
Chordal decomposition in operator-splitting methods for sparse semidefinite programs

Yang Zheng¹ · Giovanni Fantuzzi² ·
Antonis Papachristodoulou¹ · Paul Goulart¹ ·
Andrew Wynn²

Received: date / Accepted: date

Abstract We employ chordal decomposition to reformulate a large and sparse semidefinite program (SDP), either in primal or dual standard form, into an equivalent SDP with smaller positive semidefinite (PSD) constraints. In contrast to previous approaches, the decomposed SDP is suitable for the application of first-order operator-splitting methods, enabling the development of efficient and scalable algorithms. In particular, we apply the alternating direction method of multipliers (ADMM) to solve decomposed primal- and dual-standard-form SDPs. Each iteration of such ADMM algorithms requires a projection onto an affine subspace, and a set of projections onto small PSD cones that can be computed in parallel. We also formulate the homogeneous self-dual embedding (HSDE) of a primal-dual pair of decomposed SDPs, and extend a recent ADMM-based algorithm to exploit the structure of our HSDE. The resulting HSDE algorithm has the same leading-order computational cost as those for the primal or dual problems only, with the advantage of being able to identify infeasible problems and produce an infeasibility certificate. All algorithms are implemented in the open-source MATLAB solver CDCS. Numerical experiments on a range of large-scale SDPs demonstrate the computational advantages of the proposed methods compared to common state-of-the-art solvers.

YZ and GF contributed equally. A preliminary version of part of this work appeared in [51, 52]. YZ is supported by Clarendon Scholarship and Jason Hu Scholarship. GF was supported by EPSRC grant EP/J010537/1 and by an EPSRC Doctoral Prize Fellowship. AP was supported in part by EPSRC Grant EP/J010537/1 and EP/M002454/1.

Y. Zheng (✉)
Tel.: +44-07511784230
E-mail: yang.zheng@eng.ox.ac.uk

G. Fantuzzi
E-mail: giovanni.fantuzzi10@imperial.ac.uk

A. Papachristodoulou
E-mail: antonis@eng.ox.ac.uk

P. Goulart
E-mail: paul.goulart@eng.ox.ac.uk

A. Wynn
E-mail: a.wynn@imperial.ac.uk

¹ Department of Engineering Science, University of Oxford, Parks Road, Oxford, OX1 3PJ, U.K.

² Department of Aeronautics, Imperial College London, South Kensington Campus, SW7 2AZ, U.K.

Keywords sparse SDPs · chordal decomposition · operator-splitting · first-order methods

Mathematics Subject Classification (2000) 90C06 · 90C22 · 90C25 · 49M27 · 49M29

1 Introduction

Semidefinite programs (SDPs) are convex optimization problems over the cone of positive semidefinite (PSD) matrices. Given $b \in \mathbb{R}^m$, $C \in \mathbb{S}^n$, and matrices $A_1, \dots, A_m \in \mathbb{S}^n$, the standard *primal form* of an SDP is

$$\begin{aligned} \min_X \quad & \langle C, X \rangle \\ \text{subject to} \quad & \langle A_i, X \rangle = b_i, \quad i = 1, \dots, m, \\ & X \in \mathbb{S}_+^n, \end{aligned} \quad (1)$$

while the standard *dual form* is

$$\begin{aligned} \max_{y, Z} \quad & \langle b, y \rangle \\ \text{subject to} \quad & Z + \sum_{i=1}^m A_i y_i = C, \\ & Z \in \mathbb{S}_+^n. \end{aligned} \quad (2)$$

In the above and throughout this work, \mathbb{R}^m is the usual m -dimensional Euclidean space, \mathbb{S}^n is the space of $n \times n$ symmetric matrices, \mathbb{S}_+^n is the cone of PSD matrices, and $\langle \cdot, \cdot \rangle$ denotes the inner product in the appropriate space, *i.e.*, $\langle x, y \rangle = x^T y$ for $x, y \in \mathbb{R}^m$ and $\langle X, Y \rangle = \text{trace}(XY)$ for $X, Y \in \mathbb{S}^n$. SDPs have found applications in a wide range of fields, such as control theory, machine learning, combinatorics, and operations research [8]. Semidefinite programming encompasses other common types of optimization problems, including linear, quadratic, and second-order cone programs [10]. Furthermore, many nonlinear convex constraints admit SDP relaxations that work well in practice [43].

It is well-known that small and medium-sized SDPs can be solved up to any arbitrary precision in polynomial time [43] using efficient second-order interior-point methods (IPMs) [2, 24]. However, many problems of practical interest are too large to be addressed by the current state-of-the-art interior-point algorithms, largely due to the need to compute, store, and factorize an $m \times m$ matrix at each iteration.

A common strategy to address this shortcoming is to abandon IPMs in favour of simpler first-order methods (FOMs), at the expense of reducing the accuracy of the solution. For instance, Malick *et al.* introduced regularization methods to solve SDPs based on a dual augmented Lagrangian [31]. Wen *et al.* proposed an alternating direction augmented Lagrangian method for large-scale SDPs in the dual standard form [44]. Zhao *et al.* presented an augmented Lagrangian dual approach combined with the conjugate gradient method to solve large-scale SDPs [49]. More recently, O'Donoghue *et al.* developed a first-order operator-splitting method to solve the homogeneous self-dual embedding (HSDE) of a primal-dual pair of conic programs [32]. The algorithm, implemented in the C package SCS [33], has the advantage of providing certificates of primal or dual infeasibility.

A second major approach to resolve the aforementioned scalability issues is based on the observation that the large-scale SDPs encountered in applications are often structured and/or sparse [8]. Exploiting sparsity in SDPs is an active and challenging area of research [3],

with one main difficulty being that the optimal (primal) solution is typically dense even when the problem data are sparse. Nonetheless, if the *aggregate sparsity pattern* of the data is *chordal* (or has sparse *chordal extensions*), one can replace the original, large PSD constraint with a set of PSD constraints on smaller matrices, coupled by additional equality constraints [1,22,23,25]. Having reduced the size of the semidefinite variables, the converted SDP can in some cases be solved more efficiently than the original problem using standard IPMs. These ideas underly the *domain-space* and the *range-space* conversion techniques in [17,27], implemented in the MATLAB package SparseCoLO [16].

The problem with such decomposition techniques, however, is that the addition of equality constraints to an SDP often offsets the benefit of working with smaller semidefinite cones. One possible solution is to exploit the properties of chordal sparsity patterns directly in the IPMs: Fukuda *et al.* used a positive definite completion theorem [23] to develop a primal-dual path-following method [17]; Burer proposed a nonsymmetric primal-dual IPM using Cholesky factors of the dual variable Z and maximum determinant completion of the primal variable X [11]; and Andersen *et al.* developed fast recursive algorithms to evaluate the function values and derivatives of the barrier functions for SDPs with chordal sparsity [4]. Another attractive option is to solve the sparse SDP using FOMs: Sun *et al.* proposed a first-order splitting algorithm for partially decomposable conic programs, including SDPs with chordal sparsity [38]; Kalbat & Lavaei applied a first-order operator-splitting method to solve a special class of SDPs with fully decomposable constraints [26]; Madani *et al.* developed a highly-parallelizable first-order algorithm for sparse SDPs with inequality constraints, with applications to optimal power flow problems [30]; Dall’Anese *et al.* exploited chordal sparsity to solve SDPs with separable constraints using a distributed FOM [12]; finally, Sun and Vandenberghe introduced several proximal splitting and decomposition algorithms for sparse matrix nearness problems involving no explicit equality constraints [39].

In this work, we embrace the spirit of [12,26,30,32,38,39] and exploit sparsity in SDPs using a first-order operator-splitting method known as the *alternating direction method of multipliers* (ADMM). Introduced in the mid-1970s [18,20], ADMM is related to other FOMs such as dual decomposition and the method of multipliers, and it has recently found applications in many areas, including covariance selection, signal processing, resource allocation, and classification; see [9] for a review. In contrast to the approach in [38], which requires the solution of a quadratic SDP at each iteration, our approach relies entirely on first-order methods. Moreover, our ADMM-based algorithm works for generic SDPs with chordal sparsity and has the ability to detect infeasibility, which are key advantages compared to the algorithms in [12,26,30,39]. More precisely, our contributions are:

1. We apply two chordal decomposition theorems [1,23] to formulate *domain-space* and *range-space* conversion frameworks for the application of FOMs to standard-form SDPs with chordal sparsity. These are analogous to the conversion methods developed in [17,27] for IPMs, but we introduce two sets of slack variables that allow for the separation of the conic and the affine constraints when using operator-splitting algorithms. To the best of our knowledge, this extension has never been presented before, and its significant potential is demonstrated in this work.
2. We apply ADMM to solve the domain- and range-space converted SDPs, and show that the resulting iterates of the ADMM algorithms are the same up to scaling. The iterations are computationally inexpensive: the positive semidefinite (PSD) constraint is enforced via parallel projections onto small PSD cones—a much more economical strategy than that in [38]—while imposing the affine constraints requires solving a linear system with constant coefficient matrix, the factorization/inverse of which can be cached before it-

erating the algorithm. Note that the idea of enforcing a large sparse PSD constraint by projection onto multiple smaller ones has also been exploited in [12, 30] in the special context of optimal power flow problems and in [39] for matrix nearness problems.

3. We formulate the HSDE of a converted primal-dual pair of sparse SDPs. In contrast to [12, 26, 30, 38], this allows us to compute either primal and dual optimal points, or a certificate of infeasibility. We then extend the algorithm proposed in [32], showing that the structure of our HSDE can be exploited to solve a large linear system of equations extremely efficiently through a sequence of block eliminations. As a result, we obtain an algorithm that is more efficient than the method of [32], irrespectively of whether this is used on the original primal-dual pair of SDPs (before decomposition) or on the converted problems. In the former case, the advantage comes from the application of chordal decomposition to replace a large PSD cone with a set of smaller ones. In the latter case, efficiency is gained by the proposed sequence of block eliminations.
4. We present the MATLAB solver CDCS (Cone Decomposition Conic Solver), which implements our ADMM algorithms. CDCS is the first open-source first-order solver that exploits chordal decomposition and can detect infeasible problems. We test our implementation on large-scale sparse problems in SDPLIB [7], selected sparse SDPs with nonchordal sparsity pattern [4], and randomly generated SDPs with block-arrow sparsity patterns [38]. The results demonstrate the efficiency of our algorithms compared to the interior-point solvers SeDuMi [37] and the first-order solver SCS [33].

The rest of the paper is organized as follows. Section 2 reviews chordal decomposition and the basic ADMM algorithm. Section 3 introduces our conversion framework for sparse SDPs based on chordal decomposition. We show how to apply the ADMM to exploit domain-space and range-space sparsity in primal and dual SDPs in Section 4. Section 5 discusses the ADMM algorithm for the HSDE of SDPs with chordal sparsity. The computational complexity of our algorithms in terms of floating-point operations is discussed in Section 6. CDCS and our numerical experiments are presented in Section 7. Section 8 concludes the paper.

2 Preliminaries

2.1 A review of graph theoretic notions

We start by briefly reviewing some key graph theoretic concepts (see [6,21] for more details). A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is defined by a set of vertices $\mathcal{V} = \{1, 2, \dots, n\}$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is called *complete* if any two nodes are connected by an edge. A subset of vertices $\mathcal{C} \subseteq \mathcal{V}$ such that $(i, j) \in \mathcal{E}$ for any distinct vertices $i, j \in \mathcal{C}$, *i.e.*, such that the subgraph induced by \mathcal{C} is complete, is called a *clique*. The number of vertices in \mathcal{C} is denoted by $|\mathcal{C}|$. If \mathcal{C} is not a subset of any other clique, then it is referred to as a *maximal clique*. A *cycle* of length k in a graph \mathcal{G} is a set of pairwise distinct nodes $\{v_1, v_2, \dots, v_k\} \subset \mathcal{V}$ such that $(v_k, v_1) \in \mathcal{E}$ and $(v_i, v_{i+1}) \in \mathcal{E}$ for $i = 1, \dots, k - 1$. A *chord* is an edge joining two non-adjacent nodes in a cycle. A graph \mathcal{G} is undirected if $(v_i, v_j) \in \mathcal{E} \Leftrightarrow (v_j, v_i) \in \mathcal{E}$.

An undirected graph \mathcal{G} is called *chordal* (or *triangulated*, or a *rigid circuit* [42]) if every cycle of length greater than or equal to four has at least one chord. Chordal graphs include several other classes of graphs, such as acyclic undirected graphs (including trees) and complete graphs. Algorithms such as the maximum cardinality search [40] can test chordality



Fig. 1 (a) Nonchordal graph: the cycle (1-2-3-4) is of length four but has no chords. (b) Chordal graph: all cycles of length no less than four have a chord; the maximal cliques are $\mathcal{C}_1 = \{1, 2, 4\}$ and $\mathcal{C}_2 = \{2, 3, 4\}$.

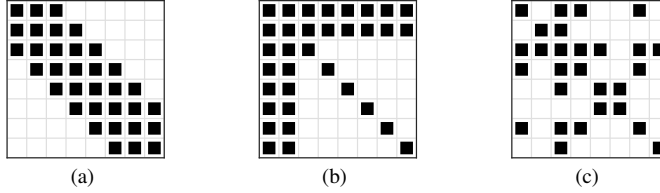


Fig. 2 Sparsity patterns of 8×8 matrices: (a) banded sparsity pattern; (b) “block-arrow” sparsity pattern; (c) a generic sparsity pattern.

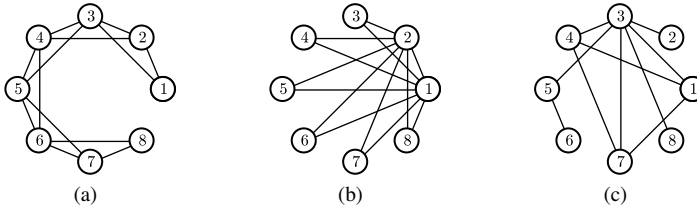


Fig. 3 Graph representation of the matrix sparsity patterns illustrated in Fig. 2(a)–(c), respectively.

and identify the maximal cliques of a chordal graph efficiently, *i.e.*, in linear time in terms of the number of nodes and edges. Non-chordal graphs can always be *chordal extended*, *i.e.*, extended to a chordal graph, by adding additional edges to the original graph. Computing the chordal extension with the minimum number of additional edges is an NP-complete problem [46], but several heuristics exist to find good chordal extensions efficiently [42].

Fig. 1 illustrates these concepts. The graph in Fig. 1(a) is not chordal, but can be chordal extended to the graph in Fig. 1(b) by adding the edge (2, 4). The chordal graph in Fig. 1(b) has two maximal cliques, $\mathcal{C}_1 = \{1, 2, 4\}$ and $\mathcal{C}_2 = \{2, 3, 4\}$. Other examples of chordal graphs are given in Fig. 3.

