Optimization Methods and Software

ISSN: 1055-6788 (Print) 1029-4937 (Online) Journal homepage: https://www.tandfonline.com/loi/goms20

SDPNAL+: A Matlab software for semidefinite programming with bound constraints (version 1.0)

Defeng Sun, Kim-Chuan Toh, Yancheng Yuan & Xin-Yuan Zhao

To cite this article: Defeng Sun, Kim-Chuan Toh, Yancheng Yuan & Xin-Yuan Zhao (2019): SDPNAL+: A Matlab software for semidefinite programming with bound constraints (version 1.0), Optimization Methods and Software, DOI: 10.1080/10556788.2019.1576176

To link to this article: https://doi.org/10.1080/10556788.2019.1576176

OPTIMIZATION METHODS & SOFTWARE
https://doi.org/10.1080/10556788.2019.1576176

Taylor & Francis
Taylor & Francis Group

SDPNAL+: A MATLAB software for semidefinite programming with bound constraints (version 1.0)

Defeng Sun[a], Kim-Chuan Toh[b], Yancheng Yuan[c] and Xin-Yuan Zhao[d]

[a]Department of Applied Mathematics, The Hong Kong Polytechnic University, Hung Hom, Hong Kong; [b]Department of Mathematics, and Institute of Operations Research and Analytics, National University of Singapore, Singapore, Singapore; [c]Department of Mathematics, National University of Singapore, Singapore, Singapore; [d]Beijing Institute for Scientific and Engineering Computing, Beijing University of Technology, Beijing, People's Republic of China

ABSTRACT

Sdpnal+ is a MATLAB software package that implements an augmented Lagrangian based method to solve large scale semidefinite programming problems with bound constraints. The implementation was initially based on a majorized semismooth Newton-CG augmented Lagrangian method, here we designed it within an inexact symmetric Gauss-Seidel based semi-proximal ADMM/ALM (alternating direction method of multipliers/augmented Lagrangian method) framework for the purpose of deriving simpler stopping conditions and closing the gap between the practical implementation of the algorithm and the theoretical algorithm. The basic code is written in MATLAB, but some subroutines in C language are incorporated via Mex files. We also design a convenient interface for users to input their SDP models into the solver. Numerous problems arising from combinatorial optimization and binary integer quadratic programming problems have been tested to evaluate the performance of the solver. Extensive numerical experiments conducted in [L.Q. Yang, D.F. Sun, and K.C. Toh, *SDPNAL+: A majorized semismooth Newton-CG augmented Lagrangian method for semidefinite programming with nonnegative constraints*, Math. Program. Comput. 7 (2015), pp. 331–366] show that the proposed method is quite efficient and robust, in that it is able to solve 98.9% of the 745 test instances of SDP problems arising from various applications to the accuracy of 10^{-6} in the relative KKT residual.

ARTICLE HISTORY
Received 22 November 2017
Accepted 29 December 2018

KEYWORDS
Semidefinite programming; augmented Lagrangian; semismooth Newton-CG method; Matlab software package

1. Introduction

Let $\mathbb{S}^n$ be the space of $n \times n$ real symmetric matrices and $\mathbb{S}^n_+$ be the cone of positive semidefinite matrices in $\mathbb{S}^n$. For any $X \in \mathbb{S}^n$, we may sometimes write $X \succeq 0$ to indicate that $X \in \mathbb{S}^n_+$. Let $\mathcal{P} = \{X \in \mathbb{S}^n : L \leq X \leq U\}$, where L,U are given $n \times n$ symmetric matrices whose elements are allowed to take the values $-\infty$ and $+\infty$, respectively. Consider the semidefinite programming (SDP) problem:

$$(\mathbf{SDP}) \quad \min \left\{ \langle C, X \rangle \mid \mathcal{A}(X) = b,\ l \leq \mathcal{B}(X) \leq u,\ X \in \mathbb{S}^n_+,\ X \in \mathcal{P} \right\},$$

CONTACT Xin-Yuan Zhao xyzhao@bjut.edu.cn

where $b \in \mathbb{R}^m$, and $C \in \mathbb{S}^n$ are given data, $\mathcal{A} : \mathbb{S}^n \to \mathbb{R}^m$ and $\mathcal{B} : \mathbb{S}^n \to \mathbb{R}^p$ are two given linear maps whose adjoints are denoted as $\mathcal{A}^*$ and $\mathcal{B}^*$, respectively. The vectors l,u are given p-dimensional vectors whose elements are allowed to take the values $-\infty$ and ∞, respectively. Note that $\mathcal{P} = \mathbb{S}^n$ is allowed, in which case there are no additional bound constraints imposed on X. We assume that the $m \times m$ symmetric matrix $\mathcal{A}\mathcal{A}^*$ is invertible, i.e. $\mathcal{A}$ is surjective.

