable S. After it was shown at the end of the 1980s that polynomial time algorithms for SDPs exist [4, 76, 83, 119], SDPs became an active research topic in the 1990s. A good overview on the recent history and applications is provided by Wolkowicz et al. [169]. SDPs are often used in combinatorial optimisation to derive convex relaxations of NP-hard problems. This is due to the existence of useful results that provide bounds on the quality of the relaxation [65]. SDPs especially in linear matrix inequality form also play an important role in control theory [26, 161], and they are used for robust controller synthesis and to analyse nonlinear and time-varying systems. Moreover, SDPs are used to derive bounds for stochastic optimisation problems with applications in finance, economics, and operations research [17, 18]. The recently growing interest in machine learning has also led to a new range of applications for SDPs, e.g. neural network verification against adversarial attacks [134], kernel matrix learning [91], sparse PCA [35], and graph clustering [40].

IPMs for SDPs have existed since the end of the 1980s and have been shown to provide robust and accurate solutions for small to medium size problems [171]. However, the recent trend to use models based on large quantities of data leads to SDPs whose dimensions challenge established solver algorithms. There are many

actively researched approaches to deal with this challenge. The first approach is to use first-order methods (FOMs). FOMs typically trade-off moderate accuracy solutions for a lower per-iteration computational cost and can therefore handle large problems more easily. The lower accuracy is not a concern in many applications where the input data is noisy or the SDP is itself a relaxation of a computationally intractable non-convex problem.

Another approach is to replace the positive semidefinite variable X in the SDP with a rectangular matrix R according to the factorisation $X = RR^\top$. Ideally the rank of R is chosen to be low to increase the computational benefit while at the same time maintaining a good approximation to the optimal solution of the original problem. The resulting problem has nonconvex objective and constraints. Burer and Monteiro [31] present an algorithm that solves the transformed nonconvex problem using the augmented Lagrangian method. For the minimization of the inner problem they use a limited memory BFGS-method [96] to avoid expensive factorisations of the Hessian. Yurtsever et al. [175] recently developed a similar approximate algorithm which is designed for very large and weakly constrained SDPs whose solutions are assumed to be low-rank. The algorithm is designed to keep the storage cost at a minimum. They approximate the solution of the inner problem in the augmented Lagrangian which requires only the computation and storage of the minimum eigenvector at each step. At each step this eigenvector is used to update the approximate PSD variable $X_k \in \mathbb{S}^n$. To further reduce storage cost they only store a sketch $S \in \mathbb{R}^{n\times R}$ of X_k with $R \ll n$ and reconstruct the solution X^* at the end. The total storage cost for this method is $\mathcal{O}(d + nR)$ where d is the number of equality constraints.

Another approach that is further discussed in this chapter is to exploit sparsity in the problem data. If the coefficient matrices A, C exhibit an aggregate sparsity structure represented by a chordal graph, then the original primal and dual forms in (4.1) and (4.2) can be decomposed into equivalent problems with a number of smaller positive semidefinite constraints. These equivalent problems involve only positive semidefinite constraints on the nonzero blocks of the sparsity pattern, which

can lead to a significant reduction in the dimension of each constraint, thereby reducing solve time. The decomposition comes at the expense of additional equality constraints.

The complete subgraphs, called cliques, in the sparsity graph determine the computational benefit of the decomposition. One has the choice to combine certain cliques to improve the decomposition and speed up the overall algorithm. Finding the optimal clique merging strategy is a difficult problem and depends on the solver algorithm and the hardware.

4.1.1 Related Work

First-order methods that can handle semidefinite programs have been developed by O'Donoghue et al. [122] and Sun et al. [150]. The ability to exploit chordal sparsity in SDPs was recognized by Fukuda et al. [53] and integrated into an IPM by Andersen et al. [5] and Nakata et al. [116]. This demonstrated substantial computational improvement in finding the solution of formerly intractable SDPs. The same decomposition technique can also be applied to solve structured SDPs with FOMs. Kalbat and Lavaei [85] developed a parallelizable algorithm for arbitrary decomposable SDPs. Moreover, ADMM was used to solve the decomposed formulation of the relaxed optimal power flow problem [102]. Recently, Zheng et al. [181] developed the ADMM-based solver `CDCS`, written in MATLAB. This work uses a homogeneous self-dual embedding of a primal-dual pair to detect infeasibility and can automatically exploit sparsity in PSD constraints.

A trade-off has to be made between the sizes of the blocks and the number of additional equality constraints. This becomes especially important when chordal decomposition is used with IPMs because, compared to FOMs, their performance is more strongly reliant on the number of additional equality constraints. Some methods to systematize clique merging step for IPMs have been proposed. However, as the relationship of the clique sizes and the number of equality constraints on the core linear algebra operations of IPM algorithms is complex, they mostly rely

on heuristics and intuition from supernode partitioning in multifrontal methods [9, 136].

Nakata et al. [116] suggest traversing the clique tree, the edge set of which is a subset of all clique pairs that have overlapping entries. For each edge in the tree Nakata et al. merge the corresponding cliques if the size of the overlapping entries relative to the cardinality of the individual cliques is higher than some predefined threshold. The threshold is chosen heuristically to balance the block sizes and the number of additional equality constraints. The methods are implemented in the `SparseCoLO` package [52]. Similarly, Sun et al. [150] suggest traversing the clique tree and merging cliques if the number of additional fill-in, additional elements that must be treated as numerical (rather than structural) zeros, and the cardinality of the supernodes are below certain thresholds. This approach is implemented in the `CHOMPACK` package [6]. A limitation of existing methods is that they rely on heuristic parameters designed for a specific interior point implementation. Furthermore, they consider only pairs of cliques that are adjacent in the clique tree.

4.1.2 Outline

Section 4.2 introduces the necessary graph concepts for this chapter. Section 4.3 shows the application of two decomposition theorems to the primal and dual SDP forms (4.1) and (4.2). A new clique graph-based merging strategy is introduced and compared with existing clique merging strategies in Section 4.4. In Section 4.5 we discuss how the merging strategy is implemented and calibrated for a first-order method. Numerical results that demonstrate the large computational advantages of chordal decomposition are shown in Section 4.6. We further compare the performance of different clique merging strategies for the same solver algorithm. Section 4.7 concludes this chapter.

4.1.3 Contributions

1. We analyse and compare existing methods to merge the cliques of the sparsity pattern after an initial decomposition to reduce the per-iteration time of the

solver algorithm. Existing methods were designed for interior-point methods and rely heavily on heuristics. We present a novel clique merging algorithm that utilizes the reduced clique graph of the sparsity structure to make effective merge decisions, considering a large number of merge possibilities. We subsequently show that this algorithm can be used effectively in combination with FOMs due to their simple relationship between per-iteration computation time and clique dimension.

2. We incorporate chordal decomposition and clique merging into the first-order solver `COSMO`. We demonstrate that our approach can speed up the per-iteration projection time for most real-world sparsity patterns in our benchmark sets by a factor of up to 2 or 3.

3. Benchmarks on very large sparse SDPs against `MOSEK` and `SCS` show orders of magnitude improvement in solve time.

4.2 Background

In this section we define some graph-related concepts that are used throughout this chapter. In particular we show how the sparsity of a matrix can be described by a graph and then analyzed using concepts from graph theory. As we will see later this relationship is the basis for the decomposition results in Section 4.3. A good overview on this topic is given in the survey paper by Vandenberghe and Andersen [159], from which we will adopt the graph-related notation. Another good overview is given by Blair and Peyton [21]. When illustrating graphs in this chapter we will display a vertex v_k simply with a circled number $\textcircled{k}$ in the figure.

4.2.1 Chordal Graphs

We define an *undirected graph* $G(V, E)$ as a pair of a finite *vertex set* V and an *edge set* of unordered pairs $E = \{\{v, u\} \mid v, u \in V, v \neq u\} \subseteq V \times V$. Elements of $v_i \in V$ are called graph *vertices* and elements $\{v, u\} \in E$ are

called graph *edges*. Notice that the definition of the edge set excludes self-loops and does not distinguish between $\{v, u\}$ and $\{u, v\}$. Consider the example graph shown in Figure 4.1(a). It has vertex set $V = \{1, 2, 3, 4, 5\}$ and edge set $E = \{\{1, 2\}, \{1, 3\}, \{1, 6\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}, \{5, 6\}\}$. By taking a subset of the vertices $V' \subseteq V$ of an undirected graph we can induce a *subgraph* $G(V', E(V'))$ where the edge set $E(V')$ only contains the edges of E that link vertices in V', i.e. $E(V') = \{\{v, u\} \in E \mid v, u \in V'\}$.

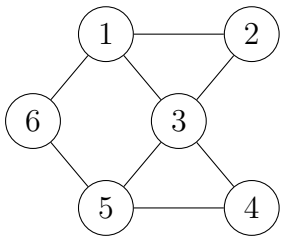
Two vertices u, v are called *adjacent* if there exists an edge between them, i.e. $\{u, v\} \in E$. The set of adjacent vertices of a vertex v is called the *neighbourhood* $\text{adj}(v) = \{u \mid \{v, u\} \in E\}$ and its cardinality is called *vertex degree* $\deg(v) = |\text{adj}(v)|$. Vertex 1 in Figure 4.1(b) has a neighbourhood of degree $\deg(1) = |\{1, 2, 6\}| = 3$. We further define a *path* between two vertices v and $w \neq v$ as a sequence of vertices $v = v_0, v_1, \ldots, v_n = w$ with $\{v_i, v_{i+1}\} \in E$. Consequently, two vertices are *connected* if one can construct at least one path between them. If all vertices of a graph are pairwise adjacent, i.e. $E = \{\{v, u\} \mid v, u \in V,\ v \neq u\}$, the graph is called *complete*.

When algorithms operate on graphs it is useful to have a notion of order among the graph vertices. An undirected graph $G(V, E)$ with n vertices can be ordered by assigning consecutive numbers $1, \ldots, n$ to the vertices. Using the ordering $\sigma \colon \{1, 2, \ldots, n\} \to V$, we define an *ordered graph* $G_\sigma(V, E, \sigma)$ as an undirected graph with an associated ordering. The inverse of the ordering $\sigma^{-1}(v)$ returns the index of the vertex within the ordering. An important ordering that is used in many numerical algebra routines and that can be found for some graphs is the *perfect elimination ordering*. To define this ordering we make use of the concept of a *simplicial* vertex, which is a vertex v whose neighbourhood $\text{adj}(v)$ is complete.

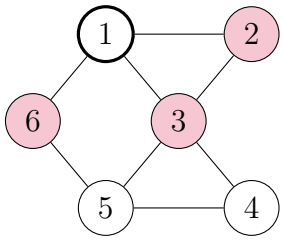
Definition 4.2.1 (Perfect elimination ordering)**.** An ordering σ of an undirected graph $G(V, E)$ is a *perfect elimination ordering* if each vertex $v_i = \sigma(i)$ is a simplicial vertex in the subgraph induced by the vertices $\{v_i, v_{i+1}, \ldots, v_n\}$.

A perfect elimination ordering for a graph is shown in Figure 4.1(c). From the definition one can construct a simple recursive algorithm to compute a perfect

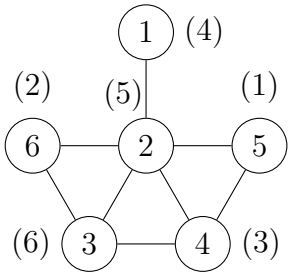
elimination ordering when one exists: For each order $i = 1, \ldots, n$, assign $\sigma(i)$ to any simplicial vertex in the unvisited graph $G(V \setminus \{\sigma(1), \ldots, \sigma(i-1)\})$. Given a



(a) Undirected graph.



(b) Neighbourhood of vertex 1.



(c) A graph with a perfect elimination ordering (shown in brackets).

Figure 4.1

valid ordering we extend the definition of vertex degree and vertex neighbourhood.

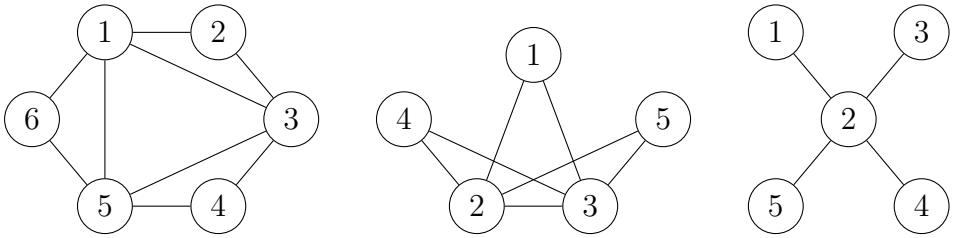
Definition 4.2.2 (Higher degree of a vertex)**.** The *higher degree of a vertex* $\deg^+(v)$ is the cardinality of the set of neighbours with a higher ordering than v, the *higher neighbourhood*, i.e. $\deg^+(v) = \left|\text{adj}^+(v)\right|$, where $\text{adj}^+(v) = \{u \in V \mid \sigma^{-1}(u) > \sigma^{-1}(v)\}$.

In Figure 4.1(c) the higher neighbourhood of vertex 4 is $\text{adj}^+(4) = \{2, 3\}$ so $\deg^+(4) = 2$.

The concept of perfect elimination ordering is closely related to a particular class of graphs known as *chordal graphs* that we will introduce in the following. First, we define a *cycle* inside a graph of length k as a path of k distinct edges joining a sequence of vertices $\{v_1, v_2, \ldots, v_k\}$ such that $\{v_1, v_k\} \in E$ and $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, k-1$. A *chord*, or "one-edge-shortcut", is an edge that joins two non-adjacent vertices in a cycle. Using the concepts of cycles and chords we define the following important class of graphs.

Definition 4.2.3 (Chordal graph)**.** An undirected graph $G(V, E)$ is called *chordal* if every cycle in the graph of length greater than three has at least one chord.

Examples of chordal graphs are shown in Figure 4.2. The graph shown in Figure 4.1(a) is not chordal since it has a chordless cycle $(1 - 3 - 5 - 6)$.



(a) Chordal graph **(b)** $(3, 2)$-fan graph **(c)** 5-star graph

Figure 4.2: Examples of chordal graphs.

Notice that this definition allows only triangular cycles in the graph, which led to the alternative term *triangulated graph*. Chordal graphs have the following important property.

Theorem 3 ([159, Theorem 4.1])**.** *A graph $G(V, E)$ is chordal if and only if G has a perfect elimination ordering.*

This property allows efficient traversal of chordal graphs and is one of the reasons chordal graphs are popular and have been studied since the 1950s. Some advantages of chordal graphs are:

- Many difficult combinatorial problems that are NP-hard for general graphs can easily be solved in polynomial time for chordal graphs, e.g. finding all maximal cliques in the graph [142]. Another example is the problem of finding an optimal vertex colouring, i.e. checking whether a graph can be coloured with k colours such that no adjacent vertices have the same colour. For chordal graphs this can be done using a simple greedy algorithm [59].
- Chordal graphs can be efficiently recognized by trying to determine a perfect elimination ordering. Two important graph-ordering algorithms are lexicographical breadth-first search [142] and the simpler maximum cardinality search (MCS) [151], both of which find a perfect elimination ordering for a chordal graph in linear time $\mathcal{O}(|V| + |E|)$.
- They can be recursively decomposed into smaller chordal subgraphs, which explains the alternative name *decomposable graphs*. We will later see how this property can be used to decompose SDPs.

To test for chordality we can use a maximum cardinality search as shown in Algorithm 5 to find an elimination ordering. If the ordering returned by the

Algorithm 5: Maximum Cardinality Search [15].

Input : A graph $G(V, E)$ with n vertices.
Output : An elimination ordering $\sigma(i) \rightarrow v$.

```
Initialise vector of cardinality of visited neighbours c(v) = 0;
for i = n to 1 do
    Find unnumbered vertex with highest cardinality v_i ← findmax c;
    Set σ^{-1}(v_i) = i;
    for all unnumbered v_j adjacent to v_i do
        c(v_j) = c(v_j) + 1;
Find inverse permutation of σ^{-1}(v): σ(i).
```

algorithm is a perfect elimination ordering, then the graph is chordal.

Notice that we can make any graph chordal by adding suitable edges to E. This is called a *chordal extension*, *chordal embedding* or *triangulation*.

Definition 4.2.4 (Chordal extension)**.** Given a graph $G(V, E)$, a chordal extension is a chordal graph $\bar{G}(V, \bar{E})$ where $E \subseteq \bar{E}$.

Finding a chordal extension with the minimum number of extra edges, or fill-in, is an NP-hard problem [174]. However, good heuristic methods to reduce the number of extra edges are available from matrix factorisation routines, e.g. the approximate minimum degree method (AMD) or the nested disection method [37]. An example for a chordal extension is shown in Figure 4.3.

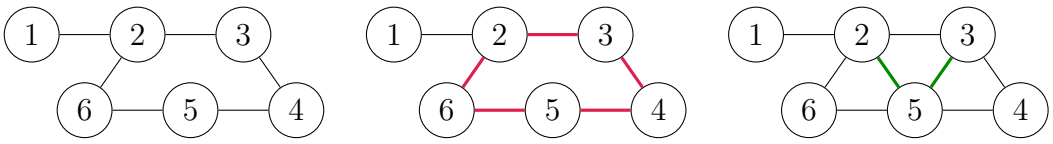


Figure 4.3: a) Nonchordal graph which has b) a cycle $(2-3-4-5-6)$ of length 5. c) The graph can be chordally extended by adding two edges $\bar{E} = \{\{2,5\},\{3,5\}\} \cup E$.

Another important concept that conveys information about how different parts of the graph are connected are *minimal vertex separators*.

Definition 4.2.5 (Minimal vertex separator)**.** A subset of vertices $S \subset V$ of an undirected graph $G(V, E)$ is a (v, u)-*vertex separator* for two vertices (v, u) if the subgraph $G(V \setminus S)$ does not contain a path that connects the vertices v, u. A (v, u)-vertex separator is a *minimal vertex separator* for the graph G if no proper subset of S is also a (v, u)-vertex separator.

The four minimal vertex separators for an example graph are shown in Figure 4.4. The chordality of a graph can be tested by considering its minimal vertex separators.

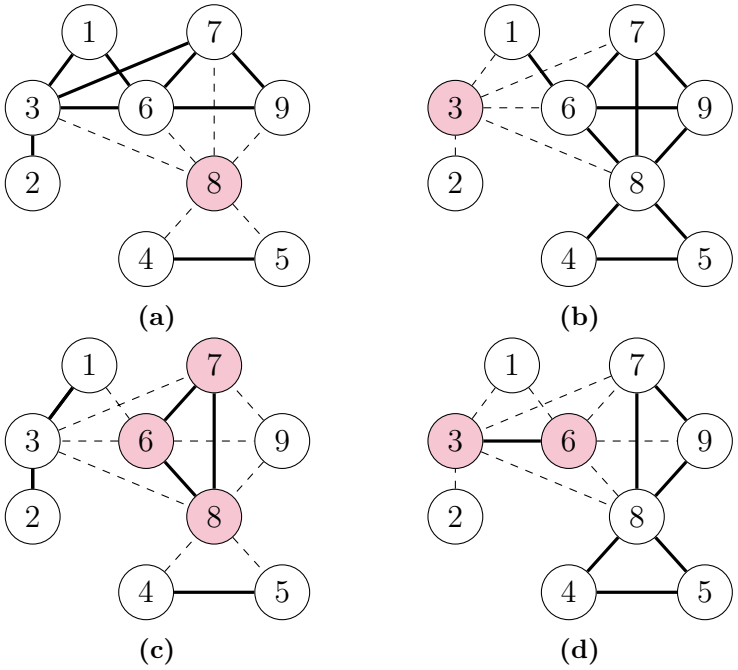


Figure 4.4: The four minimal vertex separators $\{\{8\}, \{3\}, \{6, 7, 8\}, \{3, 6\}\}$ of a chordal graph (highlighted in red). For example $\{3\}$ is a $(1, 2)$-separator.

Theorem 4 ([159, Theorem 3.1])**.** *A graph $G(V, E)$ is chordal if and only if all minimal vertex separators are complete subgraphs.*

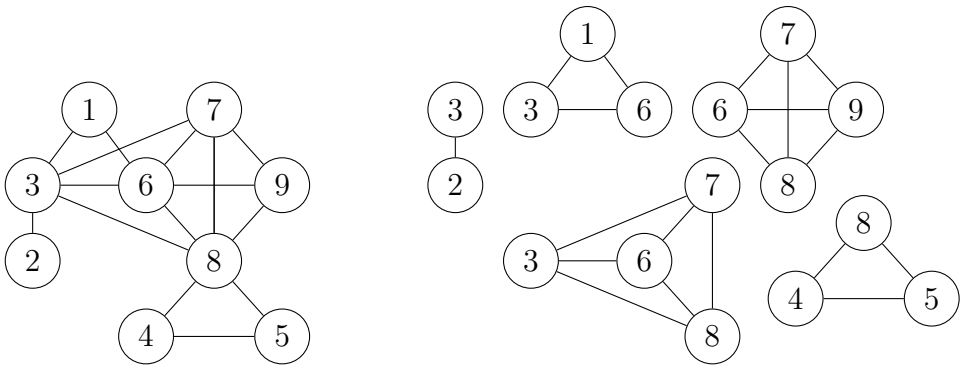
We can confirm that this is the case for the chordal graph in Figure 4.4.

4.2.2 Cliques, clique trees

We follow the convention of Vandenberghe and Andersen [159] by defining a *clique* as a subset of vertices $\mathcal{C} \subseteq V$ that induces a *maximal* complete subgraph of G.[1]

[1]The term clique in a graph context was first used by Luce and Perry [101] who analysed complete subgraphs in social networks, i.e. people who all knew each other.

The *maximality* condition means that the clique can not be extended by adding one more adjacent vertex. The number of vertices in a *clique* is given by the cardinality $|\mathcal{C}|$. If the graph is chordal and ordered $G_\sigma(V, E, \sigma)$ a clique can be identified by the *representative vertex* v^r, which is the vertex of lowest order in the clique. The other vertices of the clique are found in the higher neighbourhood of the v^r. We denote this as $\mathcal{C}(v^r) = \{v^r\} \cup \text{adj}^+(v^r)$. The example graph of Figure 4.4 is shown again in Figure 4.5 alongside its five cliques. Cliques convey significant information



(a) Chordal graph. **(b)** Cliques of the graph.

Figure 4.5

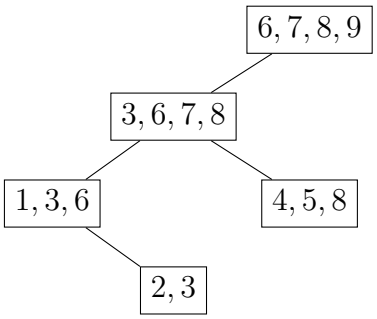
about the structure of the graph.

For a connected graph G let $\mathcal{B} = \{\mathcal{C}_1, \ldots, \mathcal{C}_p\}$ be the set of cliques. A clique tree $\mathcal{T}(\mathcal{B}, \mathcal{E})$ is formed by taking the cliques as vertices and by choosing edges from $\mathcal{E} \subseteq \mathcal{B} \times \mathcal{B}$ such that the tree satisfies the *running-intersection property*.

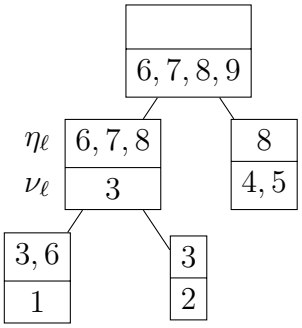
Definition 4.2.6 (Running intersection property)**.**
For each pair of cliques $\mathcal{C}_i$, $\mathcal{C}_j \in \mathcal{B}$, the intersection $\mathcal{C}_i \cap \mathcal{C}_j$ is contained in all the cliques on the path in the clique tree connecting $\mathcal{C}_i$ and $\mathcal{C}_j$.

This property is also referred to as the *clique-intersection property* in [116] and the *induced subtree property* in [159]. A close relationship between clique trees and chordal graphs is established via the following theorem.

Theorem 5 ([159, Theorem 3.5])**.** *For every chordal graph $G(V, E)$ one can construct a clique tree $\mathcal{T}(\mathcal{B}, \mathcal{E})$ that satisfies the running intersection property.*



(a) Clique tree with running intersection property for the graph in Figure 4.5(a).



(b) Clique tree partitioned into clique separators η_ℓ and clique residuals ν_ℓ.

Figure 4.6

The clique tree for the example graph from Figure 4.5(a) is shown in Figure 4.6(a).

For a clique $\mathcal{C}_\ell$ we refer to the first clique encountered on the path to the root as its *parent clique* $\mathcal{C}_{\text{par}}$. Conversely $\mathcal{C}_\ell$ is called a *child* of $\mathcal{C}_{\text{par}}$. If two cliques have the same parent clique we refer to them as *siblings*. We further define the function $\text{par} : 2^V \to 2^V$ and the multivalued function $\text{ch} : 2^V \rightrightarrows 2^V$ such that $\text{par}(\mathcal{C}_\ell)$ and $\text{ch}(\mathcal{C}_\ell)$ return the parent clique and set of child cliques of $\mathcal{C}_\ell$, respectively, where 2^V is the power set (set of all subsets) of V.

To analyse the clique tree it is helpful to partition the vertices in each clique into two sets: the *separator* $\eta_\ell = \mathcal{C}_\ell \cap \text{par}(\mathcal{C}_\ell)$, i.e. all clique elements that are also contained in the parent clique, and the set formed by the remaining vertices which is called the *clique residual* or *supernode* $\nu_\ell = \mathcal{C}_\ell \setminus \eta_\ell$. The partitioning for the example clique tree is shown in Figure 4.6(b), with the separators listed in the upper rows and the supernodes in the lower rows of the nodes. Notice that the separators are exactly the minimal vertex separators of the graph; cf. Figure 4.4. Keeping track of which vertices in a clique belong to the supernode and the separator is useful because it shows how different cliques overlap. We will use this information later in the chapter for PSD matrix completion and for clique merging strategies.

An efficient algorithm to compute the clique tree in $\mathcal{O}(n)$ was proposed by Pothen and Sun [128]. The algorithm takes as input the elimination tree and the higher degrees of every vertex and computes the representative vertex of every clique

and the parent-child relationships between the cliques. The pseudocode is shown in Algorithm 6. For the pseudocode notation we define the following functions, $\text{snd}(v) \to \nu$ returns the supernode that contains vertex v, $\text{rep}(\nu) \to v^r$ returns the representative vertex of supernode ν, $\text{ch}(v) \to C$ returns the children of v in the elimination tree.

Algorithm 6: Clique tree algorithm by Pothen and Sun [128].

Input : Postordered elimination tree for an ordered chordal graph $G_\sigma(V, E)$, higher degrees $\deg^+(v)$ for all vertices.
Output : Representative vertices $v_1^r, \ldots, v_p^r$, supernodes V^s, parent structure $\text{par}(v_i^r)$.

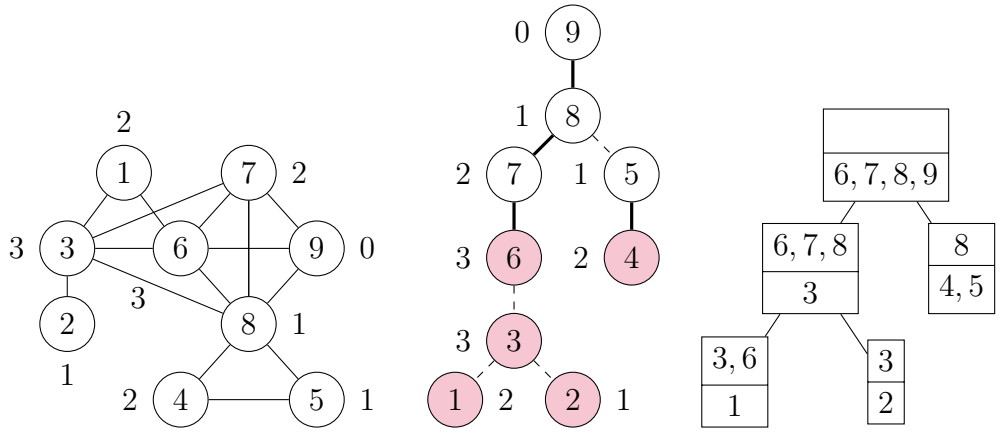
```
Initialise empty set of supernodes V^s = {};
for v ∈ V do
    Find children: C ← ch(v);
    if deg+(v) + 1 > deg+(u) for any u ∈ ch(v) then
        v is a representative vertex v^r ← v;
        Create a new supernode ν_ℓ = {v}, V^s ← ν_ℓ ;
    else
        Pick one û ∈ C such that deg+(v) + 1 = deg+(û);
        Find representative vertex v^r ← rep(snd(û));
        snd(û) = snd(û) ∪ {v};
        C = C \ {û};
    for u ∈ C do
        Find representative vertex of u: w ← rep(snd(u));
        Assign parent to w: par(w) ← v^r;
```

We illustrate the algorithm in the following example.

Example 4.2.1 (Computing a clique tree)**.** In this example we demonstrate how to compute a clique tree for the graph $G(V, e)$ in Figure 4.5(a). This requires four steps:

1. Compute the higher degrees of every vertex.
2. Compute the elimination tree.
3. Determine the representative vertices / clique supernodes and parent structure using Algorithm 6.
4. Find the clique separators.

We start by determining a perfect elimination ordering for the graph in Figure 4.5(a). For this particular graph one perfect elimination ordering is given by the natural ordering of the vertices, i.e. $\sigma(i) = i$. This defines the ordered graph $G_\sigma(V, E)$. The ordering allows us to determine the higher degree $\deg^+(v)$ for every $v \in V$. The higher degrees are shown in Figure 4.7(a).



(a) Higher degrees of example graph. **(b)** Elimination tree. **(c)** Supernodal elimination tree.