2.2 Sparse matrix cones and chordal decomposition

The sparsity pattern of a symmetric matrix $X \in \mathbb{S}^n$ can be represented by an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and vice-versa. For example, the sparsity patterns illustrated in Fig. 2 correspond to the graphs in Fig. 3. With a slight abuse of terminology, we refer to the graph \mathcal{G} as the sparsity pattern of X . Given a clique \mathcal{C}_k of \mathcal{G} , we define a matrix $E_{\mathcal{C}_k} \in \mathbb{R}^{|\mathcal{C}_k| \times n}$ as

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

where $C_k(i)$ is the i -th vertex in C_k , sorted in the natural ordering. Given $X \in \mathbb{S}^n$, the matrix E_{C_k} can be used to select a principal sub-matrix defined by the clique C_k , *i.e.*, $E_{C_k} X E_{C_k}^T \in \mathbb{S}^{|C_k|}$. In addition, the operation $E_{C_k}^T Y E_{C_k}$ creates an $n \times n$ symmetric matrix from a $|C_k| \times |C_k|$ matrix. For example, the chordal graph in Fig. 1(b) has a maximal clique $C_1 = \{1, 2, 4\}$, and for $X \in \mathbb{S}^4$ and $Y \in \mathbb{S}^3$ we have

$$E_{C_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad E_{C_1} X E_{C_1}^T = \begin{bmatrix} X_{11} & X_{12} & X_{14} \\ X_{21} & X_{22} & X_{24} \\ X_{41} & X_{42} & X_{44} \end{bmatrix}, \quad E_{C_1}^T Y E_{C_1} = \begin{bmatrix} Y_{11} & Y_{12} & 0 & Y_{13} \\ Y_{21} & Y_{22} & 0 & Y_{23} \\ 0 & 0 & 0 & 0 \\ Y_{31} & Y_{32} & 0 & Y_{33} \end{bmatrix}.$$

Given an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, let $\mathcal{E}^* = \mathcal{E} \cup \{(i, i), i \in \mathcal{V}\}$ be a set of edges that includes all self-loops. We define the space of sparse symmetric matrices with sparsity pattern \mathcal{G} as

$$\mathbb{S}^n(\mathcal{E}, 0) := \{X \in \mathbb{S}^n : X_{ij} = X_{ji} = 0 \text{ if } (i, j) \notin \mathcal{E}^*\},$$

and the cone of sparse PSD matrices as

$$\mathbb{S}_+^n(\mathcal{E}, 0) := \{X \in \mathbb{S}^n(\mathcal{E}, 0) : X \succeq 0\},$$

where the notation $X \succeq 0$ indicates that X is PSD. Moreover, we consider the cone

$$\mathbb{S}_+^n(\mathcal{E}, ?) := \mathbb{P}_{\mathbb{S}^n(\mathcal{E}, 0)}(\mathbb{S}_+^n)$$

given by the projection of the PSD cone onto the space of sparse matrices $\mathbb{S}^n(\mathcal{E}, 0)$ with respect to the usual Frobenius matrix norm (this is the norm induced by the usual trace inner product on the space of symmetric matrices). It is not difficult to see that $X \in \mathbb{S}_+^n(\mathcal{E}, ?)$ if and only if it has a positive semidefinite completion, *i.e.*, if there exists a PSD matrix M such that $M_{ij} = X_{ij}$ when $(i, j) \in \mathcal{E}^*$.

For any undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the cones $\mathbb{S}_+^n(\mathcal{E}, ?)$ and $\mathbb{S}_+^n(\mathcal{E}, 0)$ are dual to each other with respect to the trace inner product in the space of sparse matrices $\mathbb{S}^n(\mathcal{E}, 0)$ [42]. In other words,

$$\begin{aligned} \mathbb{S}_+^n(\mathcal{E}, ?) &\equiv \{X \in \mathbb{S}^n(\mathcal{E}, 0) : \langle X, Z \rangle \geq 0, \forall Z \in \mathbb{S}_+^n(\mathcal{E}, 0)\}, \\ \mathbb{S}_+^n(\mathcal{E}, 0) &\equiv \{Z \in \mathbb{S}^n(\mathcal{E}, 0) : \langle Z, X \rangle \geq 0, \forall X \in \mathbb{S}_+^n(\mathcal{E}, ?)\}. \end{aligned}$$

If \mathcal{G} is chordal, then $\mathbb{S}_+^n(\mathcal{E}, ?)$ and $\mathbb{S}_+^n(\mathcal{E}, 0)$ can be equivalently decomposed into a set of smaller but coupled convex cones according to the following theorems.

Theorem 1 ([23, theorem 7]) *Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a chordal graph and let $\{C_1, C_2, \dots, C_p\}$ be the set of its maximal cliques. Then, $X \in \mathbb{S}_+^n(\mathcal{E}, ?)$ if and only if*

$$E_{C_k} X E_{C_k}^T \in \mathbb{S}_+^{|C_k|}, \quad k = 1, \dots, p.$$

Theorem 2 ([1, theorem 2.3], [22, theorem 4], [25, theorem 1]) *Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a chordal graph and let $\{C_1, C_2, \dots, C_p\}$ be the set of its maximal cliques. Then, $Z \in \mathbb{S}_+^n(\mathcal{E}, 0)$ if and only if there exist matrices $Z_k \in \mathbb{S}_+^{|C_k|}$ for $k = 1, \dots, p$ such that*

$$Z = \sum_{k=1}^p E_{C_k}^T Z_k E_{C_k}.$$

Note that these results can be proven individually, but can also be derived from each other using the duality of the cones $\mathbb{S}_+^n(\mathcal{E}, ?)$ and $\mathbb{S}_+^n(\mathcal{E}, 0)$ [27]. In this paper, the terminology *chordal* (or *clique*) *decomposition of a sparse matrix cone* will refer to the application of Theorem 1 or Theorem 2 to replace a large sparse PSD cone with a set of smaller but coupled PSD cones. Chordal decomposition of sparse matrix cones underpins much of the recent research on sparse SDPs [4, 17, 27, 30, 38, 42], most of which relies on the conversion framework for IPMs proposed in [17, 27].

To illustrate the concept, consider the chordal graph in Fig. 1(b). By Theorem 1,

$$\begin{bmatrix} X_{11} & X_{12} & 0 & X_{14} \\ X_{12} & X_{22} & X_{23} & X_{24} \\ 0 & X_{23} & X_{33} & X_{34} \\ X_{14} & X_{24} & X_{34} & X_{44} \end{bmatrix} \in \mathbb{S}_+^n(\mathcal{E}, ?) \Leftrightarrow \begin{bmatrix} X_{11} & X_{12} & X_{14} \\ X_{12} & X_{22} & X_{24} \\ X_{14} & X_{24} & X_{44} \end{bmatrix} \succeq 0, \quad \begin{bmatrix} X_{22} & X_{23} & X_{24} \\ X_{23} & X_{33} & X_{34} \\ X_{24} & X_{34} & X_{44} \end{bmatrix} \succeq 0.$$

Similarly, Theorem 2 guarantees that (after eliminating some of the variables)

$$\begin{bmatrix} Z_{11} & Z_{12} & 0 & Z_{14} \\ Z_{12} & Z_{22} & Z_{23} & Z_{24} \\ 0 & Z_{23} & Z_{33} & Z_{34} \\ Z_{14} & Z_{24} & Z_{34} & Z_{44} \end{bmatrix} \in \mathbb{S}_+^n(\mathcal{E}, 0) \Leftrightarrow \begin{cases} \begin{bmatrix} Z_{11} & Z_{12} & Z_{14} \\ Z_{12} & a_1 & a_3 \\ Z_{14} & a_3 & a_2 \end{bmatrix} \succeq 0, & \begin{bmatrix} b_1 & Z_{23} & b_3 \\ Z_{23} & Z_{33} & Z_{34} \\ b_3 & Z_{34} & b_2 \end{bmatrix} \succeq 0, \\ a_i + b_i = Z_{ii}, i \in \{1, 2\}, \\ a_3 + b_3 = Z_{24} \end{cases}$$

for some constants a_1, a_2, a_3 and b_1, b_2, b_3 . Note that the PSD constraints obtained after the chordal decomposition of X (resp. Z) are coupled via the elements X_{22}, X_{44} , and $X_{24} = X_{42}$ (resp. Z_{22}, Z_{44} , and $Z_{24} = Z_{42}$).

2.3 The Alternating Direction Method of Multipliers

The computational “engine” employed in this work is the alternating direction method of multipliers (ADMM). ADMM is an operator-splitting method developed in the 1970s, and it is known to be equivalent to other operator-splitting methods such as Douglas-Rachford splitting and Spingarn’s method of partial inverses; see [9] for a review. The ADMM algorithm solves the optimization problem

$$\begin{aligned} \min_{x,y} \quad & f(x) + g(y) \\ \text{subject to} \quad & Ax + By = c, \end{aligned} \quad (3)$$

where f and g are convex functions, $x \in \mathbb{R}^{n_x}, y \in \mathbb{R}^{n_y}, A \in \mathbb{R}^{n_c \times n_x}, B \in \mathbb{R}^{n_c \times n_y}$ and $c \in \mathbb{R}^{n_c}$. Given a penalty parameter $\rho > 0$ and a dual multiplier $z \in \mathbb{R}^{n_c}$, the ADMM algorithm finds a saddle point of the augmented Lagrangian

$$\mathcal{L}_\rho(x, y, z) := f(x) + g(y) + z^T (Ax + By - c) + \frac{\rho}{2} \|Ax + By - c\|^2$$

by minimizing \mathcal{L} with respect to the primal variables x and y separately, followed by a dual variable update:

$$x^{(n+1)} = \arg \min_x \mathcal{L}_\rho(x, y^{(n)}, z^{(n)}), \quad (4a)$$

$$y^{(n+1)} = \arg \min_y \mathcal{L}_\rho(x^{(n+1)}, y, z^{(n)}), \quad (4b)$$

$$z^{(n+1)} = z^{(n)} + \rho (Ax^{(n+1)} + By^{(n+1)} - c). \quad (4c)$$

The superscript (n) indicates that a variable is fixed to its value at the n -th iteration. Note that since z is fixed in (4a) and (4b), one may equivalently minimize the modified Lagrangian

$$\hat{\mathcal{L}}_\rho(x, y, z) := f(x) + g(y) + \frac{\rho}{2} \left\| Ax + By - c + \frac{1}{\rho} z \right\|^2.$$

Under very mild conditions, the ADMM converges to a solution of (3) with a rate $\mathcal{O}(\frac{1}{n})$ [9, Section 3.2]. ADMM is particularly suitable when (4a) and (4b) have closed-form expressions, or can be solved efficiently. Moreover, splitting the minimization over x and y often allows distributed and/or parallel implementations of steps (4a)–(4c).

3 Chordal decomposition of sparse SDPs

The sparsity pattern of the problem data for the primal-dual pair of standard-form SDPs (1)–(2) can be described using the so-called *aggregate sparsity pattern*. We say that the pair of SDPs (1)–(2) has an aggregate sparsity pattern $\mathcal{G}(\mathcal{V}, \mathcal{E})$ if

$$C \in \mathbb{S}^n(\mathcal{E}, 0) \quad \text{and} \quad A_i \in \mathbb{S}^n(\mathcal{E}, 0), \quad i = 1, \dots, m. \quad (5)$$

In other words, the aggregate sparsity pattern \mathcal{G} is the union of the individual sparsity patterns of the data matrices C, A_1, \dots, A_m . Throughout the rest of this paper, we assume that the aggregate sparsity pattern \mathcal{G} is chordal (or that a suitable chordal extension has been found), and that it has p maximal cliques $\mathcal{C}_1, \dots, \mathcal{C}_p$. In addition, we assume that the matrices A_1, \dots, A_m are linearly independent.

It is not difficult to see that the aggregate sparsity pattern defines the sparsity pattern of any feasible dual variable Z in (2), *i.e.*, any dual feasible Z must have sparsity pattern \mathcal{G} . Similarly, while the primal variable X in (1) is usually dense, the value of the cost function and the equality constraints depend only on the entries X_{ij} with $(i, j) \in \mathcal{E}$, and the remaining entries simply guarantee that X is PSD. Recalling the definition of the sparse matrix cones $\mathbb{S}_+^n(\mathcal{E}, ?)$ and $\mathbb{S}_+^n(\mathcal{E}, 0)$, we can therefore recast the primal-form SDP (1) as

$$\begin{aligned} \min_X \quad & \langle C, X \rangle \\ \text{subject to} \quad & \langle A_i, X \rangle = b_i, \quad i = 1, \dots, m, \\ & X \in \mathbb{S}_+^n(\mathcal{E}, ?), \end{aligned} \quad (6)$$

and the dual-form SDP (2) as

$$\begin{aligned} \max_{y, Z} \quad & \langle b, y \rangle \\ \text{subject to} \quad & Z + \sum_{i=1}^m A_i y_i = C, \\ & Z \in \mathbb{S}_+^n(\mathcal{E}, 0). \end{aligned} \quad (7)$$

This formulation was first proposed by Fukuda *et al.* [17], and was later discussed in [4, 27, 38]. Note that (6) and (7) are a primal-dual pair of linear conic problems because the cones $\mathbb{S}_+^n(\mathcal{E}, ?)$ and $\mathbb{S}_+^n(\mathcal{E}, 0)$ are dual to each other.

3.1 Domain-space decomposition

As we have seen in Section 2, Theorem 1 allows us to decompose the sparse matrix cone constraint $X \in \mathbb{S}_+^n(\mathcal{E}, ?)$ into p standard PSD constraints on the submatrices of X defined by the cliques $\mathcal{C}_1, \dots, \mathcal{C}_p$. In other words,

$$X \in \mathbb{S}_+^n(\mathcal{E}, ?) \Leftrightarrow E_{\mathcal{C}_k} X E_{\mathcal{C}_k}^T \in \mathbb{S}_+^{|\mathcal{C}_k|}, \quad k = 1, \dots, p.$$

These p constraints are implicitly coupled since $E_{\mathcal{C}_i} X E_{\mathcal{C}_i}^T$ and $E_{\mathcal{C}_q} X E_{\mathcal{C}_q}^T$ have overlapping elements if $\mathcal{C}_i \cap \mathcal{C}_q \neq \emptyset$. Upon introducing slack variables $X_k, k = 1, \dots, p$, we can rewrite this as

$$X \in \mathbb{S}_+^n(\mathcal{E}, ?) \Leftrightarrow \begin{cases} X_k = E_{\mathcal{C}_k} X E_{\mathcal{C}_k}^T, & k = 1, \dots, p, \\ X_k \in \mathbb{S}_+^{|\mathcal{C}_k|}, & k = 1, \dots, p. \end{cases} \quad (8)$$

The primal optimization problem (6) is then equivalent to the SDP

$$\begin{aligned} & \min_{X, X_1, \dots, X_p} \langle C, X \rangle \\ & \text{subject to} \quad \langle A_i, X \rangle = b_i, \quad i = 1, \dots, m, \\ & \quad \quad \quad X_k = E_{\mathcal{C}_k} X E_{\mathcal{C}_k}^T, \quad k = 1, \dots, p, \\ & \quad \quad \quad X_k \in \mathbb{S}_+^{|\mathcal{C}_k|}, \quad k = 1, \dots, p. \end{aligned} \quad (9)$$

Adopting the same terminology used in [17], we refer to (9) as the *domain-space* decomposition of the primal-standard-form SDP (1).

Remark 1 The main difference between the conversion method proposed in this section and that in [17, 27] is that the large matrix X is not eliminated. Instead, in the domain-space decomposition of [17, 27], X is eliminated by replacing the constraints

$$X_k = E_{\mathcal{C}_k} X E_{\mathcal{C}_k}^T, \quad k = 1, \dots, p,$$

with the requirement that the entries of any two different sub-matrices X_j, X_k must match if they map to the same entry in X . Mathematically, this condition can be written as

$$E_{\mathcal{C}_j \cap \mathcal{C}_k} \left(E_{\mathcal{C}_k}^T X_k E_{\mathcal{C}_k} - E_{\mathcal{C}_j}^T X_j E_{\mathcal{C}_j} \right) E_{\mathcal{C}_j \cap \mathcal{C}_k}^T = 0, \quad \forall j, k \text{ such that } \mathcal{C}_j \cap \mathcal{C}_k \neq \emptyset. \quad (10)$$

Redundant constraints in (10) can be eliminated using the *running intersection property* of the cliques [6, 17], and the decomposed SDP can be solved efficiently by IPMs in certain cases [17, 27]. However, applying FOMs to (9) effectively after the elimination of X is not straightforward because the PSD matrix variables X_1, \dots, X_p are coupled via (10). In [38], for example, an SDP with a quadratic objective had to be solved at each iteration to impose the PSD constraints, requiring an additional iterative solver. Even when this problem is resolved, *e.g.*, by using the algorithm of [32], the size of the KKT system enforcing the affine constraints is increased dramatically by the consensus conditions (10), sometimes so much that memory requirements are prohibitive on desktop computing platforms [17]. In contrast, we show in Section 4 that if a set of slack variables X_k are introduced in (8) and X is not eliminated from (9), then the PSD constraint can be imposed via projections onto small PSD cones. At the same time, the affine constraints require the solution of an $m \times m$ linear system of equations, as if no consensus constraints were introduced. This makes our

conversion framework more suitable for FOMs than that of [17, 27], as all steps in many common operator-splitting algorithms have an efficiently computable explicit solution. Of course, the equalities $X_k = E_{C_k} X E_{C_k}^T$, $k = 1, \dots, p$ are satisfied only within moderate tolerances when FOMs are utilized, and the accumulation of small errors might make it more difficult to solve the original SDP to a given degree of accuracy compared to the methods in [17, 27, 38, 44]. Therefore, the trade-off between the gains in computational complexity and the reduction in accuracy should be carefully considered when choosing the most suitable approach to solve a given large-scale SDP. Nonetheless, our numerical experiments of Section 7 demonstrate that working with (9) is often a competitive strategy.