Note that (**SDP**) is equivalent to

$$\text{(P)} \quad \min \left\{ \langle C, X \rangle \mid \mathcal{A}(X) = b,\ \mathcal{B}(X) - s = 0,\ X \in \mathbb{S}^n_+,\ X \in \mathcal{P},\ s \in \mathcal{Q} \right\},$$

where $\mathcal{Q} = \{s \in \mathbb{R}^p : l \le s \le u\}$. The dual of (P), ignoring the minus sign in front of the minimization, is given by

$$\text{(D)} \quad \min \left\{ \delta^*_{\mathcal{P}}(-Z) + \delta^*_{\mathcal{Q}}(-v) + \left\langle -b, y \right\rangle \;\middle|\; \begin{array}{l} \mathcal{A}^*(y) + \mathcal{B}^*(\bar{y}) + S + Z = C,\ -\bar{y} + v = 0, \\ S \in \mathbb{S}^n_+,\ Z \in \mathbb{S}^n,\ y \in \mathbb{R}^m,\ \bar{y} \in \mathbb{R}^p,\ v \in \mathbb{R}^p \end{array} \right\},$$

where for any $Z \in \mathbb{S}^n$, $\delta^*_{\mathcal{P}}(-Z)$ is defined by

$$\delta^*_{\mathcal{P}}(-Z) = \sup\{\langle -Z, W \rangle \mid W \in \mathcal{P}\}$$

and $\delta^*_{\mathcal{Q}}(\cdot)$ is defined similarly. We note that our solver is designed based on the assumption that (P) and (D) are feasible.

While we have presented the problem (**SDP**) with a single variable block X, our solver is capable of solving the following more general problem with N blocks of variables:

$$\begin{array}{ll} \min & \displaystyle\sum_{j=1}^{N} \left\langle C^{(j)}, X^{(j)} \right\rangle \\ \text{s.t.} & \displaystyle\sum_{j=1}^{N} \mathcal{A}^{(j)}(X^{(j)}) = b, \quad l \le \sum_{j=1}^{N} \mathcal{B}^{(j)}(X^{(j)}) \le u, \\ & X^{(j)} \in \mathcal{K}^{(j)},\ X^{(j)} \in \mathcal{P}^{(j)},\ j = 1, \ldots, N, \end{array} \tag{1}$$

where $\mathcal{A}^{(j)} : \mathcal{X}^{(j)} \to \mathbb{R}^m$, and $\mathcal{B}^{(j)} : \mathcal{X}^{(j)} \to \mathbb{R}^p$ are given linear maps, $\mathcal{P}^{(j)} := \{X^{(j)} \in \mathcal{X}^{(j)} \mid L^{(j)} \le X^{(j)} \le U^{(j)}\}$ and $L^{(j)}, U^{(j)} \in \mathcal{X}^{(j)}$ are given symmetric matrices where the elements are allowed to take the values $-\infty$ and ∞, respectively. Here $\mathcal{X}^{(j)} = \mathbb{S}^{n_j}$ ($\mathbb{R}^{n_j}$), and $\mathcal{K}^{(j)} = \mathcal{X}^{(j)}$ or $\mathcal{K}^{(j)} = \mathbb{S}^{n_j}_+$ ($\mathbb{R}^{n_j}_+$). For later expositions, we should note that when $\mathcal{X}^{(j)} = \mathbb{S}^{n_j}$, the linear map $\mathcal{A}^{(j)} : \mathbb{S}^{n_j} \to \mathbb{R}^m$ can be expressed in the form of

$$\mathcal{A}^{(j)}(X^{(j)}) = \left[\begin{array}{c} \left\langle A^{(j)}_1, X^{(j)} \right\rangle, \ldots, \left\langle A^{(j)}_m, X^{(j)} \right\rangle \end{array} \right]^T, \tag{2}$$

where $A^{(j)}_1, \ldots, A^{(j)}_m \in \mathbb{S}^{n_j}$ are given constraint matrices. The corresponding adjoint $(\mathcal{A}^{(j)})^* : \mathbb{R}^m \to \mathbb{S}^{n_j}$ is then given by

$$(\mathcal{A}^{(j)})^* y = \sum_{k=1}^{m} y_k A^{(j)}_k.$$

In this paper, we introduce our MATLAB software package SDPNAL+ for solving (**SDP**) or more generally (1), where the maximum matrix dimension is assumed to be moderate