Figure 4.7

For an ordered graph, the elimination tree is rooted in the vertex with the highest order $\sigma(n)$. The parent for each vertex $v_i \in V$ is defined as the vertex v_j in v_i's higher neighbourhood with the lowest ordering index, i.e.

$$\operatorname{par}(v_i) := \operatorname{argmin}\{\sigma^{-1}(v_j) \mid v_j \in \operatorname{adj}^+(v_i)\}.$$

The elimination tree for the ordering in Figure 4.7(a) is shown in Figure 4.7(b). The elimination tree can be computed in $\mathcal{O}(n)$ [97]. Again we show the higher degrees for each vertex. Notice that by construction this tree is topologically ordered, i.e. every vertex in the tree has a lower ordering than its parent. The elimination tree and the higher degrees allow us to apply Algorithm 6 to find the supernodes and the supernode parent structure. The supernodes are given by:

$$\nu_1 = \{1\}, \quad \nu_2 = \{2\}, \quad \nu_3 = \{3\}, \quad \nu_4 = \{4, 5\}, \quad \nu_5 = \{6, 7, 8, 9\}.$$

The representative vertex v_i^r for each supernode ν_i is the first element in the node. The supernode parent structure is defined in terms of the representative vertex of each supernode, i.e. if $\operatorname{par}(v_j^r) = v_i^r$, then $\operatorname{par}(\operatorname{snd}(v_j^r)) = \operatorname{snd}(v_i^r)$:

$$\text{par}(1) = 3, \quad \text{par}(2) = 3, \quad \text{par}(3) = 6, \quad \text{par}(4) = 6.$$

The representative vertices and supernode partition are highlighted in Figure 4.7(b). Using the parent structure we define a supernodal elimination tree, with the supernodes ν_i as the vertices. In Figure 4.7(c) the supernodes are shown in the lower row of the vertices. The upper row, the clique residuals, are determined by the vertices in the higher neighbourhood of the representative vertex $\text{adj}^+(v_i^r)$ minus the elements in the lower row:

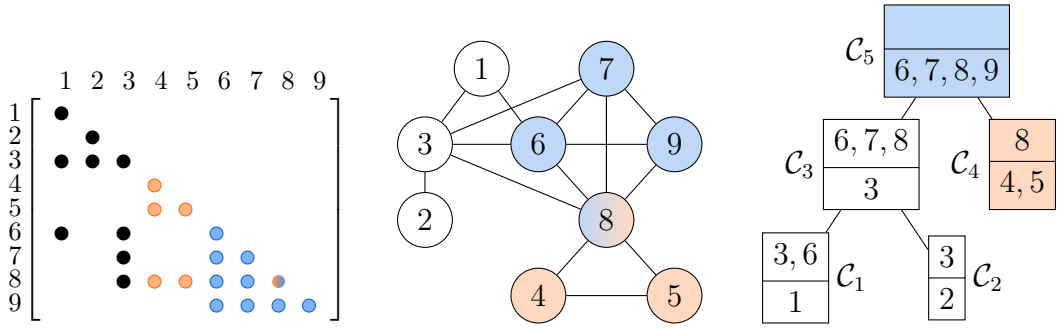
$$\eta_i \coloneqq \text{sep}(v_i^r) = \text{adj}^+(v_i^r) \setminus \nu_i.$$

Consequently, the tree in Figure 4.7(c) forms a valid clique tree with each node representing one clique.

The example started by determining a valid elimination tree based on a particular elimination ordering. Notice that for the tree in Figure 4.7(b) it holds that the order of every vertex is smaller than of its parent, i.e. $v \prec_\sigma \text{par}(v)$. This is called a *topological ordering*. After the elimination tree is computed we are allowed to reorder the vertices using any topological reordering and still get the same elimination tree and clique tree structure. A *postordering* of the elimination tree is particularly helpful in a number of algorithms that are presented in the next section. In a postordered tree, descendants of vertices are given consecutive orders. Consequently, a postordering can be generated by assigning the highest order to the root and then performing a depth-first traversal of the elimination tree.

4.2.3 Sparse matrices and graphs

In this section we will establish the connection between the graph concepts from Section 4.2.1, Section 4.2.2 and the sparsity pattern of a symmetric matrix. An undirected graph with n vertices can be used to represent the sparsity pattern of a symmetric matrix $S \in \mathbb{S}^n$. Every nonzero entry $S_{ij} \neq 0$ in the lower triangular part of the matrix introduces an edge $(i, j) \in E$. In Figure 4.8 we show the lower triangular nonzeros of a sparsity pattern that corresponds to the graph from the previous section. Furthermore, notice that we coloured the vertices / nonzero



(a) Nonzeros on the lower triangle of a symmetric matrix. **(b)** Corresponding graph. **(c)** Corresponding clique tree.

Figure 4.8: Sparsity pattern with corresponding graph and clique tree. The elements of the cliques $\mathcal{C}_4$ and $\mathcal{C}_5$ are shown highlighted.

elements of cliques $\mathcal{C}_4$ and $\mathcal{C}_5$. This shows how the cliques of the graph constitute complete subblocks in the matrix sparsity pattern. Moreover, the clique separator $\eta_4 = \{8\}$ of $\mathcal{C}_4$ tells us which matrix elements overlap with the parent clique. In this case only in the element $S_{8,8}$. Another example would be the overlapping elements between $\mathcal{C}_1$ and $\mathcal{C}_3$, which are given by the matrix elements $S_{3,3}, S_{3,6}, S_{6,6}$.

With this relationship between graphs and sparsity in mind, one can formally define the building blocks to constrain matrices to have a sparsity pattern that is specified by the edge set of an undirected graph $G(V, E)$. We define the following spaces of sparse symmetric matrices.

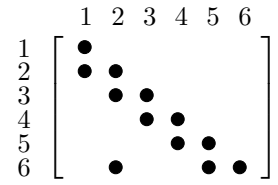
Definition 4.2.7 (Space of sparse symmetric matrices)**.** Given an undirected graph $G(V, E)$ we define the *space of sparse symmetric matrices* as

$$\mathbb{S}^n(E, 0) \coloneqq \{S \in \mathbb{S}^n \mid S_{ij} = S_{ji} = 0 \text{ if } i \neq j \text{ and } (i, j) \notin E\}. \tag{4.3}$$

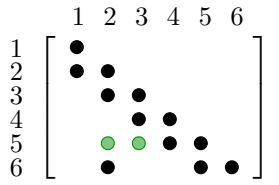
Note that the numerical value of diagonal entries S_{ii} and the off-diagonal entries S_{ij} with $(i, j) \in E$ may be zero or nonzero.

For the rest of the chapter we will assume that the graph $G(V, E)$ that defines the sparsity pattern is connected. If the graph had k disconnected components, one could analyse the sparsity pattern induced by each subgraph $G_1(V_1, E_1), \ldots, G_k(V_k, E_k)$ with partitioned vertex set $\bigcup_{i=1}^k V_i = V$, $V_i \cap V_j = \emptyset$ and edge set $\bigcup_{i=1}^k E_i = E$, $E_i \cap E_j = \emptyset$ separately.

We extend the terminology established in the previous section to matrices in $\mathbb{S}^n(E,0)$. A sparse chordal matrix is a matrix in $\mathbb{S}^n(E,0)$ where E specifies a chordal graph. Furthermore, we can perform a chordal extension of a sparsity pattern, by extending the corresponding graph. The lower-triangular sparsity patterns corresponding to the chordal extension example in Figure 4.3 are shown in Figure 4.9.



(a) Sparsity pattern corresponding to nonchordal graph in Figure 4.3(a).



(b) Chordal extension of the sparsity pattern according to graph in Figure 4.3(c).

Figure 4.9

Next, we define another sparse matrix space by extending the definition (4.3) with a positive semidefiniteness condition.

Definition 4.2.8 (Space of sparse positive semidefinite matrices)**.** Given an undirected graph $G(V,E)$ we define the *space of sparse positive semidefinite matrices* as

$$\mathbb{S}^n_+(E,0) \coloneqq \{S \in \mathbb{S}^n(E,0) \mid S \succeq 0\}.$$

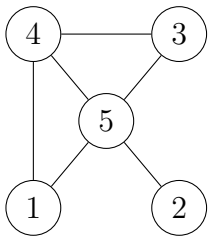
Since $\mathbb{S}^n_+(E,0)$ is the intersection of a subspace and a convex cone $\mathbb{S}^n_+(E,0) = \mathbb{S}^n_+ \cap \mathbb{S}^n(E,0)$ it is itself a closed convex cone.

The conditions for a proper cone were listed in Definition 2.1.3, i.e. closed, pointed, and nonempty interior. Clearly, $\mathbb{S}^n_+(E,0)$ is pointed as S and $-S$ can only be positive semidefinite if $S = \mathbf{0}_{n\times n}$. Moreover, its interior is $\mathbf{int}(\mathbb{S}^n_+(E,0)) \coloneqq \mathbb{S}^n_{++}(E,0)$, the set of strictly positive semidefinite matrices with sparsity pattern E

$$\mathbb{S}^n_{++}(E,0) \coloneqq \{S \in \mathbb{S}^n(E,0) \mid S \succ 0\}.$$

Consequently, $\mathbb{S}^n_+(E,0)$ is proper in $\mathbb{S}^n(E,0)$ (whenever E is nonempty). An example of a matrix in the cone of sparse positive semidefinite matrices for a specific

graph is shown in Figure 4.10. We emphasize that matrix elements that correspond



(a)

$$S = \begin{bmatrix} 1 & 0 & 0 & 2 & -1 \\ 0 & 2 & 0 & 0 & 3 \\ 0 & 0 & 3 & 1 & -2 \\ 2 & 0 & 1 & 7 & -5 \\ -1 & 3 & -2 & -5 & 12 \end{bmatrix} \succeq 0$$

(b)

Figure 4.10: Sparsity graph $G(V, E)$ and example matrix $S \in \mathbb{S}^n_+(E, 0)$ with eigenvalues $\lambda(S) = \{0.31, 0.72, 2.77, 5.03, 16.17\}$.

to edges in E are allowed to have a *numerical* value of zero, e.g. for the matrix in Figure 4.10(b) we could set $S_{45} = S_{54} = 0$ and obtain another valid example.

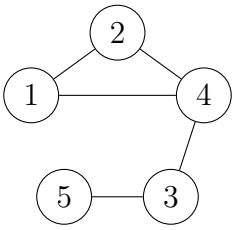
The dual cone of $\mathbb{S}^n_+(E, 0)$ is the positive semidefinite completable matrix cone, which is also a proper cone [159, Chapter 10].

Definition 4.2.9 (Space of positive semidefinite completable matrices)**.** Given an undirected graph $G(V, E)$ we define the *space of positive semidefinite completable matrices* as

$$\mathbb{S}^n_+(E, ?) := \left\{ Y \mid \exists\, \hat{Y} \in \mathbb{S}^n_+, Y_{ij} = \hat{Y}_{ij}, \text{ if } i = j \text{ or } (i, j) \in E \right\}.$$

For a matrix $Y \in \mathbb{S}^n_+(E, ?)$ we can find a positive semidefinite completion $\hat{Y}$ by choosing appropriate values for all entries $(i, j) \notin E$.

For the example graph in Figure 4.11(a), a suitable PSD-completable matrix Y is shown in Figure 4.11(b) with completion $\hat{Y}$ in Figure 4.11(c). The cones $\mathbb{S}^n_+(E, ?)$



(a)

$$\begin{bmatrix} 27 & -2 & ? & 4 & ? \\ -2 & 11 & ? & -9 & ? \\ ? & ? & 18 & 5 & 21 \\ 4 & -9 & 5 & 31 & ? \\ ? & ? & 21 & ? & 48 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 27 & -2 & 12 & 4 & 31 \\ -2 & 11 & -11 & -9 & -11 \\ 12 & -11 & 18 & 5 & 21 \\ 4 & -9 & 5 & 31 & 22 \\ 31 & -11 & 21 & 22 & 48 \end{bmatrix}$$

(c)

Figure 4.11: a) Sparsity graph $G(V, E)$, b) example matrix $Y \in \mathbb{S}^n_+(E, ?)$ and c) positive semidefinite completion $\hat{Y}$ with eigenvalues $\lambda(\hat{Y}) = \{0.02, 0.92, 15.49, 27.94, 90.63\}$.

and $\mathbb{S}^n_+(E,0)$ are well known to be duals of each other [159, Chapter 10]. Their duality has important consequences when these cones are used in the context of primal-dual algorithms for SDPs. If the primal matrix variable is constrained to be positive semidefinite with a given sparsity pattern, then the corresponding dual matrix variable will be positive semidefinite completable with the same sparsity pattern.

The positive semidefinite completion $\hat{Y}$ for a given matrix $Y \in \mathbb{S}^n_+(E,?)$ is in general not unique. An algorithm to find a completion that maximizes the determinant of $\hat{Y}$ is described in [159, Sec. 10.3] and shown in Algorithm 7. The algorithm

Algorithm 7: Positive semidefinite completion.

Input : Completable matrix $Y \in \mathbb{S}^n_+(E,?)$, chordal sparsity graph $G(V,E)$ and a postordered supernodal elimination tree with n representative vertices $v_i^r \in V^r$.

Output : Positive semidefinite completion $\hat{Y}$.

```
Initialise Ŷ = Y;
for v_i^r ∈ V in descending order do
    ν ← snd(v_i^r), η ← sep(v_i^r);
    C = ν ∪ η;
    ω = {j | v_i^r < j ≤ n} \ C;
    Ŷ_{ω,ν} = Ŷ_{ω,η} Ŷ†_{η,η} Ŷ_{η,ν};
    Ŷ_{ν,ω} = Ŷ^⊤_{ω,ν};
```

processes the representative vertices of the postordered supernode elimination tree in descending order i. At each step the algorithm gathers the indices of the supernode $\nu = \mathrm{snd}(v_i^r)$ and of the separator $\eta = \mathrm{sep}(v_i^r)$. The important index set for the completion is ω that collects all the unknown elements in the row below the representative vertex. These are the entries that are to be completed. Using this information we complete the unknown submatrix block $\hat{Y}_{\omega,\nu}$ for the columns given by the supernode. Using symmetry of $\hat{Y}$ we can also complete the transposed submatrix block.

The steps of the completion algorithm are shown for the example graph in the following example.

Example 4.2.2 (Positive semidefinite completion)**.** In this example we compute the positive semidefinite completion $\hat{Y}$ for a matrix $Y \in \mathbb{S}_+^n(E, ?)$ with the chordal sparsity pattern and supernodal elimination tree discussed in Figure 4.8. Notice that the cliques in Figure 4.8(c) are postordered. Following this ordering we determine ν (lower row), η (upper row) and ω for each clique and then fill in the submatrix according to:

$$\begin{aligned} \hat{Y}_{\omega,\nu} &= \hat{Y}_{\omega,\eta} \hat{Y}_{\eta,\eta}^{\dagger} \hat{Y}_{\eta,\nu} \\ \hat{Y}_{\nu,\omega} &= \hat{Y}_{\omega,\nu}^{\top} \end{aligned} \tag{4.4}$$

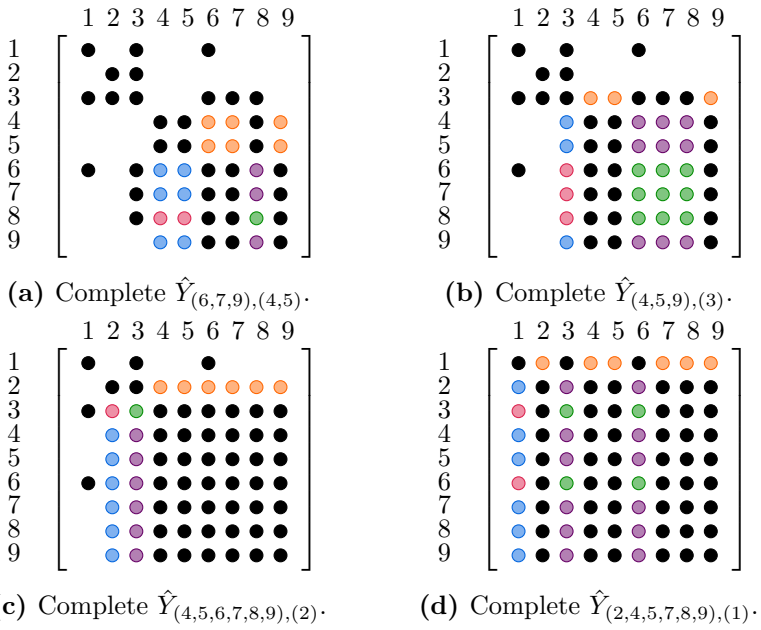
The matrix Y is gradually completed. The steps for each clique are shown in Figure 4.12. The matrix blocks corresponding to the terms in (4.4) are highlighted. No completion steps are necessary for the root of the supernode tree, clique $\mathcal{C}_5$, as it always represents a dense block in the bottom right corner. The completion for $\mathcal{C}_4$ with $v_4^r = 4$ is shown in Figure 4.12(a). The following blocks are completed:

$$\begin{aligned} \hat{Y}_{(6,7,9),(4,5)} &= \hat{Y}_{(6,7,9),(8)} \hat{Y}_{(8),(8)}^{\dagger} \hat{Y}_{(8),(4,5)} \\ \hat{Y}_{(4,5),(6,7,9)} &= \hat{Y}_{(6,7,9),(4,5)}^{\top}. \end{aligned}$$

Notice that the processing order of the cliques matters, e.g. the completed entries in Figure 4.12(b) are used to complete the entries in the next step shown in Figure 4.12(c).

4.3 Chordal Decomposition

As noted in Section 3.4.3, for large SDPs the eigendecomposition in the projection step (3.25) is the principal performance bottleneck for a first- or second-order algorithm. However, since large-scale SDPs often exhibit a certain structure or sparsity pattern, a sensible strategy is to exploit any such structure to alleviate this bottleneck. If the aggregated sparsity pattern is *chordal*, Agler's [1] and Grone's [70] theorems can be used to decompose a large PSD constraint into a collection of smaller PSD constraints and additional coupling constraints. The projection step



(a) Complete $\hat{Y}_{(6,7,9),(4,5)}$.

(b) Complete $\hat{Y}_{(4,5,9),(3)}$.

(c) Complete $\hat{Y}_{(4,5,6,7,8,9),(2)}$.

(d) Complete $\hat{Y}_{(2,4,5,7,8,9),(1)}$.

Figure 4.12: Steps of positive semidefinite completion Algorithm 7 applied to the example sparsity pattern in Figure 4.8(a). The blue and orange entries are being completed.

applied to the set of smaller PSD constraints is usually significantly faster than when applied to the original constraint. Since the projections are independent of each other, further performance improvement can be achieved by carrying them out in parallel. Our approach is to apply chordal decomposition to a standard form SDP in the form (3.14) to produce a decomposed problem. The decomposed problem is then transformed back to a problem in the form (3.14) but with a collection of smaller PSD cone constraints and coupling constraints. This process allows us to apply chordal decomposition as a preprocessing step before the problem is handed to the solver. As we will see in Section 4.3.1, a chordal sparsity pattern can be imposed on any PSD constraint.

4.3.1 Decomposition theorems

In Section 4.2.3 we established a connection between graphs and matrix sparsity. In particular we highlighted some of the advantageous properties of chordal graphs, such as the ability to calculate a supernodal elimination tree, which encodes information about all the graph cliques and how they overlap. In this section we establish a link between the sparsity pattern imposed on a positive semidefinite optimisation

variable by a set of constraints and chordal graphs. The decomposability property of a chordal graph which allows us to split it into cliques can also be applied to split a positive semidefinite matrix into a number of nonzero subblocks. The connection is formalized in the following two important matrix decomposition theorems. We first consider an SDP in standard primal form:

$$\begin{array}{ll} \text{minimize} & \langle C, X \rangle \\ \text{subject to} & \langle A_k, X \rangle = b_k, \quad k = 1, \ldots, m \\ & X \in \mathbb{S}^n_+, \end{array} \tag{4.5}$$

with variable X, coefficient matrices $A_k, C \in \mathbb{S}^n$ and vector $b \in \mathbb{R}^m$. The corresponding dual problem is

$$\begin{array}{ll} \text{maximize} & b^\top y \\ \text{subject to} & \sum_{k=1}^m A_k y_k + S = C \\ & S \in \mathbb{S}^n_+, \end{array} \tag{4.6}$$

with dual variable $y \in \mathbb{R}^m$ and slack variable S. Let us assume that the problem matrices in (4.5) and (4.6) each have their own sparsity pattern

$$A_k \in \mathbb{S}^n(E_{A_k}, 0) \text{ and } C \in \mathbb{S}^n(E_C, 0).$$

The *aggregate sparsity* of the problem is given by the graph $G(V, E)$ with edge set

$$E = E_{A_1} \cup E_{A_2} \cup \cdots \cup E_{A_m} \cup E_C.$$

In general $G(V, E)$ will not be chordal, but a *chordal extension* can be found by adding edges to the graph. We denote the extended graph as $G(V, \bar{E})$, where $E \subseteq \bar{E}$. Finding the minimum number of additional edges required to make the graph chordal is an NP-complete problem [174]. Consider a 0–1 matrix M with edge set E. A commonly used heuristic method to compute a chordal extension is to perform a symbolic Cholesky factorisation of M [53], with the edge set of the Cholesky factor then providing the chordal extension $\bar{E}$. To simplify the notation in the remainder of the chapter, we henceforward assume that E represents a chordal graph or has been appropriately extended.

Using the aggregate sparsity of the problem we can modify the constraints on the matrix variables in (4.5) and (4.6) to be in the respective sparse positive semidefinite matrix spaces:

$$X \in \mathbb{S}^n_+(E,?) \text{ and } S \in \mathbb{S}^n_+(E,0). \tag{4.7}$$

We further define the entry-selector matrices $T_\ell \in \mathbb{R}^{|\mathcal{C}_\ell| \times n}$ for a clique $\mathcal{C}_\ell$ as

$$(T_\ell)_{ij} \coloneqq \begin{cases} 1, & \text{if } \mathcal{C}_\ell(i) = j \\ 0, & \text{otherwise,} \end{cases}$$

where $\mathcal{C}_\ell(i)$ is the ith vertex of $\mathcal{C}_\ell$. We can express the constraints in (4.7) in terms of multiple smaller coupled constraints using Grone's and Agler's theorems.

Theorem 6 (Grone's theorem [70])**.** *Let $G(V,E)$ be a chordal graph with a set of maximal cliques $\{\mathcal{C}_1,\dots,\mathcal{C}_p\}$. Then $X \in \mathbb{S}^n_+(E,?)$ if and only if*

$$X_\ell = T_\ell X T_\ell^\top \in \mathbb{S}^{|\mathcal{C}_\ell|}_+,$$

for all $\ell = 1,\dots,p$.

Applying this theorem to (4.5) while restricting X to the positive semidefinite completable matrix cone as in (4.7) yields the decomposed problem:

$$\begin{array}{lll} \text{minimize} & \langle C, X\rangle & \\ \text{subject to} & \langle A_k, X\rangle = b_k, & k = 1,\dots,m \\ & X_\ell = T_\ell X T_\ell^\top, & \ell = 1,\dots,p \\ & X_\ell \in \mathbb{S}^{|\mathcal{C}_\ell|}_+, & \ell = 1,\dots,p, \end{array} \tag{4.8}$$

An application of Theorem 6 to a small SDP is shown in the following example.

Example 4.3.1 (Grone's theorem)**.** Consider the following 4×4 LMI-constraint based on a variable $x \in \mathbb{R}^3$:

$$X(x) = \begin{bmatrix} x_1 & x_2+1 & ? & ? \\ x_2+1 & x_3 & -1 & ? \\ ? & -1 & x_1+x_2+5 & -x_1+3 \\ ? & ? & -x_1+3 & x_1-x_3 \end{bmatrix} \in \mathbb{S}^4_+(E,?). \tag{4.9}$$

The corresponding sparsity graph has three cliques, $\mathcal{C}_1 = \{1,2\}$, $\mathcal{C}_2 = \{2,3\}$,and $\mathcal{C}_3 = \{3,4\}$. Following Grone's theorem the LMI in (4.9) can be equivalently

represented by the following three LMI constraints

$$X_1(x) = \begin{bmatrix} x_1 & x_2+1 \\ x_2+1 & x_3 \end{bmatrix} \succeq 0, \quad X_2(x) = \begin{bmatrix} x_3 & -1 \\ -1 & x_1+x_2+5 \end{bmatrix} \succeq 0,$$
$$X_3(x) = \begin{bmatrix} x_1+x_2+5 & -x_1+3 \\ -x_1+3 & x_1-x_3 \end{bmatrix} \succeq 0.$$

For the dual problem we utilise Agler's theorem, which is the dual to Theorem 6.

Theorem 7 (Agler's theorem [1])**.** *Let $G(V,E)$ be a chordal graph with a set of maximal cliques $\{\mathcal{C}_1,\ldots,\mathcal{C}_p\}$. Then $S \in \mathbb{S}^n_+(E,0)$ if and only if there exist matrices $S_\ell \in \mathbb{S}^{|\mathcal{C}_\ell|}_+$ for $\ell = 1,\ldots,p$ such that*

$$S = \sum_{\ell=1}^{p} T_\ell^\top S_\ell T_\ell.$$

Note that the matrices T_ℓ serve to extract the submatrix S_ℓ such that $S_\ell = T_\ell S T_\ell^\top$ has rows and columns corresponding to the vertices of the clique C_ℓ. With this theorem, we transform the dual form SDP in (4.6) with the restriction on S in (4.7) to arrive at:

$$\begin{array}{ll} \text{maximize} & b^\top y \\ \text{subject to} & \displaystyle\sum_{k=1}^{m} A_k y_k + \sum_{\ell=1}^{p} T_\ell^\top S_\ell T_\ell = C \\ & S_\ell \in \mathbb{S}^{|\mathcal{C}_\ell|}_+, \quad \ell = 1,\ldots,p. \end{array} \tag{4.10}$$

An application of Theorem 7 to a small SDP is shown in the following example.

Example 4.3.2 (Agler's theorem)**.** Consider the following 5×5 LMI-constraint based on a variable $y \in \mathbb{R}^4$:

$$S(y) = \begin{bmatrix} y_1 & -y_2 & 0 & y_4 & 0 \\ -y_2 & y_3+y_4 & 0 & 0 & 0 \\ 0 & 0 & y_3 & 2+y_1 & -y_1 \\ y_4 & 0 & 2+y_1 & 4-y_2 & y_3 \\ 0 & 0 & -y_1 & y_3 & 5+y_2 \end{bmatrix} \in \mathbb{S}^5_+ \tag{4.11}$$

The three cliques of the sparsity graph of the matrix in (4.11) are $\mathcal{C}_1 = \{1,2\}$, $\mathcal{C}_2 = \{1,4\}$, $\mathcal{C}_3 = \{3,4,5\}$. Using Agler's theorem the LMI can be equivalently represented

by the three smaller LMIs

$$S_1(y,\theta) = \begin{bmatrix} y_1 - \theta_1 & -y_2 \\ -y_2 & y_3 + y_4 \end{bmatrix} \succeq 0, \qquad S_2(y,\theta) = \begin{bmatrix} \theta_1 & y_4 \\ y_4 & 4 - y_2 - \theta_2 \end{bmatrix} \succeq 0,$$
$$S_3(y,\theta) = \begin{bmatrix} y_3 & 2 + y_1 & -y_1 \\ 2 + y_1 & \theta_2 & y_3 \\ -y_1 & y_3 & 5 + y_2 \end{bmatrix} \succeq 0,$$

which use two additional variables θ_1, θ_2 to account for the overlapping entries in $(1,1)$ and $(4,4)$. It is easy to see that the entry-wise sum of the three matrix blocks yields S. Notice that the decomposition into three cliques is somewhat arbitrary. We are free to treat any structural zero in (4.11) as a numerical zero which consequently introduces an edge into the sparsity graph. For example, treating the zeros in position $(2,4)$ and $(4,2)$ as numerical zeros would effectively merge $\mathcal{C}_1$ and $\mathcal{C}_2$ and the decomposition would have just two blocks. The idea of effective clique merging is discussed in more detail in Section 4.4.

4.3.2 Backtransformation

If the decomposition of the problem is handled as a preprocessing step before it is handed to the solver algorithm, it might be necessary to transform the decomposed problem back into a compatible problem form. In this section we consider the decomposed dual SDP problem in (4.10) and show how to transform it back into the standard solver form (3.14), defined in the previous chapter and shown again here:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2} x^\top P x + q^\top x \\ \text{subject to} & Ax + s = b \\ & s \in \mathcal{K}. \end{array}$$

For the undecomposed problem in (4.6), this is achieved by first relabelling y as x. We then choose $P = 0_{n \times n}$, $q = -b$, define $A = [\operatorname{svec}(A_1), \ldots, \operatorname{svec}(A_m)]$, $b = \operatorname{svec}(C)$, $s = \operatorname{svec}(S)$, and $\mathcal{K} = \mathcal{S}_+^n$.

To transform the decomposed dual problem (4.10), we will make use of the fact that the decision variable $S \in \mathbb{S}^n$ and all the submatrices S_ℓ are symmetric and consider instead $s_\ell = \operatorname{svec}(S_\ell)$. The main challenge is to keep track of the overlapping entries

in the individual blocks and ensure they sum to the original entry in S. Different possible transformations achieving this are described in [88].