3.2 Range-space decomposition

A *range-space* decomposition of the dual-standard-form SDP (2) can be formulated by applying Theorem 2 to the sparse matrix cone constraint $Z \in \mathbb{S}_+^n(\mathcal{E}, 0)$ in (7):

$$Z \in \mathbb{S}_+^n(\mathcal{E}, 0) \Leftrightarrow Z = \sum_{k=1}^p E_{C_k}^T Z_k E_{C_k} \text{ for some } Z_k \in \mathbb{S}_+^{|C_k|}, k = 1, \dots, p.$$

We then introduce slack variables V_k , $k = 1, \dots, p$ and conclude that $Z \in \mathbb{S}_+^n(\mathcal{E}, 0)$ if and only if there exists matrices $Z_k, V_k \in \mathbb{S}^{|C_k|}$, $k = 1, \dots, p$, such that

$$Z = \sum_{k=1}^p E_{C_k}^T V_k E_{C_k}, \quad Z_k = V_k, \quad k = 1, \dots, p, \quad Z_k \in \mathbb{S}_+^{|C_k|}, \quad k = 1, \dots, p.$$

The range-space decomposition of (2) is then given by

$$\begin{aligned} & \max_{y, Z_1, \dots, Z_p, V_1, \dots, V_p} \langle b, y \rangle \\ & \text{subject to} \quad \sum_{i=1}^m A_i y_i + \sum_{k=1}^p E_{C_k}^T V_k E_{C_k} = C, \\ & \quad Z_k - V_k = 0, \quad k = 1, \dots, p, \\ & \quad Z_k \in \mathbb{S}_+^{|C_k|}, \quad k = 1, \dots, p. \end{aligned} \tag{11}$$

Similar comments as in Remark 1 hold: the slack variables V_1, \dots, V_p are essential to formulate a decomposition framework suitable for the application of FOMs, although their introduction might complicate solving (2) to a desired accuracy.

Remark 2 Although the domain- and range-space decompositions (9) and (11) have been derived individually, they are in fact a primal-dual pair of SDPs. The duality between the original SDPs (1) and (2) is inherited by the decomposed SDPs (9) and (11) by virtue of the duality between Theorem 1 and Theorem 2. This elegant picture is illustrated in Fig. 4.

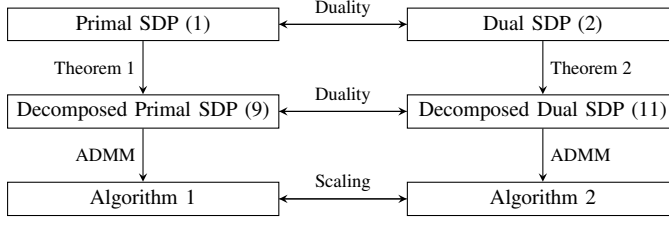


Fig. 4 Duality between the original primal and dual SDPs, and the decomposed primal and dual SDPs.

4 ADMM for domain- and range-space decompositions of sparse SDPs

In this section, we demonstrate how ADMM can be applied to solve the domain-space decomposition (9) and the range-space decomposition (11) efficiently. Furthermore, we show that the resulting domain- and range-space algorithms are equivalent, in the sense that one is just a scaled version of the other (cf. Fig. 4). Throughout this section, $\delta_{\mathcal{K}}(x)$ will denote the indicator function of a set \mathcal{K} , *i.e.*,

$$\delta_{\mathcal{K}}(x) = \begin{cases} 0, & \text{if } x \in \mathcal{K}, \\ +\infty, & \text{otherwise.} \end{cases}$$

For notational neatness, however, we write δ_0 when $\mathcal{K} \equiv \{0\}$.

To ease the exposition further, we consider the usual vectorized forms of (9) and (11). Specifically, we let $\text{vec} : \mathbb{S}^n \rightarrow \mathbb{R}^{n^2}$ be the usual operator mapping a matrix to the stack of its columns and define the vectorized data

$$c := \text{vec}(C), \quad A := [\text{vec}(A_0) \dots \text{vec}(A_m)]^T.$$

Note that the assumption that A_1, \dots, A_m are linearly independent matrices means that A has full row rank. For all $k = 1, \dots, p$, we also introduce the vectorized variables

$$x := \text{vec}(X), \quad x_k := \text{vec}(X_k), \quad z_k := \text{vec}(Z_k), \quad v_k := \text{vec}(V_k),$$

and define “entry-selector” matrices $H_k := E_{C_k} \otimes E_{C_k}$ for $k = 1, \dots, p$ that project x onto the subvectors x_1, \dots, x_p , *i.e.*, such that

$$x_k = \text{vec}(X_k) = \text{vec}(E_{C_k} X E_{C_k}^T) = H_k x.$$

Note that for each $k = 1, \dots, p$, the rows of H_k are orthonormal, and that the matrix $H_k^T H_k$ is diagonal. Upon defining

$$\mathcal{S}_k := \left\{ x \in \mathbb{R}^{|C_k|^2} : \text{vec}^{-1}(x) \in \mathbb{S}_+^{|C_k|} \right\},$$

such that $x_k \in \mathcal{S}_k$ if and only if $X_k \in \mathbb{S}_+^{|C_k|}$, we can rewrite (9) as

$$\begin{aligned} & \min_{x, x_1, \dots, x_p} \langle c, x \rangle \\ & \text{subject to} \quad Ax = b, \\ & \quad x_k = H_k x, \quad k = 1, \dots, p, \\ & \quad x_k \in \mathcal{S}_k, \quad k = 1, \dots, p, \end{aligned} \tag{12}$$

while (11) becomes

$$\begin{aligned}
& \max_{y, z_1, \dots, z_p, v_1, \dots, v_p} \langle b, y \rangle \\
& \text{subject to} \quad A^T y + \sum_{k=1}^p H_k^T v_k = c, \\
& \quad z_k - v_k = 0, \quad k = 1, \dots, p, \\
& \quad z_k \in \mathcal{S}_k, \quad k = 1, \dots, p.
\end{aligned} \tag{13}$$

4.1 ADMM for the domain-space decomposition

We start by moving the constraints $Ax = b$ and $x_k \in \mathcal{S}_k$ in (12) to the objective using the indicator functions $\delta_0(\cdot)$ and $\delta_{\mathcal{S}_k}(\cdot)$, respectively, *i.e.*, we write

$$\begin{aligned}
& \min_{x, x_1, \dots, x_p} \langle c, x \rangle + \delta_0(Ax - b) + \sum_{k=1}^p \delta_{\mathcal{S}_k}(x_k) \\
& \text{subject to} \quad x_k = H_k x, \quad k = 1, \dots, p.
\end{aligned} \tag{14}$$

This problem is in the standard form for the application of ADMM. Given a penalty parameter $\rho > 0$ and a Lagrange multiplier λ_k for each constraint $x_k = H_k x$, $k = 1, \dots, p$, we consider the (modified) augmented Lagrangian

$$\begin{aligned}
\mathcal{L}(x, x_1, \dots, x_p, \lambda_1, \dots, \lambda_p) := & \langle c, x \rangle + \delta_0(Ax - b) \\
& + \sum_{k=1}^p \left[\delta_{\mathcal{S}_k}(x_k) + \frac{\rho}{2} \left\| x_k - H_k x + \frac{1}{\rho} \lambda_k \right\|^2 \right], \tag{15}
\end{aligned}$$

and group the variables as $\mathcal{X} := \{x\}$, $\mathcal{Y} := \{x_1, \dots, x_p\}$, and $\mathcal{Z} := \{\lambda_1, \dots, \lambda_p\}$. According to (4), each iteration of the ADMM requires the minimization of the Lagrangian in (15) with respect to the \mathcal{X} - and \mathcal{Y} -blocks separately, followed by an update of the multipliers \mathcal{Z} . At each step, the variables not being optimized over are fixed to their most current value. Note that splitting the primal variables x, x_1, \dots, x_p in the two blocks \mathcal{X} and \mathcal{Y} defined above is essential to solving the \mathcal{X} and \mathcal{Y} minimization sub-problems (4a) and (4b); more details will be given in Remark 3 after describing the \mathcal{Y} -minimization step in Section 4.1.2.

4.1.1 Minimization over \mathcal{X}

Minimizing the augmented Lagrangian (15) over \mathcal{X} is equivalent to the equality-constrained quadratic program

$$\begin{aligned}
& \min_x \langle c, x \rangle + \frac{\rho}{2} \sum_{k=1}^p \left\| x_k^{(n)} - H_k x + \frac{1}{\rho} \lambda_k^{(n)} \right\|^2 \\
& \text{subject to} \quad Ax = b.
\end{aligned} \tag{16}$$

Letting ρy be the multiplier for the equality constraint (we scale the multiplier by ρ for convenience), and defining

$$D := \sum_{k=1}^p H_k^T H_k, \tag{17}$$

the optimality conditions for (16) can be written as the KKT system

$$\begin{bmatrix} D & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^p H_k^T (x_k^{(n)} + \rho^{-1} \lambda_k^{(n)}) - \rho^{-1} c \\ b \end{bmatrix}. \quad (18)$$

Recalling that the product $H_k^T H_k$ is a diagonal matrix for all $k = 1, \dots, p$ we conclude that so is D , and since A has full row rank by assumption (18) can be solved efficiently, for instance by block elimination. In particular, eliminating x shows that the only matrix to be inverted/factorized is

$$AD^{-1}A^T \in \mathbb{S}^m. \quad (19)$$

Incidentally, we note that the first-order algorithms of [32, 44] require the factorization of a similar matrix with the same dimension. Since this matrix is the same at every iteration, its Cholesky factorization (or any other factorization of choice) can be computed and cached before starting the ADMM iterations. For some families of SDPs, such as the SDP relaxation of MaxCut problems and sum-of-squares (SOS) feasibility problems [50], the matrix $AD^{-1}A^T$ is diagonal, so solving (18) is inexpensive even when the SDPs are very large. If factorizing $AD^{-1}A^T$ is too expensive, the linear system (18) can alternatively be solved by an iterative method, such as the conjugate gradient method [36].

4.1.2 Minimization over \mathcal{Y}

Minimizing the augmented Lagrangian (15) over \mathcal{Y} is equivalent to solving p independent conic problems of the form

$$\begin{aligned} \min_{x_k} \quad & \left\| x_k - H_k x^{(n+1)} + \rho^{-1} \lambda_k^{(n)} \right\|^2 \\ \text{subject to} \quad & x_k \in \mathcal{S}_k. \end{aligned} \quad (20)$$

In terms of the original matrix variables X_1, \dots, X_p , each of these p sub-problems amounts to a projection on a PSD cone. More precisely, if $\mathbb{P}_{\mathbb{S}_+^{|\mathcal{C}_k|}}$ denotes the projection onto the PSD cone $\mathbb{S}_+^{|\mathcal{C}_k|}$ and $\text{mat}(\cdot) = \text{vec}^{-1}(\cdot)$, we have

$$x_k^{(n+1)} = \text{vec} \left\{ \mathbb{P}_{\mathbb{S}_+^{|\mathcal{C}_k|}} \left[\text{mat} \left(H_k x^{(n+1)} - \rho^{-1} \lambda_k^{(n)} \right) \right] \right\}. \quad (21)$$

Since the size of each cone $\mathbb{S}_+^{|\mathcal{C}_k|}$ is small for typical sparse SDPs and the projection onto it can be computed with an eigenvalue decomposition, the variables x_1, \dots, x_p can be updated efficiently. Moreover, the computation can be carried out in parallel. In contrast, the algorithms for generic SDPs developed in [31, 32, 44] require projections onto the (much larger) original PSD cone \mathbb{S}_+^n .

Remark 3 As anticipated in Remark 1, retaining the global variable x in the domain-space decomposed SDP to enforce the consensus constraints between the entries of the subvectors x_1, \dots, x_p (i.e., $x_k = H_k x$) is fundamental. In fact, it allowed us to separate the conic constraints from the affine constraints in (12) when applying the splitting strategy of ADMM, making the minimization over \mathcal{Y} easy to compute and parallelizable. In contrast, when x is eliminated as in the conversion method of [17, 27], the conic constraints and the affine constraints cannot be easily decoupled when applying the first-order splitting method: in [38] a quadratic SDP had to be solved at each iteration, which limits its scalability.

Algorithm 1 ADMM for the domain-space decomposition of sparse primal-form SDPs

```

1: Set  $\rho > 0$ ,  $\epsilon_{\text{tol}} > 0$ , a maximum number of iterations  $n_{\text{max}}$ , and initial guesses  $x^{(0)}, x_1^{(0)}, \dots, x_p^{(0)}$ ,
    $\lambda_1^{(0)}, \dots, \lambda_p^{(0)}$ .
2: Data preprocessing: chordal extension, chordal decomposition, and factorization of the KKT system (18).
3: for  $n = 1, 2, \dots, n_{\text{max}}$  do
4:   Compute  $x^{(n)}$  using (18).
5:   for  $k = 1, \dots, p$  do
6:     Compute  $x_k^{(n)}$  using (21).
7:     Compute  $\lambda_k^{(n)}$  using (22).
8:   end for
9:   Update the residuals  $\epsilon_c, \epsilon_\lambda$ .
10:  if  $\max(\epsilon_c, \epsilon_\lambda) \leq \epsilon_{\text{tol}}$  then
11:    break
12:  end if
13: end for

```

4.1.3 Updating the multipliers \mathcal{Z}

The final step in the n -th ADMM iteration is to update the multipliers $\lambda_1, \dots, \lambda_p$ with the usual gradient ascent rule: for each $k = 1, \dots, p$,

$$\lambda_k^{(n+1)} = \lambda_k^{(n)} + \rho \left(x_k^{(n+1)} - H_k x^{(n+1)} \right). \quad (22)$$

This computation is inexpensive and easily parallelized.

4.1.4 Stopping conditions

The ADMM algorithm is stopped after the n -th iteration if the relative primal/dual error measures.

$$\epsilon_c = \frac{\left(\sum_{k=1}^p \|x_k^{(n)} - H_k x^{(n)}\|^2 \right)^{1/2}}{\max \left\{ \left(\sum_{k=1}^p \|x_k^{(n)}\|^2 \right)^{1/2}, \left(\sum_{k=1}^p \|H_k x^{(n)}\|^2 \right)^{1/2} \right\}}, \quad (23a)$$

$$\epsilon_\lambda = \rho \left(\sum_{k=1}^p \|x_k^{(n)} - x_k^{(n-1)}\|^2 \right)^{1/2} \left(\sum_{k=1}^p \|\lambda_k^{(n)}\|^2 \right)^{-1/2}, \quad (23b)$$

are smaller than a specified tolerance, ϵ_{tol} . The reader is referred to [9] for a detailed discussion of stopping conditions for ADMM algorithms. In conclusion, a primal-form SDP with domain-space decomposition (12) can be solved using the steps summarized in Algorithm 1.