(say less than 5000) but the number of linear constraints $m+p$ can be large (say more than a million). One of our main contributions here is that the current algorithm has substantially extended the capability of SDPNAL+ to solve the general problem (1) compared to the original version in [24], wherein the algorithm is designed to solve a problem with only linear equality constraints and $\mathcal{P} = \{X \in \mathbb{S}^n \mid X \geq 0\}$ or $\mathcal{P} = \mathbb{S}^n$. Moreover, the implementation in [24] was based on a majorized semismooth Newton-CG augmented Lagrangian method developed in that paper. Here, for the purpose of deriving simpler stopping conditions, we redesign the algorithm by employing an inexact semi-proximal alternating direction method of multipliers (sPADMM) (or the semi-proximal augmented Lagrangian (sPALM) if the bound constraints are absent) framework developed in [2] for multi-block convex composite conic programming problems. Currently, the algorithm which we have implemented is a 2-phase algorithm based on the augmented Lagrangian function for (D). In the first phase, we employ the inexact symmetric Gauss-Seidel based sPADMM to solve the problem to a modest level of accuracy. Note that while the main purpose of the first phase algorithm is to generate a good initial point to warm-start the second phase algorithm, it can be used on its own to solve a problem. The algorithm we have implemented in the second phase is an inexact sPADMM for which the main subproblem in each iteration is solved by a semismooth Newton-CG method.

The development of SDPNAL+ in [24], which is built on the earlier work on SDPNAL in [25], has in fact spurred much of the recent progresses in designing efficient convergent ADMM-type algorithms for solving multi-block convex composite conic programming, such as [2,7,16]. Those works in turn shaped the recent algorithmic design of SDPNAL+. Indeed, the algorithm in the first phase of SDPNAL+ is the same as the convergent ADMM-type method developed in [16] when the subproblems in each iteration are solved analytically. For the algorithm in the second phase, it is an economical variant of the majorized semismooth Newton-CG (SNCG) augmented Lagrangian method designed in [24] to solve (D) for which only one SNCG subproblem is solved in each iteration.

Another contribution of this paper is our development of a basic interface for the users to input their SDP models into the SDPNAL+ solver. While there are currently two well developed MATLAB based user interfaces for SDP problems, namely, CVX [4] and YALMIP [8], there are strong motivations for us to develop our own interface here. A new interface is necessary to facilitate the modelling of an SDP problem for SDPNAL+ because of latter's flexibility to directly accept inequality constraints of the form '$l \leq \mathcal{B}(X) \leq u$', and bound constraints of the form '$L \leq X \leq U$'. The flexibility can significantly simplify the generation of the data in the SDPNAL+ format as compared to what need to be done in CVX or YALMIP to reformulate them as equality constraints through introducing extra variables. In addition, the final number of equality constraints present in the data input to SDPNAL+ can also be substantially fewer than those present in CVX or YALMIP. It is important to note here that the number of equality constraints present in the generated problem data can greatly affect the computational efficiency of the solvers, especially for interior-point based solvers. An illustration of the benefits just mentioned will be given at the end of Section 5.

Our SDPNAL+ solver is designed for solving feasible problems of the form presented in (P) and (D). It is capable of solving large scale SDPs with m or p up to a few millions but n is assumed to be moderate (up to a few thousands). Extensive numerical experiments conducted in [24] show that a variety of large scale SDPs can be solved by SDPNAL+ much more efficiently than the best alternative methods [10,21].

The SDPNAL+ package can be downloaded from the following website:
http://www.math.nus.edu.sg/ mattohkc/SDPNALplus.html

Installation and general information such as citations, can be found at the above link. The test instances which we have used to evaluate the performance of our solver can also be found at the above website.