All the necessary information about the overlapping entries is stored in the clique tree $\mathcal{T}(\mathcal{B}, \mathcal{E})$ that represents the sparsity pattern of S. We assume that the clique tree is post-ordered with cliques $\mathcal{C}_1, \ldots, \mathcal{C}_p$. Define a vector to represent slack variables for overlapping entries $\theta := [\theta_1, \ldots, \theta_{n_o}]^\top$, where $n_o := \sum_{\ell=1}^{p} \frac{|\eta_\ell|(|\eta_\ell|+1)}{2}$ is the total number of overlapping entries in the upper triangle of the sparsity pattern. The s vector of the decomposed problem is created by stacking the vectorized subblocks s_ℓ according to the postordering of the clique tree. Define $\omega(i, j, \ell)$ as the index of s corresponding to the (i, j)th element of block S_ℓ. The equality constraint in problem (4.10) can be represented in the equivalent standard form (3.14) as:

$$\left[\begin{array}{ccc|c} \operatorname{svec}(\tilde{A}_1^1) & \cdots & \operatorname{svec}(\tilde{A}_m^1) & \\ \vdots & \ddots & \vdots & L \\ \operatorname{svec}(\tilde{A}_1^p) & \cdots & \operatorname{svec}(\tilde{A}_m^p) & \end{array}\right] \begin{bmatrix} x \\ \theta \end{bmatrix} + \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_p \end{bmatrix} = \begin{bmatrix} \operatorname{svec}(\tilde{B}^1) \\ \vdots \\ \operatorname{svec}(\tilde{B}^p) \end{bmatrix}, \tag{4.12}$$

with

$$\tilde{A}_{k,ij}^\ell := \begin{cases} A_{k,ij}^\ell & \text{if } (i,j) \notin \operatorname{sep}(\mathcal{C}_\ell) \\ 0 & \text{otherwise,} \end{cases} \qquad \tilde{B}_{ij}^\ell := \begin{cases} B_{ij}^\ell & \text{if } (i,j) \notin \operatorname{sep}(\mathcal{C}_\ell) \\ 0 & \text{otherwise.} \end{cases}$$

The matrices $\tilde{A}_k^\ell$ and $\tilde{B}^\ell$ take on the values of A_k and B in the locations corresponding to the elements in the submatrix S_ℓ. If a matrix entry of clique $\mathcal{C}_l$ in $A_{k,ij}^\ell$ and B_{ij}^ℓ is overlapped by an entry of the parent clique, i.e. both (i, j) are included in $\operatorname{sep}(\mathcal{C}_l)$, it is set to zero. Each column of the linking matrix $L \in \mathbb{R}^{m_d \times n_o}$ links one overlapping entry in the clique tree. L is generated by first collecting all the matrix indices $(i, j)_\ell$ of the separators $\operatorname{sep}(\mathcal{C}_\ell)$ in the clique tree:

$$O := \bigcup_{\ell=1}^{p} \{(i,j)_\ell \in \operatorname{sep}(\mathcal{C}_\ell) \mid i \leq j\}.$$

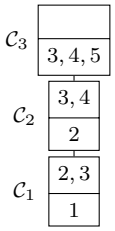
Then L is constructed column-by-column, each representing one overlapping entry. The column vector l_c is equal to 1 in the row r corresponding to the $(i, j)_\ell$-th entry of S_ℓ and -1 in the row corresponding to the same entry of the parent block S_k,

where $\mathcal{C}_\ell = \mathrm{ch}(\mathcal{C}_k)$. Thus, each element in l_c is defined by:

$$l_{rc} := \begin{cases} 1 & \text{if } r = \omega(i,j,\ell) \\ -1 & \text{if } r = \omega(i,j,\mathrm{par}(\ell)) \\ 0 & \text{otherwise,} \end{cases} \quad \text{for each } (i,j)_\ell \in O.$$

As an example consider a problem with $m = 1$ and $p = 3$ and the simple sparsity pattern and clique tree given by:

$$\begin{bmatrix} B_{11}-A_{11}x & B_{12}-A_{12}x & B_{13}-A_{13}x & 0 & 0 \\ & B_{22}-A_{22}x & B_{23}-A_{23}x & B_{24}-A_{24}x & 0 \\ & & B_{33}-A_{33}x & B_{34}-A_{34}x & B_{35}-A_{35}x \\ & & & B_{44}-A_{44}x & B_{45}-A_{45}x \\ & & & & B_{55}-A_{55}x \end{bmatrix}$$



Using the transformation in (4.12) this constraint can be represented by three constraints on the submatrices S_1, S_2 and S_3 and six overlap variables $\theta_1, \ldots, \theta_6$:

$$S_3 = \begin{bmatrix} B_{33} - A_{33}x - \theta_1 & B_{34} - A_{34}x - \theta_2 & B_{35} - A_{35}x \\ & B_{44} - A_{44}x - \theta_3 & B_{45} - A_{45}x \\ & & B_{55} - A_{55}x \end{bmatrix} \in \mathbb{S}^3_+$$

$$S_2 = \begin{bmatrix} B_{22} - A_{22}x - \theta_4 & B_{23} - A_{23}x - \theta_5 & B_{24} - A_{24}x \\ & \theta_1 - \theta_6 & \theta_2 \\ & & \theta_3 \end{bmatrix} \in \mathbb{S}^3_+$$

$$S_1 = \begin{bmatrix} B_{11} - A_{11}x & B_{12} - A_{12}x & B_{13} - A_{13}x \\ & \theta_4 & \theta_5 \\ & & \theta_6 \end{bmatrix} \in \mathbb{S}^3_+$$

Notice how the overlap variables drop out if the original matrix S is reassembled according to Theorem 7 by adding the entries of each subblock.

4.4 Clique Merging

In Section 4.3 we showed how conic problems with structured PSD constraints can be decomposed to obtain an equivalent problem with many, usually significantly smaller, PSD constraints and how this can reduce the number of operations to project the respective matrix blocks onto the positive semidefinite cone. In this section we introduce a merging procedure that allows us to improve the matrix decomposition in order to further speed up the projection step.

Given an initial decomposition with (chordally extended) edge set E and a set of cliques $\{\mathcal{C}_1, \ldots, \mathcal{C}_p\}$, we are free to re-merge cliques back into larger blocks. This is equivalent to treating *structural* zeros in the problem as *numerical* zeros, leading to additional edges in the graph. Looking at the decomposed problem in (4.8) and (4.10), the effects of merging two cliques $\mathcal{C}_i$ and $\mathcal{C}_j$ are twofold:

1. We replace two positive semidefinite matrix constraints of dimensions $|\mathcal{C}_i|$ and $|\mathcal{C}_j|$ with one constraint on a larger clique with dimension $|\mathcal{C}_i \cup \mathcal{C}_j|$, where the increase in dimension depends on the size of the overlap.
2. We remove consistency constraints for the overlapping entries between $\mathcal{C}_i$ and $\mathcal{C}_j$, thus reducing the size of the linear system of equality constraints.

When merging cliques these two factors have to be balanced, and the optimal merging strategy depends on the particular SDP solution algorithm employed. To search for favourable merge candidates Nakata et al. [116] and Sun et al. [150] use a clique tree; we will refer to their two approaches as `SparseCoLO` and the *parent-child* strategy, respectively, in the following sections. We will then propose a new merging method in Section 4.4.2 whose performance is superior to both methods when used with a first-order method such as ADMM. Following Section 4.2.2 we denote the set of cliques $\mathcal{B}$ which act as the nodes of an undirected clique graph (or clique tree) with edge set $\mathcal{E} \subseteq \mathcal{B} \times \mathcal{B}$. Given a clique merging strategy, Algorithm 8 describes how to merge a subset of cliques $\mathcal{B}_m$ and update $\mathcal{B}$ and $\mathcal{E}$ accordingly. The algorithm first creates a merged clique $\mathcal{C}_m = \bigcup_{\mathcal{C} \in \mathcal{B}_m} \mathcal{C}$ by computing the set

Algorithm 8: Function mergeCliques$(\mathcal{B}, \mathcal{E}, \mathcal{B}_m)$.

Input : A set of cliques $\mathcal{B}$ with edge set $\mathcal{E}$, a subset of cliques $\mathcal{B}_m = \{\mathcal{C}_{m,1}, \mathcal{C}_{m,2}, \ldots, \mathcal{C}_{m,r}\} \subseteq \mathcal{B}$ to be merged.

Output : A reduced set of cliques $\hat{\mathcal{B}}$ with edge set $\hat{\mathcal{E}}$ and the merged clique $\mathcal{C}_m$.

$\hat{\mathcal{E}} \leftarrow \mathcal{E}$;

$\mathcal{C}_m \leftarrow \mathcal{C}_{m,1} \cup \mathcal{C}_{m,2} \cup \cdots \cup \mathcal{C}_{m,r}$;

$\hat{\mathcal{B}} \leftarrow (\mathcal{B} \setminus \mathcal{B}_m) \cup \{\mathcal{C}_m\}$;

Remove edges $\{(\mathcal{C}_i, \mathcal{C}_j) \mid i \neq j,\ \mathcal{C}_i, \mathcal{C}_j \in \mathcal{B}_m\}$ in $\hat{\mathcal{E}}$;

Replace edges $\{(\mathcal{C}_i, \mathcal{C}_j) \mid \mathcal{C}_i \in \mathcal{B}_m, \mathcal{C}_j \notin \mathcal{B}_m\}$ with $(\mathcal{C}_m, \mathcal{C}_j)$ in $\hat{\mathcal{E}}$;

union of the individual cliques in $\mathcal{B}_m$. It then replaces the individual cliques of $\mathcal{B}_m$ in $\mathcal{B}$ with $\mathcal{C}_m$. The last step replaces all edges to merged cliques in $\mathcal{B}_m$ with an edge to $\mathcal{C}_m$.

4.4.1 Existing clique tree-based strategies

The parent-child strategy described by Sun et al. [150] traverses the clique tree in a depth-first order and merges a clique $\mathcal{C}_\ell$ with its parent clique $\mathcal{C}_{\mathrm{par}(\ell)} \coloneqq \mathrm{par}(\mathcal{C}_\ell)$ if at least one of the two following conditions are met:

$$\begin{aligned}\left(\left|\mathcal{C}_{\mathrm{par}(\ell)}\right| - |\eta_\ell|\right)\left(|\mathcal{C}_\ell| - |\eta_\ell|\right) &\leq t_{\mathrm{fill}},\\ \max\left\{|\nu_\ell|, \left|\nu_{\mathrm{par}(\ell)}\right|\right\} &\leq t_{\mathrm{size}},\end{aligned}$$

with heuristic parameters t_{fill} and t_{size}. These conditions keep the amount of extra fill-in generated by the merge and the supernode cardinalities below the specified thresholds. The `SparseCoLO` strategy described by Nakata et al. [116] and Fujisawa et al. [51] considers parent-child as well as sibling relationships. Given a parameter $\sigma > 0$, two cliques $\mathcal{C}_i, \mathcal{C}_j$ are merged if the following merge criterion holds

$$\min\left\{\frac{|\mathcal{C}_i \cap \mathcal{C}_j|}{|\mathcal{C}_i|}, \frac{|\mathcal{C}_i \cap \mathcal{C}_j|}{|\mathcal{C}_j|}\right\} \geq \sigma. \tag{4.13}$$

This approach traverses the clique tree depth-first, performing the following steps for each clique $\mathcal{C}_\ell$:

1. For each clique pair $\{(\mathcal{C}_i, \mathcal{C}_j) \mid \mathcal{C}_i, \mathcal{C}_j \in \mathrm{ch}(\mathcal{C}_\ell)\}$, check if (4.13) holds, then:
   - $\mathcal{C}_i$ and $\mathcal{C}_j$ are merged, or
   - if $(\mathcal{C}_i \cup \mathcal{C}_j) \supseteq \mathcal{C}_\ell$, then $\mathcal{C}_i$, $\mathcal{C}_j$, and $\mathcal{C}_\ell$ are merged.
2. For each clique pair $\{(\mathcal{C}_i, \mathcal{C}_\ell) \mid \mathcal{C}_i \in \mathrm{ch}(\mathcal{C}_\ell)\}$, merge $\mathcal{C}_i$ and $\mathcal{C}_\ell$ if (4.13) is satisfied.

We note that the open-source implementation of the `SparseCoLO` algorithm described in [116] follows the algorithm outlined here but also employs a few additional heuristics, e.g. a final consolidation step where very small cliques are merged until a predefined minimum clique size is reached.

An advantage of these two approaches is that the clique tree can be computed easily and the conditions are inexpensive to evaluate. However, a disadvantage is that choosing parameters that work well on a variety of problems and solver algorithms is difficult. Secondly, clique trees are not unique and in some cases it is beneficial to merge cliques that are not directly related via parent-child or sibling relationships on the particular clique tree that was computed. This is shown in the following example.

Example 4.4.1 (Nephew-uncle merge)**.** Consider the chordal graph $G(V, E)$ consisting of three connected subgraphs:

$$\begin{aligned} G_a(V_a, E_a), &\text{ with } V_a = \{3, 4, \ldots, m_a\}, \\ G_b(V_b, E_b), &\text{ with } V_b = \{m_a + 2, m_a + 3, \ldots, m_b\}, \\ G_c(V_c, E_c), &\text{ with } V_c = \{m_b + 1, m_b + 2, \ldots, m_c\}, \end{aligned}$$

and some additional vertices $\{1, 2, m_a + 1\}$. The graph is connected as shown in Figure 4.13(a), where the complete subgraphs are represented as nodes V_a, V_b, V_c. A corresponding clique tree is shown in Figure 4.13(b). By choosing the cardinality

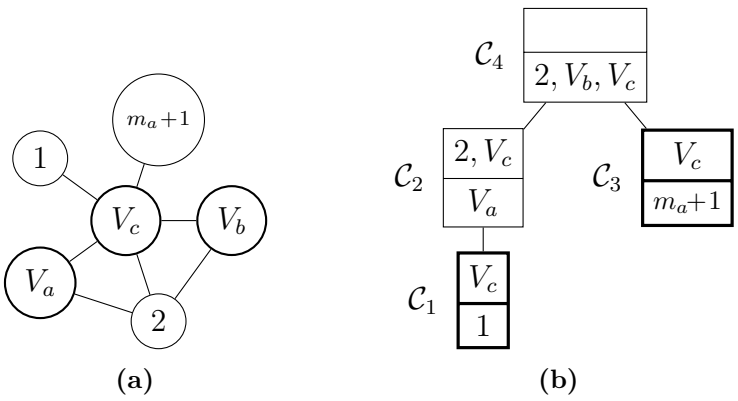


Figure 4.13: Sparsity graph (a) that can lead to clique tree (b) with an advantageous "nephew-uncle" merge between $\mathcal{C}_1$ and $\mathcal{C}_3$.

$|V_c|$, the overlap between cliques $\mathcal{C}_1 = \{1, 2\} \cup V_c$ and $\mathcal{C}_3 = \{m_a + 1\} \cup V_c$ can be made arbitrarily large while $|V_a|$, $|V_b|$ can be chosen so that any other merge is disadvantageous. However, neither the parent-child strategy nor `SparseCoLO` would consider merging $\mathcal{C}_1$ and $\mathcal{C}_3$ since they are in a "nephew-uncle" relationship. In fact

for the particular sparsity graph in Figure 4.13(a) eight different clique trees can be computed. Only in half of them do the cliques $\mathcal{C}_1$ and $\mathcal{C}_3$ appear in a parent-child relationship. Therefore, a merge strategy that only considers parent-child relationships would miss this favourable merge in half the cases.

4.4.2 A new clique graph-based strategy

To overcome the limitations of existing strategies we propose a merging strategy based on the *reduced clique graph* $\mathcal{G}(\mathcal{B}, \xi)$, which is given by the union of all possible clique trees of G; see [72] for a detailed discussion. To formally define the edge set of a reduced clique tree we first define the concept of a *separating pair*.

Definition 4.4.1 (Separating pair)**.** Following Habib and Stacho [72], we say that two cliques $\mathcal{C}_i$, $\mathcal{C}_j$ form a separating pair $\mathcal{P}_{ij} = \{\mathcal{C}_i, \mathcal{C}_j\}$ if their intersection is nonempty and if every path in the underlying graph G from a vertex in $\mathcal{C}_i \setminus \mathcal{C}_j$ to a vertex in $\mathcal{C}_j \setminus \mathcal{C}_i$ contains a vertex of the intersection $\mathcal{C}_i \cap \mathcal{C}_j$.

Similar to the clique tree, the vertices of the reduced clique graph are given by the maximal cliques $\mathcal{B}$ of the underlying sparsity graph. The edge set ξ is formed by introducing edges between any two cliques $(\mathcal{C}_i, \mathcal{C}_j)$ that form a separating pair $\mathcal{P}_{ij}$. Figure 4.14 shows the reduced clique graph $\mathcal{G}(\mathcal{B}, \xi)$ for an example graph. We

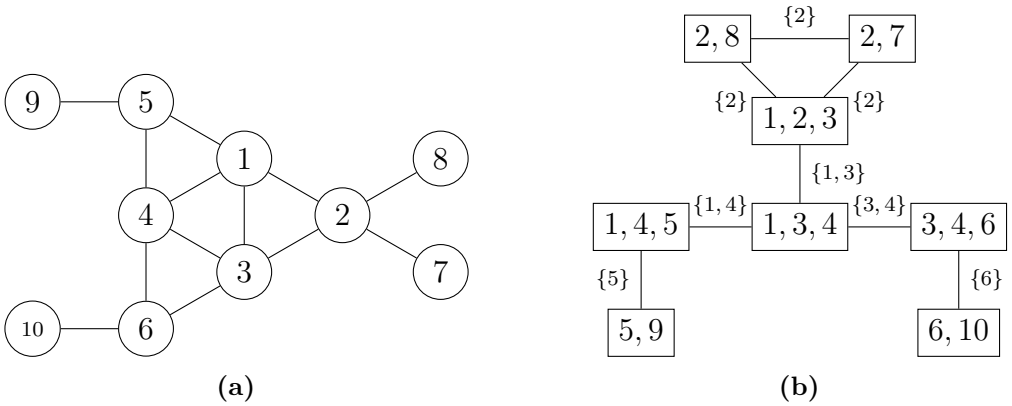


Figure 4.14: (a) Example graph and (b) corresponding reduced clique graph $\mathcal{G}(\mathcal{B}, \xi)$. Every edge between $\mathcal{C}_i$ and $\mathcal{C}_j$ is labelled with the intersection $\mathcal{C}_i \cap \mathcal{C}_j$.

note that ξ is a subset of the edges present in the *clique intersection graph* which is obtained by introducing edges for every two cliques that intersect. For example,

notice that in Figure 4.14(b) there is no edge between the cliques $\{1, 4, 5\}$ and $\{3, 4, 6\}$ even though they intersect with vertex 4. The reason is that these two cliques do not form a separating pair, e.g. in Figure 4.14(a) you can draw a path from vertex 5 to vertex 6 without passing vertex 4, e.g. $5 - 1 - 3 - 6$.

We are using the reduced clique graph instead of the clique intersection graph, because the reduced clique graph has the property that it remains a valid reduced clique graph of the altered sparsity pattern after performing a permissible merge between two cliques. This is not always the case for the clique intersection graph. Some merges can even break the chordality of the underlying sparsity graph; see Figure 4.15.

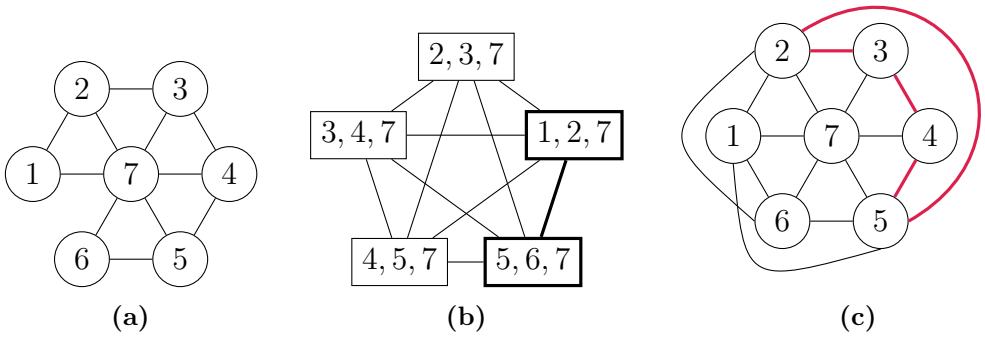


Figure 4.15: Merging of cliques $\{1, 2, 7\}$ and $\{5, 6, 7\}$ in the clique intersection graph (b) leads to non-chordal sparsity graph in (c) with a 4-cycle $2 - 3 - 4 - 5$.

Algorithm 9 shows how to compute the reduced clique graph-based on the set of cliques $\mathcal{B}$ and the set of minimum vertex separators $\mathcal{S}$ [71].

Algorithm 9: Computing the reduced clique graph.

Input : Set of cliques $\mathcal{B}$ and sortable list of separators $\mathcal{S}$.
Output : Reduced clique graph $\mathcal{G}(\mathcal{B}, \xi)$.

```
Sort S ∈ 𝒮 from highest cardinality to lowest;
Initialise ξ = ∅;
for S ∈ 𝒮 do
    for C_i ≠ C_j ∈ ℬ and (C_i, C_j) ∉ ξ do
        if C_i ∩ C_j ⊇ S and {C_i, C_j} is separating pair then
            ξ ← ξ ∪ (C_i, C_j);
```

For convenience, we will refer to the reduced clique graph simply as the clique graph in the following sections. Based on the permissibility condition for edge reduction in [72], we define a permissibility condition for clique merges.

Definition 4.4.2 (Permissible merge)**.**
Given a reduced clique graph $\mathcal{G}(\mathcal{B}, \xi)$, a merge between two cliques $(\mathcal{C}_i, \mathcal{C}_j) \in \xi$ is permissible if for every common neighbour $\mathcal{C}_k$ it holds that $\mathcal{C}_i \cap \mathcal{C}_k = \mathcal{C}_j \cap \mathcal{C}_k$.

The clique graph and the permissible merge condition gives us a pool of candidate merge pairs. Next, we need a way to evaluate the impact of a potential merge on the solve time of the algorithm. To do this we define an *edge weighting function* $e\colon 2^V \times 2^V \to \mathbb{R}$

$$e\left(\mathcal{C}_i, \mathcal{C}_j\right) = w_{ij}.$$

that assigns a weight w_{ij} to each edge $(\mathcal{C}_i, \mathcal{C}_j) \in \xi$. This function is used to estimate the per-iteration computational savings of merging a pair of cliques depending on the targeted algorithm and hardware. It evaluates to a positive number if a merge reduces the per-iteration time and to a negative number otherwise. For a first-order method, whose per-iteration cost is dominated by an eigenvalue factorisation with complexity $\mathcal{O}\left(|\mathcal{C}|^3\right)$, a simple choice would be:

$$e(\mathcal{C}_i, \mathcal{C}_j) = |\mathcal{C}_i|^3 + |\mathcal{C}_j|^3 - |\mathcal{C}_i \cup \mathcal{C}_j|^3. \tag{4.14}$$

More sophisticated weighting functions can be determined empirically; see Section 4.6.2. After a weight has been computed for each edge $(\mathcal{C}_i, \mathcal{C}_j)$ in the clique graph, we merge cliques as outlined in Algorithm 10. This strategy considers the edges in terms of their weights, starting with the permissible clique pair $(\mathcal{C}_i, \mathcal{C}_j)$ with the highest weight w_{ij}. If the weight is positive, the two cliques are merged and the edge weights for all edges connected to the merged clique $\mathcal{C}_m = \mathcal{C}_i \cup \mathcal{C}_j$ are updated. This process continues until no edges with positive weights remain.

The clique graph for the clique tree in Figure 4.8(c) is shown in Figure 4.16(a) with the edge weighting function in (4.14). Following Algorithm 10 the edge with the largest weight is considered first and the corresponding cliques are merged, i.e.

Algorithm 10: Clique graph-based merging strategy.

Input : A weighted clique graph $\mathcal{G}(\mathcal{B}, \xi)$.
Output : A merged clique graph $\mathcal{G}(\hat{\mathcal{B}}, \hat{\xi})$.

$\hat{\mathcal{B}} \leftarrow \mathcal{B}$ and $\hat{\xi} \leftarrow \xi$;
STOP $\leftarrow$ false;
while *!STOP* **do**
  choose permissible edge $(\mathcal{C}_i, \mathcal{C}_j)$ with maximum w_{ij};
  if $w_{ij} > 0$ **then**
    $\mathcal{B}_m \leftarrow \{\mathcal{C}_i, \mathcal{C}_j\}$;
    $\hat{\mathcal{B}}, \hat{\xi}, \mathcal{C}_m \leftarrow \text{mergeCliques}\left(\hat{\mathcal{B}}, \hat{\xi}, \mathcal{B}_m\right)$;
    for *each edge* $(\mathcal{C}_m, \mathcal{C}_\ell) \in \hat{\xi}$ **do**
      update $w_{m\ell} \leftarrow e(\mathcal{C}_m, \mathcal{C}_\ell)$;
  else
    STOP $\leftarrow$ true;

$\{3, 6, 7, 8\}$ and $\{6, 7, 8, 9\}$. Note that the merge is permissible because both cliques intersect with the only common neighbour $\{4, 5, 8\}$ in the same way. The revised clique graph $\mathcal{G}(\hat{\mathcal{B}}, \hat{\xi})$ is shown in Figure 4.16(b). Since no edges with positive weights remain, the algorithm stops.

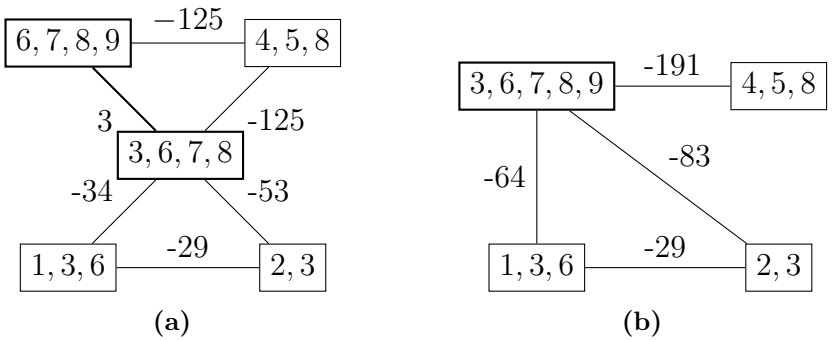


Figure 4.16: (a) Reduced clique graph $\mathcal{G}(\mathcal{B}, \xi)$ of the clique tree in Figure 4.8(c) with edge weighting function $e(\mathcal{C}_i, \mathcal{C}_j) = |\mathcal{C}_i|^3 + |\mathcal{C}_j|^3 - |\mathcal{C}_i \cup \mathcal{C}_j|^3$ and (b) clique graph $\mathcal{G}(\hat{\mathcal{B}}, \hat{\xi})$ after merging the cliques $\{3, 6, 7, 8\}$ and $\{6, 7, 8, 9\}$ and updating edge weights.

After Algorithm 10 has terminated, it is possible to recompute a valid clique tree from the revised clique graph. This can be done in two steps. First, the edge weights in $\mathcal{G}(\hat{\mathcal{B}}, \hat{\xi})$ are replaced with the cardinality of their intersection:

$$\tilde{w}_{ij} = |\mathcal{C}_i \cap \mathcal{C}_j|, \text{ for all } (\mathcal{C}_i, \mathcal{C}_j) \in \hat{\xi}.$$

Second, a clique tree is then given by any *maximum weight spanning tree* of the newly weighted clique graph and can be calculated *e.g.* using Kruskal's algorithm described in [90].

Our merging strategy has some clear advantages over competing approaches. Since the clique graph covers a wider range of merge candidates, it will consider edges that do not appear in clique tree-based approaches such as the "nephew-uncle" example in Figure 4.13. Moreover, the edge weighting function allows one to make a merge decision based on the particular solver algorithm and hardware used. One downside is that this approach is more computationally expensive than the other methods. However, numerical experiments show that the time spent on finding the clique graph, merging the cliques, and recomputing the clique tree typically only represents a very small fraction of the total computational savings relative to other merging methods when solving SDPs.

We emphasize that our clique graph-based merging strategy is independent of the solver algorithm. Thus, it can be adapted to work with other solvers, e.g. `MOSEK`, by specifying a different edge weighting function (4.14). The edge weighting function has to reflect how the change in both clique sizes and number of equality constraints affect the computation time of the main algorithm steps, which could be approximated analytically or experimentally.

4.5 Implementation

Chordal decomposition and clique merging is automatically applied as a preprocessing step in `COSMO` if a problem with decomposable PSD constraints is passed. The user can choose between three merge strategies: no merging, parent-child merging, or clique graph merging. If clique graph merging is chosen, they have the option to provide an edge-weighting function $e(\mathcal{C}_i, \mathcal{C}_j)$ to customise the merging strategy to their hardware. The solver algorithm performs the high-level steps shown in Algorithm 11. In a first step, the solver checks each PSD constraint for sparsity. Assuming the constraint is decomposable and representable by a

Algorithm 11: High-level solver steps with decomposition and clique merging.

Input : Problem data in solver format (3.14), merge strategy, edge-weighting function.
Output : Optimal solution (x^*, s^*, y^*), certificates of infeasibility $(\delta x, \delta y)$.

for *each PSD constraint* **do**
  if *constraint decomposable* **then**
    $G(V, E) \leftarrow$ compute sparsity graph;
    $G(V, \bar{E}) \leftarrow$ compute chordal extension;
    $\mathcal{G}(\mathcal{B}, \xi) \leftarrow$ compute reduced clique graph with Algorithm 9;
    merge cliques according to merge strategy, e.g. Algorithm 10;
    recompute a valid clique tree;
transform decomposed cones back into solver format (3.14), see Section 4.3.2;
$(x_d^*, s_d^*, y_d^*) \leftarrow$ solve decomposed problem with ADMM, see Algorithm 3;
if *optimal solution found* **then**
  $(x^*, s^*, y^*) \leftarrow$ undo the decomposition;
  for *each PSD constraint* **do**
    positive semidefinite completion of the dual variable y^*;
  return (x^*, s^*, y^*)
else
  return infeasibility certificates $(\delta x, \delta y)$;

graph with n vertices and m edges, the solver computes the sparsity graph and then uses a symbolic Cholesky factorisation, with complexity $\mathcal{O}(n^{3/2})$ [95] to find a chordal extension. Then the cliques and separators of the chordal graph are determined, e.g. using MCS at a cost of $\mathcal{O}(n + m)$ [159]. If clique graph merging is chosen as the merging strategy, the algorithm computes the reduced clique graph using Algorithm 9 with complexity $\mathcal{O}(mn)$ [71] and iteratively merges cliques according to Algorithm 10. Once all useful merges are processed a valid clique tree is computed from the reduced clique graph using Kruskal's algorithm at a cost of $\mathcal{O}(|\xi| \log(|\mathcal{B}|))$ [90]. Each merge step is upper bounded by $\mathcal{O}(|\xi| \log(|\xi|))$ for finding the highest weighted edge in $\mathcal{G}(\mathcal{B}, \xi)$. Consequently, assuming that the constraint is fairly sparse, i.e. $m \ll n^2$, the chordal decomposition and merge operations will be cheaper than the computationally expensive eigendecomposition of the projection step.

After the final decomposition is found for each decomposable PSD constraint, the solver transforms the decomposed constraints into the standard constraint problem format $Ax + s = b$, $s \in \mathcal{K}$. Non-decomposable PSD constraints and other cone constraints are left untouched as can be seen in the following example.

Example 4.5.1. The considered problem has three inequality constraints, one second-order cone constraint of dimension 5, and three PSD constraints of dimensions 4, 5, and 4. The slack variable $s \in \mathcal{K}$ is partitioned in the following way:

$$s = (s_l, s_q, s_{\text{psd},1}, s_{\text{psd},2}, s_{\text{psd},3}) \in \mathbb{R}^3_+ \times \mathcal{K}^5_{soc} \times \mathbb{S}^4_+ \times \mathbb{S}^5_+ \times \mathbb{S}^4_+.$$

The decomposition step decomposes the first PSD constraint into three constraints each of dimension 2. The second PSD constraint does not exhibit any structure. The third PSD constraint is decomposed into two constraints of dimension 3. After transformation into a solver compatible format the transformed slack variable s_t is partitioned as

$$\begin{aligned} s_t &= (s_l, s_q, s^1_{\text{psd},1}, s^2_{\text{psd},1}, s^3_{\text{psd},1}, s_{\text{psd},2}, s^1_{\text{psd},3}, s^2_{\text{psd},3}) \\ &\in \mathbb{R}^3_+ \times \mathcal{K}^5_{soc} \times \mathbb{S}^2_+ \times \mathbb{S}^2_+ \times \mathbb{S}^2_+ \times \mathbb{S}^5_+ \times \mathbb{S}^3_+ \times \mathbb{S}^3_+. \end{aligned}$$

After the decomposition and backtransformation step, the solver attempts to compute an optimal solution (x^*_d, s^*_d, y^*_d) of the decomposed problem. If an optimal solution is found, the decomposition is undone and the solution of the original problem (x^*, y^*, s^*) is returned. If the problem is infeasible, the solver returns the appropriate certificates as discussed in Section 3.6.