4.2 ADMM for the range-space decomposition

An ADMM algorithm similar to Algorithm 1 can be developed for the range-space decomposition (13) of a dual-standard-form sparse SDP. As in Section 4.1, we start by moving

all but the consensus equality constraints $z_k = v_k$, $k = 1, \dots, p$, to the objective using indicator functions. This leads to

$$\begin{aligned} \min \quad & -\langle b, y \rangle + \delta_0 \left(c - A^T y - \sum_{k=1}^p H_k^T v_k \right) + \sum_{k=1}^p \delta_{\mathcal{S}_k}(z_k) \\ \text{subject to} \quad & z_k = v_k, \quad k = 1, \dots, p. \end{aligned} \quad (24)$$

Given a penalty parameter $\rho > 0$ and a Lagrange multiplier λ_k for each of the constraints $z_k = v_k$, $k = 1, \dots, p$, we consider the (modified) augmented Lagrangian

$$\begin{aligned} \mathcal{L}(y, v_1, \dots, v_p, z_1, \dots, z_p, \lambda_1, \dots, \lambda_p) := & -\langle b, y \rangle \\ & + \delta_0 \left(c - A^T y - \sum_{k=1}^p H_k^T v_k \right) + \sum_{k=1}^p \left[\delta_{\mathcal{S}_k}(z_k) + \frac{\rho}{2} \left\| z_k - v_k + \frac{1}{\rho} \lambda_k \right\|^2 \right], \end{aligned} \quad (25)$$

and consider three groups of variables, $\mathcal{X} := \{y, v_1, \dots, v_p\}$, $\mathcal{Y} := \{z_1, \dots, z_p\}$, and $\mathcal{Z} := \{\lambda_1, \dots, \lambda_p\}$. Similar to Section 4.1, each iteration of the ADMM algorithm for (13) consists of minimizations over \mathcal{X} and \mathcal{Y} , and an update of the multipliers \mathcal{Z} . Each of these steps admits an inexpensive closed-form solution, as we demonstrate next.

4.2.1 Minimization over \mathcal{X}

Minimizing (25) over block \mathcal{X} is equivalent to solving the equality-constrained quadratic program

$$\begin{aligned} \min_{y, v_1, \dots, v_p} \quad & -\langle b, y \rangle + \frac{\rho}{2} \sum_{k=0}^p \left\| z_k^{(n)} - v_k + \frac{1}{\rho} \lambda_k^{(n)} \right\|^2 \\ \text{subject to} \quad & c - A^T y - \sum_{k=1}^p H_k^T v_k = 0. \end{aligned} \quad (26)$$

Let ρx be the multiplier for the equality constraint. After some algebra, the optimality conditions for (26) can be written as the KKT system

$$\begin{bmatrix} D & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c - \sum_{k=1}^p H_k^T \left(z_k^{(n)} + \rho^{-1} \lambda_k^{(n)} \right) \\ -\rho^{-1} b \end{bmatrix}, \quad (27)$$

plus a set of p uncoupled equations for the variables v_k ,

$$v_k = z_k^{(n)} + \frac{1}{\rho} \lambda_k^{(n)} + H_k x, \quad k = 1, \dots, p. \quad (28)$$

The KKT system (27) is the same as (18) after rescaling $x \mapsto -x$, $y \mapsto -y$, $c \mapsto \rho^{-1} c$ and $b \mapsto \rho b$. Consequently, the numerical cost of (26) is the same as in Section 4.1.1 plus the cost of (28), which is inexpensive and can be parallelized. Moreover, as in Section 4.1.1, the factors of the coefficient matrix required to solve the KKT system (27) can be pre-computed and cached before iterating the ADMM algorithm.

4.2.2 Minimization over \mathcal{Y}

As in Section 4.1.2, the variables z_1, \dots, z_p are updated with p independent projections,

$$z_k^{(n+1)} = \text{vec} \left\{ \mathbb{P}_{\mathbb{S}_+^{|C_k|}} \left[\text{mat} \left(v_k^{(n+1)} - \rho^{-1} \lambda_k^{(n)} \right) \right] \right\}, \quad (29)$$

where $\mathbb{P}_{\mathbb{S}_+^{|C_k|}}$ denotes projection on the PSD cone $\mathbb{S}_+^{|C_k|}$. Again, these projections can be computed efficiently and in parallel.

Remark 4 As anticipated in Section 3.2, introducing the set of slack variables v_k and the consensus constraints $z_k = v_k$, $k = 1, \dots, p$ is essential to obtain an efficient algorithm for range-space decomposed SDPs. The reason is that the splitting strategy of the ADMM decouples the conic and affine constraints, and the conic variables can be updated using the simple conic projection (29).

4.2.3 Updating the multipliers \mathcal{Z}

The multipliers λ_k , $k = 1, \dots, p$, are updated (possibly in parallel) with the inexpensive gradient ascent rule

$$\lambda_k^{(n+1)} = \lambda_k^{(n)} + \rho \left(z_k^{(n+1)} - v_k^{(n+1)} \right). \quad (30)$$

4.2.4 Stopping conditions

Similar to Section 4.1.4, we stop our ADMM algorithm after the n -th iteration if the relative primal/dual error measures

$$\epsilon_c = \frac{\left(\sum_{k=1}^p \|z_k^{(n)} - v_k^{(n)}\|^2 \right)^{1/2}}{\max \left\{ \left(\sum_{k=1}^p \|z_k^{(n)}\|^2 \right)^{1/2}, \left(\sum_{k=1}^p \|v_k^{(n)}\|^2 \right)^{1/2} \right\}}, \quad (31a)$$

$$\epsilon_\lambda = \rho \left(\sum_{k=1}^p \|z_k^{(n)} - z_k^{(n-1)}\|^2 \right)^{1/2} \left(\sum_{k=1}^p \|\lambda_k^{(n)}\|^2 \right)^{-1/2}, \quad (31b)$$

are smaller than a specified tolerance, ϵ_{tol} . The ADMM algorithm to solve the range-space decomposition (13) of a dual-form sparse SDP is summarized in Algorithm 2.

4.3 Equivalence between the primal and dual ADMM algorithms

Since the computational cost of (28) is the same as (22), all ADMM iterations for the dual-form SDP with range-space decomposition (13) have the same cost as those for the primal-form SDP with domain-space decomposition (12), plus the cost of (30). However, if one minimizes the dual augmented Lagrangian (25) over z_1, \dots, z_p *before* minimizing it over y, v_1, \dots, v_p , then (28) can be used to simplify the multiplier update equations to

$$\lambda_k^{(n+1)} = \rho H_k x^{(n+1)}, \quad k = 1, \dots, p. \quad (32)$$

Algorithm 2 ADMM for the range-space decomposition of sparse dual-form SDPs

```

1: Set  $\rho > 0$ ,  $\epsilon_{\text{tol}} > 0$ , a maximum number of iterations  $n_{\text{max}}$  and initial guesses  $y^{(0)}$ ,  $z_1^{(0)}$ ,  $\dots$ ,  $z_p^{(0)}$ ,  $\lambda_1^{(0)}$ ,  $\dots$ ,  $\lambda_p^{(0)}$ .
2: Data preprocessing: chordal extension, chordal decomposition, and factorization of the KKT system (27).
3: for  $n = 1, 2, \dots, n_{\text{max}}$  do
4:   for  $k = 1, \dots, p$  do
5:     Compute  $z_k^{(n)}$  using (29).
6:   end for
7:   Compute  $y^{(n)}$ ,  $x$  using (26).
8:   for  $k = 1, \dots, p$  do
9:     Compute  $v_k^{(n)}$  using (28)
10:    Compute  $\lambda_k^{(n)}$  using (32) (no cost).
11:  end for
12:  Update the residuals  $\epsilon_c$  and  $\epsilon_\lambda$ .
13:  if  $\max(\epsilon_c, \epsilon_\lambda) \leq \epsilon_{\text{tol}}$  then
14:    break
15:  end if
16: end for

```

Given that the products H_1x, \dots, H_px have already been computed to update v_1, \dots, v_p in (28), updating the multipliers $\lambda_1, \dots, \lambda_p$ requires only a scaling operation. Then, after swapping the order of \mathcal{X} - and \mathcal{Y} -block minimization of (25) and recalling that (18) and (27) are scaled versions of the same KKT system, the ADMM algorithms for the primal and dual standard form SDPs can be considered scaled versions of each other; see Fig. 4 for an illustration. In fact, the equivalence between ADMM algorithms for the original (*i.e.*, before chordal decomposition) primal and dual SDPs was already noted in [45].

Remark 5 Although the iterates of Algorithm 1 and Algorithm 2 are the same up to scaling, the convergence performance of these two algorithms differ in practice because first-order methods are sensitive to the scaling of the problem data and of the iterates.

5 Homogeneous self-dual embedding of domain- and range-space decomposed SDPs

Algorithms 1 and 2, as well as other first-order algorithms that exploit chordal sparsity [26, 30, 38], can solve feasible problems, but cannot detect infeasibility in their current formulation. Although some recent ADMM methods resolve this issue [5, 28], an elegant way to deal with an infeasible primal-dual pair of SDPs—which we pursue here—is to solve their homogeneous self-dual embedding (HSDE) [48].

The essence of the HSDE method is to search for a non-zero point in the intersection of a convex cone and a linear space; this is non-empty because it always contains the origin, meaning that the problem is always feasible. Given such a non-zero point, one can either recover optimal primal and dual solutions of the original pair of optimization problems, or construct a certificate of primal or dual infeasibility. HSDEs have been widely used to develop IPMs for SDPs [37, 47], and more recently O’Donoghue *et al.* have proposed an operator-splitting method to solve the HSDE of general conic programs [32].

In this section, we formulate the HSDE of the domain- and range-space decomposed SDPs (12) and (13), which is a primal-dual pair of SDPs. We also apply ADMM to solve this HSDE; in particular, we extend the algorithm of [32] to exploit chordal sparsity without increasing its computational cost (at least to leading order) compared to Algorithms 1 and 2.

5.1 Homogeneous self-dual embedding

To simplify the formulation of the HSDE of the decomposed (vectorized) SDPs (12) and (13), we let $\mathcal{S} := \mathcal{S}_1 \times \cdots \times \mathcal{S}_p$ be the direct product of all semidefinite cones and define

$$s := \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix}, \quad z := \begin{bmatrix} z_1 \\ \vdots \\ z_p \end{bmatrix}, \quad t := \begin{bmatrix} v_1 \\ \vdots \\ v_p \end{bmatrix}, \quad H := \begin{bmatrix} H_1 \\ \vdots \\ H_p \end{bmatrix}.$$

When strong duality holds, the tuple $(x^*, s^*, y^*, t^*, z^*)$ is optimal if and only if all of the following conditions hold:

1. (x^*, s^*) is primal feasible, *i.e.*, $Ax^* = b$, $s^* = Hx^*$, and $s^* \in \mathcal{S}$. For reasons that will become apparent below, we introduce slack variables $r^* = 0$ and $w^* = 0$ of appropriate dimensions and rewrite these conditions as

$$Ax^* - r^* = b, \quad s^* + w^* = Hx^*, \quad s^* \in \mathcal{S}, \quad r^* = 0, \quad w^* = 0. \quad (33)$$

2. (y^*, t^*, z^*) is dual feasible, *i.e.*, $A^T y^* + H^T t^* = c$, $z^* = t^*$, and $z^* \in \mathcal{S}$. Again, it is convenient to introduce a slack variable $h^* = 0$ of appropriate size and write

$$A^T y^* + H^T t^* + h^* = c, \quad z^* - t^* = 0, \quad z^* \in \mathcal{S}, \quad h^* = 0. \quad (34)$$

3. The duality gap is zero, *i.e.*

$$c^T x^* - b^T y^* = 0. \quad (35)$$

The idea behind the HSDE [48] is to introduce two non-negative and complementary variables τ and κ and embed the optimality conditions (33), (34) and (35) into the linear system $v = Qu$ with u, v and Q defined as

$$u := \begin{bmatrix} x \\ s \\ y \\ t \\ \tau \end{bmatrix}, \quad v := \begin{bmatrix} h \\ z \\ r \\ w \\ \kappa \end{bmatrix}, \quad Q := \begin{bmatrix} 0 & 0 & -A^T & -H^T & c \\ 0 & 0 & 0 & I & 0 \\ A & 0 & 0 & 0 & -b \\ H & -I & 0 & 0 & 0 \\ -c^T & 0 & b^T & 0 & 0 \end{bmatrix}. \quad (36)$$

Any nonzero solution of this embedding can be used to recover an optimal solution for (9) and (11), or provide a certificate for primal or dual infeasibility, depending on the values of τ and κ ; details are omitted for brevity, and the interested reader is referred to [32].

The decomposed primal-dual pair of (vectorized) SDPs (12)-(13) can therefore be recast as the self-dual conic feasibility problem

$$\begin{aligned} & \text{find } (u, v) \\ & \text{subject to } v = Qu, \\ & (u, v) \in \mathcal{K} \times \mathcal{K}^*, \end{aligned} \quad (37)$$

where, writing $n_d = \sum_{k=1}^p |\mathcal{C}_k|^2$ for brevity, $\mathcal{K} := \mathbb{R}^{n^2} \times \mathcal{S} \times \mathbb{R}^m \times \mathbb{R}^{n_d} \times \mathbb{R}_+$ is a cone and $\mathcal{K}^* := \{0\}^{n^2} \times \mathcal{S} \times \{0\}^m \times \{0\}^{n_d} \times \mathbb{R}_+$ is its dual.

5.2 A simplified ADMM algorithm

The feasibility problem (37) is in a form suitable for the application of ADMM, and moreover steps (4a)-(4c) can be greatly simplified by virtue of its self-dual character [32]. Specifically, the n -th iteration of the simplified ADMM algorithm for (37) proposed in [32] consists of the following three steps, where $\mathbb{P}_{\mathcal{K}}$ denotes projection onto the cone \mathcal{K} :

$$\hat{u}^{(n+1)} = (I + Q)^{-1} \left(u^{(n)} + v^{(n)} \right), \quad (38a)$$

$$u^{(n+1)} = \mathbb{P}_{\mathcal{K}} \left(\hat{u}^{(n+1)} - v^{(n)} \right), \quad (38b)$$

$$v^{(n+1)} = v^{(n)} - \hat{u}^{(n+1)} + u^{(n+1)}. \quad (38c)$$

Note that (38b) is inexpensive, since \mathcal{K} is the cartesian product of simple cones (zero, free and non-negative cones) and small PSD cones, and can be efficiently carried out in parallel. The third step is also computationally inexpensive and parallelizable. On the contrary, even when the preferred factorization of $I + Q$ (or its inverse) is cached before starting the iterations, a direct implementation of (38a) may require substantial computational effort because

$$Q \in \mathbb{S}^{n^2 + 2n_d + m + 1}$$

is a very large matrix (e.g., $n^2 + 2n_d + m + 1 = 2\,360\,900$ for problem `rs365` in Section 7.3). Yet, it is evident from (36) that Q is highly structured and sparse, and these properties can be exploited to speed up step (38a) using a series of block-eliminations and the matrix inversion lemma [10, Section C.4.3].

5.2.1 Solving the “outer” linear system

The affine projection step (38a) requires the solution of a linear system (which we refer to as the “outer” system for reasons that will become clear below) of the form

$$\begin{bmatrix} M & \zeta \\ -\zeta^T & 1 \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \end{bmatrix} = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}, \quad (39)$$

where

$$M := \begin{bmatrix} I & -\hat{A}^T \\ \hat{A} & I \end{bmatrix}, \quad \zeta := \begin{bmatrix} \hat{c} \\ -\hat{b} \end{bmatrix}, \quad \hat{A} := \begin{bmatrix} A & 0 \\ H & -I \end{bmatrix}, \quad \hat{c} := \begin{bmatrix} c \\ 0 \end{bmatrix}, \quad \hat{b} := \begin{bmatrix} b \\ 0 \end{bmatrix} \quad (40)$$

and we have split

$$u^{(n)} + v^{(n)} = \begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix}. \quad (41)$$

Note that \hat{u}_2 and ω_2 are scalars. Eliminating \hat{u}_2 from the first block equation in (39) yields

$$(M + \zeta\zeta^T)\hat{u}_1 = \omega_1 - \omega_2\zeta, \quad (42a)$$

$$\hat{u}_2 = \omega_2 + \zeta^T\hat{u}_1. \quad (42b)$$

Moreover, applying the matrix inversion lemma [10, Section C.4.3] to (42a) shows that

$$\hat{u}_1 = \left[I - \frac{(M^{-1}\zeta)\zeta^T}{1 + \zeta^T(M^{-1}\zeta)} \right] M^{-1}(\omega_1 - \omega_2\zeta). \quad (43)$$

Note that the vector $M^{-1}\zeta$ and the scalar $1 + \zeta^T(M^{-1}\zeta)$ depend only on the problem data, and can be computed before starting the ADMM iterations (since M is quasi-definite it can be inverted, and any symmetric matrix obtained as a permutation of M admits an LDL factorization). Instead, recalling from (41) that $\omega_1 - \omega_2\zeta$ changes at each iteration because it depends on the iterates $u^{(n)}$ and $v^{(n)}$, the vector $M^{-1}(\omega_1 - \omega_2\zeta)$ must be computed at each iteration. Consequently, computing \hat{u}_1 and \hat{u}_2 requires the solution of an “inner” linear system for the vector $M^{-1}(\omega_1 - \omega_2\zeta)$, followed by inexpensive vector inner products and scalar-vector operations in (43) and (42b).