We have evaluated the performance of SDPNAL+ on various classes of large scale SDP problems arising from the relaxation of combinatorial problems such as maximum stable set problems, quadratic assignment problems, frequency assignment problems, and binary integer quadratic programming problems. The solver has also been tested on large SDP problems arising from robust clustering problems, rank-one tensor approximation problems, as well as electronic structure calculations in quantum chemistry. The detailed numerical results can be found at the above website. Based on the numerical evaluation of SDPNAL+ on 745 SDP problems, we can observe that the solver is fairly robust (in the sense that it is able to solve most of the tested problems to the accuracy of 10^{-6} in the relative KKT residual) and highly efficient in solving the tested classes of problems.

The remaining parts of this paper are organized as follows. In the next section, we describe the installation and present some general information on our software. Section 2 gives some details on the main solver function `sdpnalplus.m`. In Section 3, we describe the algorithm implemented in SDPNAL+ and discuss some implementation issues. In Section 4, we present a basic interface for the users to input their SDP models into the SDPNAL+ solver. In Section 5, we present a few SDP examples to illustrate the usage of our software, and how to input the SDP models into our interface. Section 6 gives a summary of the numerical results obtained by SDPNAL+ in solving 745 test instances of SDP problems arising from various sources. Finally, we conclude the paper in Section 7.

2. Data structure and main solver

SDPNAL+ is an enhanced version of the SDPNAL solver developed by Zhao, Sun and Toh [25]. The internal implementation of SDPNAL+ thus follows the data structures and design framework of SDPNAL. A casual user need not understand the internal implementation of SDPNAL+.

2.1. *The main function:* `sdpnalplus.m`

In the SDPNAL+ solver, the main routine is `sdpnalplus.m`, whose calling syntax is as follows:

```
[obj,X,s,y,S,Z,ybar,v,info,runhist] = ...
sdpnalplus(blk,At,C,b,L,U,Bt,l,u,OPTIONS,X,s,y,S,Z,ybar,v);
```

Input arguments.

- `blk`: a cell array describing the conic block structure of the SDP problem.
- `At, C, b, L, U, Bt, l, u`: data of the problem (**SDP**).
  If $L \leq X$ but X is unbounded above, one can set `U = inf` or `U = []`. Similarly, if the linear map $\mathcal{B}$ is not present, one can set `Bt = []`, `l = []`, `u = []`.

- `OPTIONS`: a structure array of parameters (optional).
- `X, s, y, S, Z, ybar, v`: an initial iterate (optional).

Output arguments. The names chosen for the output arguments explain their contents. The argument `X` is a solution to (P) which satisfies the constraints $X \in \mathbb{S}^n_+$ and $X \in \mathcal{P}$ approximately up to the desired accuracy tolerance. The argument `info` is a structure array which records various performance measures of the solver. For example

```
info.etaRp, info.etaRd, info.etaK1, info.etaK2
```

correspond to the measures η_P, η_D, $\eta_{\mathcal{K}}$, $\eta_{\mathcal{P}}$ defined later in (4), respectively. The argument `runhist` is a structure array which records the history of various performance measures during the course of running `sdpnalplus.m`. For example,

```
runhist.primobj, runhist.dualobj, runhist.relgap
runhist.primfeasorg, runhist.dualfeasorg
```

record the primal and dual objective values, complementarity gap, primal and dual infeasibilities at each iteration, respectively.

2.2. Generation of starting point by `admmplus.m`

If an initial point `(X,s,y,S,Z,ybar,v)` is not provided for `sdpnalplus.m`, we call the function `admmplus.m`, which implements a convergent 3-block ADMM proposed in [16], to generate a starting point. The routine `admmplus.m` has a similar calling syntax as `sdpnalplus.m` given as follows:

```
 [obj,X,s,y,S,Z,ybar,v,info,runhist] = ...
admmplus(blk,At,C,b,L,U,Bt,l,u,OPTIONS,X,s,y,S,Z,ybar,v);
```

Note that if an initial point `(X,s,y,S,Z,ybar,v)` is not supplied to `admmplus.m`, the default initial point is `(0,0,0,0,0,0,0)`.

We should mention that although we use `admmplus.m` for the purpose of warm-starting `sdpnalplus.m`, the user has the freedom to use `admmplus.m` alone to solve the problem (SDP).