When an ADMM algorithm is used to solve the decomposed problem most of the per-iteration computations are done in the function that projects the iterate s^k onto the constraint cone $\mathcal{K}$. It was mentioned before that if the problem's constraint cone $\mathcal{K}$ is a Cartesian product of multiple smaller cones, then the projection step in (3.25) decomposes into multiple independent projections of parts of s_k onto the respective cones. Thus, the projections can be carried out in parallel. This property is especially useful if the problem naturally has many PSD constraints or if one (or

several) PSD constraints are decomposed into multiple smaller ones, see Figure 4.17. One can see that the advantages of decomposing a large PSD constraint into a

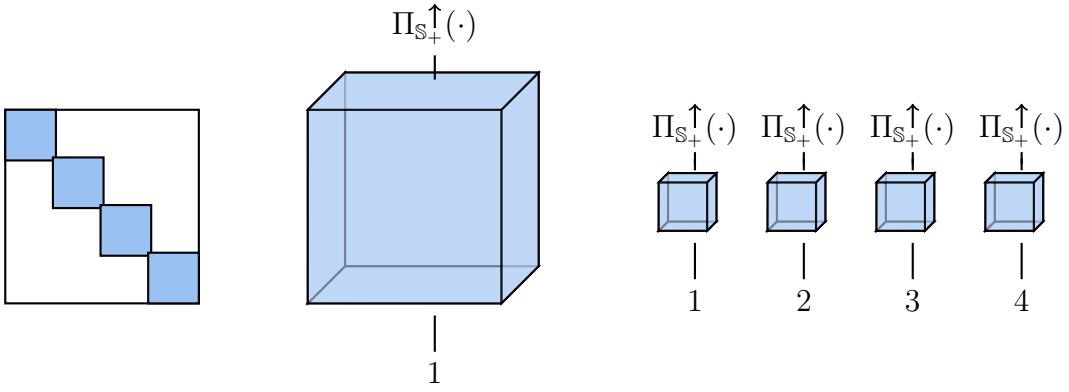


Figure 4.17: Sparsity pattern (left), projection using a single thread (middle), chordal decomposition and multithreaded projections (right). The volume represents the amount of computation for one projection.

number of smaller PSD constraints are two-fold. First, the amount of time to project all the individual matrix blocks will typically be lower than projecting the original matrix. Second, the decomposition enables the parallel execution of the projections on multiple CPU threads.

For the eigendecomposition involved in the projection step of a PSD constraint, the LAPACK [8] function `sveyr` is used, which can also utilise multiple threads. Consequently, this leads to two-level parallelism in the computation, i.e. on the higher level the projection functions are carried out in parallel and each projection function independently calls `sveyr`. Determining the optimal allocation of the CPU cores to each of these tasks depends on the number of PSD constraints and their dimensions and is a difficult problem. The impact of using different number of Julia threads and different BLAS settings on the accumulated projection time is shown in Table 4.1. The algorithm variables are stored in vector form. The implemented projection function includes the de-vectorisation of the input data to reshape it into matrix form. It also includes the vectorisation of the matrix after it has been projected using (2.13).

For the BLAS settings we compare the cases that each call to `sveyr` is restricted to one single thread or that it gets automatically chosen by BLAS. The impact of the

Table 4.1: Impact of multithreading and BLAS settings on projection times.

		BLAS single-threaded[2]		BLAS multi-threaded[3]	
	threads[1]	solve time (s)	proj. time (s)	solve time (s)	proj. time (s)
Same size blocks	1	56.29	30.72	58.33	32.06
	2	42.04	16.16	43.05	16.41
	4	34.10	8.51	37.08	8.87
	8	30.39	4.77	36.41	8.54
	16	31.12	4.47	37.64	9.72
Dominant block	1	27.58	8.48	26.63	6.53
	2	28.08	7.46	28.48	8.30
	4	26.03	6.92	26.70	6.60
	8	27.81	7.04	28.96	7.89
	16	26.44	6.92	34.44	12.45
maxG32	1	17.00	13.82	19.74	16.24
	2	11.95	8.80	33.85	30.30
	4	8.04	4.87	44.66	37.41
	8	6.27	3.02	53.05	44.37
	16	5.76	2.44	77.23	67.90
mcp500-4	1	9.34	8.10	7.64	6.24
	2	8.59	7.35	9.25	7.87
	4	8.39	7.16	8.42	7.03
	8	8.23	6.97	8.82	7.42
	16	8.17	6.91	11.60	10.09

[1] number of Julia threads with MKL BLAS on 8 physical (16 logical) Intel Xeon E5-2560 cores;
[2] BLAS calls are restricted to one CPU thread;
[3] BLAS calls can request number of threads automatically (independent of Julia threads)

settings is shown for four different problems. We generate a best case SDP problem with block-arrow sparsity pattern, that can be decomposed into an equivalent problem with 120 10×10 PSD constraints. Next, we generate a similar problem, that decomposes into a problem with one dominant 400×400 block and 80 10×10 blocks. We further benchmarked the set of decomposable SDPLib problems and selected one example where the impact of multithreading on the projection time was large (`maxG32`) and one where it was small (`mcp500-4`).

The results show that if the BLAS calls are restricted to a single thread, adding more physical Julia threads generally improves the projection time and the overall solve time. As expected, this is especially the case for the SDP with blocks of similar size. It also works well for `maxG32`, which after decomposition and clique merging

has eight large blocks with a dimension between 82 and 102 and 656 smaller blocks of much smaller size. For these two problems multithreaded BLAS seems to cause a slowdown, likely due to an oversubscription of CPU threads. For the SDP with a dominant block and `mcp500-4`, the benefit of adding more Julia threads is much less significant. Indeed, after decomposition and merging `mcp500-4` ends up with one block of dimension 393 and 86 smaller blocks with dimensions less than 44. For the problems with a dominant block, it seems to be beneficial to use multiple BLAS threads to speed up the projection of any large blocks and then to serially project the smaller blocks afterwards.

Overall it seems that a good configuration, that can handle various problems, is to run `sveyr` single-threaded and to perform the projection in parallel on all available physical cores.

4.6 Benchmark results

This section presents numerical results that showcase the computational benefits of chordal decomposition and clique merging. We benchmark our solver `COSMO` against the interior-point solver `MOSEK` v9.0 and the accelerated first-order ADMM solver `SCS` v2.1.1. The same hardware, error measures and solver configurations were used as in Section 3.9.

In Section 4.6.1 we show how chordal decomposition can speed up the solver for SDPs that exhibit a block-arrow sparsity pattern. The problem structure is chordal by default and well-suited for chordal decomposition. Moreover, for this problem type no clique merging was used as the structure decomposes with minimal overlap. In Section 4.6.2 we consider different non-chordal sparsity patterns from various applications. In particular we solve large structured problems from the SDPLib benchmark set [23] and generate very large non-chordal SDPs with sparsity patterns from the `SuiteSparse` Matrix Collections [39]. For these two problem sets we first compare our clique graph-based merging strategy to the other strategies discussed

in Section 4.4. Afterwards, we benchmark COSMO with clique graph merging against other conic solvers.

4.6.1 Block-arrow sparse SDPs

To demonstrate the benefits of the chordal decomposition discussed in Section 4.3, we consider randomly generated SDPs of the form (4.6) with a block-arrow aggregate sparsity pattern similar to test problems in [5, 181]. Figure 4.18 shows the sparsity pattern of the PSD constraint. The sparsity pattern is generated based on the following parameters: block size d, number of blocks N_b and width of the arrow head w. Note that the graph corresponding to the sparsity pattern is always chordal and that, for this sparsity pattern, clique merging yields no benefit. In the following

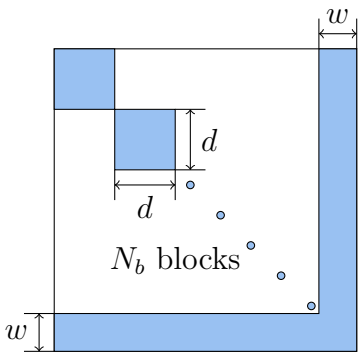


Figure 4.18: Parameters of block-arrow sparsity pattern. The shaded area represents the non-zeros of the sparsity pattern.

we study the effects of independently increasing the block size d and the number of blocks N_b. The parameters for the two test cases are:

- Varying the number of blocks: $N_b = 50, 60, \ldots, 140$, $d = 10$, $w = 20$, and $m = 100$. This leads to PSD cone sizes between 520–1420 and number of nonzeros in the constraint matrix A between 6.5×10^5–1.8×10^6.
- Varying the block size: $d = 10, 12, \ldots, 28$, $N_b = 50$, $w = 10$, and $m = 100$. This leads to PSD cone sizes between 510–1410 and number of nonzeros in the constraint matrix A between 3.9×10^5–1.7×10^6.

The solve times, the iterations and maximum errors for all solvers are shown in Table 4.2. The solve times for all solvers are also shown in Figure 4.19

Table 4.2: Benchmark results for block arrow sparse SDPs with varying number of blocks N_b and block size d.

	Solve time (s)				Iterations				Max error[2]			
d	COSMO	COSMO[cd]	MOSEK	SCS	COSMO	COSMO[cd]	MOSEK	SCS	COSMO	COSMO[cd]	MOSEK	SCS
10	12.71	**1.22**	16.5	56.99	125	175	9	1340	1.50×10^{-4}	5.60×10^{-4}	1.16×10^{-3}	9.01×10^{-4}
12	23.95	**2.3**	28.72	68.72	125	225	10	1120	2.29×10^{-4}	7.52×10^{-4}	3.65×10^{-5}	5.38×10^{-4}
14	27.4	**2.39**	41.66	147.72	150	175	10	1780	7.92×10^{-5}	8.60×10^{-4}	6.84×10^{-4}	5.46×10^{-4}
16	36.4	**3.6**	58.04	254.93	150	275	10	2360	4.76×10^{-5}	5.34×10^{-4}	2.64×10^{-4}	6.91×10^{-4}
18	42.96	**4.42**	85.81	472.77	150	275	11	3440	2.66×10^{-4}	5.77×10^{-4}	2.55×10^{-4}	3.25×10^{-4}
20	41.13	**5.08**	112.56	251.39	125	275	11	1460	2.61×10^{-4}	5.35×10^{-4}	6.33×10^{-5}	7.26×10^{-4}
22	50.7	**11.47**	142.14	520.81	125	700	11	2460	2.87×10^{-4}	5.85×10^{-4}	3.31×10^{-4}	4.86×10^{-4}
24	125.12	**10.52**	178.75	445.37	275	500	11	1780	7.46×10^{-5}	7.23×10^{-4}	3.57×10^{-4}	5.10×10^{-4}
26	88.74	**6.84**	220.26	623.46	150	200	11	2060	8.17×10^{-5}	3.27×10^{-4}	2.89×10^{-6}	1.39×10^{-4}
28	98.83	**7.47**	263.52	1626.82	150	200	11	4660	3.73×10^{-5}	8.93×10^{-4}	2.62×10^{-5}	4.59×10^{-4}
N_b	COSMO	COSMO[cd]	MOSEK	SCS	COSMO	COSMO[cd]	MOSEK	SCS	COSMO	COSMO[cd]	MOSEK	SCS
50	17.31	**3.73**	28.69	115.36	150	300	10	2540	1.33×10^{-4}	6.28×10^{-4}	2.21×10^{-4}	9.12×10^{-4}
60	21.66	**6.4**	46.38	217.75	150	475	11	3380	1.42×10^{-4}	8.74×10^{-4}	7.27×10^{-5}	8.81×10^{-4}
70	34.51	**9.89**	64.8	177.24	175	700	11	2060	4.12×10^{-5}	6.14×10^{-4}	1.72×10^{-5}	8.99×10^{-4}
80	37.22	**5.78**	85.29	175.38	175	275	11	1540	5.15×10^{-5}	7.13×10^{-4}	2.08×10^{-4}	4.57×10^{-4}
90	63.32	**12.85**	113.24	234.56	225	550	11	1580	4.80×10^{-5}	1.12×10^{-3}	4.79×10^{-5}	6.40×10^{-4}
100	67.3	**25.16**	140.81	328.2	200	1100	11	1800	6.38×10^{-5}	9.54×10^{-4}	3.28×10^{-5}	8.59×10^{-4}
110	117.7	**39.86**	172.99	761.22	275	1650	11	3140	3.78×10^{-5}	6.80×10^{-4}	1.35×10^{-4}	3.53×10^{-4}
120	81.31	**29.06**	209.96	1191.64	150	1050	11	4620	3.59×10^{-4}	1.07×10^{-3}	3.66×10^{-4}	2.71×10^{-4}
130	115.21	**26.61**	269.06	1094.53	175	925	12	3580	8.48×10^{-5}	1.31×10^{-3}	8.59×10^{-5}	9.67×10^{-4}
140	114.95	**26.87**	290.23	***[†]	150	800	11	***	1.20×10^{-4}	1.92×10^{-3}	4.99×10^{-4}	***

[cd] with chordal decomposition; [2] $\max\{\epsilon_1, \epsilon_2, \epsilon_3\}$; [†] time limit reached

and Figure 4.20. The line `COSMO(CD)` corresponds to the solver with chordal decomposition enabled.

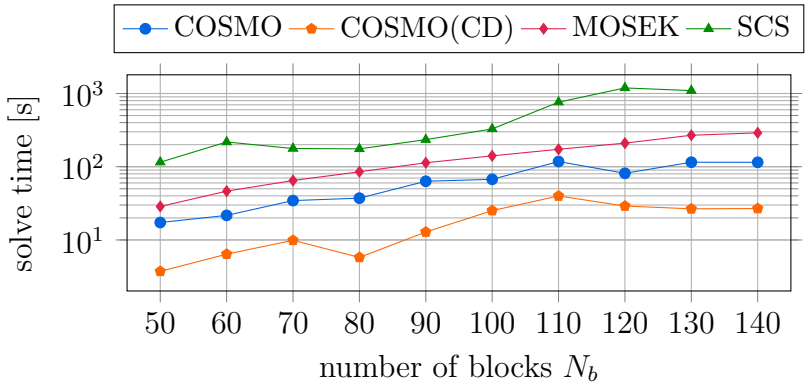


Figure 4.19: Solve time for increasing number of blocks N_b of block-arrow sparsity pattern.

The figures show that `COSMO` with and without chordal decomposition solves this problem type consistently faster than `MOSEK` and `SCS`. The reason why `COSMO` performs better than the other first-order solver `SCS` can be explained by the significantly lower number of iterations (Table 4.2). Furthermore, one can see that in both cases the solver time for each solver rises when the number of blocks and the block sizes are increased. The increase is smaller for `COSMO(CD)` which is more

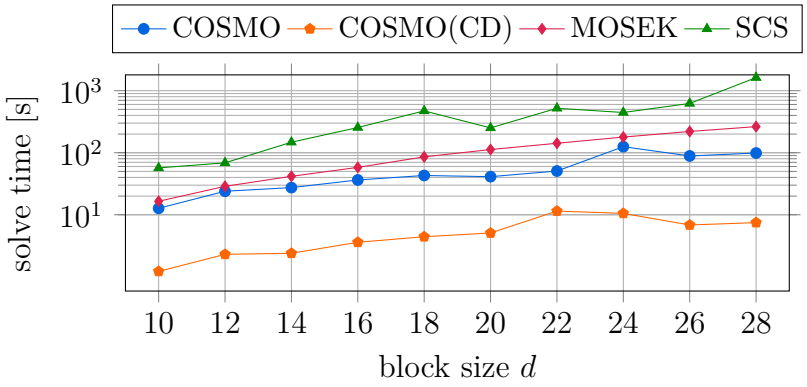


Figure 4.20: Solve time for increasing block size d of block-arrow sparsity pattern.

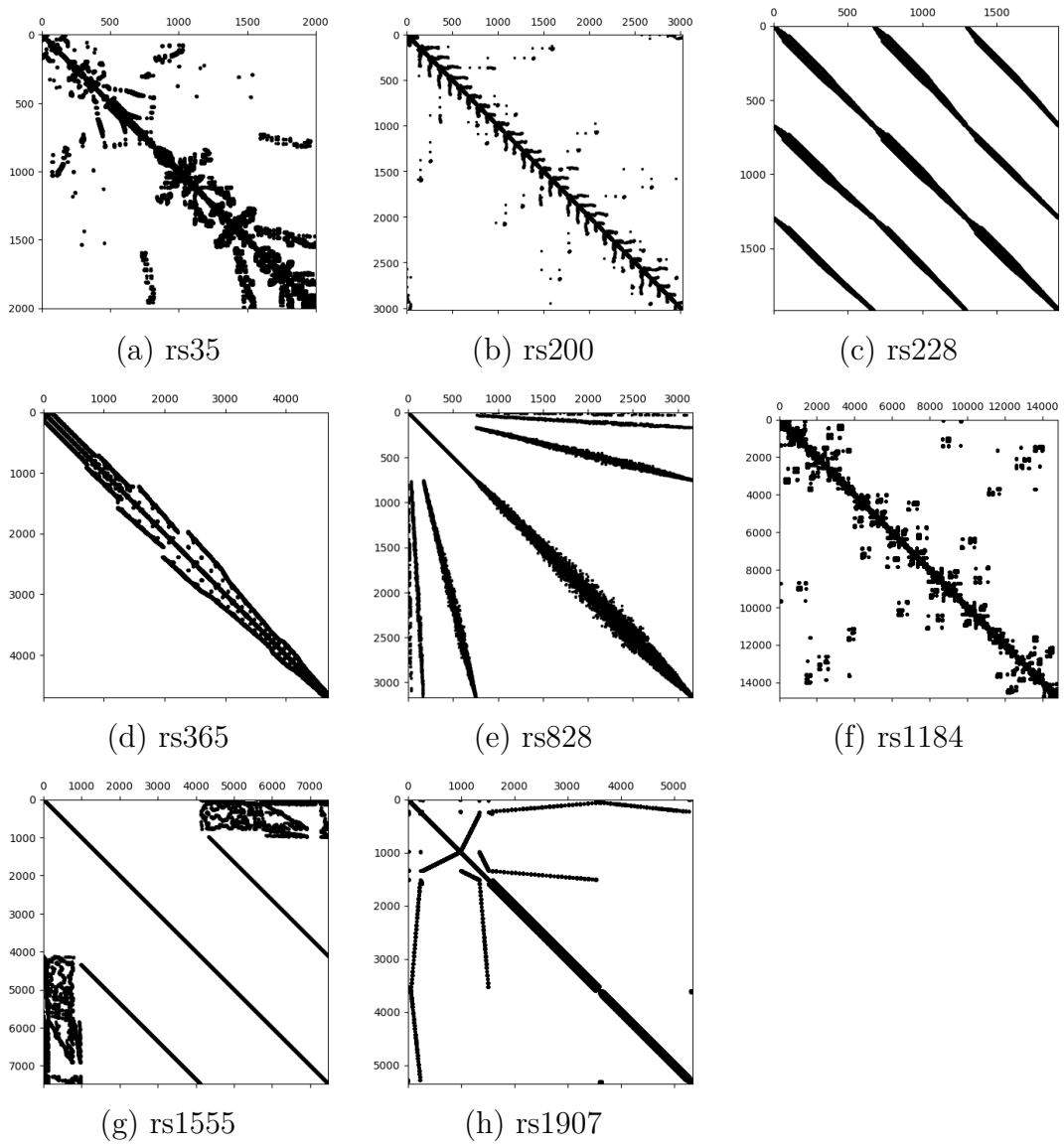
affected by the number of iterations than the problem dimension.

4.6.2 Non-chordal problems with clique merging

To compare our proposed clique graph-based merge approach with the clique tree-based strategies of [116] and [150], all three methods discussed in Section 4.4 were used to preprocess large sparse SDPs from SDPLib, a collection of SDP benchmark problems [23]. This problem set contains maximum cut problems, SDP relaxations of quadratic programs and Lovász theta problems. Moreover, we consider a set of test SDPs generated from (non-chordal) sparsity patterns of matrices from the `SuiteSparse` Matrix Collections [39]. The sparsity patterns for these problems are shown in Figure 4.21. Details on the problem dimensions are shown in Table 4.3.

Table 4.3: PSD cone size, number of affine constraints, number of nonzeros in A for sparse SDPs.

problem	cone size	aff. constr.	nnz in A	problem	cone size	aff. constr.	nnz in A
maxG11	800	800	8.00×10^2	maxG32	2000	2000	2.00×10^3
maxG51	1000	1000	1.00×10^3	mcp500-1	500	500	5.00×10^2
mcp500-2	500	500	5.00×10^2	mcp500-3	500	500	5.00×10^2
mcp500-4	500	500	5.00×10^2	qpG11	1600	800	1.60×10^3
qpG51	2000	1000	2.00×10^3	thetaG11	801	2401	1.04×10^4
thetaG51	1001	6910	3.65×10^4				
rs1184	14 822	5	1.83×10^6	rs1555	7479	200	7.34×10^6
rs1907	5357	200	2.13×10^7	rs200	3025	200	2.39×10^6
rs228	1919	200	3.43×10^6	rs35	2003	200	8.59×10^6
rs365	4704	200	1.09×10^7	rs828	3169	200	2.40×10^6



(a) rs35 (b) rs200 (c) rs228

(d) rs365 (e) rs828 (f) rs1184

(g) rs1555 (h) rs1907

Figure 4.21: Aggregate sparsity pattern of non-chordal SDPs created from matrices of the `SuiteSparse` Matrix Collection. The patterns are labeled with their ID number.

Both problem sets were used in the past to benchmark structured SDPs [5, 181]. This section discusses how the different decompositions affect the per-iteration computation times of the solver. In a second step we compare the solver time of `COSMO` with our clique graph merging strategy to those of `MOSEK` and `SCS`.

For the strategy described in [116] we used the `SparseCoLO` package to decompose the problem. The parent-child method discussed in [150] and the clique graph-based method described in Section 4.4.2 are available as options in `COSMO`. We further investigate the effect of using different edge weighting functions. The major

operation affecting the per-iteration time is the projection step. This step involves an eigenvalue decomposition of the matrices corresponding to the cliques. Since the eigenvalue decomposition of a symmetric matrix of dimension N has a complexity of $\mathcal{O}(N^3)$, we define a *nominal* edge weighting function as in (4.14). However, the exact relationship will be different because the projection function involves copying of data and is affected by hardware properties such as cache size. We therefore also consider an empirically *estimated* edge weighting function. To determine the relationship between matrix size and projection time, the execution time of the relevant function inside `COSMO` was measured for different matrix sizes. We then approximated the relationship between projection time, t_{proj}, and matrix size, N, as a polynomial:

$$t_{\text{proj}}(N) = aN^3 + bN^2,$$

where a, b were estimated using least-squares (Figure 4.22). The estimated weighting function is then defined as

$$e(\mathcal{C}_i, \mathcal{C}_j) = t_{\text{proj}}(|\mathcal{C}_i|) + t_{\text{proj}}(|\mathcal{C}_j|) - t_{\text{proj}}(|\mathcal{C}_i \cup \mathcal{C}_j|). \tag{4.15}$$

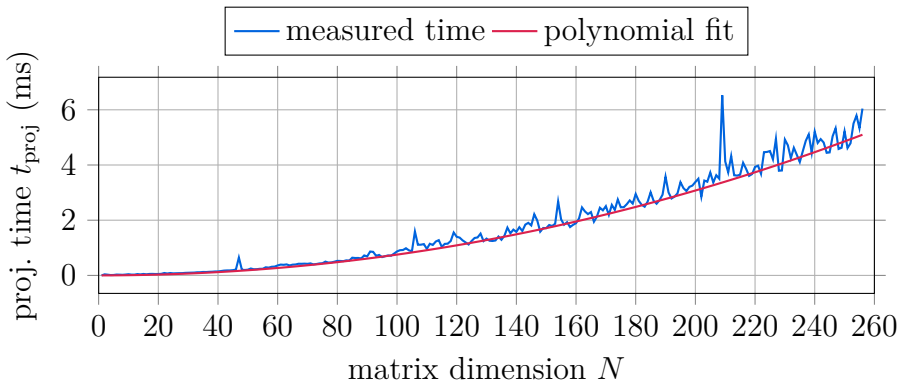


Figure 4.22: Measured and estimated relationship between matrix size and execution time of the projection function in `COSMO`.

Six different cases were considered: no decomposition (`NoDe`), no clique merging (`NoMer`), decomposition using `SparseCoLO` (`SpCo`), parent-child merging (`ParCh`), and the clique graph-based method with nominal edge weighting (`CG1`) and estimated edge weighting (`CG2`). `SparseCoLO` was used with default parameters. All cases were

run single-threaded. Since the per-iteration projection times for some problems lie in the millisecond range every problem was benchmarked ten times and the median values are reported. Table 4.4 shows the solve time, the mean projection time, the number of iterations, the number of cliques after merging, and the maximum clique size of the sparsity pattern for each problem and strategy. The solve time includes the time spent on decomposition and clique merging. We do not report the total solver time when `SparseCoLO` was used for the decomposition because this has to be done in a separate preprocessing step in `MATLAB`, which was orders of magnitude slower than the other methods.

The results show that our clique graph-based methods lead to a reduction in overall solver time. The method with estimated edge weighting function `CG2` achieves the lowest average projection times for the majority of problems. In four cases `ParCh` has a narrow advantage. The geometric mean of the ratios of projection time of `CG2` compared to the best non-graph method is 0.701, with a minimum ratio of 0.407 for problem `mcp500-2`. There does not seem to be a clear pattern that relates the projection time to the number of cliques or the maximum clique size of the decomposition. This is expected as the optimal merging strategy depends on the properties of the initial decomposition such as the overlap between the cliques. The merging strategies `ParCh`, `CG1` and `CG2` generally result in similar maximum clique sizes compared to `SparseCoLO`, with `CG1` being the most conservative in the number of merges.

Next, we benchmarked `COSMO` with merging strategy `CG2` against two other solvers, `MOSEK`, and `SCS`. Table 4.5 shows the solve time, number of iterations and maximum error.

The decomposition helps `COSMO` to solve the majority of problems faster than `MOSEK` and `SCS`. This is even more significant for the larger problems that were generated from the `SuiteSparse` Matrix Collection. The decomposition does not seem to provide a major benefit for the slightly denser problems `mcp500-3` and `mcp500-4`. Furthermore, `COSMO` seems to converge slowly for `qpG51` and `thetaG51`. Similar

Table 4.4: Benchmark results for non-chordal sparse SDPs from SDPLib and SDPs generated with sparsity patterns from the `SuiteSparse` Matrix Collection for different merging strategies.

problem	Median Solve time (s)						Median Projection time (ms)					
	NoDe[1]	NoMer[2]	SpCo[3]	ParCh[4]	CG1[5]	CG2[6]	NoDe	NoMer	SpCo	ParCh	CG1	CG2
maxG11	35.85	4.69	∗∗∗	3.33	3.09	**2.93**	143.3	18.4	12.7	**10.5**	13.4	11.2
maxG32	407.82	25.95	∗∗∗	15.62	**14.38**	19.56	1257.7	73.2	73.1	49.9	51.6	**43.1**
maxG51	40.61	32.8	∗∗∗	9.56	**6.93**	9.62	245.8	230.4	219.3	122.2	65.5	**58.5**
mcp500-1	21.46	1.33	∗∗∗	0.7	0.89	**0.6**	56.0	7.7	7.4	**4.6**	6.1	4.9
mcp500-2	13.55	12.28	∗∗∗	8.37	2.6	**2.42**	54.9	45.4	32.1	28.0	14.3	**11.4**
mcp500-3	11.28	32.45	∗∗∗	30.13	7.01	**5.14**	51.1	104.9	105.0	83.4	28.2	**21.2**
mcp500-4	14.57	72.27	∗∗∗	13.87	**5.84**	7.67	59.2	253.8	193.8	141.2	44.1	**35.8**
qpG11	142.31	7.3	∗∗∗	4.79	**4.62**	4.7	305.1	20.0	12.3	**10.9**	15.6	13.2
qpG51	450.74	186.49	∗∗∗	**89.81**	150.86	132.68	523.6	247.2	246.9	134.1	73.9	**65.2**
thetaG11	332.06	9.43	∗∗∗	9.43	9.46	**6.81**	477.5	21.9	18.9	**12.5**	16.3	14.5
thetaG51	1062.91	110.64	∗∗∗	107.02	**37.22**	85.92	252.8	230.8	∗∗∗[†]	131.7	66.5	**53.7**
rs1184	∗∗∗[m]	1227.48	∗∗∗	882.27	632.96	**569.29**	∗∗∗[m]	4192.8	3495.8	3424.1	2483.9	**2301.3**
rs1555	∗∗∗[†]	79.83	∗∗∗	**65.93**	80.72	83.84	∗∗∗[†]	316.7	242.7	268.8	160.7	**132.1**
rs1907	∗∗∗[†]	233.86	∗∗∗	197.99	178.79	**166.23**	∗∗∗[†]	483.8	490.3	455.3	382.7	**352.1**
rs200	640.0	31.33	∗∗∗	21.09	24.78	**19.29**	3366.2	121.6	93.0	82.9	93.2	**71.7**
rs228	206.2	40.88	∗∗∗	27.79	25.29	**18.6**	1220.2	116.0	59.4	76.2	67.1	**50.9**
rs35	296.52	196.93	∗∗∗	146.56	88.86	**71.25**	1269.8	548.1	358.2	404.6	272.5	**223.2**
rs365	∗∗∗[†]	159.75	∗∗∗	127.77	110.48	**92.5**	∗∗∗[†]	433.1	364.6	351.1	289.9	**262.0**
rs828	603.55	29.86	∗∗∗	19.24	23.25	**17.81**	3716.7	113.2	80.0	71.1	87.5	**64.2**

problem	Iterations						Number of cliques / Maximum clique size					
	NoDe	NoMer	SpCo	ParCh	CG1	CG2	NoDe	NoMer	SpCo	ParCh	CG1	CG2
maxG11	225	225	500	275	200	225	1/800	598/24	13/80	198/32	473/28	407/36
maxG32	300	300	425	250	225	375	1/2000	1498/76	21/210	481/76	1164/92	664/102
maxG51	150	100	75	50	75	125	1/1000	674/326	181/322	163/326	448/362	313/395
mcp500-1	350	150	150	125	125	100	1/500	457/39	451/44	105/43	437/54	361/65
mcp500-2	225	225	200	250	150	175	1/500	363/138	144/138	96/140	316/156	226/171
mcp500-3	200	250	250	300	200	200	1/500	259/242	101/242	71/242	211/263	135/285
mcp500-4	225	225	375	75	100	175	1/500	161/340	63/346	46/341	105/368	87/393
qpG11	400	325	525	375	250	300	1/1600	1398/24	813/80	287/32	1273/28	1207/36
qpG51	750	600	800	550	1800	1825	1/2000	1674/326	1182/304	275/326	1448/362	1313/395
thetaG11	675	375	2275	650	500	400	1/801	598/25	13/81	198/33	494/29	423/41
thetaG51	3825	325	∗∗∗[†]	575	375	1250	1/1001	676/324	150/323	157/324	424/358	267/396
rs1184	∗∗∗[m]	225	200	200	200	200	1/14822	2236/500	78/1330	1043/500	664/608	258/632
rs1555	∗∗∗[†]	150	150	150	150	175	1/7479	6891/184	3350/187	2556/184	5529/236	4858/276
rs1907	∗∗∗[†]	200	200	175	175	200	1/5357	577/261	47/585	419/261	441/324	219/405
rs200	175	125	125	125	125	125	1/3025	1632/95	94/216	444/95	1123/112	583/119
rs228	150	125	125	125	125	125	1/1919	790/88	48/180	255/88	369/95	129/127
rs35	200	175	200	175	150	150	1/2003	589/343	53/735	189/343	214/457	106/520
rs365	∗∗∗[†]	175	175	175	175	175	1/4704	1230/296	110/350	539/296	688/349	346/474
rs828	150	125	150	125	125	125	1/3169	1875/86	112/174	501/86	1378/102	708/126

[†] time limit reached; [m] out of memory error; [1] no decomposition; [2] no merging; [3] `SparseCoLO` merging;
[4] parent-child merging; [5] clique graph with nominal edge weighting (4.14);
[6] clique graph with estimated edge weighting (4.15);

observations for `mcp500-3`, `mcp500-4` and `thetaG51` have been made by Andersen et al. [5]. Finally, many of the larger problems were not solvable within the time limit or caused out-of-memory problems if no decomposition was used in `MOSEK` and `SCS`.