5.2.2 Solving the “inner” linear system

Recalling the definition of M from (40), the “inner” linear system to calculate \hat{u}_1 in (43) has the form

$$\begin{bmatrix} I & -\hat{A}^T \\ \hat{A} & I \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = \begin{bmatrix} \nu_1 \\ \nu_2 \end{bmatrix}. \quad (44)$$

Here, σ_1 and σ_2 are the unknowns and represent suitable partitions of the vector $M^{-1}(\omega_1 - \omega_2\zeta)$ in (43), which is to be calculated, and we have split

$$\omega_1 - \omega_2\zeta = \begin{bmatrix} \nu_1 \\ \nu_2 \end{bmatrix}.$$

Applying block elimination to remove σ_1 from the second equation in (44), we obtain

$$(I + \hat{A}^T \hat{A})\sigma_1 = \nu_1 + \hat{A}^T \nu_2, \quad (45a)$$

$$\sigma_2 = -\hat{A}\sigma_1 + \nu_2. \quad (45b)$$

Recalling the definition of \hat{A} and recognizing that

$$D = H^T H = \sum_{k=1}^p H_k^T H_k$$

is a diagonal matrix, as already noted in Section 4.1.1, we also have

$$I + \hat{A}^T \hat{A} = \begin{bmatrix} (I + D + A^T A) & -H^T \\ -H & 2I \end{bmatrix}.$$

Block elimination can therefore be used once again to solve (45a), and simple algebraic manipulations show that the only matrix to be factorized (or inverted) is

$$I + \frac{1}{2}D + A^T A \in \mathbb{S}^{n^2}. \quad (46)$$

Note that this matrix depends only on the problem data and the chordal decomposition, so it can be factorized/inverted before starting the ADMM iterations. In addition, it is of the “diagonal plus low rank” form because $A \in \mathbb{R}^{m \times n^2}$ with $m < n^2$ (in fact, often $m \ll n^2$). This means that the matrix inversion lemma can be used to reduce the size of the matrix to factorize/invert even further: letting $P = I + \frac{1}{2}D$ be the diagonal part of (46), we have

$$(P + A^T A)^{-1} = P^{-1} - P^{-1} A^T (I + A P^{-1} A^T)^{-1} A P^{-1}.$$

In summary, after a series of block eliminations and applications of the matrix inversion lemma, step (38a) of the ADMM algorithm for (37) only requires the solution of an $m \times m$ linear system of equations with coefficient matrix

$$I + A \left(I + \frac{1}{2} D \right)^{-1} A^T \in \mathbb{S}^m, \quad (47)$$

plus a sequence of matrix-vector, vector-vector, and scalar-vector multiplications. A detailed count of floating-point operations is given in Section 6.

5.2.3 Stopping conditions

The ADMM algorithm described in the previous section can be stopped after the n -th iteration if a primal-dual optimal solution or a certificate of primal and/or dual infeasibility is found, up to a specified tolerance ϵ_{tol} . As noted in [32], rather than checking the convergence of the variables u and v , it is desirable to check the convergence of the original primal and dual SDP variables using the primal and dual residual error measures normally considered in interior-point algorithms [37]. For this reason, we employ different stopping conditions than those used in Algorithms 1 and 2, which we define below using the following notational convention: we denote the entries of u and v in (36) that correspond to x , y , τ , and z , respectively, by u_x , u_y , u_τ , and v_z .

If $u_\tau^{(n)} > 0$ at the n -th iteration of the ADMM algorithm, we take

$$x^{(n)} = \frac{u_x^{(n)}}{u_\tau^{(n)}}, \quad y^{(n)} = \frac{u_y^{(n)}}{u_\tau^{(n)}}, \quad z^{(n)} = \frac{H^T v_z^{(n)}}{u_\tau^{(n)}} \quad (48)$$

as the candidate primal-dual solutions, and define the relative primal residual, dual residual, and duality gap as

$$\epsilon_p := \frac{\|Ax^{(n)} - b\|_2}{1 + \|b\|_2}, \quad (49a)$$

$$\epsilon_d := \frac{\|A^T y^{(n)} + z^{(n)} - c\|_2}{1 + \|c\|_2}, \quad (49b)$$

$$\epsilon_g := \frac{|c^T x^{(n)} - b^T y^{(n)}|}{1 + |c^T x^{(n)}| + |b^T y^{(n)}|}. \quad (49c)$$

Also, we define the residual in consensus constraints as

$$\epsilon_c := \max\{(23a), (31a)\}. \quad (50)$$

We terminate the algorithm if $\max\{\epsilon_p, \epsilon_d, \epsilon_g, \epsilon_c\}$ is smaller than ϵ_{tol} . If $u_\tau^{(n)} = 0$, instead, we terminate the algorithm if

$$\max \left\{ \|Au_x^{(n)}\|_2 + \frac{c^T u_x^{(n)}}{\|c\|_2} \epsilon_{\text{tol}}, \|A^T u_y^{(n)} + H^T v_z^{(n)}\|_2 - \frac{b^T u_y^{(n)}}{\|b\|_2} \epsilon_{\text{tol}} \right\} \leq 0. \quad (51)$$

Certificates of primal or dual infeasibility (with tolerance ϵ_{tol}) are then given, respectively, by the points $u_y^{(n)}/(b^T u_y^{(n)})$ and $-u_x^{(n)}/(c^T u_x^{(n)})$. These stopping criteria are similar to those used by many other conic solvers, and coincide with those used in SCS [33] except for the addition of the residual in the consensus constraints (50). The complete ADMM algorithm to solve the HSDE of the primal-dual pair of domain- and range-space decomposed SDPs is summarized in Algorithm 3.

Algorithm 3 ADMM for the HSDE of sparse SDPs with chordal decomposition

```

1: Set  $\epsilon_{\text{tol}} > 0$ , a maximum number of iterations  $n_{\text{max}}$  and initial guesses  $\hat{u}^{(0)}, u^{(0)}, v^{(0)}$ .
2: Data preprocessing: chordal extension, chordal decomposition and factorization of the matrix in (47).
3: for  $n = 1, \dots, n_{\text{max}}$  do
4:   Compute  $\hat{u}^{(n+1)}$  using the sequence of block eliminations (39)-(47).
5:   Compute  $u^{(n+1)}$  using (38b).
6:   Compute  $v^{(n+1)}$  using (38c).
7:   if  $u_{\tau}^{(n)} > 0$  then
8:     Compute  $\epsilon_p, \epsilon_d, \epsilon_g, \epsilon_c$ .
9:     if  $\max\{\epsilon_p, \epsilon_d, \epsilon_g, \epsilon_c\} \leq \epsilon_{\text{tol}}$  then
10:      break
11:    end if
12:  else
13:    if (51) holds then
14:      break
15:    end if
16:  end if
17: end for

```

6 Complexity analysis via flop count

The computational complexity of each iteration of Algorithms 1-3 can be assessed by counting the total number of required *floating-point operations* (flops)—that is, the number of additions, subtractions, multiplications, or divisions of two floating-point numbers [10, Appendix C.1.1]—as a function of problem dimensions. For (18) and (27) we have

$$A \in \mathbb{R}^{m \times n^2}, \quad b \in \mathbb{R}^m, \quad c \in \mathbb{R}^{n^2}, \quad D \in \mathbb{S}^{n^2}, \quad H_k \in \mathbb{R}^{|C_k|^2 \times n^2} \text{ for } k = 1, \dots, p,$$

while the dimensions of the variables are

$$x \in \mathbb{R}^{n^2}, \quad y \in \mathbb{R}^m, \quad x_k, \lambda_k \in \mathbb{R}^{|C_k|^2} \text{ for } k = 1, \dots, p.$$

In this section, we count the flops in Algorithms 1–3 as a function of m , n , p , and $n_d = \sum_{k=1}^p |C_k|^2$. We do not consider the sparsity in the problem data, both for simplicity and because sparsity is problem-dependent. Thus, the matrix-vector product Ax is assumed to cost $(2n^2 - 1)m$ flops (for each row, we need n^2 multiplications and $n^2 - 1$ additions), while $A^T y$ is assumed to cost $(2m - 1)n^2$ flops. In practice, of course, these matrix-vector products may require significantly fewer flops if A is sparse, and sparsity *should* be exploited in any implementation to reduce computational cost. The only exception that we make concerns the matrix-vector products $H_k x$ and $H_k^T x_k$ because each H_k , $k = 1, \dots, p$, is an “entry-selector” matrix that extracts the subvector $x_k \in \mathbb{R}^{|C_k|^2}$ from $x \in \mathbb{R}^{n^2}$. Hence, the operations $H_k x$ and $H_k^T x_k$ require no actual matrix multiplications but only indexing operations (plus, possibly, making copies of floating-point numbers depending on the implementation), so they cost no flops according to our definition. However, we do not take into account that the vectors $H_k^T x_k \in \mathbb{R}^{n^2}$, $k = 1, \dots, p$, are often sparse, because their sparsity depends on the particular problem at hand. It follows from these considerations that computing the summation $\sum_{k=1}^p H_k^T x_k$ costs $(p - 1)n^2$ flops.

Using these rules, in the Appendix we prove the following results.

Proposition 1 *Given the Cholesky factorization of $AD^{-1}A^T = LL^T$, where L is lower triangular, solving the linear systems (18) and (27) via block elimination costs $(4m + p + 3)n^2 + 2m^2 + 2n_d$ flops.*

Proposition 2 *Given the constant vector $\hat{\zeta} := (M^{-1}\zeta)/(1 + \zeta^T M^{-1}\zeta) \in \mathbb{R}^{n^2+2n_d+m}$ and the Cholesky factorization $I + A(I + \frac{1}{2}D)^{-1}A^T = LL^T$, where L is lower triangular, solving (38a) using the sequence of block eliminations (39)–(47) requires $(8m + 2p + 11)n^2 + 2m^2 + 7m + 21n_d - 1$ flops.*

These propositions reveal that the computational complexity of the affine projections in Algorithms 1 and 2, which amount to solving the linear systems (18) and (27), is comparable to that of the affine projection (38a) in Algorithm 3. In fact, since typically $m \ll n^2$, we expect that the affine projection step of Algorithm 3 will be only approximately twice as expensive as the corresponding step in Algorithms 1 and 2 in terms of the number of flops, and therefore also in terms of CPU time (the numerical results presented in Table 11, Section 7.4 will confirm this expectation).

Similarly, the following result (also proved in the Appendix) guarantees that the leading-order costs of the conic projections in Algorithms 1–3 are identical and, importantly, depend only on the size and number of the maximal cliques in the chordal decomposition, *not* on the dimension n of the original PSD cone in (1)–(2).

Proposition 3 *The computational costs of the conic projections in Algorithms 1–3 require $\mathcal{O}(\sum_{k=1}^p |\mathcal{C}_k|^3)$ floating-point operations.*

In particular, the computational burden grows as a linear function of the number of cliques when their size is fixed, and as a cubic function of the clique size.

Finally, we emphasize that Propositions 1–3 suggest that Algorithms 1–3 should solve a primal-dual pair of sparse SDPs more efficiently than the general-purpose ADMM method for conic programs of [32], irrespective of whether this is used before or after chordal decomposition. In the former case, the benefit comes from working with smaller PSD cones: one block-elimination in equation (28) of [32] allows solving affine projection step (38a) in $\mathcal{O}(mn^2)$ flops, which is typically comparable to the flop count of Propositions 1 and 2,¹ but the conic projection step costs $\mathcal{O}(n^3)$ flops, which for typical sparse SDPs is significantly larger than $\mathcal{O}(\sum_{k=1}^p |\mathcal{C}_k|^3)$. In the latter case, instead, the conic projection (38b) costs the same for all methods, but projecting the iterates onto the affine constraints becomes much more expensive according to our flop count when the sequences of block eliminations described in Section 5 is not exploited fully.

7 Implementation and numerical experiments

We implemented Algorithms 1–3 in an open-source MATLAB solver which we call CDCS (Cone Decomposition Conic Solver). We refer to our implementation of Algorithms 1–3 as CDCS-primal, CDCS-dual and CDCS-hsde, respectively. This section briefly describes CDCS and presents numerical results on sparse SDPs from SDPLIB [7], large and sparse SDPs with nonchordal sparsity patterns from [4], and randomly generated SDPs with block-arrow sparsity pattern. Such problems have also been used as benchmarks in [4, 38].

In order to highlight the advantages of chordal decomposition, first-order algorithms, and their combination, the three algorithms in CDCS are compared to the interior-point solver SeDuMi [37], and to the single-threaded direct implementation of the first-order algorithm of [32] provided by the conic solver SCS [33]. All experiments were carried out on a PC with a 2.8 GHz Intel Core i7 CPU and 8GB of RAM and the solvers were called

¹ This can be seen more clearly after using the crude bound $n_d \leq pn^2$ in Propositions 1 and 2 and recalling that, for typical problems, $m \ll n^2$ and $p \ll n$.

with termination tolerance $\epsilon_{\text{tol}} = 10^{-3}$, number of iterations limited to 2000, and their default remaining parameters. The purpose of comparing CDCS to a low-accuracy IPM is to demonstrate the advantages of combining FOMs with chordal decomposition, while a comparison to the high-performance first-order conic solver SCS highlights the advantages of chordal decomposition alone. When possible, accurate solutions ($\epsilon_{\text{tol}} = 10^{-8}$) were also computed using SeDuMi; these can be considered “exact”, and used to assess how far the solution returned by CDCS is from optimality. Note that tighter tolerances could be used also with CDCS and SCS to obtain a more accurate solution, at the expense of increasing the number of iterations required to meet the convergence requirements. More precisely, given the proven convergence rate of general ADMM algorithms, any termination tolerance ϵ_{tol} is generally reached in at most $\mathcal{O}(\frac{1}{\epsilon_{\text{tol}}})$ iterations.

7.1 CDCS

To the best of our knowledge, CDCS is the first open-source first-order conic solver that exploits chordal decomposition for the PSD cones and is able to handle infeasible problems. Cartesian products of the following cones are supported: the cone of free variables \mathbb{R}^n , the non-negative orthant \mathbb{R}_+^n , second-order cones, and PSD cones. The current implementation is written in MATLAB and can be downloaded from

<https://github.com/oxfordcontrol/cdcs>.

Note that although many steps of Algorithms 1–3 can be carried out in parallel, our implementation is sequential. Interfaces with the optimization toolboxes YALMIP [29] and SOSTOOLS [34] are also available.

7.1.1 Implementation details

CDCS applies chordal decomposition to all PSD cones. Following [42], the sparsity pattern of each PSD cone is chordal extended using the MATLAB function `chol` to compute a symbolic Cholesky factorization of the approximate minimum-degree permutation of the cone’s adjacency matrix, returned by the MATLAB function `symamd`. The maximal cliques of the chordal extension are then computed using a `.mex` function from SparseCoLO [16].

As far as the steps of our ADMM algorithms are concerned, projections onto the PSD cone are performed using the MATLAB routine `eig`, while projections onto other supported cones only use vector operations. The Cholesky factors of the $m \times m$ linear system coefficient matrix (permuted using `symamd`) are cached before starting the ADMM iterations. The permuted linear system is solved at each iteration using the routines `cs_ksolve` and `cs_ltsolve` from the CSparse library [13].