2.3. Arrays of input data

The format of the input data in SDPNAL+ is similar to those in SDPT3 [18,20]. For each SDP problem, the conic block structure of the problem data is described by a cell array named `blk`. If the kth block `X{k}` of the variable `X` is a nonnegative vector block with dimension n_k, then we set

$$\begin{aligned}
&\texttt{blk\{k,1\}} = \texttt{'l'}, \texttt{blk\{k,2\}} = n_k,\\
&\texttt{At\{k\}} = [n_k \times m \text{ sparse}], \texttt{Bt\{k\}} = [n_k \times p \text{ sparse}],\\
&\texttt{C\{k\}, L\{k\}, U\{k\}, X\{k\}, S\{k\}, Z\{k\}} = [n_k \times 1 \text{ double or sparse}].
\end{aligned}$$

If the jth block `X{j}` of the variable `X` is a semidefinite block consisting of a single block of size s_j, then the content of the jth block is given as follows:

$$\texttt{blk\{j,1\}} = \texttt{'s'}, \texttt{blk\{j,2\}} = s_j,$$
$$\texttt{At\{j\}} = [\bar{s}_j \times m \text{ sparse}], \texttt{Bt\{k\}} = [\bar{s}_j \times p \text{ sparse}],$$
$$\texttt{C\{j\}, L\{j\}, U\{j\}, X\{j\}, S\{j\}, Z\{j\}} = [s_j \times s_j \text{ double or sparse}],$$

where $\bar{s}_j = s_j(s_j+1)/2$. By default, the contents of the cell arrays `L` and `U` are set to be empty arrays. But if $\texttt{X\{j\}} \geq 0$ is required, then one can set

$$\texttt{L\{j\}} = \texttt{0}, \quad \texttt{U\{j\}} = [].$$

One can also set $L = 0$ to indicate that $\texttt{X\{j\}} \geq 0$ for all $j = 1, \ldots, N$ in (1).

We should mention that for the sake of computational efficiency, we store all the constraint matrices associated with the jth semidefinite block in vectorized form as a single $\bar{s}_j \times m$ matrix `At{j}`, where the kth column of this matrix corresponds to the kth constraint matrix $A_k^{(j)}$, i.e.

$$\texttt{At\{j\}} = [\text{svec}(A_1^{(j)}), \ldots, \text{svec}(A_m^{(j)})],$$

and $\text{svec} : \mathcal{S}^{s_j} \to \mathbb{R}^{\bar{s}_j}$ is the vectorization operator on symmetric matrices defined by

$$\text{svec}(X) = [X_{11}, \sqrt{2}X_{12}, X_{22}, \ldots, \sqrt{2}X_{1,s_j}, \ldots, \sqrt{2}X_{s_j-1,s_j}, X_{s_j,s_j}]^T. \tag{3}$$

We store `Bt` in the same format as `At`. The function `svec.m` provided in SDPNAL+ can easily convert a symmetric matrix into the vector storage scheme described in (3). Note that while we store the constraint matrices in vectorized form, the semidefinite blocks in the variables `X`, `S` and `Z` are stored either as matrices or in vectorized forms according to the storage scheme of the input data `C`.

Other than inputting the data `(At,b,C,L,U)` of an SDP problem individually, SDPNAL+ also provides the functions `read_sdpa.m` and `read_sedumi.m` to convert problem data stored in the SDPA [23] and SeDuMi [15] format into our cell-array data format just described. For example, for the problem `theta62.dat-s` in the folder `/datafiles`, the user can call the m-file `read_sdpa.m` to load the SDP data as follows:

```
>> [blk,At,C,b] = read_sdpa('./datafiles/theta62.dat-s');
>> OPTIONS.tol = 1e-6;
>> [obj,X,s,y,S,Z,ybar,v,info,runhist] =
sdpnalplus(blk,At,C,b, [],[],[],[],[],OPTIONS);
```

2.4. *The structure array* `OPTIONS` *for parameters*

Various parameters used in our solver `sdpnalplus.m` are set in the structure array `OPTIONS`. For details, see `SDPNALplus_parameters.m`. The important parameters which the user is likely to reset are described next.