Table 4.5: Benchmark results for non-chordal sparse SDPs from SDPLib and SDPs generated with sparsity patterns from the `SuiteSparse` Matrix Collection.

	Solve time (s)			Iterations			Max error[2]		
problem	COSMO[1]	MOSEK	SCS	COSMO[1]	Mosek	SCS	COSMO[1]	MOSEK	SCS
maxG11	**1.47**	4.45	131.8	225	5	1220	7.84×10^{-4}	1.98×10^{-3}	8.74×10^{-4}
maxG32	**6.25**	50.84	840.79	375	5	1220	9.74×10^{-4}	2.76×10^{-3}	6.87×10^{-4}
maxG51	**8.09**	9.92	36.56	125	8	180	2.31×10^{-3}	2.83×10^{-4}	8.85×10^{-4}
mcp500-1	**0.24**	1.7	29.28	100	7	580	1.02×10^{-3}	1.52×10^{-3}	6.31×10^{-4}
mcp500-2	**1.68**	1.75	17.36	175	7	380	6.82×10^{-4}	7.37×10^{-4}	3.30×10^{-4}
mcp500-3	4.41	**1.68**	8.36	200	6	180	2.23×10^{-3}	4.18×10^{-4}	6.86×10^{-4}
mcp500-4	8.2	**1.76**	7.4	175	7	160	1.65×10^{-3}	2.79×10^{-4}	3.43×10^{-4}
qpG11	**2.36**	26.23	734.7	300	7	1820	4.57×10^{-4}	1.28×10^{-3}	9.77×10^{-4}
qpG51	121.6	**96.42**	527.55	1825	14	800	7.56×10^{-3}	4.80×10^{-4}	9.83×10^{-4}
thetaG11	**2.32**	8.53	142.53	400	9	1380	1.43×10^{-3}	1.11×10^{-3}	8.31×10^{-5}
thetaG51	71.21	**50.08**	967.43	1250	11	3240	1.38×10^{-1}	4.03×10^{-4}	9.94×10^{-4}
rs1184	**224.86**	$***^{m}$	$***^{m}$	200	***	***	5.65×10^{-4}	***	***
rs1555	**66.6**	$***^{\dagger}$	$***^{m}$	175	***	***	5.32×10^{-4}	***	***
rs1907	**104.61**	$***^{\dagger}$	$***^{m}$	200	***	***	2.66×10^{-4}	***	***
rs200	**12.47**	752.27	$***^{\dagger}$	125	11	***	1.87×10^{-4}	6.11×10^{-4}	***
rs228	**12.86**	395.24	982.5	125	11	1620	2.15×10^{-4}	6.34×10^{-4}	8.27×10^{-4}
rs35	**54.88**	919.19	$***^{\dagger}$	150	12	***	2.48×10^{-4}	3.26×10^{-4}	***
rs365	**62.65**	$***^{\dagger}$	$***^{\dagger}$	175	***	***	3.08×10^{-4}	***	***
rs828	**10.84**	825.03	$***^{\dagger}$	125	11	***	1.98×10^{-4}	8.48×10^{-4}	***

[1] with chordal decomposition and clique merging strategy `CG2`; [2] $\max\{\epsilon_1, \epsilon_2, \epsilon_3\}$; [†] time limit reached; [m] out of memory error;

4.7 Conclusions

In this chapter we gave a brief description of how graph theory concepts link to the sparsity of a matrix. We further examined how the concept of cliques and decomposition theorems can be used to chordally decompose large structured SDPs into equivalent problems that can be easier to solve by orders of magnitude. After introducing existing clique tree-based merging strategies, we described a novel clique graph merging strategy. The method is clique tree agnostic, considers a wider range of potential merges and is customisable to the solver algorithm and hardware used. Benchmark tests show that our approach is able to reduce the projection time and the solve time of our first-order solver significantly compared to existing clique merging methods, achieving savings of up to 60%. We further show how multithreading and chordal decomposition can be utilized to solve very large SDPs arising from real applications within minutes where other solvers run out of memory or need hours to find a solution.

An extension to our method would include information about the number of available CPU threads in the edge weighting function. This would allow us to optimise the strategy for the parallel execution of the block-specific projection steps.

5

Acceleration methods for fixed-point operators

Contents

5.1 Introduction

First-order optimisation methods such as Bregman methods, proximal gradient method and ADMM / Douglas-Rachford splitting, have been a competitive alternative to interior-point methods, especially for large convex optimisation problems [58,

122, 148]. They are therefore a good option for many (recent) applications of convex optimisation that lead to large optimisation problems. These include image processing [66, 121], SDPs in neural network safety verification [48], robust controller synthesis [124] or large QPs, SOCPs, or SDPs in multi-period portfolio optimisation [29]. However, two drawbacks of first-order methods are their slow convergence to high accuracy solutions and their dependence on algorithm parameters and problem scaling. While modest accuracy solutions are sufficient in many real world applications there are still cases where higher accuracies are desirable.

To improve convergence it helps to look at the wider class of *fixed-point iterations* applied in the context of *monotone non-expansive operators*; see Ryu and Boyd [144] for an overview. Indeed, many FOMs can be represented as a variant of a fixed-point iteration applied to a suitable operator. For example Eckstein [44] showed that the Douglas-Rachford splitting method is a special case of the *proximal point algorithm* applied to an operator splitting. The proximal point algorithm itself is a fixed-point iteration of the resolvent operator of a subdifferential mapping. The same is true for ADMM, which can be derived from the application of Douglas-Rachford splitting to the dual problem [54].

Recent research has focused on speeding up the convergence of these fixed-point iterations and with it the underlying FOMs. One approach is to interpret the search for fixed points of an operator $F(v)\colon \mathbb{R}^n \to \mathbb{R}^n$ as finding the zeros of the corresponding residual operator $r(v) = (\mathrm{Id} - F)(v)$ and then applying Newton-type methods that have fast asymptotic convergence rates in practice:

$$v_{k+1} \coloneqq v_k + d_k, \text{ with direction } d_k \text{ from } J_k d_k = r(v_k),$$

where J_k is the (possibly approximate) Jacobian of r at v_k. Each iteration of the quasi-Newton method then requires at least one evaluation of the operator F, i.e. one iteration of the underlying FOM.

However, using a quasi-Newton method, e.g. *Broyden's method*, introduces two new difficulties that were already discussed by Powell [131] in 1970.

The first shortcoming is that (quasi-) Newton methods can diverge, if the initial estimate v_0 is too far from the solution. There are two main approaches to avoid divergence and globalize Newton's method. The first is to use a line search method, as described by Haselgrove [73], that replaces the update rule for v with:

$$v_{k+1} = v_k + \lambda_k d_k,$$

where the step size λ_k is determined in a search process to ensure the new estimate is better by some merit function, e.g. the residual norm, than the current one, e.g.

$$\|r(v_k + \lambda_k d_k)\|_2^2 < \|r(v_k)\|_2^2 .$$

The second approach is to interleave the pure Newton step with pure gradient steps whenever the candidate direction seems to be disadvantageous. This approach was proposed by Levenberg/Marquardt [93, 106] in the context of solving non-linear least-squares problems and subsequently used in the *hybrid algorithm* by Powell [131].

The second shortcoming of quasi-Newton methods is that when the Jacobian of the function, here the residual operator, is numerically approximated, it can become singular or near-singular. This prohibits the computation of further meaningful descent directions. A way to combat this is by using regularisation techniques. Either by regularising the iterates before storing them, like in Powell regularisation [131], or in certain cases by regularising a least-squares subproblem [50].

Many recent publications on acceleration methods for non-smooth operators can be seen as offering solutions designed to mitigate the two challenges discussed by Powell.

5.1.1 Related Work

There have been a number of recent publications on the topic of acceleration methods for FOMs, mostly using (multi-)secant methods, such as *Broyden's method* or *Anderson Acceleration* (AA). The recent survey paper by d'Aspremont et al. [34]

discusses a range of acceleration techniques for convex optimisation algorithms. Well-known acceleration methods based on the gradient descent method are the *Heavy ball method* [127], *Nesterov's accelerated gradient method* [118], as well as *accelerated proximal gradient method* [13] . All of these methods require differentiability of at least one part of the objective of the underlying optimisation problem.

Giselsson et al. [63] propose a line search method for FOMs that can be written in terms of non-expansive operators. Their approach is to search along the last residual direction and take a larger step than the classic method would permit, allowing their algorithm to skip many iterations. The main computational bottleneck of their approach is the fact that the evaluation of each candidate point along the search ray triggers one evaluation of the underlying algorithm. Consequently, they suggest only using a line search if the operator can be split into two parts $F(v) = F_2F_1(v)$, where F_2 is cheap to evaluate and F_1 is affine. In that case it is possible to check many points on the ray cheaply.

Themelis and Patrinos [152] propose a Newton-type algorithm called *SuperMann* that globalizes fixed-point iterations of non-expansive operators and enjoys superlinear convergence under certain conditions. They use a restarted limited memory Broyden's method to approximate the Jacobian and Powell regularisation to keep their approximation non-singular. They further introduce a number of conditions that they check to allow unsupervised (or blind) updates, which prevents too many additional evaluations of the residual operator. Similar to Powell's hybrid algorithm, they fall back onto the original fixed-point iteration if the new accelerated point seems to diverge from the solution.

Ali et al. [3] apply a semismooth Newton method to the fixed point operator that is used in the `SCS` solver. The Newton direction is computed using Clarke's generalized Jacobian of the operator which involves finding the generalized Jacobian of the projection operator. Therefore, the method assumes a semismooth operator and relies on expensive line search steps. They show performance improvements over the original `SCS` algorithm for QPs and SOCPs, but not for SDPs.

A number of recent publications suggest a similar algorithm, but instead of using Newton's or Broyden's method they derive their update equation for the approximate Jacobian based on the Acceleration method by Anderson [7] from 1965 which does not require a line search. While the method was successfully applied in electronic structure computation, known as *Pulay mixing* [133], *direct inversion in the iterative subspace* (DIIS) [132], or *Anderson mixing*, it only recently gained attention in the optimisation community. Most notably by the papers of Walker and Ni [167] and Fang and Saad [47].

The *Anderson acceleration* algorithm updates the next iterate using an affine combination of past iterates, where the weights are determined by minimizing the weighted sum of past residuals. It turns out that this can be interpreted as a (type-II) Broyden update [47]. Anderson acceleration methods have been proposed to speed up the Douglas-Rachford method that is used in `SCS` [177]. The authors combine (type-I) AA steps with the execution of the original algorithm whenever the current residual decreases sufficiently. To ensure a non-singular Jacobian they use a Gram-Schmidt orthogonalization strategy. While the authors prove global convergence of their algorithm, their current implementation in `SCS` v2.0 lacks the safeguarding steps outlined in [177]. Aside from ADMM / Douglas-Rachford splitting, AA has also been used in combination with other FOMs. Mai and Johansson [103] applied it to the classical and the Bregman proximal gradient method, Higham [81] to the alternating projection method, and Henderson and Varadhan [77] to the expectation-minimization algorithm for maximum likelihood estimation.

5.1.2 Outline

In Section 5.2 we introduce the AA method and its relationship with Broyden's method. In Section 5.3 we present a safeguarded and restarted acceleration method to deal with the two typical shortcomings of quasi-Newton methods and provide details on the implementation. In Section 5.4 we demonstrate the benefits of our

safeguarded and accelerated version of ADMM applied to the problem format (3.14) on a number of benchmark sets. Section 5.5 concludes the chapter.

5.1.3 Contributions

With the work in this chapter we make the following contributions:

- We develop a safeguarded AA wrapper around the ADMM algorithm used in `COSMO`.
- We show how a restarted limited-memory acceleration scheme integrates well with the ρ-adaptation and infeasibility detection techniques used in our variant of ADMM.
- We provide numerical evidence on more than 500 test problems on how AA reduces the mean number of iterations for each problem set by a factor 1.7 to 8.5 and the mean total solve time by a factor of up to 6 for higher accuracy solutions. Meanwhile, we show that the additional time spent to calculate the Anderson directions can be kept between 3 % to 15 % for large QPs and SDPs. Thus, the results make a particularly strong case for using AA to solve SDPs and large QPs.

5.2 Background

This section describes different acceleration methods for fixed-point operators. First, we connect fixed points and fixed-point iterations to FOMs. Then we establish the connection between the residual operator and the application of (quasi-)Newton methods to accelerate convergence. We introduce classical and multisecant Broyden's method and show how AA can be viewed as a particular variant of Broyden's method.

5.2.1 Fixed-point iterations

We are interested in finding the fixed point of a mapping $F\colon \mathcal{D} \to \mathbb{R}^n$. From Definition 2.6.4, we have the set of fixed-points:

$$\mathbf{Fix}\, F \coloneqq \{v \in \mathbf{dom}\, F \mid v = Fv\}\,.$$

The following theorem provides conditions for the existence of a unique fixed point and a way to find it.

Theorem 8 (Banach-Picard fixed point theorem [12, Theorem 1.50])**.** *Let $(\mathcal{X}, d)$ be a complete metric space with metric d and let $F : \mathcal{X} \to \mathcal{X}$ be Lipschitz continuous with constant $L \in [0, 1[$. Given $v_0 \in \mathcal{X}$, set*

$$v_{k+1} = Fv_k.$$

Then there exists $v \in \mathcal{X}$ such that the following hold:

- *v is the unique fixed point of F.*
- *$(\forall v \in \mathbb{N})\, d(v_{k+1}, v) \leq Ld(v_k, v)$, i.e. the mapping is contractive.*
- *the sequence $(v_k)_{k\in\mathbb{N}}$ converges linearly to v.*

Thus, if finding an optimal solution using a FOM can be expressed as a fixed point problem with a contractive operator F, the Picard iteration $v_{k+1} = Fv_k$ will converge linearly to the fixed-point. However, many first-order methods translate merely into nonexpansive operators. A fixed-point iteration of a nonexpansive operator, see Definition 2.6.2, does not necessarily converge to a fixed point, e.g. consider the nonexpansive operator $F = -\mathrm{Id}$ and $v_0 \neq 0$. However, it was shown that the damped, averaged, or Krasnoselskii-Mann iteration [89, 104]

$$v_{k+1} = (1 - \alpha_k)v_k + \alpha_k F v_k,$$

with $\alpha_k \in (0, 1)$ converges to a unique fixed point for nonexpansive operators. A number of common FOMs can be written as Picard iterations of contractive operators or averaged fixed-point iterations of nonexpansive operators. The *proximal gradient method* solves the optimisation problem

$$\text{minimize} \quad f(v) + g(v), \tag{5.1}$$

with $v \in \mathbb{R}^n$ and convex, closed, proper functions $f \in \mathbb{R}^n \to \mathbb{R}$ and $g \in \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$. We further assume that f is differentiable with Lipschitz constant L. We

can derive an operator form of the method by rewriting the problem of finding the zeros of the subdifferential of the composite function as a fixed-point problem

$$\begin{aligned} 0 \in (\nabla f + \partial g)(v) &\Leftrightarrow 0 \in (\mathrm{Id} + \alpha\partial g)(v) - (\mathrm{Id} - \alpha\nabla f)(v) \\ &\Leftrightarrow (\mathrm{Id} + \alpha\partial g)(v) \ni (\mathrm{Id} - \alpha\nabla f)(v) \\ v &= (\mathrm{Id} + \alpha\partial g)^{-1}(\mathrm{Id} - \alpha\nabla f)(v) \\ v &= R_{\partial g}(\mathrm{Id} - \alpha\nabla f)(v) = \mathbf{prox}_g(v - \alpha\nabla f(v)). \end{aligned}$$

The gradient step inside the proximal operator is contractive for $\alpha \in (0, 2/L)$ and the proximal operator is the resolvent operator applied to ∂g, which is nonexpansive. Thus, the composition is contractive and the Banach-Picard iteration can be applied to find a fixed point.

Similarly, in Section 3.2.3 we described the Douglas-Rachford splitting for an optimisation problem of the form (5.1) but with f not necessarily differentiable and possibly valued on the extended real number line. The fixed-point form

$$0 \in (\partial f + \partial g)(v) \Leftrightarrow z = (\frac{1}{2}\mathrm{Id} + \frac{1}{2}C_{\partial f}C_{\partial g})(z),\ v = R_{\partial g}(z)$$

is an averaged iteration of two nonexpansive Cayley operators applied to the subdifferentials ∂f and ∂g. An overview of common FOMs in operator problem form and the corresponding fixed-point iteration are shown in Table 5.1. A more extensive overview of monotone operators and associated algorithms can be found in Ryu and Boyd [144]. This shows that the Krasnoselskii-Mann iteration can be viewed as a meta-algorithm that generalizes the methods in Table 5.1. Common ideas to improve the linear convergence were discussed in Chapter 3, i.e. problem scaling and step size adaptation.

When the fixed-point iteration is a minimization problem of a smooth convex function f, then pure and quasi-Newton methods with line search, and accelerated gradient descent methods, or momentum methods can be applied to improve the convergence [34]. However, the convergence properties of these methods do not generalize easily to the nonsmooth case. A different approach is to view the

Table 5.1: Common first-order methods as fixed point iterations of monotone operators.

Method	Operator form*	Fixed point iteration
Gradient method†	$0 \in \nabla f(v)$	$v_{k+1} = (\mathrm{Id} - \alpha \nabla f)(v_k)$
↪ Dual ascent		
Proximal point method	$0 \in A(v)$	$v_{k+1} = R_A(v_k) = (Id + \alpha A)^{-1}$
↪ Method of multipliers		
↪ Iterative refinement		
Forward-backward splitting	$0 \in (A + B)(v)$	$v_{k+1} = R_B(v_k - \alpha A v_k)$
↪ Proximal gradient method**	$0 \in (\partial f + \partial g)(v)$	
↪ Alternating projection method		
↪ Projected gradient method		
↪ Iterative shrinkage-thresholding algorithm (ISTA)		
Peaceman-Rachford splitting	$0 \in (A + B)(x)$	$z_{k+1} = C_A C_B(z_k)$
		$v_{k+1} = R_B(z_{k+1})$
Douglas-Rachford splitting	$0 \in (A + B)(x)$	$z_{k+1} = 1/2\mathrm{Id} + 1/2 C_A C_B(z_k)$
		$v_{k+1} = R_B(z_{k+1})$
↪ ADMM		
↪ Consensus		

* A, B maximal monotone operators, R_A, C_A Resolvent and Cayley operator of A, see Section 2.6
† f strongly convex and strongly smooth with parameter L, $\alpha \in (0, 2/L)$
** f, g convex, closed, and proper

problem of finding the fixed points of F as the problem of finding the roots of the residual operator $r(v) = (\mathrm{Id} - F)(v)$. A quasi-Newton method can then be used to approximate the Jacobian of r at v_k and local superlinear convergence can be achieved near the solution. In the spirit of Fang and Saad [47], we present the Anderson Acceleration method as one candidate method among a family of multisecant methods that can be used to economically approximate the Jacobian. We further discuss approaches to stabilize the method when used in a global optimisation setting.

5.2.2 Quasi-Newton methods

Newton's method can be used to find the roots of a differentiable function $f\colon \mathbb{R}^n \to \mathbb{R}^n$ using the iteration

$$v_{k+1} = v_k - J_f(v_k)^{-1} f(v_k), \tag{5.2}$$

where $J_f(v_k)$ is the Jacobian matrix at v_k. However, in the general nonsmooth case the Jacobian of the residual operator is unavailable or expensive to compute. Moreover, for high-dimensional problems solving (5.2) can quickly become prohibitive as it requires an LU factorisation with complexity $\mathcal{O}(\frac{2}{3}n^3)$. Quasi-Newton methods mitigate these issues by approximating the Jacobian. This is done using a series of computationally cheap low-rank updates to keep an updated approximation $B_k \approx J_f(v_k)$ of the Jacobian or its inverse $H_k \approx J_f(v_k)^{-1}$:

$$\begin{aligned} v_{k+1} &= v_k - B_k^{-1} f(v_k), \\ B_{k+1} &= u(B_k, v_{k+1}, v_k, f(v_{k+1}), f(v_k)), \end{aligned} \tag{5.3}$$

where $u\colon \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ is the update rule for the approximation.

5.2.3 Broyden's methods

Broyden's method defines an update rule for the Jacobian approximation update B_{k+1} based on two conditions. First, the secant condition of finite differences

$$B_{k+1}\Delta v_k = \Delta f_k, \tag{5.4}$$

where $\Delta v_k \coloneqq v_{k+1} - v_k$ and $\Delta f_k \coloneqq f(v_{k+1}) - f(v_k)$. Second, a condition requiring that no new information is inserted along any directions orthogonal to Δv_k:

$$B_{k+1} w = B_k, \quad \text{for all } w^\top \Delta v_k = 0. \tag{5.5}$$

The second condition is equivalent to requiring that the update does not change too much in one step. B_{k+1} is chosen as the matrix satisfying (5.4) and minimizing $\|B_{k+1} - B_k\|_F^2$ [41]. This yields the following analytical solution:

$$B_{k+1} = B_k + \frac{(\Delta f_k - B_k \Delta v_k)\, \Delta v_k^\top}{\Delta v_k^\top \Delta v_k}. \tag{5.6}$$

Adding the update rule to (5.3) yields Broyden's first method [47]

$$\begin{aligned} v_{k+1} &= v_k - B_k^{-1} f(v_k), \\ B_{k+1} &= B_k + \frac{(\Delta f_k - B_k \Delta v_k)\, \Delta v_k^\top}{\Delta v_k^\top \Delta v_k}. \end{aligned} \tag{5.7}$$

The update in (5.7) still requires solution of a linear system, but B_k is likely a low-rank modification of a diagonal matrix, and therefore has a complexity of $\mathcal{O}(n^2)$. By using the Sherman-Morrison formula, (5.6) can be expressed in terms of the inverse H_k:

$$H_{k+1} = H_k + \frac{(\Delta v_k - H_k \Delta f_k)\, \Delta v_k^\top H_k}{\Delta v_k^\top H_k \Delta f_k}.$$

To avoid these transformations, Broyden devised a second method that directly approximates and updates the inverse. Minimizing instead a least change objective of the inverses $\|H_{k+1} - H_k\|_F^2$ under the secant condition

$$H_{k+1} \Delta f_k = \Delta v_k$$

yields the update rule

$$\begin{aligned} v_{k+1} &= v_k - H_k f(v_k), \\ H_{k+1} &= H_k + \frac{(\Delta v_k - H_k \Delta f_k)\, \Delta f_k^\top}{\Delta f_k^\top \Delta f_k}. \end{aligned} \tag{5.8}$$

The update rule for both the first and second method are based on the secant condition at the current step. Vanderbilt and Louie [163] argue that this approach loses information from previous iterations and suggest that the secant condition for B_{k+1} should also hold for older iterates. Eyert [46] followed this idea and developed a generalized Broyden's method which allows higher-rank updates. We assume that the update should preserve m previous secant conditions

$$B_k \mathcal{V}_k = \mathcal{F}_k \tag{5.9}$$

where the past differences are stacked as column vectors to form the matrices

$$\mathcal{F}_k \coloneqq \begin{bmatrix} \Delta f_{k-m} & \cdots & \Delta f_{k-1} \end{bmatrix}, \quad \mathcal{V}_k \coloneqq \begin{bmatrix} \Delta v_{k-m} & \cdots & \Delta v_{k-1} \end{bmatrix}. \tag{5.10}$$

We formulate the same no-change condition as (5.5) but orthogonal to all m previous differences:

$$(B_k - B_{k-m})w = 0 \quad \text{for all } w \in \mathcal{V}_k^{\perp}. \tag{5.11}$$

Much like the classical Broyden's method, the matrix B_k satisfying both (5.9) and (5.11) can be found by a rank-m update from B_{k-m} [47]:

$$B_k = B_{k-m} + (\mathcal{F}_k - B_{k-m}\mathcal{V}_k)\left(\mathcal{V}_k^{\top}\mathcal{V}_k\right)^{-1}\mathcal{V}_k^{\top}.$$

This is also the solution that minimizes $\|B_k - B_{k-m}\|_F$ subject to (5.9). To avoid inversion we can use the Woodbury formula to update the inverse:

$$H_k = H_{k-m} + (\mathcal{V}_k - H_{k-m}\mathcal{F}_k)\left(\mathcal{V}_k^{\top}H_{k-m}\mathcal{F}_k\right)^{-1}\mathcal{V}_k^{\top}H_{k-m}. \tag{5.12}$$

Similar to the classical Broyden method, the counterpart for the generalized type-I method is obtained by directly minimizing $\|H_k - H_{k-m}\|_F$ under the condition $H_k\mathcal{F}_k = \mathcal{V}_k$ and has the update rule

$$H_k = H_{k-m} + (\mathcal{V}_k - H_{k-m}\mathcal{F}_k)\left(\mathcal{F}_k^{\top}\mathcal{F}_k\right)^{-1}\mathcal{F}_k^{\top}. \tag{5.13}$$

We can write the full update rule for the generalized type-II Broyden's method by substituting (5.13) into (5.8):

$$\begin{aligned} v_{k+1} &= v_k - H_k f(v_k) \\ &= v_k - H_{k-m}f(v_k) - (\mathcal{V}_k - H_{k-m}\mathcal{F}_k)\left(\mathcal{F}_k^{\top}\mathcal{F}_k\right)^{-1}\mathcal{F}_k^{\top}f(v_k) \\ &= v_k - H_{k-m}f(v_k) - (\mathcal{V}_k - H_{k-m}\mathcal{F}_k)\,\eta_k. \end{aligned} \tag{5.14}$$

Notice that unlike the type-I method, the type-II counterpart involves solving a least-squares problem for η_k, i.e. $\left(\mathcal{F}_k^{\top}\mathcal{F}_k\right)\eta_k = \mathcal{F}_k^{\top}f(v_k)$. This can be done efficiently using an updated QR decomposition. In the next section we show how classical Anderson Acceleration can be interpreted as a generalized type-II Broyden's method with a specific estimate for H_0. Based on this relationship it is also possible to derive a type-I Anderson method.

5.2.4 Anderson acceleration

AA is an extrapolation method proposed by Anderson [7] in 1965, originally to improve the convergence of iterative procedures to solve nonlinear integral equations. Since then it has been used in many different fields, such as material sciences computational quantum chemistry and physics [132]. Closely related methods to AA are the Eddy-Mesina algorithm, Minimal Polynomial Extrapolation, or Reduced Rank Extrapolation [34]. Despite the success in specific domains, such as material science, physics [46], and computational chemistry [133] it remained largely unknown in the optimisation community. In 2009 Fang and Saad [47] showed that AA can be viewed as part of the family of generalized Broyden's methods.

Before we define AA, we consider again the fixed-point iteration problem:

$$v_{k+1} = F(v_k)$$

with possibly nonsmooth, nonexpansive operator $F\colon \mathbb{R}^n \to \mathbb{R}^n$. Furthermore, we denote the residual operator as $r_k \coloneqq r(v_k) = v_k - F(v_k)$ and treat the fixed-point problem as a root-finding problem of the residual operator. The main idea of AA is to choose the accelerated candidate point v_{k+1}^{acc} as an affine combination of $m_k + 1$ previous iterates

$$v_{k+1} = \sum_{i=0}^{m_k} \alpha_k^i F(v_{k-m_k+i}).$$

The weights $\alpha_k = \left[\alpha_k^0, \ldots, \alpha_k^{m_k}\right]$ are determined by minimizing the norm of the affine combination of past residuals. This yields the constrained minimization problem

$$\begin{array}{ll} \text{minimize} & \|R_k \alpha_k\|_2^2 \\ \text{subject to} & \mathbf{1}_{m_k+1}^\top \alpha_k = 1, \end{array} \tag{5.15}$$

where $R_k = \begin{bmatrix} r_{k-m_k} & \cdots & r_k \end{bmatrix}$ is the matrix of past residual vectors. The classic AA iteration is show in Algorithm 12. Notice that each iteration of the acceleration scheme requires one evaluation of the operator to obtain the next residual vector r_{k+1}.

There are many variations to this algorithm. The full-memory version takes the whole history into consideration. Walker and Ni [167] showed that the full-memory

Algorithm 12: Anderson acceleration applied to a fixed-point iteration.

Input : Nonexpansive operator $F\colon \mathbb{R}^n \to \mathbb{R}^n$, initial estimate $v_0 \in \mathbb{R}^n$.

for $k = 0,1,\ldots$ **do**
- Set $m_k = \min\{m, k\}$;
- Update $R_k \leftarrow [r_{k-m_k} \cdots r_k]$;
- Solve for $m_k + 1$ weights α_k: $\min_{\mathbf{1}^\top \alpha_k} \|R_k \alpha_k\|_2^2$;
- Update $v_{k+1} = \sum_{i=0}^{m_k} \alpha_k^i F(v_{k-m_k+i})$;

version for an affine operator $F(v_k) = Av_k + b$ is identical to the generalized minimal residual method (GMRES). In practice, and especially for large dimensions n, a limited-memory variant with a fixed memory length is used. Limited-memory approaches can be distinguished by the way past iterates are retained. Once the memory buffer is full, rolling memory approaches overwrite the oldest iterate with the current one. Restarted memory approaches instead clear the whole history when the memory size is reached and then begin to fill it up with new iterates.