CDCS solves the decomposed problems (12) and/or (13) using any of Algorithms 1–3, and then attempts to construct a primal-dual solution of the original SDPs (1) and (2) with a maximum determinant completion routine (see [17, Section 2], [42, Chapter 10.2]) adapted from SparseCoLO [16]. We adopted this approach for simplicity of implementation, even though we cannot guarantee that the principal sub-matrices $E_{C_k} X^* E_{C_k}^T$ of the partial matrix X^* returned by CDCS as the candidate solution are strictly positive definite (a requirement for the maximum determinant completion to exist). This may cause the current completion routine to fail, although for all cases in which we have observed failure, this was due to CDCS returning a candidate solution with insufficient accuracy that was not actually PSD-completable. In any case, our current implementation issues a warning when

the matrix completion routine fails; future versions of CDCS will include alternative completion methods, such as that discussed in [42, Chapter 10.3] and the minimum-rank PSD completion [30, Theorem 1], which work also in the lack of strict positive definiteness.

7.1.2 Adaptive penalty strategy

While the ADMM algorithms proposed in the previous sections converge independently of the choice of penalty parameter ρ , in practice its value strongly influences the number of iterations required for convergence. Unfortunately, analytic results for the optimal choice of ρ are not available except for very special problems [19, 35]. Consequently, in order to improve the convergence rate and make performance less dependent on the choice of ρ , CDCS employs the dynamic adaptive rule.

$$\rho^{k+1} = \begin{cases} \mu \rho^{(n)} & \text{if } \|\epsilon_p^{(n)}\|_2 \geq \nu \|\epsilon_d^{(n)}\|_2, \\ \mu^{-1} \rho^{(n)} & \text{if } \|\epsilon_d^{(n)}\|_2 \geq \nu \|\epsilon_p^{(n)}\|_2, \\ \rho^{(n)} & \text{otherwise.} \end{cases}$$

Here, $\epsilon_p^{(n)}$ and $\epsilon_d^{(n)}$ are the primal and dual residuals at the n -th iteration, while μ and ν are parameters no smaller than 1. Note that since ρ does not enter any of the matrices being factorized/inverted, updating its value is computationally inexpensive.

The idea of the rule above is to adapt ρ to balance the convergence of the primal and dual residuals to zero; more details can be found in [9, Section 3.4.1]. Typical choices for the parameters (the default in CDCS) are $\mu = 2$ and $\nu = 10$ [9].

7.1.3 Scaling the problem data

The relative scaling of the problem data also affects the convergence rate of ADMM algorithms. CDCS scales the problem data after the chordal decomposition step using a strategy similar to [32]. In particular, the decomposed SDPs (12) and (13) can be rewritten as:

$$\begin{aligned} \min_{\hat{x}} \quad & \hat{c}^T \hat{x} & \max_{\hat{y}, \hat{z}} \quad & \hat{b}^T \hat{y} \\ \text{subject to} \quad & \hat{A} \hat{x} = \hat{b} & \text{subject to} \quad & \hat{A}^T \hat{y} + \hat{z} = \hat{c} \\ & \hat{x} \in \mathbb{R}^{n^2} \times \mathcal{K}, & & \hat{z} \in \{0\}^{n^2} \times \mathcal{K}^* \end{aligned} \quad (52a,b)$$

where

$$\hat{x} = \begin{bmatrix} x \\ s \end{bmatrix}, \quad \hat{y} = \begin{bmatrix} y \\ t \end{bmatrix}, \quad \hat{z} = \begin{bmatrix} 0 \\ z \end{bmatrix}, \quad \hat{c} = \begin{bmatrix} c \\ 0 \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \quad \hat{A} = \begin{bmatrix} A & 0 \\ H & -I \end{bmatrix}.$$

CDCS solves the scaled decomposed problems

$$\begin{aligned} \min_{\hat{x}} \quad & \sigma(D\hat{c})^T \hat{x} & \max_{\hat{y}, \hat{z}} \quad & \rho(Eb)^T \hat{y} \\ \text{subject to} \quad & E\hat{A}D\hat{x} = \rho E\hat{b} & \text{subject to} \quad & D\hat{A}^T E\hat{y} + \bar{z} = \sigma D\hat{c} \\ & \hat{x} \in \mathbb{R}^{n^2} \times \mathcal{K}, & & \bar{z} \in \{0\}^{n^2} \times \mathcal{K}^*, \end{aligned} \quad (53a,b)$$

obtained by scaling vectors \hat{b} and \hat{c} by positive scalars ρ and σ , and the primal and dual equality constraints by positive definite, diagonal matrices D and E . Note that such a rescaling does not change the sparsity pattern of the problem. As already observed in [32], a good

Table 1 Details of the SDPLIB problems considered in this work.

	Small		Infeasible		Large and sparse			
	theta1	theta2	infd1	infd2	maxG11	maxG32	qpG11	qpG51
Original cone size, n	50	100	30	30	800	2000	1600	2000
Affine constraints, m	104	498	10	10	800	2000	800	1000
Number of cliques, p	1	1	1	1	598	1499	1405	1675
Maximum clique size	50	100	30	30	24	60	24	304
Minimum clique size	50	100	30	30	5	5	1	1

Table 2 Results for two small SDPs, theta1 and theta2, in SDPLIB.

	theta1			theta2		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	0.281	14	23.00	1.216	15	32.88
SeDuMi (low)	0.161	8	23.00	0.650	8	32.88
SCS (direct)	0.057	140	22.99	0.244	200	32.89
CDCS-primal	0.297	163	22.92	0.618	188	32.94
CDCS-dual	0.284	154	22.83	0.605	178	32.89
CDCS-hsde	0.230	156	23.03	0.392	118	32.88

choice for E , D , σ and ρ is such that the rows of \bar{A} and \bar{b} have Euclidean norm close to one, and the columns of \bar{A} and \bar{c} have similar norms. If D and D^{-1} are chosen to preserve membership to the cone $\mathbb{R}^{n^2} \times \mathcal{K}$ and its dual, respectively (how this can be done is explained in [32, Section 5]), an optimal point for (52) can be recovered from the solution of (53):

$$\hat{x}^* = \frac{D\bar{x}^*}{\rho}, \quad \hat{y}^* = \frac{E\bar{y}^*}{\sigma}, \quad \hat{z}^* = \frac{D^{-1}\bar{z}^*}{\sigma}.$$

7.2 Sparse SDPs from SDPLIB

Our first experiment is based on large-scale benchmark problems from SDPLIB [7]: two Lovász ϑ number SDPs (theta1 and theta2), two infeasible SDPs (infd1 and infd2), two MaxCut problems (maxG11 and maxG32), and two SDP relaxations of box-constrained quadratic programs (qpG11 and qpG51). Table 1 reports the dimensions of these problems, as well as chordal decomposition details. Problems theta1 and theta2 are dense, so have only one maximal clique; all other problems are sparse and have many maximal cliques of size much smaller than the original cone.

The numerical results are summarized in Tables 2–7. Table 2 shows that the small dense SDPs theta1 and theta2, were solved in approximately the same CPU time by all solvers. Note that since these problems only have one maximal clique, SCS and CDCS-hsde use similar algorithms, and performance differences are mainly due to the implementation (most notably, SCS is written in C). Table 3 confirms that CDCS-hsde successfully detects infeasible problems, while CDCS-primal and CDCS-dual do not have this ability.

The CPU time, number of iterations and terminal objective value for the four large-scale sparse SDPs maxG11, maxG32, qpG11 and qpG51 are listed in Table 4. All algorithms in CDCS were faster than either SeDuMi or SCS, especially for problems with smaller maximum clique size as one would expect in light of the complexity analysis of Section 6. Notably, CDCS solved maxG11, maxG32, and qpG11 in less than 100 s, a speedup of approximately $9\times$, $43\times$, and $66\times$ over SCS. In addition, even though FOMs are only meant to

Table 3 Results for two infeasible SDPs in SDPLIB. An objective value of +Inf denotes infeasibility. Results for the primal-only and dual-only algorithms in CDCS are not reported since they cannot detect infeasibility.

	infp1			infp2		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	0.127	2	+Inf	0.033	2	+Inf
SeDuMi (low)	0.120	2	+Inf	0.031	2	+Inf
SCS (direct)	0.067	20	+Inf	0.031	20	+Inf
CDCS-hsde	0.109	118	+Inf	0.114	101	+Inf

Table 4 Results for four large-scale sparse SDPs in SDPLIB, maxG11, maxG32, qpG11 and qpG51.

	maxG11			maxG32		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	88.9	13	629.2	1 266	14	1 568
SeDuMi (low)	48.7	7	628.7	624	7	1 566
SCS (direct)	93.9	1 080	629.1	2 433	2 000	1 568
CDCS-primal	22.2	230	629.5	84	311	1 569
CDCS-dual	16.9	220	629.2	61	205	1 567
CDCS-hsde	10.9	182	629.3	56	291	1 568

	qpG11			qpG51		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	650	14	2 449	1 895	22	1 182
SeDuMi (low)	357	8	2 448	1 530	18	1 182
SCS (direct)	1 065	2 000	2 449	2 220	2 000	1 288
CDCS-primal	29	249	2 450	482	1 079	1 145
CDCS-dual	21	193	2 448	396	797	1 201
CDCS-hsde	16	219	2 449	865	2 000	1 182

provide moderately accurate solutions, the terminal objective value returned by CDCS-hsde was always within 0.2% of the high-accuracy optimal value computed using SeDuMi. This is an acceptable difference in many practical applications.

To provide further evidence to assess the relative performance of the tested solvers, Tables 5 and 6 report the constraint violations for the original (not decomposed) SDPs, alongside the error in the consensus constraints for the decomposed problems. Specifically:

1. For CDCS-primal, we measure how far the partial matrix $X = \text{mat}(x) \in \mathbb{S}^n(\mathcal{E}, ?)$ is from being PSD-completable. This is the only quantity of interest because the equality constraints in (1) are satisfied exactly by virtue of the second block equation in (18). Instead of calculating the distance between X and the cone $\mathbb{S}_+^n(\mathcal{E}, ?)$ exactly using, for instance, the methods of [39]) we bound it from above by computing the smallest non-negative constant α such that $X + \alpha I \in \mathbb{S}_+^n(\mathcal{E}, ?)$; indeed, for such α it is clear that $\min_{Y \in \mathbb{S}_+^n(\mathcal{E}, 0)} \|Y - X\|_F \leq \|(X + \alpha I) - X\|_F = \alpha\sqrt{n}$. This strategy is more economical because, letting $\lambda_{\min}(M)$ be the minimum eigenvalue of a matrix M , Theorem 1 implies that

$$\alpha = -\min\{0, \lambda_{\min}[\text{mat}(H_1x)], \dots, \lambda_{\min}[\text{mat}(H_px)]\}.$$

To mitigate the dependence on the scaling of X , Table 5 lists the normalized error

$$\epsilon_\alpha := \frac{\alpha}{1 + \|X\|_F}. \quad (54)$$

Table 5 Residuals for the solutions returned by CDCS with $\epsilon_{\text{tol}} = 10^{-3}$ and the maximum number of iterations fixed to 2000. The residual ϵ_c , defined in (23a), (31a), and (50) for CDCS-primal, CDCS-dual, and CDCS-hsde respectively, measures the error in the consensus constraints of the decomposed SDPs. The quantities ϵ_p and ϵ_d are defined in (49a) and (49b), respectively, and measure the primal-dual residuals of the equality constraints for the original SDPs (before decomposition). Finally, ϵ_α computed using (54) measures the violation of the semidefiniteness constraint in the primal SDP.

	CDCS-primal		CDCS-dual		CDCS-hsde			
	ϵ_c	ϵ_α	ϵ_c	ϵ_d	ϵ_α	ϵ_p	ϵ_d	ϵ_c
maxG11	9.97e-4	2.72e-4	1.34e-4	2.48e-4	2.84e-4	5.52e-4	3.46e-4	9.98e-4
maxG32	9.96e-4	2.22e-4	4.75e-4	11.1e-4	1.78e-4	2.74e-4	2.68e-4	9.97e-4
qpG11	3.70e-4	3.62e-4	9.94e-4	20.0e-4	2.76e-4	9.66e-4	3.60e-4	8.40e-4
qpG51	9.34e-4	0.54e-4	9.91e-4	4.02e-4	0.15e-4	13.3e-4	31.3e-4	2.10e-4
rs35	9.99e-4	7.05e-4	6.41e-4	9.36e-4	0.01e-4	0.31e-4	13.6e-4	9.99e-4
rs200	9.92e-4	6.73e-4	1.48e-4	2.47e-4	2.74e-4	9.37e-4	9.95e-4	9.71e-4
rs228	9.97e-4	5.92e-4	2.14e-4	2.71e-4	1.75e-4	8.82e-4	9.83e-4	9.86e-4
rs365	9.98e-4	2.98e-4	4.61e-4	9.99e-4	0.36e-4	2.50e-4	9.99e-4	9.97e-4
rs1555	2.21e-4	1.55e-4	9.97e-4	19.6e-4	0.98e-4	11.0e-4	1.89e-4	9.99e-4
rs1907	9.98e-4	3.27e-4	6.24e-4	12.0e-4	0.23e-4	3.78e-4	9.96e-4	9.90e-4

Table 6 Residuals ϵ_p and ϵ_d , defined as in (49a) and (49b), for the solutions returned by SeDuMi and SCS. Both solvers were called with $\epsilon_{\text{tol}} = 10^{-3}$, and the maximum number of iterations for SCS is 2000. Entries marked *** denote failure due to memory limitations.

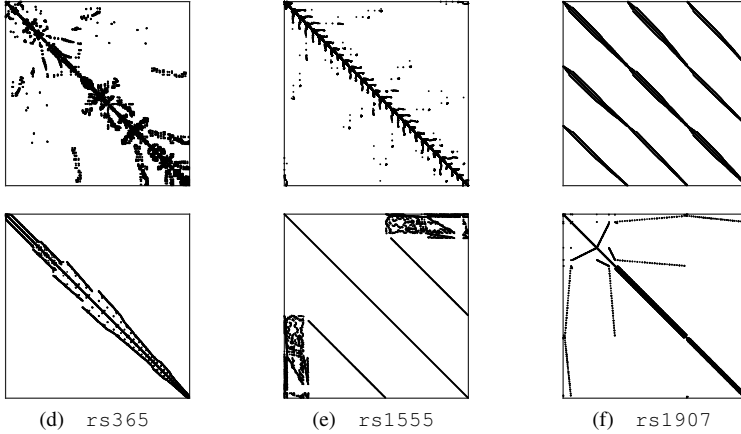
	SeDuMi		SCS	
	ϵ_p	ϵ_d	ϵ_p	ϵ_d
maxG11	8.36e-6	5.95e-7	4.71e-6	9.98e-4
maxG32	8.45e-6	3.85e-7	2.46e-6	3.20e-3
qpG11	1.13e-5	2.90e-7	1.32e-5	2.85e-2
qpG51	4.23e-6	5.41e-7	1.15e-4	2.36e-1
rs35	4.33e-7	1.73e-10	1.41e-5	7.60e-2
rs200	5.41e-6	4.83e-9	1.55e-5	4.43e-1
rs228	3.40e-6	1.89e-9	2.09e-5	1.72e-1
rs365	***	***	2.70e-5	7.20e-1
rs1555	***	***	3.67e-6	9.22e-1
rs1907	***	***	3.29e-5	7.96e-1

- For CDCS-dual, given the candidate solutions y and $Z = \text{mat}(\sum_{k=1}^p H_k^T z_k) = \sum_{k=1}^p E_{C_k} \text{mat}(z_k) E_{C_k}^T$ we report the violation of the equality constraints in (2) given by the relative dual residual ϵ_d , defined as in (49b). Note that (29) guarantees that the matrices $\text{mat}(z_1), \dots, \text{mat}(z_p)$ are PSD, so Z is also PSD.
- For the candidate solution (48) returned by CDCS-hsde, we list the relative primal and dual residuals ϵ_p and ϵ_d defined in (49a) and (49b), as well as the error measure ϵ_α computed with (54).
- For SeDuMi and SCS, we report only the primal and dual residuals (49a)–(49b) since the PSD constraints are automatically satisfied in both the primal and the dual problems.