(1) `OPTIONS.tol`: accuracy tolerance to terminate the algorithm, default is 10^{-6}.
(2) `OPTIONS.maxiter`: maximum number of iterations allowed, default is 20000.
(3) `OPTIONS.maxtime`: maximum time (in seconds) allowed, default is 10000.
(4) `OPTIONS.tolADM`: accuracy tolerance to use for `admmplus.m` when generating a starting point for the algorithm in the second phase of `sdpnalplus.m` (default $= 10^{-4}$).

(5) `OPTIONS.maxiterADM`: maximum number of ADMM iterations allowed for generating a starting point. When there are no bound constraints on X ($\mathcal{P} = \mathbb{S}^n$) and no linear inequality constraints corresponding to $\mathcal{B}(X)$ (hence $\mathcal{Q} = \emptyset$), the default value is roughly equal to 200; otherwise, the default value is 2000.
(6) `OPTIONS.printlevel`: different levels of details to print the intermediate information during the run. It can be the integers 0,1,2, with 1 being the default. Setting to the highest value 2 will result in printing the complete details.
(7) `OPTIONS.stopoption`: options to stop the solver. The default is `OPTIONS.stopoption = 1`, for which the solver may be stopped prematurely when stagnation occurs. To prevent the solver from stopping prematurely before the required accuracy is attained, set `OPTONS.stopoption = 0`.
(8) `OPTIONS.AATsolve.method`: options to solve a linear system involving the coefficient matrix $\mathcal{A}\mathcal{A}^*$, with `OPTIONS.AATsolve.method = 'direct' (default) or 'iterative'.` For the former option, a linear system of the form $\mathcal{A}\mathcal{A}^* y = h$ is solved by the sparse Cholesky factorization, while for the latter option, it is solved by a diagonally preconditioned PSQMR iterative solver.

2.5. Stopping criteria

In SDPNAL+, we measure the accuracy of an approximate optimal solution $(X, s, y, \bar{y}, S, Z, v)$ for (P) and (D) by using the following relative residual based on the KKT optimality conditions:

$$\eta = \max\{\eta_P, \eta_D, \eta_{\mathcal{K}}, \eta_{\mathcal{P}}\}, \tag{4}$$

where $\mathcal{K} = \mathbb{S}^n_+$,

$$\eta_P = \max\left\{\frac{\|\mathcal{A}(X) - b\|}{1 + \|b\|}, \frac{\|\mathcal{B}(X) - s\|}{1 + \|s\|}\right\},$$
$$\eta_D = \max\left\{\frac{\|\mathcal{A}^*(y) + \mathcal{B}^*(\bar{y}) + S + Z - C\|}{1 + \|C\|}, \frac{\|\bar{y} - v\|}{1 + \|v\|}\right\},$$
$$\eta_{\mathcal{K}} = \frac{1}{5}\frac{\|X - \Pi_{\mathcal{K}}(X - S)\|}{1 + \|X\| + \|S\|}, \quad \eta_{\mathcal{P}} = \frac{1}{5}\max\left\{\frac{\|X - \Pi_{\mathcal{P}}(X - Z)\|}{1 + \|X\| + \|Z\|}, \frac{\|s - \Pi_{\mathcal{Q}}(s - v)\|}{1 + \|s\| + \|v\|}\right\}.$$

Additionally, we compute the relative gap by

$$\eta_g = \frac{|\texttt{pobj} - \texttt{dobj}|}{1 + |\texttt{pobj}| + |\texttt{dobj}|}. \tag{5}$$

For a given accuracy tolerance specified in `OPTIONS.tol`, we terminate both `sdpnalplus.m` and `admmplus.m` when

$$\eta \leq \texttt{OPTIONS.tol}. \tag{6}$$

2.6. Caveats

There are a few points which we should emphasize on our solver.

- It is important to note that SDPNAL+ is a research software. It is not intended nor designed to be a general purpose software at the moment. The solver is designed based on the assumption that the primal and dual SDP problems (P) and (D) are feasible, and that Slater's constraint qualification holds. The solver is expected to be robust if the primal and dual SDP problems are both non-degenerate at the optimal solutions. However, if either one of them, particularly if the primal problem, is degenerate or if the Slater's condition fails, then the solver may not be able to solve the problems to high accuracy.
- Another point to note is that our solver is designed with the emphasis on handling problems with positive semidefinite variables efficiently. Little attention has been paid on optimizing the solver to handle linear programming problems.
- While in theory our solver can easily be extended to solve problems with second-order cone constraints, it is not capable of solving such problems at the moment although we plan to extend our solver to handle second-order cone programming problems in the future.