The connection between AA and Broyden's method was first shown by Eyert [46]. The relationship becomes clear if one uses a change of variables to avoid the explicit affine combination constraint on α^i. Define a vector $\eta = (\eta^0, \ldots \eta^{m_k - 1}) \in \mathbb{R}^{m_k}$ such that

$$\alpha^0 = \eta^0,\ \alpha^i = \eta^i - \eta^{i-1}, \ldots,\ \alpha^{m_k} = 1 - \eta^{m_k - 1}$$

to implicitly encode the constraint. The norm minimization problem in (5.15) can be written in terms of η

$$\begin{aligned} \left\| \alpha^0 r_{k-m_k} + \alpha^1 r_{k-m_k-1} + \cdots + \alpha^{m_k} r_k \right\|_2^2 = \\ \left\| r_k - \left(\eta^0 \Delta r_{k-m_k} + \eta^1 \Delta r_{k-m_k-1} + \cdots + \eta^{m_k-1} \Delta r_{k-1} \right) \right\|_2^2 = \|r_k - \mathcal{R}_k \eta\|_2^2, \end{aligned} \tag{5.16}$$

where $\Delta r_k = r(v_{k+1}) - r(v_k)$ and $\mathcal{R}_k = \begin{bmatrix} \Delta r_{k-m_k} & \ldots & \Delta r_{k-1} \end{bmatrix}$. Next, the AA update rule is expressed using η and the sum is written in matrix-vector form

$$\begin{aligned} v_{k+1} &= \sum_{i=0}^{m_k} \alpha_k^i F(v_{k-m_k+i}) = F(v_k) - \sum_{i=0}^{m_k-1} \eta_k^i \left(F(v_{k-m_k+i+1}) - F(v_{k-m_k+i}) \right) \\ &= v_k - r_k - (\mathcal{V}_k - \mathcal{R}_k)\eta_k. \end{aligned}$$

where $\mathcal{V}_k$ is defined as before in (5.10). Assuming that $\mathcal{R}_k$ has full rank, we substitute η_k for the solution of the least-squares problem (5.16)

$$v_{k+1} = v_k - r_k - (\mathcal{V}_k - \mathcal{R}_k)(\mathcal{R}_k^\top \mathcal{R}_k)^{-1}\mathcal{R}_k^\top r_k \tag{5.17}$$

$$= v_k - \left(I + (\mathcal{V}_k - \mathcal{R}_k)(\mathcal{R}_k^\top \mathcal{R}_k)^{-1}\mathcal{R}_k^\top\right) r_k. \tag{5.18}$$

Comparing (5.18) and (5.14) shows that at each step classical AA performs a generalized rank-m Broyden type-II update of the identity matrix, i.e. it minimizes $\|H_k - I\|_F^2$. In other words, AA approximates the inverse Jacobian of the residual operator. We can write this in quasi-Newton form:

$$v_{k+1} = v_k - H_k r_k$$
$$\text{with } H_k = -I + (\mathcal{V}_k + \mathcal{R}_k)(\mathcal{R}_k^\top \mathcal{R}_k)^{-1}\mathcal{R}_k^\top.$$

Using this relationship one can derive a Broyden type-I equivalent update rule of AA, which minimizes $\|B_k - I\|_F^2$ [47], compare (5.12)

$$H_k = -I + (\mathcal{V}_k + \mathcal{R}_k)(\mathcal{V}_k^\top \mathcal{R}_k)^{-1}\mathcal{V}_k^\top. \tag{5.19}$$

Note however, that this form does not allow the use of QR decomposition to evaluate the inverse term. Anderson Acceleration type-I has for example been used to speed up Douglas Rachford splitting in [177].

Convergence results

When discussing existing convergence results for AA one has to distinguish between type-I and type-II, as well as full-memory and limited-memory variants. Recently published convergence results on AA rely on additional assumptions on the operator F. Toth and Kelley [155] prove convergence of AA type-II with memory storage length m, notated Anderson(m). The method is locally R-linearly convergent if the fixed-point map is a contraction and the coefficients η remain bounded. Removing the condition on boundedness of η, they show Q-linear convergence for Anderson(m) only for linear operators.

Under the assumption of continuous differentiability of F around the solution, Gay and Schnabel [60] show local Q-superlinear convergence of full-memory AA type-I and Rohwedder and Schneider [139] show Q-linear convergence of Anderson(m). By noting the equivalence to GMRES for affine operators, Potra and Engler [129] show convergence of full-memory AA type-II in a finite number of steps. However, for our design of Anderson(m) around a general conic ADMM algorithm we do not have a contractive operator and cannot make further differentiability assumptions on F. Unfortunately, Mai and Johansson [103] show that a naive application of Anderson(m) cannot guarantee global convergence results. Therefore, the acceleration method has to be embedded into a stabilisation scheme to ensure global convergence, which is discussed in Section 5.3.

5.3 Safeguarded acceleration

In this section we discuss our particular variant of AA and the necessary modifications to use it with the particular operator splitting discussed in Chapter 3. In particular, we discuss the steps necessary to handle the two main issues of quasi-Newton methods:

- lack of global convergence guarantees
- tendency of the Jacobian approximation to become singular or near-singular.

In the following we discuss our choice of the Anderson type and introduce a safeguarding condition to ensure nonexpansiveness of the accelerated iteration. Moreover, we discuss measures to prevent singularity of the Jacobian approximation, and how to schedule infeasibility checks and step size adaptation.

Anderson acceleration variant

We implemented both the type-I and the type-II variant of AA in generalized Broyden form; compare (5.19) and (5.17). Neither variant was reliably superior in terms of convergence over the problem sets considered. Thus, by default we use

the type-II variant as it has the advantage that the equations to determine the coefficients

$$\eta_k = (\mathcal{R}_k^\top \mathcal{R}_k)^{-1} \mathcal{R}_k^\top r_k \tag{5.20}$$

can be written as a least-squares problem and therefore solved efficiently using a continuously updated QR decomposition. We assume an overdetermined least-squares matrix $\mathcal{R}_k \in \mathbb{R}^{n \times m_k}$, where $n \geq m_k$. The QR decomposition and solve steps are given by

$$\mathcal{R}_k = Q_k R_k \quad \text{and} \quad R_k \eta_k = Q_k^\top r_k, \tag{5.21}$$

with orthonormal $Q_k \in \mathbb{R}^{n \times m_k}$ and upper triangular $R_k \in \mathbb{R}^{m_k \times m_k}$. The shape of R_k allows a fast back-substitution step to solve for η_k in (5.21). We use the modified Gram-Schmidt process to compute the factorisation, which requires $2nm_k^2$ flops [67].

At each step of AA we only need to incorporate the latest Δr_k into $\mathcal{R}_k$. We are using a restarted memory approach, where starting from an empty memory we accumulate up to $m_{\max}$ column vectors in $\mathcal{R}_k$. In Section 5.4 we discuss experimental results for different choices of $m_{\max}$. Once that amount is reached the whole memory is deleted and the method is restarted. Consequently, at each iteration the matrix $\mathcal{R}_k$ changes only by one column or gets deleted completely. This property can be used to efficiently maintain updated QR-factors. We allocate the memory for an $n \times m_{\max}$ matrix Q, and an $m_{\max} \times m_{\max}$ matrix R which are to be reused throughout the acceleration scheme. Aside from the iteration counter k of the outer fixed-point algorithm, we also maintain a counter j that points at the next empty column of $\mathcal{R}$, Q and R which is to be filled by the acceleration scheme at that iteration. Moreover, when using the matrices at a certain iteration we only consider the block of the matrix filled with valid data, i.e. columns 1 to j for Q and rows and columns 1 to j for R. To maintain Q and R we simply perform one Gram-Schmidt step to orthonormalize the latest Δr, as shown in Algorithm 13. One step of AA then costs at most an additional $\frac{3}{2}m_k^2 + 12m_k n + 3n$ flops [82]. Since for large conic problems we usually have $m_k \ll n$ and an ADMM operator generally has a complexity between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, the acceleration has a comparably low

Algorithm 13: Modified Gram-Schmidt step to update Q and R

Input : Memory size $m_{\max}$, column pointer $1 \leq j \leq m_{\max}$, Δr, Q, R
Output : Updated matrices Q and R

if $j = 1$ **then**
  $R \leftarrow \|\Delta r\|_2$;
  $Q \leftarrow \Delta r / \|\Delta r\|_2$;
else
  $\Delta r \leftarrow \Delta r$;
  for $i = 1$ *to* $j - 1$ **do**
    $R_{ij} \leftarrow \langle Q_i, \Delta r \rangle$;
    $\Delta r \leftarrow \Delta r - \langle Q_i, \Delta r \rangle Q_i$;
  $R_{jj} \leftarrow \|\Delta r\|_2$;
  $Q_j \leftarrow \Delta r / \|\Delta r\|_2$;

extra cost. We use Q_i to denote the i-th column vector of Q. Note that Δr gets overwritten in the process, but is not needed afterwards. If at iteration k of the outer loop the acceleration method considers the past j previous Δr, we solve the following system for η_k

$$R_{(i,\dots,j),(i,\dots,j)}\eta_k = Q^{\top}_{(:),(i,\dots,j)}r_k,$$

and on the next iteration η_{k+1}

$$R_{(i,\dots,j+1),(i,\dots,j+1)}\eta_{k+1} = Q^{\top}_{(:),(i,\dots,j+1)}r_{k+1},$$

where we found the $j+1$-th columns of R and Q using Algorithm 13. In the case of a rolling memory scheme, a slightly more complex implementation to maintain Q and R is required. The reason is that one has to correct the orthonormal basis vectors for the fact that the oldest iterate is deleted before Δr can be processed; see Walker [166] for an implementation using Givens rotations.

Globalisation strategy

As discussed in Section 5.2.4, AA applied to an operator based on a non-smooth optimisation problem does not guarantee global convergence. The two sources of instability are accelerated candidate points that diverge from the solution compared to the previous iterate and badly conditioned Jacobian approximations.

We mitigate the first issue by performing a safeguarding step at each iteration. Assume that at iteration k AA produced an accelerated candidate point v_k^{acc} using (5.18). We check the quality of that point by comparing the norm of the residual operator with the last available residual norm of an accepted step

$$\|r(v_k^{\text{acc}})\|_2 = \|v_k^{\text{acc}} - F(v_k^{\text{acc}})\|_2 \leq \tau \|r(v_k)\|_2 \tag{5.22}$$

with expansiveness tolerance $\tau \in (0, 1)$. If the test succeeds the candidate point is accepted as the next iterate $v_{k+1} = v_k^{\text{acc}}$. Otherwise, a normal fixed point step is performed such that $v_{k+1} = F(v_k)$. Note that checking condition (5.22) is computationally expensive since it involves evaluating $F(v_k^{\text{acc}})$ and $F(v_k)$, i.e. two iterations of the operator for each iteration of the algorithm. This makes the condition unusable in practice as even a 50 % decrease in the number of iterations would not result in any reduction in solve time. To avoid checking condition (5.22) the authors in [50, 177] replace the term $\tau \|r(v_k)\|_2$ by a summable and exponentially decaying series based on $\|r(v_0)\|_2$. The authors in [152] add a linearly decreasing error term q_k to $\tau \|r(v_k)\|_2$ and only check the condition periodically or if past progress stalled.

We use a different approach and check at every step the relaxed safeguarding condition

$$\|v_k^{\text{acc}} - F(v_k^{\text{acc}})\|_2 \leq \tau \|r(v_{k-1})\|_2 \tag{5.23}$$

using the previous residual norm $\|r(v_{k-1})\|_2$ and $\tau \in (0, 2)$. While $\tau > 1$ does not guarantee convergence we observed improved convergence in practice. Note that every check of (5.23) only requires evaluation of $F(v_k^{acc})$. We see in Section 5.4 that in practice most candidate points pass the safeguarding check, in which case $F(v_k^{\text{acc}})$ can be used for the next evaluation of AA and is essentially free. Consequently, this safeguarding step only results in extra operator evaluations if candidate points are rejected. The idea of using not only the last iterate $r(v_k)$ in the safeguarding check is also common in non-monotone line search techniques for Newton's method. It is based on the observation that in practice enforcing monotonicity of the objective function can slow the rate of convergence in the intermediate stages of Newton's

method. The authors in [69] replace the standard line search condition based on the function value of the last iterate by a modified condition based on the maximum function value of a set of m previous iterates.

Singularity of the Jacobian approximation

Another source of instability of AA is the tendency of the iterates $r(v_k)$ to become closely aligned. This happens especially in regions when the algorithm does not make much progress, e.g. when the algorithm repeatedly projects on the same active constraint set. One consequence is that the column vectors that make up the history of residual differences, $\mathcal{R}_k = \begin{bmatrix}\Delta r_{k-m_k} & \dots & \Delta r_{k-1}\end{bmatrix}$ lead to bad conditioning of $\mathcal{R}_k$ and its factorisation.

The two main ideas to mitigate the effects of bad conditioning of $\mathcal{R}_k$ are regularisation and safeguarding. Instead of solving the least-squares problem in (5.16), one can apply a Tikonov regularisation step to solve the regularised problem:

$$\text{minimize } \|\mathcal{R}_k\eta_k - r_k\|_2^2 + \lambda \|\eta\|_2^2, \tag{5.24}$$

with regularisation parameter λ, which biases the solution η_k to have smaller norm, which can be solved as:

$$\eta_k = (\mathcal{R}_k^\top \mathcal{R}_k + \lambda I_{m_k})^{-1} \mathcal{R}_k^\top r_k \tag{5.25}$$

using normal equations. Another option is to simply monitor the condition number of R_k, e.g. using incremental condition estimation [20], or the norm of the solution η_k. If $\text{cond}(R_k) > \text{tol}_{\text{cond}}$ or $\|\eta_k\|_2 > \eta_{\max}$, we do not attempt to compute a new accelerated candidate point and instead perform an ordinary fixed-point iteration. This approach can also be used to trigger a restart of the method, i.e. resetting the column pointer $j = 1$. In numerical tests there do not appear to be substantial differences between these techniques in terms of their safeguarding quality. We thus use the bound $\|\eta_k\|_2 \leq 10^4$ as a safeguarding mechanism, because the check can be performed cheaply, and because the regularised problem (5.24) has to be solved using the normal equations, which is slower than solving (5.20) in practice.

Scheduling of step size adaptation and infeasibility detection

The FOM represented by the operator that is wrapped in AA might rely on successive iterates produced by the operator rather than the acceleration scheme to perform step size adaptation or infeasibility detection [10]; see Section 3.6 and Section 3.7. Most FOMs adapt the step size parameter of the underlying operator as a measure to improve the rate of convergence in practice [58, 122, 148]. The step size adaptation discussed in Section 3.7 uses the ratio of scaled primal and dual residual norms to update the step size parameter ρ at fixed intervals, e.g. every 25 iterations. For most FOMs in operator-form, such as the Douglas-Rachford splitting, the step size parameter of the algorithm arises from the scaling parameter of the resolvent operator $R_F = (\mathrm{Id} + \rho F)^{-1}$, see Section 3.2.3. Thus, changing the step size also changes the operator representing the algorithm. As AA tries to approximate the Jacobian of the residual operator, this operator change will invalidate previously collected iterates.

If the FOM uses successive differences in iterates and separating hyperplane conditions to derive infeasibility certificates, then it needs access to successive differences in iterates that are unaltered by acceleration. Similar to the step size adaptation, the infeasibility conditions are checked at fixed intervals.

To integrate these two measures into the accelerated algorithm, we schedule step size adaptation and infeasibility detection to be performed when the acceleration scheme is restarted. This can be done since every restart involves at least two non-accelerated iterations. For example consider a configuration where AA is restarted every 10 iterations, and step size adaptation is performed every 25 iterations. After 25 iterations the algorithm would schedule a step size adaptation at the next available non-accelerated iteration. This would then happen after the second restart at iteration 30. Note this could also happen at any other iteration between 25 and 30 if the scheme is restarted for another reason. The same scheduling approach can be used for infeasibility detection. All considerations of this section are summarized in Algorithm 14.

Algorithm 14: Safeguarded AA with memory restarts and scheduling.

Input : v_0, f_0, fixed-point iteration $F_\rho\colon \mathbb{R}^n \to \mathbb{R}^n$ with $v_{k+1} = f_k = F_\rho(v_k)$, allocated memory for $\mathcal{V}_0$, $\mathcal{R}_0$, Q, and R, column pointer $j = 1$, parameters: $\mathrm{tol}_{cond}, \eta_{\max}, \tau, k_{\max}$

```
acc_success = false;
for k = 1, ..., k_max do
    Update history: V_j ← [V_{j-1}, Δv_{k-1}], R_j ← [R_{j-1}, Δr_{k-1}], j ← j + 1;
    if j > 2 then
        Update QR factors: Q_k, R_k using Algorithm 13;
        Compute η_k from R_k η_k = Q_k^⊤ r_k;
        if cond(R_k) > tol_cond or ‖η_k‖_2 > η_max then
            acc_success ← false;
        else
            Accelerate: v_k^acc = f_k − (V_j − R_j) η_k;
            acc_success ← true;
        if acc_success then
            Fixed point iteration: f_k^acc = F_ρ(v_k^acc) using (3.10);
            Compute residual: r_k^acc = v_k^acc − f_k^acc;
            if ‖r_k^acc‖_2 ≤ τ ‖r(v_{k-1})‖_2 then
                v_{k+1} ← v_k^acc, f_{k+1} ← f_k^acc, and r_{k+1} ← r_k^acc;
            else
                acc_success ← false;
    if not acc_success or j ≤ 2 then
        if ρ-adaptation scheduled then
            adapt step size ρ;
            j ← 1;
        Safeguarding: v_{k+1} ← f_k, f_{k+1} = F_ρ(v_{k+1}), r_{k+1} = v_{k+1} − f_{k+1};
    if infeasibility detection scheduled and j = 2 then
        perform infeasibility checks;
    if j > m_max then
        j ← 1 ;
```

The for-loop in Algorithm 14 can be terminated if the residual norm of the iterate $\|r(v_k)\|_2 \leq \epsilon$ is below a certain treshold ϵ or one can use termination conditions based on the primal and dual residuals of the underlying optimisation algorithm, see Section 3.6.

5.4 Benchmark results

To evaluate the impact of acceleration on the convergence of a first-order solver to higher solution accuracies, we implemented the safeguarding mechanism of Algorithm 14 in `COSMO` v0.8. Different variants of AA are implemented as the standalone package `COSMOAccelerators`, which is available at

https://github.com/oxfordcontrol/COSMOAccelerators.jl

and which is used as a dependency of the `COSMO` package. The following benchmarks compare `COSMO` without acceleration, with acceleration but without safeguarding condition (5.22), and acceleration with active safeguarding as specified in Section 5.3. The experiments were run using Julia v1.5 on computing nodes of the University of Oxford ARC-HTC cluster with 16 logical Intel Xeon E5-2560 cores and 64GB of DDR3 RAM. We configured `COSMO` with the same algorithm parameters as in Section 3.9, but increased the accuracy to $\epsilon = 10^{-6}$ for QPs and $\epsilon = 10^{-5}$ for SDPs, checking for convergence every 25 iterations.

The acceleration method was configured with maximum memory length $m_{\max} = 15$. We used the safeguarding parameter $\tau = 2$, and maximum norm $\eta_{\max} = 10^4$ for the Anderson parameters, as they worked reasonably well on many different problem types.

A number of different QP and SDP problem sets are tested to evaluate the impact of AA on the number of iterations to achieve the desired accuracy. Moreover, we are interested to see if any reduction in iterations of the pure ADMM algorithm translates into a reduction in total solve time, i.e. to verify that the extra time spent to compute Anderson candidate points is small compared to the overall time saved. The section closes with an analysis of the impact of different acceleration memory length parameters $m_{\max}$ on the convergence and overall solve time of the solver.

5.4.1 Quadratic programs

We compare three different problem sets. The Maros and Mészáros problem set [105], see Section 3.9.1, which includes convex QPs from a variety of application, model

predictive control problems based on the MPC Benchmarking Collection [49], and Markowitz portfolio optimisation problems of the form described in (3.37). Since these problem sets contain problems of varying sizes we report some statistics of the constraint matrices A in Table 5.2.

Table 5.2: Problem statistics of constraint matrix A for SDP and QP problem sets.

		nonzeros in A			num variables			num constraints		
	#	min	μ	max	min	μ	max	min	μ	max
Maros	135	4	1.6×10^4	2.4×10^5	2	3.7×10^3	4.0×10^4	3	6.4×10^3	8.1×10^4
MPC	308	2.9×10^1	7.1×10^2	8.4×10^3	1.1×10^1	1.3×10^2	8.1×10^2	1.1×10^1	1.9×10^2	1.4×10^3
Portfolio	10	1.4×10^5	4.9×10^6	1.3×10^7	5.1×10^3	2.8×10^4	5.1×10^4	5.1×10^3	2.8×10^4	5.1×10^4
SPCA	10	2.5×10^4	2.3×10^5	5.3×10^5	1.0×10^4	9.2×10^4	2.1×10^5	1.5×10^4	1.4×10^5	3.2×10^5
Block	20	7.8×10^5	2.2×10^6	3.6×10^6	5.1×10^2	9.7×10^2	1.4×10^3	1.0×10^2	1.0×10^2	1.0×10^2
Lovasz	15	1.3×10^3	1.8×10^4	1.1×10^5	1.1×10^3	1.7×10^4	1.0×10^5	1.3×10^3	1.8×10^4	1.0×10^5
Mittelmann	62	2.0×10^3	2.0×10^5	4.5×10^6	4.2×10^2	9.6×10^3	10.0×10^4	5.6×10^3	1.5×10^6	1.4×10^7

* with arithmetic mean μ

Markowitz portfolio optimisation We generate Markowitz portfolio optimisation problems of the form (3.37) with $k = 50, 100, \ldots, 600$ factors and $n = 100k$ assets. The covariance matrix $\Sigma = D + FF^\top$ is formed based on a diagonal matrix $D \in \mathbb{R}^{n \times n}$ with diagonal elements $D_{ii} \sim \mathcal{U}(0, \sqrt{k})$ and $F \in \mathbb{R}^{n \times k} \sim \mathcal{N}(0, 1)$ with 50 % nonzero elements. The expected return vector $\mu \in \mathbb{R}^n \sim \mathcal{N}(0, 1)$ is randomly generated and the risk-return parameter is chosen as $\gamma = 1.0$. Consequently, the nonzeros in the constraint matrix A of the transformed problem range between 1×10^5 to 2×10^7 which is considerably larger than for the other two QP problem sets.

5.4.2 Semidefinite programs

To measure the impact of acceleration on the solver convergence for SDPs, we generate problems based on sparse principal component analysis and compute the Lovász theta function for a number of publicly available graphs. We also test the effect of acceleration on the chordally decomposed block arrow sparse SDPs from Section 4.6.1. We further form a testset based on the smaller non-decomposable problems used in Hans Mittelmann's SDP benchmarks [111]. Problem statistics of the constraint matrix A for each problem set are shown in Table 5.2.

Sparse principal component analysis We generate SDPs based on the sparse principal component analysis problem which determines sparse directions $v_i \in \mathbb{R}^p$ that maximize the variance of the (normalized) data set $X \in \mathbb{R}^{N\times p}$. The first sparse eigenvector with constrained cardinality is found by solving

$$\begin{array}{ll} \text{maximize} & v^\top \Sigma v \\ \text{subject to} & \|v\|_2 = 1 \\ & \|v\|_0 \leq k, \end{array} \tag{5.26}$$

where $\Sigma = X^\top X$ is the empirical covariance matrix of X. Further directions can be found by solving the problem with an updated matrix $\Sigma_1 = \Sigma - \left(v^{*\top}\Sigma v^*\right) v^* v^{*\top}$. Due to the cardinality constraint, problem (5.26) is NP-hard [153]. Thus, we focus instead on solving a relaxed version, that uses the lifting procedure [35],

$$\begin{array}{ll} \text{maximize} & \operatorname{tr}(\Sigma V) \\ \text{subject to} & \operatorname{tr}(\Sigma V) = 1 \\ & \mathbf{1}_p^\top |V| \, \mathbf{1}_p \leq k \\ & V \succeq 0, \end{array}$$

where we chose $V = vv^\top, V \succeq 0$, then ignored the constraint that $\operatorname{rank}(V) = 1$ and approximated the cardinality constraint by a convex 1-norm constraint. Moreover, instead of choosing k a priori we follow the approach in [35] and use a robustness formulation by moving the 1-norm constraint into the objective

$$\begin{array}{ll} \text{maximize} & \operatorname{tr}(\Sigma V) - \theta \mathbf{1}_p^\top |V| \, \mathbf{1}_p \\ \text{subject to} & \operatorname{tr}(\Sigma V) = 1 \\ & V \succeq 0, \end{array} \tag{5.27}$$

with penalty parameter $\theta > 0$. We solve (5.27) for the first sparse eigenvector for randomly generated data matrices $X \in \mathbb{R}^{N\times p} \sim \mathcal{N}(0, 1)$ with $p = 5, 7, 9, \ldots, 23$ features, $N = 20p$ samples, 30 % nonzero elements, and penalty parameter $\theta = 2$.

Lovász theta function The Lovász theta function is an important concept in graph theory. Consider an undirected graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$ and edge set E. A *stable set* is a subset of $S \subseteq V$ such that the induced subgraph does not contain edges. The *stability number* $\alpha(G)$ of the graph is equal to the cardinality of the largest stable set. It is closely related to the *Shannon capacity* $\Theta(G)$ that models the amount of information that a noisy communication channel

can carry if certain signal values can be confused with each other. Unfortunately, the determination of $\alpha(G)$ is an NP-hard problem [74] and the complexity of computing $\Theta(G)$ remains unknown. The Lovász theta function $\vartheta(G)$ was introduced by Lovász [100], can be computed in polynomial time, and represents an upper bound on both the stability number and the Shannon capacity:

$$\alpha(G) \leq \Theta(G) \leq \vartheta(G).$$

The value of $\vartheta(G)$ is equal to the optimal value p^* of the following SDP:

$$\begin{array}{ll} \text{maximize} & \operatorname{tr}(\mathbf{1}_{n\times n} X) \\ \text{subject to} & \operatorname{tr}(X) = 1 \\ & X_{ij} = 0 \text{ for } (i,j) \in E, \\ & X \succeq 0, \end{array} \tag{5.28}$$

with matrix variable X and edge set E. For this benchmark we compute $\vartheta(G)$ by solving (5.28) for a set of undirected graphs from the SuiteSparse Matrix Collection [39]; see Section A.1. For these benchmarks we did not exploit the inherent sparsity of the problem, for example using techniques from Chapter 4.

Figure 5.1 gives an overview of the impact of acceleration with memory length $m_{\max} = 15$ on the mean number of iterations for every problem set. We note that

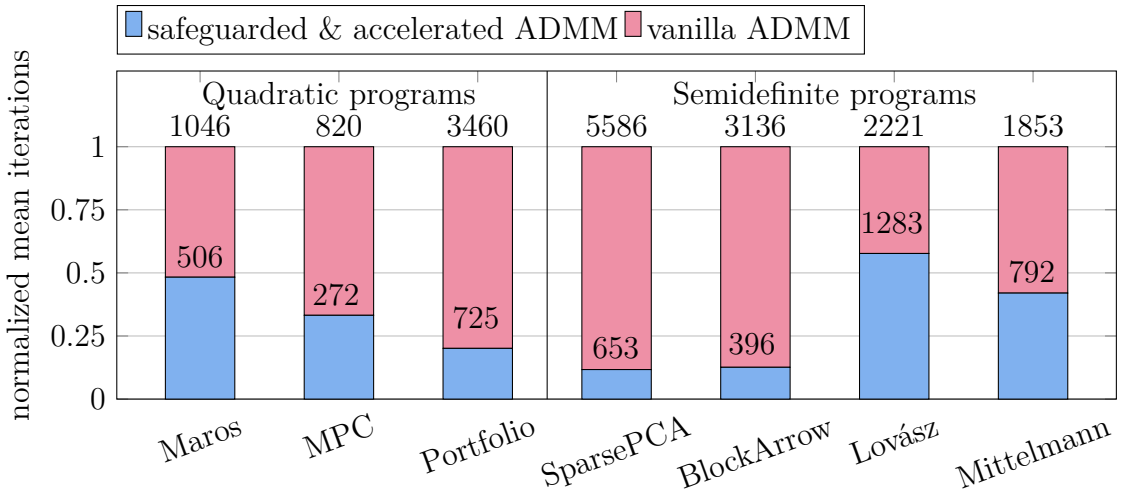


Figure 5.1: Mean iterations of non-accelerated and accelerated ADMM solver for the subsets of solved problems in different problem sets. The bars show the relative improvement of acceleration for a solution accuracy of 10^{-6} (QPs) and 10^{-5} (SDPs).

the mean is calculated based on the subset of problems in each set that was solved by both algorithms. Otherwise, the mean would be skewed in favour of the method that solved more problems. As we will see, this actually favours the non-accelerated method as it tends to solve fewer problems in each set. The bar chart shows that the acceleration helped to reduce the mean number of iterations for each problem set substantially. The smallest reduction of 42 % is observed for the Lovász theta function problems, while the biggest reduction (88 %) is achieved for sparse PCA problems.

To investigate if the reduction in iterations translates into lower solve times we calculate the mean and median of the total solve time, the number of problems solved, and the mean time used to calculate the Anderson directions as a fraction of the solve time. Since all solver configurations solved a different number of problems, we calculated the mean for the number of iterations and the solve time based on the subset of problems that was solved by every solver configuration. This skews the results slightly in favour of solver configurations that solved fewer problems. For the safeguarded method we also show the number of declined candidate points due to failing the safeguarding check (5.22), as this leads to one additional evaluation of the operator.

We further compute the normalized shifted geometric mean of the solve times as defined in (3.38), with a maximum allowable time of of 5 min for QPs and 60 min for SDPs. The results for each problem set are reported in Table 5.3.

The results show that the reduction in iterations leads to a reduction in mean solve time. For some problem sets, like Maros and MPC, the median of the solve time is fairly similar to the vanilla method. This is likely due to the presence of easier problems that are solved in only a few iterations by each method.