The results in Table 5 demonstrate that, for the problems tested in this work, the residuals for the original SDPs are comparable to the convergence tolerance used in CDCS even when they are not tracked directly. The performance of SCS on our test problems is relatively poor. It is well known that the performance of ADMM algorithms is sensitive to their parameters, as well as problem scaling. We have calibrated CDSC using typical parameter values that offer a good compromise between efficiency and reliability, but we have not tried to fine-tune SCS under the assumption that good parameter values have already been chosen by its developers. Although performance may be improved through further parameter optimization, the discrepancy between the primal and dual residuals reported in Table 6 suggests

Table 7 Average CPU time per iteration (in seconds) for the SDPs from SDPLIB tested in this work.

	theta1	theta2	maxG11	maxG32	qpG11	qpG51
SCS (direct)	4.0×10^{-4}	1.2×10^{-3}	0.087	1.216	0.532	1.110
CDCS-primal	1.8×10^{-3}	3.3×10^{-3}	0.076	0.188	0.101	0.437
CDCS-dual	1.8×10^{-3}	3.4×10^{-3}	0.064	0.174	0.091	0.484
CDCS-hsde	1.5×10^{-3}	3.3×10^{-3}	0.048	0.140	0.064	0.430

**Fig. 5** Aggregate sparsity patterns of the nonchordal SDPs in [4]; see Table 8 for the matrix dimensions.

that slow convergence may be due to problem scaling for these instances. Note that CDCS and SCS adopt the same rescaling strategy, but one key difference is that CDCS applies it to the decomposed SDP rather than to the original one. Thus, CDCS has more degrees of scaling freedom than SCS, which might be the reason for the substantial improvement in convergence performance. Further investigation of the effect of scaling in ADMM-based algorithms for conic programming, however, is beyond the scope of this work.

Finally, to offer a comparison of the performance of CDCS and SCS that is insensitive both to problem scaling and to differences in the stopping conditions, Table 7 reports the average CPU time per iteration required to solve the sparse SDPs maxG11 , maxG32 , qpG11 and qpG51 , as well as the dense SDPs theta1 and theta2 . Evidently, all algorithms in CDCS are faster than SCS for the large-scale sparse SDPs (maxG11 , maxG32 , qpG11 and qpG51), and in particular CDCS-hsde improves on SCS by approximately $1.8\times$, $8.7\times$, $8.3\times$, and $2.6\times$ for each problem, respectively. This is to be expected since the conic projection step in CDCS is more efficient due to smaller semidefinite cones, but the results are remarkable considering that CDCS is written in MATLAB, while SCS is implemented in C. Additionally, the performance of CDCS could be improved even further with a parallel implementation of the projections onto small PSD cones.

7.3 Nonchordal SDPs

In our second experiment, we solved six large-scale SDPs with nonchordal sparsity patterns from [4]: rs35 , rs200 , rs228 , rs365 , rs1555 , and rs1907 . The aggregate sparsity patterns of these problems, illustrated in Fig. 5, come from the University of Florida Sparse Matrix

Table 8 Summary of chordal decomposition for the chordal extensions of the nonchordal SDPs form [4].

	rs35	rs200	rs228	rs365	rs1555	rs1907
Original cone size, n	2003	3025	1919	4704	7479	5357
Affine constraints, m	200	200	200	200	200	200
Number of cliques, p	588	1635	783	1244	6912	611
Maximum clique size	418	102	92	322	187	285
Minimum clique size	5	4	3	6	2	7

Table 9 Results for large-scale SDPs with nonchordal sparsity patterns form [4]. Entries marked *** indicate that the problem could not be solved due to memory limitations.

	rs35			rs200		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	1 391	17	25.33	4 451	17	99.74
SeDuMi (low)	986	11	25.34	2 223	8	99.73
SCS (direct)	2 378	2 000	25.08	9 697	2 000	81.87
CDCS-primal	370	379	25.27	159	577	99.61
CDCS-dual	272	245	25.53	103	353	99.72
CDCS-hsde	2 019	2 000	25.47	254	1 114	99.70
	rs228			rs365		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	1 655	21	64.71	***	***	***
SeDuMi (low)	809	10	64.80	***	***	***
SCS (direct)	2 338	2 000	62.06	34 497	2 000	44.02
CDCS-primal	94	400	64.65	321	401	63.37
CDCS-dual	84	341	64.76	240	265	63.69
CDCS-hsde	79	361	64.87	332	442	63.64
	rs1555			rs1907		
	Time (s)	# Iter.	Objective	Time (s)	# Iter.	Objective
SeDuMi (high)	***	***	***	***	***	***
SeDuMi (low)	***	***	***	***	***	***
SCS (direct)	139 314	2 000	34.20	50 047	2 000	45.89
CDCS-primal	1 721	2 000	61.22	330	349	62.87
CDCS-dual	317	317	69.54	271	252	63.30
CDCS-hsde	1 413	2 000	61.36	393	414	63.14

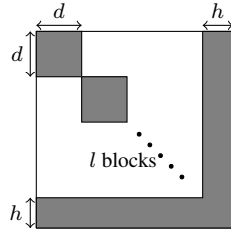
Collection [14]. Table 8 demonstrates that all six sparsity patterns admit chordal extensions with maximum cliques that are much smaller than the original cone.

Total CPU time, number of iterations, and terminal objective values are presented in Table 9. For all problems, the algorithms in CDCS (primal, dual and hsde) are all much faster than either SCS or SeDuMi. In addition, SCS never terminates successfully, while the objective value returned by CDCS is always within 2% of the high-accuracy solutions returned by SeDuMi (when this could be computed). The residuals listed in Tables 5 and 6 suggest that this performance difference might be due to poor problem scaling in SCS.

The advantages of the algorithms proposed in this work are evident from Table 10: the average CPU time per iteration in CDCS-hsde is approximately $22\times$, $24\times$, $28\times$, and $105\times$ faster compared to SCS for problems $rs200$, $rs365$, $rs1907$, and $rs1555$, respectively. The results for average CPU time per iteration also demonstrate that the computational complexity of all three algorithms in CDCS (primal, dual, and hsde) is independent of the original problem size: problems $rs35$ and $rs228$ have similar cone size n and the same number of constraints m , yet the average CPU time for the latter is approximately $5\times$ smaller. This can be explained by noticing that for all test problems considered here the number of constraints

Table 10 Average CPU time per iteration (in seconds) for the nonchordal SDPs form [4].

	rs35	rs200	rs228	rs365	rs1555	rs1907
SCS (direct)	1.188	4.847	1.169	17.250	69.590	25.240
CDCS-primal	0.944	0.258	0.224	0.715	0.828	0.833
CDCS-dual	1.064	0.263	0.232	0.774	0.791	0.920
CDCS-hsde	1.005	0.222	0.212	0.735	0.675	0.893

**Fig. 6** Block-arrow sparsity pattern (dots indicate repeating diagonal blocks). The parameters are: the number of blocks, l ; block size, d ; the width of the arrow head, h .

m is moderate, so the overall complexity of our algorithms is dominated by the conic projection. As stated in Proposition 3, this depends only on the size and number of the maximal cliques, not on the size of the original PSD cone. A more detailed investigation of how the number of maximal cliques, their size, and the number of constraints affect the performance of CDCS is presented next.

7.4 Random SDPs with block-arrow patterns

To examine the influence of the number of maximal cliques, their size, and the number of constraints on the computational cost of Algorithms 1–3, we considered randomly generated SDPs with a “block-arrow” aggregate sparsity pattern, illustrated in Fig. 6. Such a sparsity pattern is characterized by: the number of blocks, l ; the block size, d ; and the size of the arrow head, h . The associated PSD cone has dimension $ld + h$. The block-arrow sparsity pattern is chordal, with l maximal cliques all of the same size $d + h$. The effect of the number of constraints in the SDP, m , is investigated as well, and numerical results are presented below for the following scenarios:

1. Fix $l = 100$, $d = 10$, $h = 20$, and vary the number of constraints, m ;
2. Fix $m = 200$, $d = 10$, $h = 20$, and vary l (hence, the number of maximal cliques);
3. Fix $m = 200$, $l = 50$, $h = 10$, and vary d (hence, the size of the maximal cliques).

In our computations, the problem data are generated randomly using the following procedure. First, we generate random symmetric matrices A_1, \dots, A_m with block-arrow sparsity pattern, whose nonzero entries are drawn from the uniform distribution $U(0, 1)$ on the open interval $(0, 1)$. Second, a strictly primal feasible matrix $X_f \in \mathbb{S}_+^n(\mathcal{E}, 0)$ is constructed as $X_f = W + \alpha I$, where $W \in \mathbb{S}^n(\mathcal{E}, 0)$ is randomly generated with entries from $U(0, 1)$ and α is chosen to guarantee $X_f \succ 0$. The vector b in the primal equality constraints is then computed such that $b_i = \langle A_i, X_f \rangle$ for all $i = 1, \dots, m$. Finally, the matrix C in the dual constraint is constructed as $C = Z_f + \sum_{i=1}^m y_i A_i$, where y_1, \dots, y_m are drawn from $U(0, 1)$ and $Z_f \succ 0$ is generated similarly to X_f .

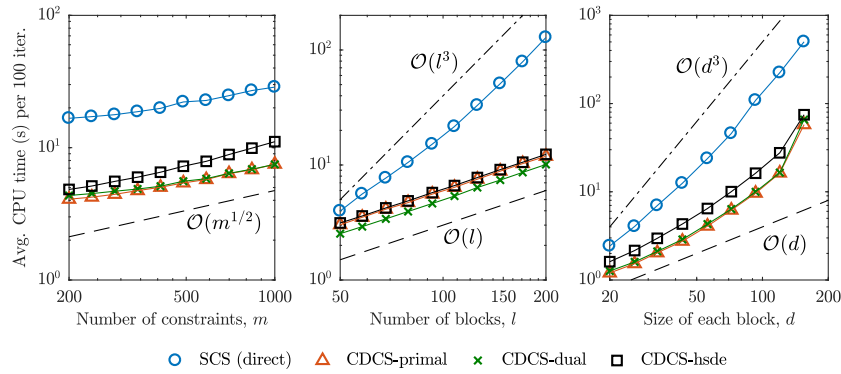


Fig. 7 Average CPU time (in seconds) per 100 iterations for SDPs with block-arrow patterns. Left to right: varying the number of constraints; varying the number of blocks; varying the block size.

Table 11 Average CPU time ($\times 10^{-2}$ s) required by the affine projection steps in CDCS-primal, CDCS-dual, and CDCS-hsde as a function of the number of constraints (m) for $l = 100$, $d = 10$, and $h = 20$.

m	200	239	286	342	409	489	585	699	836	1000
CDCS-primal	1.05	1.21	1.40	1.63	1.90	2.22	2.60	3.12	3.59	4.29
CDCS-dual	1.10	1.26	1.46	1.67	1.94	2.28	2.65	3.16	3.66	4.31
CDCS-hsde	1.84	2.14	2.55	2.95	3.50	4.12	4.85	5.80	6.81	8.04

The average CPU time per 100 iterations for the first-order solvers is plotted in Figure 7. As already observed in the previous sections, in all three test scenarios the algorithms in CDCS are faster than SCS, when the latter is used to solve the original SDPs (before chordal decomposition). Of course, as one would expect, the computational cost grows when either the number of constraints, the size of the maximal cliques, or their number is increased. Note, however, that the CPU time per iteration of CDCS grows more slowly than that of SCS as a function of the number of maximal cliques, which is the benefit of considering smaller PSD cones in CDCS. Precisely, the CPU time per iteration of CDCS increases linearly when the number of cliques l is raised, as expected from Proposition 3; instead, the CPU time per iteration of SCS grows cubically, since the eigenvalue decomposition on the original cone requires $\mathcal{O}(l^3)$ flops (note that when d and h are fixed, $(ld + h)^3 = \mathcal{O}(l^3)$). Finally, the results in Table 11 confirm the analysis in Propositions 1 and 2, according to which the CPU time required in the affine projection of CDCS-hsde was approximately twice larger than that of CDCS-primal or CDCS-dual. On the other hand, the increase in computational cost with the number of constraints m is slower than predicted by Propositions 1 and 2 due to the fact that, contrary to the complexity analysis presented in Section 6, our implementation of Algorithms 1–3 takes advantage of sparse matrix operations where possible.

8 Conclusion

In this paper, we have presented a conversion framework for large-scale SDPs characterized by chordal sparsity. This framework is analogous to the conversion techniques for IPMs of [17, 27], but is more suitable for the application of FOMs. We have then developed efficient ADMM algorithms for sparse SDPs in either primal or dual standard form, and for their homogeneous self-dual embedding. In all cases, a single iteration of our ADMM al-

gorithms only requires parallel projections onto small PSD cones and a projection onto an affine subspace, both of which can be carried out efficiently. In particular, when the number of constraints m is moderate the complexity of each iteration is determined by the size of the largest maximal clique, not the size of the original problem. This enables us to solve large, sparse conic problems that are beyond the reach of standard interior-point and/or other first-order methods.

All our algorithms have been made available in the open-source MATLAB solver CDCS. Numerical simulations on benchmark problems, including selected sparse problems from SDPLIB, large and sparse SDPs with a nonchordal sparsity pattern, and SDPs with a block-arrow sparsity pattern, demonstrate that our methods can significantly reduce the total CPU time requirement compared to the state-of-the-art interior-point solver SeDuMi [37] and the efficient first-order solver SCS [33]. We remark that the current implementation of our algorithms is sequential, but many steps can be carried out in parallel, so further computational gains may be achieved by taking full advantage of distributed computing architectures. Besides, it would be interesting to integrate some acceleration techniques (*e.g.*, [15, 41]) that promise to improve the convergence performance of ADMM in practice.

Finally, we note that the conversion framework we have proposed relies on chordal sparsity, but there exist large SDPs which do not have this property. An example with applications in many areas is that of SDPs from sum-of-squares relaxations of polynomial optimization problems. Future work should therefore explore whether and to which extent first order methods can be used to take advantage other types of sparsity and structure.

Acknowledgements The authors would like to thank the Associate Editor and the anonymous reviewers, whose invaluable comments contributed to improving the quality of our original manuscript.

Appendix

A Proof of Proposition 1. Since (18) and (27) are the same modulo scaling, we only consider the former. Also, we drop the superscript (n) to lighten the notation. Recall that $H_k^T x_k$ is an indexing operation and requires no flops, and let

$$\hat{b} := \sum_{k=1}^p H_k^T (x_k + \rho^{-1} \lambda_k) - \rho^{-1} c \in \mathbb{R}^{n^2}. \quad (55)$$

After a suitable block elimination and writing $AD^{-1}A^T = LL^T$, the solution of (18) is given by

$$LL^T y = AD^{-1} \hat{b} - b, \quad (56a)$$

$$x = D^{-1} (\hat{b} - A^T y). \quad (56b)$$

Computing x and y cost $(4m + p + 3)n^2 + 2m^2 + 2n_d$ flops, counted as the sum of:

- (i) $(p + 1)n^2 + 2n_d$ flops to form \hat{b} : no flops to multiply by H_k , $2|C_k|^2$ flops to compute $x_k + \rho^{-1} \lambda_k$, n^2 flops to calculate $\rho^{-1} c$, and $(p - 1)n^2 + n^2$ flops to sum all addends in (55).
- (ii) $(2m + 1)n^2$ flops to compute $AD^{-1} \hat{b} - b$: n^2 flops to compute $D^{-1} \hat{b}$ since D is diagonal, $(2n^2 - 1)m$ flops to multiply by A , and m flops to subtract b .
- (iii) $2m^2$ flops to compute y via forward and backward substitutions using (56a).
- (iv) $(2m + 1)n^2$ flops to compute x via (56b): $(2m - 1)n^2$ flops to find $A^T y$, n^2 flops to subtract it from \hat{b} , and n^2 flops to multiply by D^{-1} .