The extra work due to AA with a memory size of 15 varies from 2 % to 15 % for the portfolio problems and SDPs and thus has little impact on the overall solve time. For the QP problem sets that include small problems, the time spent in acceleration related function is of the order of 25 %. This means that for smaller

Table 5.3: Benchmark results for vanilla, non-safeguarded accelerated, and safeguarded accelerated ADMM for various QP and SDP problem sets.

	algorithm	solved	iter[1]	solve time[2]	% acc time[3]	gmean[4]
Maros	vanilla	75	1046.0	2.10 (0.02)		2.33
	accelerated	98	676.8	1.27 (0.02)	24.7	1.08
	safeguarded	100	505.8 (22.9)	1.16 (0.02)	25.0	1.00
MPC	vanilla	255	820.3	0.020 (0.0026)		2.74
	accelerated	230	296.9	0.012 (0.0025)	27.5	4.72
	safeguarded	285	272.0 (115.3)	0.015 (0.0029)	26.5	1.00
Portfolio	vanilla	10	3460.0	171.51 (134.16)		3.04
	accelerated	10	1042.5	69.85 (48.96)	4.8	1.31
	safeguarded	10	725.0 (30.7)	49.67 (32.29)	3.6	1.00
SPCA	vanilla	9	5586.1	132.81 (95.69)		6.72
	accelerated	10	1013.9	46.92 (15.60)	14.6	1.57
	safeguarded	10	652.8 (32.3)	23.67 (12.20)	13.5	1.00
Block	vanilla	20	3136.3	99.31 (42.11)		5.40
	accelerated	20	308.8	11.49 (8.47)	2.9	1.00
	safeguarded	20	396.3 (66.9)	18.85 (8.97)	2.3	1.27
Lovász	vanilla	12	2220.8	17.01 (7.21)		5.72
	accelerated	15	1360.4	5.79 (3.01)	8.7	1.09
	safeguarded	15	1283.3 (15.0)	4.92 (2.82)	8.5	1.00
Mittelmann	vanilla	22	1853.4	179.50 (34.81)		1.72
	accelerated	29	744.3	79.57 (31.84)	3.7	1.00
	safeguarded	31	792.0 (5.9)	88.68 (22.51)	3.4	1.01

[1] mean iteration and (extra safeguarding iterations);
[2] mean and median (based on subset of problems where all solver configurations solved the problem);
[3] geometric mean of fraction of total solve time spent in acceleration-related functions;
[4] normalized shifted geometric mean of solve time, see (3.38) (based on all problems in the problem set);

problems the number of iterations has to be reduced by at least that amount to make the acceleration worthwhile. On average this seems to be the case, but for some individual problems the acceleration might slow down the solver.

The results also show that the safeguarded acceleration leads to a higher number of problems solved. The impact of the safeguarding vs. the non-safeguarded accelerated solver in terms of preventing instability seems to be small for most problem sets. However, for the set of MPC problems, the safeguarded method solved 55 more problems than the non-safeguarded version. The geometric mean of the solve times takes into account the different number of problems solved. The impact of

safeguarded acceleration vs. the vanilla method ranges between a factor 1.72 for the Mittelmann SDPs and 6.72 for the sparse PCA problems.

Figure 5.2 shows the residual norms of the vanilla and the safeguarded method for the Mittelmann SDP problem `ros_500`. This particular problem is chosen as an example as the residual norms show a typical behaviour for problems where AA works successfully. Initially the primal and dual residual norms of both methods

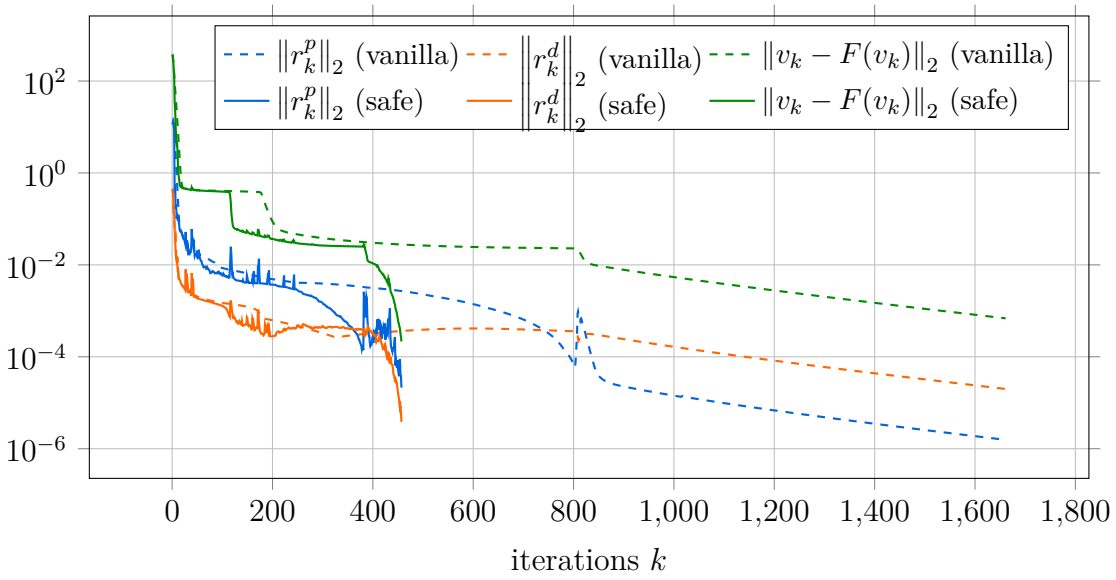


Figure 5.2: Norms of primal residual $\left\|r_p^k\right\|_2$, dual residual $\left\|r_d^k\right\|_2$, and fixed point residual $\|v_k - F(v_k)\|_2$ of the vanilla and the safeguarded accelerated method (Mittelmann: `ros_500`).

decrease in a similar fashion until an accuracy of 10^{-2} to 10^{-4}. However, the residuals of the accelerated method then drop sharply and avoid the vanilla method's period of slow convergence from iteration 400 onwards. The sharp decrease in the residual norm is likely due to the iterates reaching the region where the quasi-Newton steps approximate the Jacobian well and the method achieves superlinear convergence. The figure also indicates that the relative impact of AA will be fairly low when the solver method terminates already at a low accuracy solution of e.g. 10^{-3}. For the problems in our test sets where AA does not improve the convergence, we commonly see a badly conditioned matrix $\mathcal{R}_k$ and even method restarts do not improve the conditioning. Consequently, this leads to a lot of accelerated steps being discarded

by one of the safeguarding checks. As discussed earlier this bad conditioning can happen when successive iterates are closely aligned. When solving a QP this usually happens in regions of slow progress when the algorithm repeatedly projects onto the same incorrect set of constraints.

One of the most influential parameters of AA is the memory length $m_{\max}$, i.e. the maximum number of previous iterates that are used to find the next Anderson candidate vector. A higher number of past vectors would suggest a more accurate choice for the accelerated vector. However, more vectors, or even just two almost collinear vectors, can affect the numerical stability of the inner least-squares problem dramatically. Moreover, a higher memory length requires more time to compute the Anderson acceleration parameters η_k. To choose a sensible memory length we benchmarked our accelerated ADMM solver with memory length $m_{\max} = 5, 10, \ldots, 30$ on the discussed problem sets. The results are shown in Table 5.4.

As predicted, a higher memory length increases the time spent in acceleration related function and therefore has to be balanced against improvements in convergence. The results also show that a higher memory length does not automatically improve the convergence. Using the geometric mean column as a metric for overall performance of each solver configuration we can conclude that a memory length of $m_{\max} = 15$ is a reasonable choice across all problem sets.

5.5 Conclusions

In this chapter we propose a strategy to safeguard AA using periodic restarts, least-squares condition checking, and a safeguarding rule based on the residual operator norm. We show how these design choices integrate well with an ADMM solver that requires occasional pure operator steps for step-size adaptation and infeasibility checks. We provide an efficient implementation of Anderson acceleration with QR decomposition in the latest version of `COSMO` which combines the advantages of acceleration with already existing features, e.g. custom cones and linear solvers, as well as chordal decomposition and clique merging techniques. The effectiveness of

our approach in reducing both the mean number of iterations and solve time and increasing the number of solved problems is shown for a number of different QP and SDP problem sets.

Table 5.4: Impact of acceleration memory length on iterations and solve times.

	$m_{\max}$	solved	iter[1]	solve time[1]	% acc time[2]	gmean[3]
Maros	5	95	1204.0 (223.5)	1.17 (0.06)	19.7	1.21
	10	97	1029.9 (202.0)	1.26 (0.05)	23.3	1.13
	15	100	873.8 (205.5)	1.22 (0.05)	26.0	1.00
	20	98	976.2 (180.0)	1.32 (0.06)	27.8	1.09
	25	95	1006.6 (194.5)	1.27 (0.07)	28.8	1.18
	30	98	1002.3 (186.5)	1.29 (0.07)	30.1	1.08
MPC	5	284	546.4 (100.5)	0.016 (0.0024)	21.3	1.05
	10	278	410.3 (106.5)	0.014 (0.0023)	22.9	1.36
	15	285	434.6 (115.0)	0.015 (0.0025)	25.7	1.00
	20	285	442.9 (103.0)	0.018 (0.0027)	27.1	1.00
	25	285	438.3 (100.0)	0.017 (0.0026)	30.0	1.00
	30	285	471.1 (100.5)	0.020 (0.0027)	30.4	1.00
Portfolio	5	10	1094.1 (853.0)	76.80 (41.73)	3.1	1.43
	10	10	1020.1 (906.0)	72.00 (40.25)	2.9	1.39
	15	10	764.3 (823.5)	53.31 (31.49)	3.5	1.16
	20	10	646.5 (716.5)	46.07 (31.75)	3.5	1.00
	25	10	723.1 (700.0)	50.40 (36.40)	3.5	1.08
	30	10	752.7 (727.5)	47.11 (51.75)	3.8	1.11
SPCA	5	10	1311.3 (1078.5)	54.76 (30.19)	12.2	1.56
	10	10	1002.5 (874.0)	40.78 (22.60)	12.9	1.27
	15	10	718.2 (739.5)	27.86 (17.53)	14.0	1.00
	20	10	1029.0 (756.5)	49.02 (23.12)	14.7	1.31
	25	10	1283.5 (1172.0)	60.91 (27.93)	14.9	1.61
	30	10	1157.8 (1203.5)	46.84 (31.43)	15.6	1.58
Block	5	19	471.3 (313.0)	18.53 (9.53)	2.3	1.73
	10	20	324.8 (279.0)	11.88 (6.98)	2.5	1.00
	15	20	451.9 (300.0)	18.06 (7.74)	2.5	1.13
	20	20	705.9 (264.0)	30.90 (7.36)	3.2	1.41
	25	20	771.2 (256.0)	32.30 (7.39)	2.8	1.36
	30	20	555.1 (279.0)	19.38 (8.89)	3.0	1.26
Lovász	5	12	2020.9 (881.0)	15.92 (3.16)	6.2	6.84
	10	14	1752.6 (1385.0)	9.52 (3.90)	8.5	2.20
	15	15	1796.3 (1117.0)	8.46 (3.06)	8.5	1.00
	20	13	1502.6 (748.0)	8.38 (1.79)	9.0	2.48
	25	14	1275.2 (723.0)	5.99 (1.93)	9.9	2.10
	30	14	1472.3 (663.0)	8.37 (1.74)	10.6	1.85
Mittelmann	5	30	1746.6 (838.0)	208.90 (89.87)	2.7	1.36
	10	31	1463.7 (703.5)	183.14 (68.08)	3.0	1.25
	15	31	1400.9 (638.5)	174.79 (61.94)	3.7	1.14
	20	29	1386.7 (713.5)	170.97 (50.64)	3.8	1.28
	25	31	1342.7 (627.5)	173.63 (47.68)	3.9	1.00
	30	30	1302.2 (689.5)	170.04 (41.17)	4.0	1.10

[1] mean and median (based on subset of problems where all solver configurations solved the problem);
[2] geometric mean of fraction of total solve time spent in acceleration-related functions;
[3] normalized shifted geometric mean of solve time, see (3.38) (based on all problems in the problem set);

6 Conclusions

Contents

In chapters 3, 4, and 5 we outlined a range of techniques to scale first-order optimisation methods, both to handle larger problem dimensions and to achieve more accurate solutions. Before we discuss the main contributions of this thesis in Section 6.2 and future research directions in Section 6.3 we demonstrate the effects of incrementally adding some of these techniques to a classic ADMM solver.

6.1 Conclusion benchmarks

For these benchmarks we revisit the three problem sets of decomposable SDPs that were discussed in Section 4.6: block arrow sparse SDPs, large SDPs from SDPLib, and SDPs with nonchordal sparsity pattern from the SuiteSparse matrix collection. The benchmark compares the accelerated Douglas-Rachford splitting solver `SCS` and the interior-point solver `MOSEK` to `COSMO` with none of the techniques enabled. We then incrementally add chordal decomposition, clique merging, multithreading,

and AA to COSMO. The solvers were configured with a time limit of 30 min and to achieve accuracies of 10^{-3} and 10^{-5} respectively.

Figure 6.1 shows the solve times of each solver for the block arrow sparse SDPs with increasing number of blocks for low and higher accuracies. These SDPs have a cone size between 520–1420, 100 affine constraints, and number of nonzeros in the constraint matrix A between 6.5×10^5–1.8×10^6; see Section 4.6.1. At low

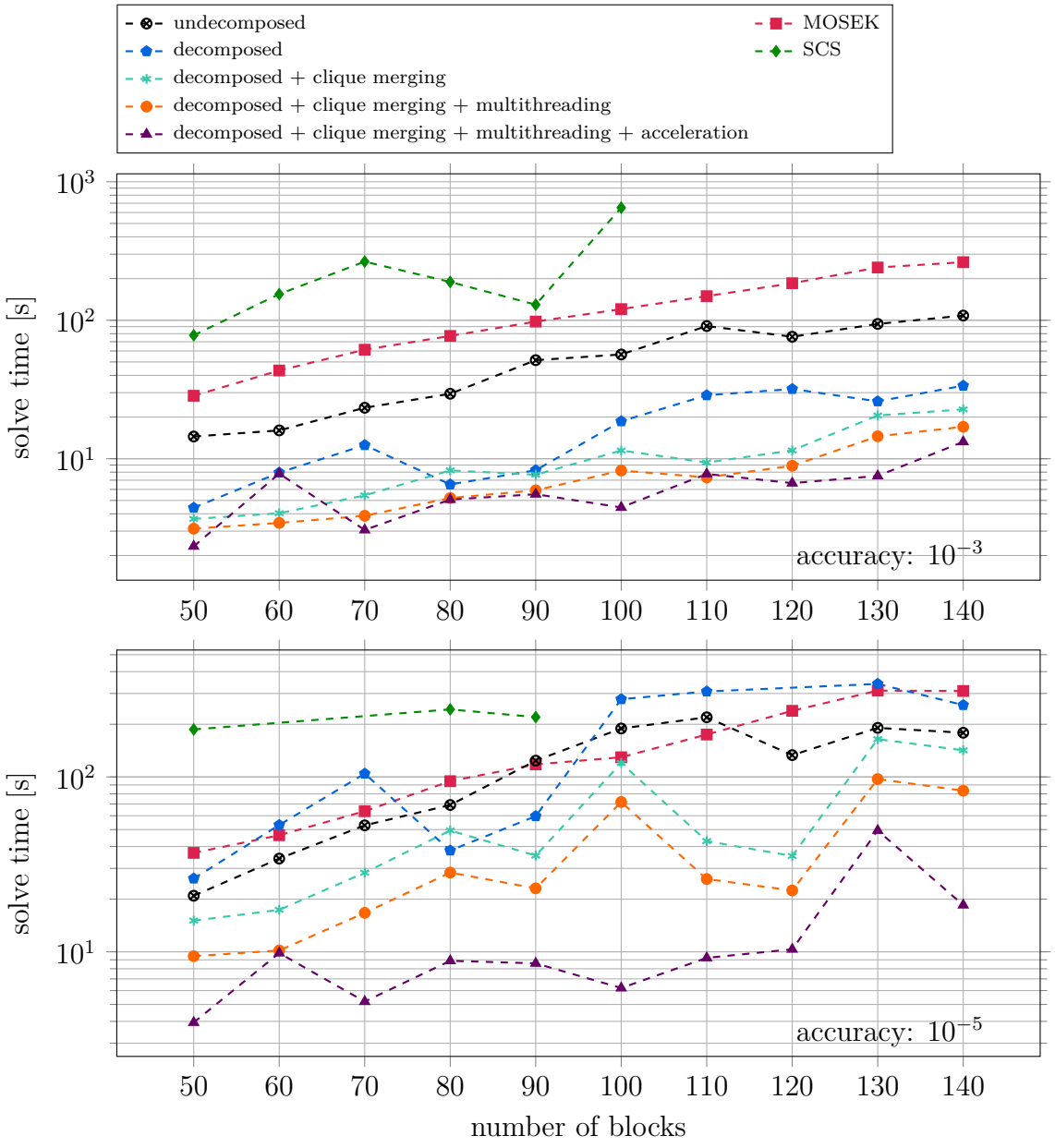


Figure 6.1: Solve time of various COSMO configurations, SCS, and MOSEK for block arrow sparse SDPs with increasing number of blocks.

accuracy we observe the expected behaviour that each additional technique seems to improve the solve time, with the biggest improvement achieved by enabling chordal decomposition. Moreover, the effect of acceleration seems to be inconsistent which is in line with the observed behaviour in Figure 5.2 of similar convergence between the non-accelerated and the accelerated case in the low accuracy region. Since the activation of multithreading does not influence the convergence of the solver, it is not surprising that it gives the most consistent improvement across all problems. For the higher accuracy case we see a large effect of the acceleration. Compared to `MOSEK`, an improvement of up to 20x is achieved for both the low and higher accuracy case. `SCS` does not converge within the time limit for higher dimensions.

The combined results for the SDPLib problems and the SDPs with nonchordal sparsity pattern from the SuiteSparse matrix collection are shown in Figure 6.2. The problem dimensions for these problems are shown in Table 4.3.

In terms of the impact of each individual technique on the solve time, similar conclusions as in the previous problem set can be drawn. For the high accuracy case we see that the acceleration generally improves the solve time, but there are some cases where this is not the case, e.g. `rs200`. The accelerated solver is the fastest method for most problems in the low accuracy case, e.g. for problem `rs828` the solve time is 88x lower than for `MOSEK`. However, in the higher accuracy case `MOSEK` benefits from the fact that the solver generally only needs 2-3 additional iterations to achieve the desired accuracy, whereas the first-order solvers need substantially more iterations. Therefore, `MOSEK` is faster than the first-order methods for the set of problems where the number of nonzeros is below 50 000. For the very large problems with more than 1 500 000 nonzeros the accelerated version of `COSMO` achieves the best solve times, aided by the chordal decomposition and comparably good convergence.

6.2 Conclusions and contributions

Many applications in control, machine learning, process engineering, operations research, statistics, and finance involve the solution of convex conic optimisation

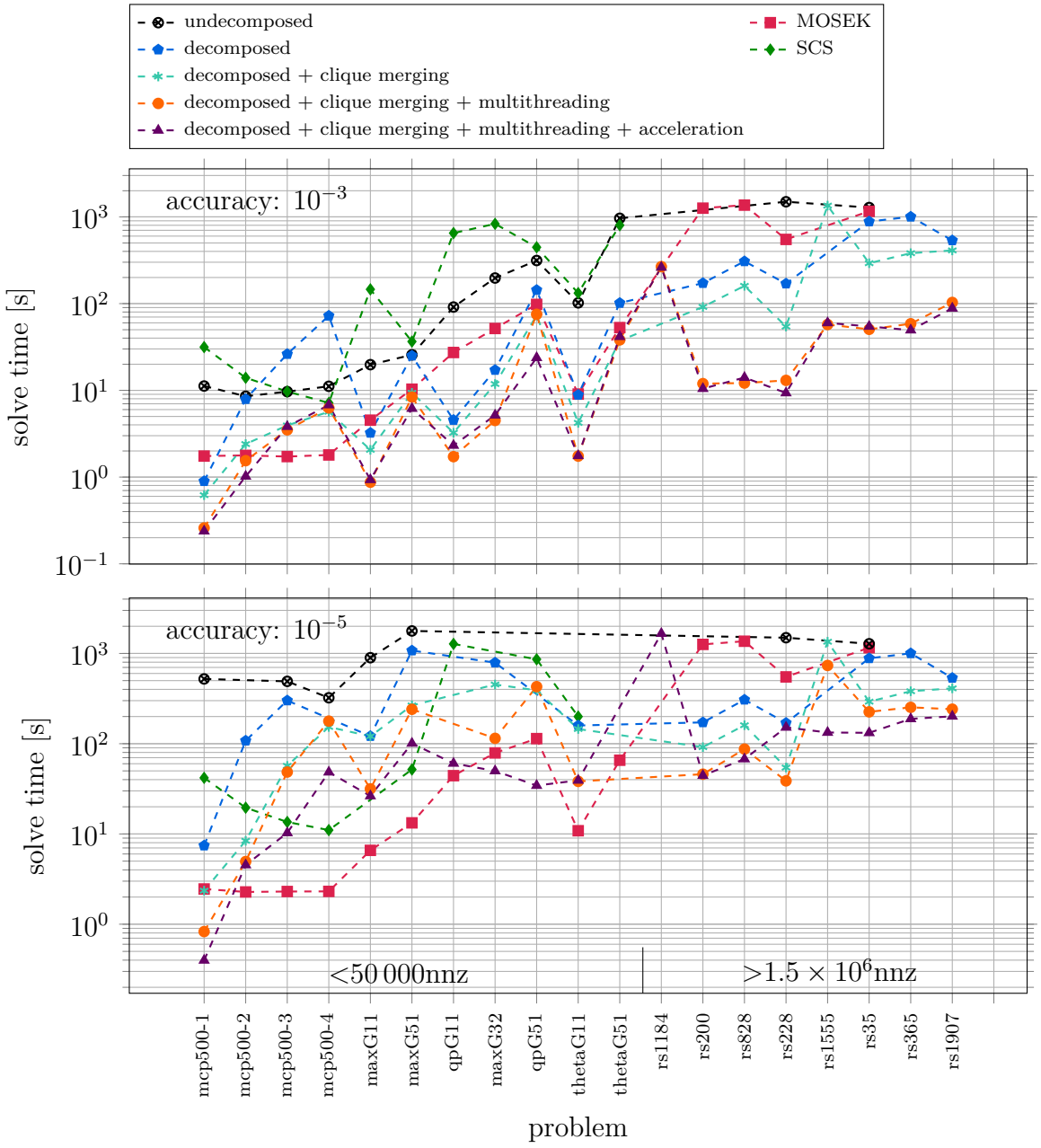


Figure 6.2: Solve time of various `COSMO` configurations, `SCS`, and `MOSEK` for decomposable SDPLib problems and SDPs with sparsity patterns from the SuiteSparse matrix collection.

problems. As the underlying data, network structures, or number of agents increase, modern solver algorithms must be able to solve the resulting very large problems in an appropriate amount of time. Although recent advancements in computing hardware can help reduce the solution time of solvers, further improvements in the underlying algorithms are still needed. This thesis discusses and investigates different techniques to either improve the convergence or reduce the per-iteration computation

cost of FOMs for large conic optimisation problems. Specific contributions are as follows:

`COSMO`: A solver package for large convex optimisation problems

We developed the conic operator splitting method (`COSMO`), an ADMM-based solver written in Julia which allows flexible problem formulation by combining a quadratic objective function with a combination of conic constraints. By supporting the zero cone, the nonnegative orthant, the second-order cone, the positive semidefinite cone, the exponential cone and its dual, the power cone and its dual, and the hyperbox, the solver is equipped to handle a vast range of convex problems that appear in applications. Moreover, we demonstrated that the solver design allows the user to define custom convex cones that allow faster projections [141] and / or more natural problem formulations. More customisation through the user is possible by choosing from a range of direct or indirect linear system solvers and by using Julia's type abstraction system to solve problems with arbitrary floating point precision.

We demonstrate the performance of `COSMO` against other state-of-the-art algorithms, especially on large SDPs and parametric problems that benefit from warm starting and factorisation caching. Moreover, we show that the ability to handle quadratic objective functions allows `COSMO` to efficiently solve QPs and SDPs with quadratic objective terms. This avoids the expensive, and in many cases sparsity-destroying, transformation using a second-order cone that other conic solvers rely on. Comparing our solver against the C-solver `OSQP`, which uses the same splitting for QPs, shows that the performance penalty due to the Julia implementation is small ($<10\,\%$). On the other hand, the Julia implementation allows us to implement and test advanced algorithmic techniques efficiently. Thus, the other contributions of this thesis, such as clique merging and acceleration techniques, have been fully integrated in the solver package and can be used selectively or in combination.

Scaling conic solvers for large structured semidefinite programs

Chordal decomposition techniques can be used to transform structured SDPs into equivalent problems involving more but smaller positive semidefinite constraints. It is therefore one of the main methods of adapting solver algorithms for very large SDPs if they inherently possess a structure or if the structure is imposed. We discuss chordal decomposition in the context of a first-order solver, building on the work of Zheng et al. [181], where the main computational bottleneck is the projection step onto the positive semidefinite cone. The initial decomposition of the positive semidefinite matrix variable into smaller blocks, represented by cliques, is not necessarily optimal and can in some cases even be detrimental to the solver performance. We develop a novel algorithm that uses the reduced clique graph to merge cliques to reduce the projection time of the solver algorithm. Comparing this method with existing heuristics, we show that our approach can speed up the per-iteration projection time by a factor of up to 3.

Our implementation in `COSMO` combines the clique merging approach with the benefits of executing projections for the individual blocks in parallel on multiple CPU threads. We show that, for nicely decomposable problems, doubling the number of available CPU threads halves the projection time as expected and that for more realistic sparsity structures multithreading still provides a significant speed-up.

Safeguarded acceleration methods for ADMM

Optimisation methods based on first-order information are often the preferred choice for very large optimisation problems outside the range that interior-point methods can handle. While they perform well for low to medium accuracies, they suffer from slow convergence if a higher accuracy is required. We show how to wrap an ADMM-based solver into an Anderson acceleration scheme, that enforces constraints on the norm of the residual operator and uses repeated restarts to globalize the method. The restarts integrate well with the requirement of the ADMM method to access pure operator steps for step size adaptation and infeasibility detection.

Benchmarks on a range of different problem sets show significant reductions in both mean iterations up to a factor of 8, and mean solve time up to a factor of 6. Profiling results show that the acceleration can be carried out with fairly insignificant computational cost for SDPs and large QPs. We further showed that it integrates well with the chordal decomposition, clique merging, and multithreading features of `COSMO`.

6.3 Future research directions

In the following we outline some possible directions for future research:

Extensions to the solver

Many convex optimisation problems of interest can be represented using extended formulations based on the standard cones discussed in this thesis. However, these formulations often add a number of auxiliary variables to the problem which lead to more nonzeros in the constraint matrix. In many cases constraints can be represented more naturally through non-standard cones, e.g. using the log-determinant cone directly instead of representing it using a PSD cone and an exponential cone. Recently, Coey et al. [33] defined the gradients and Hessians of logarithmically homogeneous self-concordant barrier functions for a number of exotic cones to allow more natural problem formulations for their interior-point solver. It would be interesting to investigate if projection functions for some of these cones can be implemented efficiently and how these formulations compare to formulations that use classical cones.

We noticed differences in the convergence of the operator splitting used in this thesis and the homogeneous self-dual embedding approach [122], where the former seems to benefit from quadratic terms in the objective and the latter often seems faster to converge on problems with linear objectives. The author of [123] recently developed an embedding that allows the original homogeneous self-dual embedding to handle quadratic objectives. A thorough comparison of the convergence of these

different available operator splittings on a wide range of problems with and without quadratic objective would be desirable.

Schubiger et al. [146] implemented a GPU version of the `OSQP` solver for QPs and demonstrated up to two orders of magnitude improvements over the CPU implementation. They used a conjugate gradient method to solve the linear system step on a GPU. Moreover, the matrix-vector operations in the projection steps were also handled by the GPU. The Julia programming language offers a fairly flexible integration of GPU operations via multiple dispatch based on the array type that contains the problem data. Aside from the positive semidefinite cone projection, all cone projections in `COSMO` are written in native Julia and mostly consist of matrix-vector multiplications and vector operations. This should allow the solver to perform all operations on the GPU, which avoids slow data transfer between CPU and GPU. This should be leveraged to support both platforms. It would be interesting to investigate the computational advantages of an Anderson accelerated first-order GPU solver for general conic problems.

Optimising clique merging for multiple threads and/or interior-point methods

In Chapter 4 we demonstrated that clique merging can be successfully used to shape the sparsity pattern of a structured SDP and customised to specific hardware. This allowed us to achieve faster projection times at every iteration. However, our current clique merging algorithm is designed for single-threaded execution. Independently of the clique merging, we showed that decomposable SDPs with similar clique sizes benefit significantly from multithreading. A future research direction would be to adapt our clique merging algorithm for multithreading, e.g. by trying to achieve similar block sizes. For problems that have one large and many smaller cliques an interesting approach could be to use all available threads to project the block corresponding to the large clique and then switch to parallel projection for the small blocks.

Moreover, our clique graph-based merging approach currently assumes that the solver algorithm is based on a first-order method. However, the edge-weighting function that scores merge candidates can easily be changed to customise the method for interior-point solvers.

In the benchmarks of Section 4.6 we noticed that in some cases the solver needs more iterations for the decomposed SDP compared to the undecomposed problem. This is likely due to the additional auxiliary variables in the decomposed problem (4.12) which link overlapping entries and introduce more slack into the problem. It might be possible to improve the convergence by spreading out the auxiliary variables more evenly among overlapping entries.

Applications with decomposable SDPs

We demonstrated the solver performance gains that are achievable for large SDPs using a FOM and chordal decomposition. A classic example of a real-world problem that involves solving a decomposable SDP is the optimal power flow problem [113]. Other examples are the Lovász theta function problem, the approximate Euclidean distance matrix completion [150], and the relaxed boolean optimisation problem [25] which appears in the maximum cut SDP relaxation [65], the graph partitioning problem [170] and in LQR control with binary inputs. Zheng [180] has applied chordal decomposition to sparse Lyapunov-type LMIs in the context of decentralized control of networked systems and to scale large sum-of-squares programs. A further research direction is to consider new applications that naturally have an imposed sparsity pattern. Moreover, for very large dimensions it can be a reasonable approach to impose a specific sparsity pattern to make computing a solution approximation feasible. Recent results explored the inner approximation of the PSD cone using diagonally dominant matrices [2], factor-width k matrices [147], or Grassmannian packing [179]. Many interesting research questions arise around comparing the quality of these approximations, quantifying conservatism, and applying these approximations to real-world SDPs.

Robust acceleration for first-order methods

There are still many open questions around quasi-Newton method based acceleration for FOMs. From a theoretical viewpoint it would be important to establish a formal convergence rate for a safeguarded AA method applied to a nonsmooth and nonexpansive operator. From a practical viewpoint more experiments are necessary to determine better heuristics for choosing acceleration parameters such as the memory size. It would also be interesting to compare the effectiveness of our safeguarding condition with the ones used in [50, 152, 177]. In particular it seems that practical performance declines slightly if these conditions are strictly enforced and the acceleration scheme is not allowed to temporarily choose "slightly worse" iterates.

We looked into ways of improving the numerical conditioning of the Jacobian approximation (5.25) which relies on a history of past residual differences. The bad conditioning often stems from the FOM reaching a region of slow progress, e.g. getting stuck on a wrong active set when a QP is solved. An interesting approach would be to inject older residual iterates or iterates orthogonal to the space spanned by the recent iterates to improve the conditioning. Another interesting approach would be to establish a connection between inferior acceleration performance and the recent work of [22], in which the author analyses four different convergence regimes for ADMM.

Appendices

A

Additional benchmark information

Contents

A.1 Lovász theta graphs

The following graphs from the SuiteSparse Matrix Collection [39] were used for the Lovász theta function benchmarks in Section 5.4: `GD06_theory`, `GD97_b`, `GD98_c`, `Journals`, `Sandi_authors`, `Trefethen_150`, `adjnoun`, `celegans_metabolic`, `football`, `grid1`, `grid1_dual`, `jazz`, `mycielskian7`, `mycielskian8`, `polbooks`

List of Figures

References

[1] J. Agler et al. “Positive semidefinite matrices with a given sparsity pattern”. In: *Linear Algebra and its Applications* 107 (1988), pp. 101–149.

[2] A. A. Ahmadi and A. Majumdar. “DSOS and SDSOS optimization: more tractable alternatives to sum of squares and semidefinite optimization”. In: *SIAM Journal on Applied Algebra and Geometry* 3.2 (2019), pp. 193–230.

[3] A. Ali, E. Wong, and J. Z. Kolter. “A semismooth Newton method for fast, generic convex programming”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 70–79.

[4] F. Alizadeh. “Combinatorial optimization with interior point methods and semi-definite matrices”. Ph.D. Thesis. University of Minnesota, 1991.

[5] M. S. Andersen, J. Dahl, and L. Vandenberghe. “Implementation of nonsymmetric interior-point methods for linear optimization over sparse matrix cones”. In: *Mathematical Programming Computation* 2.3-4 (2010), pp. 167–201.

[6] M. S. Andersen and L. Vandenberghe. *CHOMPACK: A Python package for chordal matrix computations*. 2015. URL: https://chompack.readthedocs.io/.

[7] D. G. Anderson. “Iterative procedures for nonlinear integral equations”. In: *Journal of the ACM* 12.4 (1965), pp. 547–560.

[8] E. Anderson et al. *LAPACK Users’ guide*. SIAM, 1999.

[9] C. Ashcraft and R. Grimes. “The influence of relaxed supernode partitions on the multifrontal method”. In: *ACM Transactions on Mathematical Software* 15.4 (1989), pp. 291–309.

[10] G. Banjac et al. “Infeasibility detection in the alternating direction method of multipliers for convex optimization”. In: *Journal of Optimization Theory and Applications* 183.2 (2019), pp. 490–519.

[11] S. Barman et al. “Decomposition methods for large scale LP decoding”. In: *IEEE Transactions on Information Theory* 59.12 (2013), pp. 7870–7886.

[12] H. Bauschke and P. Combettes. *Convex Analysis and Monotone Operator Theory in Hilbert Spaces*. 1st ed. Springer, 2011.

[13] A. Beck and M. Teboulle. “A fast iterative shrinkage-thresholding algorithm for linear inverse problems”. In: *SIAM Journal on Imaging Sciences* 2.1 (2009), pp. 183–202.

[14] A. Ben-Tal and A. Nemirovski. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. SIAM, 2001.

[15] A. Berry et al. “Maximum cardinality search for computing minimal triangulations of graphs”. In: *Algorithmica* 39.4 (2004), pp. 287–298.

[16] D. Bertsekas. *Convex analysis and optimization*. Athena Scientific, 2003.

[17] D. Bertsimas. "The achievable region method in the optimal control of queueing systems; formulations, bounds and policies". In: *Queueing systems* 21.3-4 (1995), pp. 337–389.

[18] D. Bertsimas and I. Popescu. "Optimal inequalities in probability theory: A convex optimization approach". In: *SIAM Journal on Optimization* 15.3 (2005), pp. 780–804.

[19] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM Review* 59.1 (2017), pp. 65–98.

[20] C. H. Bischof, P. Tak, and P. Tang. "Incremental condition estimation". In: *SIAM J. Matrix Anal. Appl* 11 (1990), pp. 312–322.

[21] J. R. S. Blair and B. Peyton. "An introduction to chordal graphs and clique trees". In: *Graph theory and sparse matrix computation*. Springer, 1993, pp. 1–29.

[22] D. Boley. "Local linear convergence of the alternating direction method of multipliers on quadratic or linear programs". In: *SIAM Journal on Optimization* 23.4 (2013), pp. 2183–2207.

[23] B. Borchers. "SDPLIB 1.2, a library of semidefinite programming test problems". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 683–690.

[24] F. Borrelli, A. Bemporad, and M. Morari. *Predictive control for linear and hybrid systems*. Cambridge University Press, 2017.

[25] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

[26] S. Boyd et al. *Linear matrix inequalities in system and control theory*. Vol. 15. Society for Industrial and Applied Mathematics, 1994.

[27] S. Boyd et al. "A tutorial on geometric programming". In: *Optimization and engineering* 8.1 (2007), pp. 67–127.

[28] S. Boyd et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1 (2011), pp. 1–122.

[29] S. Boyd et al. *Performance bounds and suboptimal policies for multi-period investment*. Citeseer, 2014.

[30] S. Boyd et al. "Multi-period trading via convex optimization". In: *Foundations and Trends in Optimization* 3.1 (2017), pp. 1–76.

[31] S. Burer and R. D. C. Monteiro. "A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization". In: *Mathematical Programming* 95.2 (2003), pp. 329–357.

[32] Y. Censor and S. A. Zenios. *Parallel optimization: Theory, algorithms, and applications*. Oxford University Press, 1997.

[33] C. Coey, L. Kapelevich, and J. P. Vielma. "Towards practical generic conic optimization". In: *arXiv preprint:2005.01136* (2020).

[34] A. d'Aspremont, D. Scieur, and A. Taylor. "Acceleration methods". In: *arXiv preprint arXiv:2101.09545* (2021).

[35] A. d'Aspremont et al. "A direct formulation for sparse PCA using semidefinite programming". In: *Advances in Neural Information Processing Systems* 17 (2004), pp. 41–48.

[36] G. Dantzig. *Linear programming and extensions*. Princeton University Press, 1963.

[37] T. A. Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[38] T. A. Davis. *SuiteSparse: A suite of sparse matrix software*. http://faculty.cse.tamu.edu/davis/suitesparse.html. 2015.

[39] T. A. Davis and Y. Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[40] T. De Bie and N. Cristianini. "Fast SDP relaxations of graph cut clustering, transduction, and other combinatorial problems". In: *Journal of Machine Learning Research* 7.Jul (2006), pp. 1409–1436.

[41] J. E. Dennis, Jr. and J. J. Moré. "Quasi-Newton methods, motivation and theory". In: *SIAM review* 19.1 (1977), pp. 46–89.

[42] I. Dunning, J. Huchette, and M. Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (2017), pp. 295–320.

[43] J. Eckstein and D. P. Bertsekas. "On the Douglas-Rachford Splitting Method and the Proximal Point Algorithm for Maximal Monotone Operators". In: *Mathematical Programming* 55.1 (1992), pp. 293–318.

[44] J. Eckstein. "Splitting methods for monotone operators with applications to parallel optimization". PhD thesis. Massachusetts Institute of Technology, 1989.

[45] H. Everett. "Generalized Lagrange multiplier method for solving problems of optimum allocation of resources". In: *Operations Research* 11.3 (1963), pp. 399–417.

[46] V. Eyert. "A comparative study on methods for convergence acceleration of iterative vector sequences". In: *Journal of Computational Physics* 124.2 (1996), pp. 271–285.

[47] H. Fang and Y. Saad. "Two classes of multisecant methods for nonlinear acceleration". In: *Numerical Linear Algebra with Applications* 16.3 (2009), pp. 197–221.

[48] M. Fazlyab, M. Morari, and G. J. Pappas. "Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming". In: *IEEE Transactions on Automatic Control* (2020).

[49] J. Ferreau. *MPC Benchmarking Collection: Open collection of model predictive control (MPC) benchmarking problems*. 2020. URL: https://github.com/ferreau/mpcBenchmarking.

[50] A. Fu, J. Zhang, and S. Boyd. "Anderson Accelerated Douglas–Rachford Splitting". In: *SIAM Journal on Scientific Computing* 42.6 (2020), A3560–A3583.

[51] K. Fujisawa, M. Fukuda, and K. Nakata. "Preprocessing sparse semidefinite programs via matrix completion". In: *Optimization Methods and Software* 21.1 (2006), pp. 17–39.

[52] K. Fujisawa et al. “User’s manual for SparseCoLO: Conversion methods for sparse conic-form linear optimization problems”. In: *Research Report B-453, Dept. of Math. and Comp. Sci. Japan, Tech. Rep.* (2009), pp. 152–8552.

[53] M. Fukuda et al. “Exploiting sparsity in semidefinite programming via matrix completion I: General framework”. In: *SIAM Journal on Optimization* 11.3 (2001), pp. 647–674.

[54] D. Gabay. “Chapter 10: Applications of the method of multipliers to variational inequalities”. In: *Studies in Mathematics and its Applications*. Vol. 15. Elsevier, 1983, pp. 299–331.

[55] D. Gabay and B. Mercier. *A dual algorithm for the solution of non linear variational problems via finite element approximation*. Institut de recherche d’informatique et d’automatique, 1975.

[56] M. Garstka, M. Cannon, and P. Goulart. “COSMO: A conic operator splitting method for large convex problems”. In: *18th European Control Conference (ECC)*. Naples, Italy, 2019, pp. 1951–1956.

[57] M. Garstka, M. Cannon, and P. Goulart. “A clique graph based merging strategy for decomposable SDPs”. In: *IFAC-PapersOnLine* 53.2 (2020). 21th IFAC World Congress, pp. 7355–7361.

[58] M. Garstka, M. Cannon, and P. Goulart. “COSMO: A Conic Operator Splitting Method for Convex Conic Problems (to appear)”. In: *Journal of Optimization Theory and Applications* (2021).

[59] F. Gavril. “Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 180–187.

[60] D. M. Gay and R. B. Schnabel. “Solving systems of nonlinear equations by Broyden’s method with projected updates”. In: *Nonlinear Programming*. Elsevier, 1978, pp. 245–281.

[61] P. Giselsson and S. Boyd. “Diagonal scaling in Douglas-Rachford splitting and ADMM”. In: *IEEE 53rd Annual Conference on Decision and Control (CDC)*. IEEE. 2014, pp. 5033–5039.

[62] P. Giselsson and S. Boyd. “Metric selection in fast dual forward-backward splitting”. In: *Automatica* 62 (2015), pp. 1–10.

[63] P. Giselsson, M. Fält, and S. Boyd. “Line search for averaged operator iteration”. In: *IEEE 55th Conference on Decision and Control (CDC)* (2016), pp. 1015–1022.

[64] R. Glowinski and A. Marroco. “Sur l’approximation, par éléments finis d’ordre un, et la résolution, par pénalisation-dualité d’une classe de problèmes de Dirichlet non linéaires”. In: *Revue française d’automatique, informatique, recherche opérationnelle. Analyse numérique* 9.2 (1975), pp. 41–76.

[65] M. X. Goemans and D. P. Williamson. “Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming”. In: *Journal of the ACM* 42.6 (1995), pp. 1115–1145.

[66] T. Goldstein and S. Osher. “The split Bregman method for L1-regularized problems”. In: *SIAM Journal on Imaging Sciences* 2.2 (2009), pp. 323–343.

[67] G. H. Golub and C. F. Van Loan. *Matrix computations*. Vol. 3. Johns Hopkins University Press, 2013.

[68] A. Greenbaum. *Iterative methods for solving linear systems*. Vol. 17. SIAM, 1997.

[69] L. Grippo, F. Lampariello, and S. Lucidi. "A nonmonotone line search technique for Newtons method". In: *SIAM Journal on Numerical Analysis* 23.4 (1986), pp. 707–716.

[70] R. Grone et al. "Positive definite completions of partial Hermitian matrices". In: *Linear Algebra and its Applications* 58 (1984), pp. 109–124.

[71] M. Habib and J. Stacho. "Polynomial-time algorithm for the leafage of chordal graphs". In: *European Symposium on Algorithms*. Springer. 2009, pp. 290–300.

[72] M. Habib and J. Stacho. "Reduced clique graphs of chordal graphs". In: *European Journal of Combinatorics* 33.5 (2012), pp. 712–735.

[73] C. B. Haselgrove. "The solution of non-linear equations and of differential equations with two-point boundary conditions". In: *The Computer Journal* 4.3 (1961), pp. 255–259.

[74] J. Håstad. "Clique is hard to approximate within 1- ε". In: *Acta Mathematica* 182.1 (1999), pp. 105–142.

[75] E. V. Haynsworth. *On the Schur complement*. Tech. rep. University of Basel, Switzerland, 1968.

[76] C. Helmberg et al. "An interior-point method for semidefinite programming". In: *SIAM Journal on Optimization* 6.2 (1996), pp. 342–361.

[77] N. C. Henderson and R. Varadhan. "Damped Anderson acceleration with restarts and monotonicity control for accelerating EM and EM-like algorithms". In: *Journal of Computational and Graphical Statistics* 28.4 (2019), pp. 834–846.

[78] M. R. Hestenes. "Multiplier and gradient methods". In: *Journal of Optimization Theory and Applications* 4.5 (1969), pp. 303–320.

[79] M. R. Hestenes and E. Stiefel. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49.6 (1952), pp. 409–436.

[80] L. T. K. Hien. "Differential properties of Euclidean projection onto power cone". In: *Mathematical Methods of Operations Research* 82 (2015), pp. 265–284.

[81] N. J. Higham. "Computing the nearest correlation matrix - a problem from finance". In: *IMA Journal of Numerical Analysis* 22.3 (2002), pp. 329–343.

[82] N. J. Higham and N. Strabi. "Anderson acceleration of the alternating projections method for computing the nearest correlation matrix". In: *Numerical Algorithms* 72.4 (2016), pp. 1021–1042.

[83] F. Jarre. "An interior-point method for minimizing the maximum eigenvalue of a linear combination of matrices". In: *SIAM Journal on Control and Optimization* 31.5 (1993), pp. 1360–1377.

[84] D. Johnson, G. Pataki, and F. Alizadeh. *Seventh DIMACS implementation challenge: Semidefinite and related problems*. dimacs.rutgers.edu/Challenges/Seventh. 2000.

[85] A. Kalbat and J. Lavaei. “A fast distributed algorithm for decomposable semidefinite programs”. In: *54th IEEE Conference on Decision and Control (CDC)*. IEEE. 2015, pp. 1742–1749.

[86] A. Kalinkin, A. Anders, and R. Anders. “Intel Math Kernel Library PARDISO for Intel Xeon Phi TM Manycore Coprocessor”. In: *Applied Mathematics* 6.08 (2015), p. 1276.

[87] N. Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the 16th annual ACM symposium on Theory of Computing*. ACM. 1984, pp. 302–311.

[88] S. Kim et al. “Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion”. In: *Mathematical programming* 129.1 (2011), pp. 33–68.

[89] M. A. Krasnoselski. “Two remarks on the method of successive approximations”. In: *Nauk* 10 (1955), pp. 123–127.

[90] J. B. Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem”. In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[91] G. R. G. Lanckriet et al. “Learning the kernel matrix with semidefinite programming”. In: *Journal of Machine learning research* 5.Jan (2004), pp. 27–72.

[92] B. Legat et al. “MathOptInterface: a data structure for mathematical optimization problems”. In: *arXiv preprint arXiv:2002.03447* (2020).

[93] K. Levenberg. “A method for the solution of certain non-linear problems in least squares”. In: *Quarterly of Applied Mathematics* 2.2 (1944), pp. 164–168.

[94] P. Lions and B. Mercier. “Splitting algorithms for the sum of two nonlinear operators”. In: *SIAM Journal on Numerical Analysis* 16.6 (1979), pp. 964–979.

[95] R. J. Lipton, D. J. Rose, and R. E. Tarjan. “Generalized nested dissection”. In: *SIAM Journal on Numerical Analysis* 16.2 (1979), pp. 346–358.

[96] D. C. Liu and J. Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical Programming* 45.1 (1989), pp. 503–528.

[97] J. Liu. “The role of elimination trees in sparse factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 11.1 (1990), pp. 134–172.

[98] Y. Liu, E. K. Ryu, and W. Yin. “A new use of Douglas–Rachford splitting for identifying infeasible, unbounded, and pathological conic programs”. In: *Mathematical Programming* 177.1 (2019), pp. 225–253.

[99] M. S. Lobo et al. “Applications of second-order cone programming”. In: *Linear algebra and its Applications* 284.1-3 (1998), pp. 193–228.

[100] L. Lovász. “On the Shannon capacity of a graph”. In: *IEEE Transactions on Information Theory* 25.1 (1979), pp. 1–7.

[101] R. D. Luce and A. D. Perry. “A method of matrix analysis of group structure”. In: *Psychometrika* 14.2 (1949), pp. 95–116.

[102] R. Madani, A. Kalbat, and J. Lavaei. “ADMM for sparse semidefinite programming with applications to optimal power flow problem”. In: *54th IEEE Conference on Decision and Control (CDC)*. IEEE. 2015, pp. 5932–5939.

[103] V. Mai and M. Johansson. "Anderson acceleration of proximal gradient methods". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 6620–6629.

[104] W. R. Mann. "Mean value methods in iteration". In: *Proceedings of the American Mathematical Society* 4.3 (1953), pp. 506–510.

[105] I. Maros and C. Mészáros. "A repository of convex quadratic programming problems". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 671–681.

[106] D. W. Marquardt. "An algorithm for least-squares estimation of nonlinear parameters". In: *Journal of the Society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441.

[107] S. Mehrotra. "On the implementation of a primal-dual interior point method". In: *SIAM Journal on Optimization* 2.4 (1992), pp. 575–601.

[108] A. Miele, E. Cragg, and A. Levy. "Use of the augmented penalty function in mathematical programming problems, part 2". In: *Journal of Optimization Theory and Applications* 8.2 (1971), pp. 131–153.

[109] A. Miele et al. "Use of the augmented penalty function in mathematical programming problems, part 1". In: *Journal of Optimization Theory and Applications* 8.2 (1971), pp. 115–130.

[110] A. Miele et al. "On the method of multipliers for mathematical programming problems". In: *Journal of Optimization Theory and Applications* 10.1 (1972), pp. 1–33.

[111] H. D. Mittelmann. "An independent benchmarking of SDP and SOCP solvers". In: *Mathematical Programming* 95.2 (2003), pp. 407–430.

[112] H. Mittelmann. *Benchmarks for optimization software*. http://plato.asu.edu/bench.html. 2020.

[113] D. K. Molzahn et al. "Implementation of a large-scale optimal power flow solver based on semidefinite programming". In: *IEEE Transactions on Power Systems* 28.4 (2013), pp. 3987–3998.

[114] MOSEK, Aps. *The MOSEK optimization toolbox for MATLAB manual. Version 8.0 (Revision 57)*. 2017.

[115] K. Murty and S. Kabadi. "Some NP-complete problems in quadratic and nonlinear programming". In: *Mathematical programming* 39.2 (1987), pp. 117–129.

[116] K. Nakata et al. "Exploiting sparsity in semidefinite programming via matrix completion II: Implementation and numerical results". In: *Mathematical Programming* 95.2 (2003), pp. 303–327.

[117] A. Nemirovski. "Advances in convex optimization: conic programming". In: *International Congress of Mathematicians*. Vol. 1. 2007, pp. 413–444.

[118] Y. E. Nesterov. "A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$". In: *Dokl. akad. nauk SSSR*. Vol. 269. 1983, pp. 543–547.

[119] Y. Nesterov and A. Nemirovsky. "A general approach to polynomial-time algorithms design for convex programming". In: *Report, Central Economical and Mathematical Institute, USSR Academy of Sciences, Moscow* (1988).

[120] Y. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming*. SIAM, 1994.

[121] D. O'Connor and L. Vandenberghe. "Primal-dual decomposition by operator splitting and applications to image deblurring". In: *SIAM Journal on Imaging Sciences* 7.3 (2014), pp. 1724–1754.

[122] B. O'Donoghue et al. "Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding". In: *Journal of Optimization Theory and Applications* 169.3 (June 2016), pp. 1042–1068.

[123] B. O'Donoghue. "Operator splitting for a homogeneous embedding of the monotone linear complementarity problem". In: *arXiv preprint arXiv:2004.02177* (2020).

[124] B. O'Donoghue, G. Stathopoulos, and S. Boyd. "A splitting method for optimal control". In: *IEEE Transactions on Control Systems Technology* 21.6 (2013), pp. 2432–2442.

[125] N. Parikh and S. Boyd. "Proximal algorithms". In: *Foundations and Trends in Optimization* 1.3 (2014), pp. 127–239.

[126] P. A. Parrilo. "Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization". PhD thesis. California Institute of Technology, 2000.

[127] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17.

[128] A. Pothen and C. Sun. "Compact clique tree data structures in sparse matrix factorizations". In: *Large-Scale Numerical Optimization* (1990), pp. 180–204.

[129] F. A. Potra and H. Engler. "A characterization of the behavior of the Anderson acceleration on linear problems". In: *linear Algebra and its Applications* 438.3 (2013), pp. 1002–1011.

[130] M. J. D. Powell. "A method for nonlinear constraints in minimization problems". In: *Optimization* (1969), pp. 283–298.

[131] M. J. D. Powell. "A hybrid method for nonlinear equations". In: *Numerical methods for nonlinear algebraic equations* (1970).

[132] P. Pulay. "Improved SCF convergence acceleration". In: *Journal of Computational Chemistry* 3.4 (1982), pp. 556–560.

[133] P. Pulay. "Convergence acceleration of iterative sequences. The case of SCF iteration". In: *Chemical Physics Letters* 73.2 (1980), pp. 393–398.

[134] A. Raghunathan, J. Steinhardt, and P. S. Liang. "Semidefinite relaxations for certifying robustness to adversarial examples". In: *Advances in Neural Information Processing Systems*. 2018, pp. 10877–10887.

[135] S. S. Rao et al. "A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping". In: *Cell* 159.7 (2014), pp. 1665–1680.

[136] J. K. Reid and J. A. Scott. "An out-of-core sparse Cholesky solver". In: *ACM Transactions on Mathematical Software (TOMS)* 36.2 (2009), p. 9.

[137] R. T. Rockafellar. *Convex analysis*. Princeton University Press, 1970.

[138] R. T. Rockafellar. "Monotone operators and the proximal point algorithm". In: *SIAM Journal on Control and Optimization* 14.5 (1976), pp. 877–898.

[139] T. Rohwedder and R. Schneider. "An analysis for the DIIS acceleration method used in quantum chemistry calculations". In: *Journal of Mathematical Chemistry* 49.9 (2011), p. 1889.

[140] N. Rontsis and P. Goulart. "Optimal Approximation of Doubly Stochastic Matrices". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2020, pp. 3589–3598.

[141] N. Rontsis, P. Goulart, and Y. Nakatsukasa. "Efficient Semidefinite Programming with approximate ADMM". In: *arXiv preprint arXiv:1912.02767* (2019).

[142] D. J. Rose, R. E. Tarjan, and G. S. Lueker. "Algorithmic aspects of vertex elimination on graphs". In: *SIAM Journal on Computing* 5.2 (1976), pp. 266–283.

[143] D. Ruiz. *A scaling algorithm to equilibrate both rows and columns norms in matrices*. Tech. rep. Rutherford Appleton Laboratory, 2001.

[144] E. Ryu and S. Boyd. "Primer on monotone operator methods". In: *Applied and Computational Mathematics* 15.1 (2016), pp. 3–43.

[145] O. Schenk et al. *PARDISO solver project*. http://www.pardiso-project.org. 2010.

[146] M. Schubiger, G. Banjac, and J. Lygeros. "GPU acceleration of ADMM for large-scale quadratic programming". In: *Journal of Parallel and Distributed Computing* 144 (2020), pp. 55–67.

[147] A. Sootla, Y. Zheng, and A. Papachristodoulou. "Block factor-width-two matrices in semidefinite programming". In: *18th European Control Conference (ECC)*. IEEE. 2019, pp. 1981–1986.

[148] B. Stellato et al. "OSQP: An Operator Splitting Solver for Quadratic Programs". In: *Mathematical Programming Computation* 12.4 (Oct. 2020), pp. 637–672.

[149] J. 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.

[150] Y. Sun, M. S. Andersen, and L. Vandenberghe. "Decomposition in conic optimization with partially separable structure". In: *SIAM Journal on Optimization* 24.2 (2014), pp. 873–897.

[151] R. E. Tarjan. "Maximum cardinality search and chordal graphs". In: *Unpublished Lecture Notes CS* 259 (1976).

[152] A. Themelis and P. Patrinos. "SuperMann: a superlinearly convergent algorithm for finding fixed points of nonexpansive operators". In: *IEEE Transactions on Automatic Control* 64.12 (2019), pp. 4875–4890.

[153] A. M. Tillmann and M. E. Pfetsch. "The computational complexity of the restricted isometry property, the nullspace property, and related concepts in compressed sensing". In: *IEEE Transactions on Information Theory* 60.2 (2013), pp. 1248–1259.

[154] K. Toh, M. J. Todd, and R. H. Tütüncü. "SDPT3 - A MATLAB software package for semidefinite programming, version 1.3". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 545–581.

[155] A. Toth and C. T. Kelley. "Convergence analysis for Anderson acceleration". In: *SIAM Journal on Numerical Analysis* 53.2 (2015), pp. 805–819.

[156] J. N. Tsitsiklis. *Problems in decentralized decision making and computation.* Tech. rep. Massachusetts Institute of Technology, 1984.

[157] J. Tsitsiklis, D. Bertsekas, and M. Athans. "Distributed asynchronous deterministic and stochastic gradient optimization algorithms". In: *IEEE Transactions on Automatic Control* 31.9 (1986), pp. 803–812.

[158] M. Udell et al. "Convex Optimization in Julia". In: *SC14 Workshop on High Performance Technical Computing in Dynamic Languages* (2014).

[159] L. Vandenberghe and M. S. Andersen. "Chordal graphs and semidefinite optimization". In: *Foundations and Trends in Optimization* 1.4 (2015), pp. 241–433.

[160] L. Vandenberghe. *Lecture notes: Optimization methods for large-scale systems (EE236C).* https://www.seas.ucla.edu/~vandenbe/ee236c.html. 2020.

[161] L. Vandenberghe and V. Balakrishnan. "Algorithms and software for LMI problems in control". In: *IEEE Control Systems Magazine* 17.5 (1997), pp. 89–95.

[162] R. J. Vanderbei. "Symmetric Quasidefinite Matrices". In: *SIAM Journal on Optimization* 5.1 (1995), pp. 100–113.

[163] D. Vanderbilt and S. G. Louie. "Total energies of diamond (111) surface reconstructions by a linear combination of atomic orbitals method". In: *Physical Review B* 30.10 (1984), p. 6118.

[164] J. P. Vielma et al. "Extended formulations in mixed integer conic quadratic programming". In: *Mathematical Programming Computation* 9.3 (2017), pp. 369–418.

[165] S. Vigerske. *MINLPLIB2 library - A Library of Mixed-Integer and Continuous Nonlinear Programming Instances.* 2001. URL: https://www.minlplib.org/.

[166] H. F. Walker. *Anderson acceleration: Algorithms and implementations.* Tech. rep. MS-6-15-50. Worcester Polytechnic Institute, Mathematical Sciences Department, 2011.

[167] H. F. Walker and P. Ni. "Anderson acceleration for fixed-point iterations". In: *SIAM Journal on Numerical Analysis* 49.4 (2011), pp. 1715–1735.

[168] P. Wolfe. "The simplex method for quadratic programming". In: *Econometrica: Journal of the Econometric Society* (1959), pp. 382–398.

[169] H. Wolkowicz, R. Saigal, and L. Vandenberghe. *Handbook of Semidefinite Programming: Theory, Algorithms, and Applications.* Vol. 27. Springer Science & Business Media, 2012.

[170] H. Wolkowicz and Q. Zhao. "Semidefinite programming relaxations for the graph partitioning problem". In: *Discrete Applied Mathematics* 96 (1999), pp. 461–479.

[171] M. Wright. "The interior-point revolution in optimization: history, recent developments, and lasting consequences". In: *Bulletin of the American Mathematical Society* 42.1 (2005), pp. 39–56.

[172] S. J. Wright. *Primal-dual interior-point methods*. Vol. 54. SIAM, 1997.

[173] S. Wright and J. Nocedal. "Numerical optimization". In: *Springer Science* 35 (1999).

[174] M. Yannakakis. "Computing the minimum fill-in is NP-complete". In: *SIAM Journal on Algebraic Discrete Methods* 2.1 (1981), pp. 77–79.

[175] A. Yurtsever et al. "Scalable semidefinite programming". In: *SIAM Journal on Mathematics of Data Science* 3.1 (2021), pp. 171–200.

[176] R. Zass and A. Shashua. "Doubly stochastic normalization for spectral clustering". In: *Advances in Neural Information Processing Systems*. 2007, pp. 1569–1576.

[177] J. Zhang, B. O'Donoghue, and S. Boyd. "Globally convergent type-I Anderson acceleration for nonsmooth fixed-point iterations". In: *SIAM Journal on Optimization* 30.4 (2020), pp. 3170–3197.

[178] X. Zhao, D. Sun, and K. Toh. "A Newton-CG augmented Lagrangian method for semidefinite programming". In: *SIAM Journal on Optimization* 20.4 (2010), pp. 1737–1765.

[179] T. Zheng, J. Guthrie, and E. Mallada. "Tight Inner Approximations of the Positive-Semidefinite Cone via Grassmannian Packing". In: *arXiv preprint: 2105.12021* (2021).

[180] Y. Zheng. "Chordal sparsity in control and optimization of large-scale systems". PhD thesis. University of Oxford, 2019.

[181] Y. Zheng et al. "Chordal decomposition in operator-splitting methods for sparse semidefinite programs". In: *Mathematical Programming* 180.1 (2020), pp. 489–532.