B Proof of Proposition 2. Consider the ‘‘inner’’ system (44) first. Partition the vectors σ_1 and σ_2 as

$$\sigma_1 = \begin{bmatrix} \sigma_{11} \\ \sigma_{12} \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} \sigma_{21} \\ \sigma_{22} \end{bmatrix},$$

where $\sigma_{11} \in \mathbb{R}^{n^2}$, σ_{12} , $\sigma_{22} \in \mathbb{R}^{n_d}$, and $\sigma_{21} \in \mathbb{R}^m$. The vectors ν_1 and ν_2 on the right-hand side of (44) can be partitioned in a similar way. Recalling the definition of the matrix \hat{A} from (40), (45b) becomes

$$\begin{bmatrix} \sigma_{21} \\ \sigma_{22} \end{bmatrix} = \begin{bmatrix} \nu_{21} - A\sigma_{11} \\ \nu_{22} - H\sigma_{11} + \sigma_{12} \end{bmatrix}. \quad (57)$$

To calculate σ_{11} and σ_{12} one needs to solve (45a), which after partitioning all variables can be rewritten as

$$\begin{bmatrix} (I + D + A^T A) & -H^T \\ -H & 2I \end{bmatrix} \begin{bmatrix} \sigma_{11} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \nu_{11} + A^T \nu_{21} + H^T \nu_{22} \\ \nu_{12} - \nu_{22} \end{bmatrix}. \quad (58)$$

Eliminating σ_{12} from the first block equation results in

$$\left(I + \frac{1}{2}D + A^T A \right) \sigma_{11} = \nu_{11} + A^T \nu_{21} + \frac{1}{2}H^T (\nu_{12} + \nu_{22}), \quad (59a)$$

$$\sigma_{12} = \frac{1}{2}(\nu_{12} - \nu_{22} + H\sigma_{11}). \quad (59b)$$

After defining $P := I + \frac{1}{2}D$ and $\eta := \nu_{11} + A^T \nu_{21} + \frac{1}{2}H^T (\nu_{12} + \nu_{22})$ to lighten the notation, an application of the matrix inversion lemma to (59a) yields

$$\sigma_{11} = P^{-1}\eta - P^{-1}A^T(I + AP^{-1}A^T)^{-1}AP^{-1}\eta. \quad (60)$$

We are now in a position to count the flops required to solve the ‘‘inner’’ linear system. First, computing σ_{11} via (60) requires a total $(6m + p + 3)n^2 + 2m^2 - m$ flops, counted as follows:

- (i) $(2m + p + 1)n^2$ flops to form η ;
- (ii) n^2 flops to compute $P^{-1}\eta$, since P is an $n^2 \times n^2$ diagonal matrix;
- (iii) $(2n^2 - 1)m$ flops to calculate $AP^{-1}\eta$;
- (iv) $2m^2$ flops to form the vector $(I + AP^{-1}A^T)^{-1}AP^{-1}\eta$ using forward and backward substitutions (we assume that the Cholesky decomposition $I + AP^{-1}A^T = LL^T$ has been cached);
- (v) $(2m - 1)n^2$ flops to find $A^T(I + AP^{-1}A^T)^{-1}AP^{-1}\eta$;
- (vi) $2n^2$ flops to compute σ_{11} via (60) given $P^{-1}\eta$ and $A^T(I + AP^{-1}A^T)^{-1}AP^{-1}\eta$.

Once σ_{11} is known, σ_{12} is found from (59b) with $3n_d$ flops because the product $H\sigma_{11}$ is simply an indexing operation and costs no flops. Given σ_{11} and σ_{12} , computing σ_{21} and σ_{22} from (57) requires $2mn^2 + 2n_d$ flops, so the ‘‘inner’’ linear system (44) costs a total of $(8m + 2p + 3)n^2 + 2m^2 - m + 5n_d$ flops.

After the inner system has been solved, we see that computing \hat{u}_1 from (43) requires $(8m + 2p + 9)n^2 + 2m^2 + 5m + 17n_d - 1$ flops in total:

- (i) $2(n^2 + 2n_d + m)$ flops to compute $\omega_1 - \omega_2\zeta$;
- (ii) $(8m + 2p + 3)n^2 + 2m^2 - m + 5n_d$ flops to solve the ‘‘inner’’ linear system $M^{-1}(\omega_1 - \omega_2\zeta)$;
- (iii) $2(n^2 + 2n_d + m) - 1$ flops to compute $\zeta^T M^{-1}(\omega_1 - \omega_2\zeta) \in \mathbb{R}$;
- (iv) $n^2 + 2n_d + m$ flops to calculate $\hat{\zeta} \cdot \zeta^T M^{-1}(\omega_1 - \omega_2\zeta)$;
- (v) $n^2 + 2n_d + m$ flops to compute $\hat{u}_1 = M^{-1}(\omega_1 - \omega_2\zeta) - \hat{\zeta} \cdot \zeta^T M^{-1}(\omega_1 - \omega_2\zeta)$.

Summing this to the $2(n^2 + 2n_d + m)$ flops required to calculate \hat{u}_2 using (42b) yields the desired result.

C Proof of Proposition 3. The conic projection (21) in Algorithm 1 amounts to projecting the matrices

$$\text{mat} \left(H_k x^{(n+1)} - \rho^{-1} \lambda_k^{(n)} \right) \in \mathbb{S}^{|C_k|}, \quad k = 1, \dots, p$$

onto the PSD cone $\mathbb{S}_+^{|C_k|}$. Computing $H_k x^{(n+1)} - \rho^{-1} \lambda_k^{(n)}$ requires $2|C_k|^2$ flops, while a PSD projection using a full eigenvalue decomposition costs $\mathcal{O}(|C_k|^3)$ flops to leading order, so the overall number of flops is $\mathcal{O}(\sum_{k=1}^p |C_k|^3)$. The same argument holds for the conic projection (29) in Algorithm 2.

In Algorithm 3, instead, the projection is onto the cone $\mathcal{K} := \mathbb{R}^{n^2} \times \mathcal{S} \times \mathbb{R}^m \times \mathbb{R}^{n_d} \times \mathbb{R}_+$. Nothing needs to be done to project onto \mathbb{R}^{n^2} , \mathbb{R}^m and \mathbb{R}^{n_d} , while the projection of $a \in \mathbb{R}$ onto \mathbb{R}_+ is given by $\max\{0, a\}$ and requires no flops according to our definition. Finally, projecting onto \mathcal{S} requires eigenvalue decompositions of the matrices $\text{mat}(x_k)$, $k = 1, \dots, p$, with a leading-order cost of $\mathcal{O}(\sum_{k=1}^p |C_k|^3)$ flops.

References

1. Agler, J., Helton, W., McCullough, S., Rodman, L.: Positive semidefinite matrices with a given sparsity pattern. *Linear Algebra Appl.* **107**, 101–149 (1988)
2. Alizadeh, F., Haeblerly, J.P.A., Overton, M.L.: Primal-dual interior-point methods for semidefinite programming: convergence rates, stability and numerical results. *SIAM J. Optim.* **8**(3), 746–768 (1998)
3. Andersen, M., Dahl, J., Liu, Z., Vandenberghe, L.: Interior-point methods for large-scale cone programming. In: *Optimization for machine learning*, pp. 55–83. MIT Press (2011)
4. Andersen, M.S., Dahl, J., Vandenberghe, L.: Implementation of nonsymmetric interior-point methods for linear optimization over sparse matrix cones. *Math. Program. Comput.* **2**(3-4), 167–201 (2010)
5. Banjac, G., Goulart, P., Stellato, B., Boyd, S.: Infeasibility detection in the alternating direction method of multipliers for convex optimization. *optimization-online.org* (2017). URL http://www.optimization-online.org/DB_HTML/2017/06/6058.html
6. Blair, J.R., Peyton, B.: An introduction to chordal graphs and clique trees. In: *Graph theory and sparse matrix computation*, pp. 1–29. Springer (1993)
7. Borchers, B.: SDPLIB 1.2, a library of semidefinite programming test problems. *Optim. Methods Softw.* **11**(1-4), 683–690 (1999)
8. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: *Linear Matrix Inequalities in System and Control Theory*. SIAM (1994)
9. Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends® Mach. Learn.* **3**(1), 1–122 (2011)
10. Boyd, S., Vandenberghe, L.: *Convex optimization*. Cambridge University Press (2004)
11. Burer, S.: Semidefinite programming in the space of partial positive semidefinite matrices. *SIAM J. Optim.* **14**(1), 139–172 (2003)
12. Dall’Anese, E., Zhu, H., Giannakis, G.B.: Distributed optimal power flow for smart microgrids. *IEEE Trans. Smart Grid* **4**(3), 1464–1475 (2013)
13. Davis, T.: *Direct Methods for Sparse Linear Systems*. SIAM (2006)
14. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* **38**(1), 1 (2011)
15. Fält, M., Giselsson, P.: Line search for generalized alternating projections. arXiv preprint arXiv:1609.05920 (2016)
16. Fujisawa, K., Kim, S., Kojima, M., Okamoto, Y., Yamashita, M.: User’s manual for SparseCoLO: Conversion methods for sparse conic-form linear optimization problems. Tech. rep., Research Report B-453, Tokyo Institute of Technology, Tokyo 152-8552, Japan (2009)
17. Fukuda, M., Kojima, M., Murota, K., Nakata, K.: Exploiting sparsity in semidefinite programming via matrix completion I: General framework. *SIAM J. Optim.* **11**(3), 647–674 (2001)
18. Gabay, D., Mercier, B.: A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Comput. Math. Appl.* **2**(1), 17–40 (1976)
19. Ghadimi, E., Teixeira, A., Shames, I., Johansson, M.: Optimal parameter selection for the alternating direction method of multipliers (ADMM): quadratic problems. *IEEE Trans. Automat. Contr.* **60**(3), 644–658 (2015)
20. Glowinski, R., Marroco, A.: Sur l’approximation, par éléments finis d’ordre un, et la résolution, par pénalisation-dualité d’une classe de problèmes de dirichlet non linéaires. *Revue française d’automatique, informatique, recherche opérationnelle. Analyse numérique* **9**(2), 41–76 (1975)
21. Godsil, C., Royle, G.F.: *Algebraic graph theory*. Springer Science & Business Media (2013)
22. Griewank, A., Toint, P.L.: On the existence of convex decompositions of partially separable functions. *Math. Program.* **28**(1), 25–49 (1984)
23. Grone, R., Johnson, C.R., Sá, E.M., Wolkowicz, H.: Positive definite completions of partial hermitian matrices. *Linear Algebra Appl.* **58**, 109–124 (1984)
24. Helmberg, C., Rendl, F., Vanderbei, R.J., Wolkowicz, H.: An interior-point method for semidefinite programming. *SIAM J. Optim.* **6**(2), 342–361 (1996)
25. Kakimura, N.: A direct proof for the matrix decomposition of chordal-structured positive semidefinite matrices. *Linear Algebra Appl.* **433**(4), 819–823 (2010)
26. Kalbat, A., Lavaei, J.: A fast distributed algorithm for decomposable semidefinite programs. In: *Proc. 54th IEEE Conf. Decis. Control*, pp. 1742–1749 (2015)
27. Kim, S., Kojima, M., Mevissen, M., Yamashita, M.: Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion. *Math. Program.* **129**(1), 33–68 (2011)
28. Liu, Y., Ryu, E.K., Yin, W.: A new use of douglas-rachford splitting and ADMM for identifying infeasible, unbounded, and pathological conic programs. arXiv preprint arXiv:1706.02374 (2017)
29. Lofberg, J.: YALMIP: A toolbox for modeling and optimization in MATLAB. In: *IEEE Int. Symp. Comput. Aided Control Sys. Des.*, pp. 284–289. IEEE (2004)

30. Madani, R., Kalbat, A., Lavaei, J.: ADMM for sparse semidefinite programming with applications to optimal power flow problem. In: Proc. 54th IEEE Conf. Decis. Control, pp. 5932–5939 (2015)
31. Malick, J., Povh, J., Rendl, F., Wiegale, A.: Regularization methods for semidefinite programming. *SIAM J. Optim.* **20**(1), 336–356 (2009)
32. O’Donoghue, B., Chu, E., Parikh, N., Boyd, S.: Conic optimization via operator splitting and homogeneous self-dual embedding. *J. Optim. Theory Appl.* **169**(3), 1042–1068 (2016)
33. O’Donoghue, B., Chu, E., Parikh, N., Boyd, S.: SCS: Splitting conic solver, version 1.2.6. <https://github.com/cvxgrp/scs> (2016)
34. Papachristodoulou, A., Anderson, J., Valmorbida, G., Prajna, S., Seiler, P., Parrilo, P.: SOSTOOLS version 3.0.0 sum of squares optimization toolbox for MATLAB. arXiv preprint arXiv:1310.4716 (2013)
35. Raghunathan, A.U., Di Cairano, S.: Alternating direction method of multipliers for strictly convex quadratic programs: Optimal parameter selection. In: Proc. American Control Conf., pp. 4324–4329. IEEE (2014)
36. Saad, Y.: Iterative methods for sparse linear systems. SIAM (2003)
37. Sturm, J.F.: Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optim. Methods Softw.* **11**(1-4), 625–653 (1999)
38. Sun, Y., Andersen, M.S., Vandenberghe, L.: Decomposition in conic optimization with partially separable structure. *SIAM J. Optim.* **24**(2), 873–897 (2014)
39. Sun, Y., Vandenberghe, L.: Decomposition methods for sparse matrix nearness problems. *SIAM J. Matrix Anal. Appl.* **36**(4), 1691–1717 (2015)
40. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.* **13**(3), 566–579 (1984)
41. Themelis, A., Patrinos, P.: SuperMann: a superlinearly convergent algorithm for finding fixed points of nonexpansive operators. arXiv preprint arXiv:1609.06955 (2016)
42. Vandenberghe, L., Andersen, M.S.: Chordal graphs and semidefinite optimization. *Found. Trends® Optim.* **1**(4), 241–433 (2014)
43. Vandenberghe, L., Boyd, S.: Semidefinite programming. *SIAM Review* **38**(1), 49–95 (1996)
44. Wen, Z., Goldfarb, D., Yin, W.: Alternating direction augmented lagrangian methods for semidefinite programming. *Math. Program. Comput.* **2**(3-4), 203–230 (2010)
45. Yan, M., Yin, W.: Self equivalence of the alternating direction method of multipliers. In: *Splitting Methods in Communication, Imaging, Science, and Engineering*, pp. 165–194. Springer (2016)
46. Yannakakis, M.: Computing the minimum fill-in is NP-complete. *SIAM J. Algebraic Discrete Methods* **2**, 77–79 (1981)
47. Ye, Y.: Interior point algorithms: theory and analysis. John Wiley & Sons (2011)
48. Ye, Y., Todd, M.J., Mizuno, S.: An $\mathcal{O}(\sqrt{nl})$ -iteration homogeneous and self-dual linear programming algorithm. *Math. Oper. Res.* **19**(1), 53–67 (1994)
49. Zhao, X.Y., Sun, D., Toh, K.C.: A newton-cg augmented lagrangian method for semidefinite programming. *SIAM J. Optim.* **20**(4), 1737–1765 (2010)
50. Zheng, Y., Fantuzzi, G., Papachristodoulou, A.: Exploiting sparsity in the coefficient matching conditions in Sum-of-Squares programming using ADMM. *IEEE Control Systems Letters* **1**(1), 80–85 (2017)
51. Zheng, Y., Fantuzzi, G., Papachristodoulou, A., Goulart, P., Wynn, A.: Fast ADMM for homogeneous self-dual embedding of sparse SDPs. *IFAC PapersOnLine* **50**(1), 8411–8416 (2017)
52. Zheng, Y., Fantuzzi, G., Papachristodoulou, A., Goulart, P., Wynn, A.: Fast ADMM for semidefinite programs with chordal sparsity. In: Proc. American Control Conf., pp. 3335–3340. IEEE (2017)