3. Algorithmic design and implementation

For simplicity, we will describe the algorithmic design for the problem (D) instead of the dual of the more general problem (1). Our algorithm is developed based on the augmented Lagrangian function for (D), which is defined as follows: given a penalty parameter $\sigma > 0$, for $(Z, v, y, \bar{y}) \in \mathbb{S}^n \times \mathbb{R}^p \times \times \mathbb{R}^m \times \mathbb{R}^p$, and $(X, s) \in \mathbb{S}^n \times \mathbb{R}^p$,

$$
\begin{aligned}
&L_\sigma(Z, v, S, y, \bar{y}; X, s) \\
&= \begin{cases} \delta^*_{\mathcal{P}}(-Z) + \delta^*_{\mathcal{Q}}(-v) + \langle -b, y\rangle + \delta_{\mathbb{S}^n_+}(S) - \dfrac{1}{2\sigma}\|X\|^2 - \dfrac{1}{2\sigma}\|s\|^2 \\ \quad + \dfrac{\sigma}{2}\|\mathcal{A}^*(y) + \mathcal{B}^*(\bar{y}) + S + Z - C + \sigma^{-1}X\|^2 + \dfrac{\sigma}{2}\|v - \bar{y} + \sigma^{-1}s\|^2. \end{cases}
\end{aligned}
$$

As mentioned in the Introduction, the algorithm implemented in SDPNAL+ is a 2-phase algorithm where the first phase is a convergent inexact sGS-sPADMM algorithm [2] whose template is described next.

First-phase algorithm. Given an initial iteration $(Z^0, v^0, S^0, y^0, \bar{y}^0, X^0, s^0)$, perform the following steps in each iteration.

Step 1. Let $R_1^k = \mathcal{A}^*(y^k) + \mathcal{B}^*(\bar{y}^k) + S^k + Z^k - C + \sigma^{-1}X^k$ and $R_2^k = v^k - \bar{y}^k + \sigma^{-1}s^k$. Compute $(Z^{k+1}, v^{k+1}) = \operatorname{argmin} L_\sigma(Z, v, S^k, y^k, \bar{y}^k; X^k, s^k)$ as follows:

$$
\begin{aligned}
Z^{k+1} &= \operatorname{argmin}\left\{\delta^*_{\mathcal{P}}(-Z) + \frac{\sigma}{2}\|Z - Z^k + R_1^k\|^2\right\} = \sigma^{-1}\Pi_{\mathcal{P}}(\sigma(R_1^k - Z^k)) - (R_1^k - Z^k), \\
v^{k+1} &= \operatorname{argmin}\left\{\delta^*_{\mathcal{Q}}(-v) + \frac{\sigma}{2}\|v - v^k + R_2^k\|^2\right\} = \sigma^{-1}\Pi_{\mathcal{Q}}(\sigma(R_2^k - v^k)) - (R_2^k - v^k).
\end{aligned}
$$

Step 2a. Compute

$$
(y^{k+1}_{\text{tmp}}, \bar{y}^{k+1}_{\text{tmp}}) \approx \operatorname{argmin}\left\{L_\sigma(Z^{k+1}, v^{k+1}, S^k, y, \bar{y}; X^k, s^k)\right\}.
$$

$$
\underbrace{\begin{bmatrix} \mathcal{A}\mathcal{A}^* & \mathcal{A}\mathcal{B}^* \\ \mathcal{B}\mathcal{A}^* & \mathcal{B}\mathcal{B}^* + \mathcal{I} \end{bmatrix}}_{\mathcal{M}} \begin{bmatrix} y \\ \bar{y} \end{bmatrix} = \begin{bmatrix} h_1 := \sigma^{-1}b - \mathcal{A}(S^k + Z^{k+1} - C + \sigma^{-1}X^k) \\ h_2 := v^{k+1} + \sigma^{-1}s^k - \mathcal{B}(S^k + Z^{k+1} - C + \sigma^{-1}X^k) \end{bmatrix}. \quad (7)
$$

In our implementation, we solve the linear system via the sparse Cholesky factorization of $\mathcal{M}$ if it can be computed at a moderate cost. Otherwise, we use a preconditioned CG method to solve (7) approximately so that the residual norm satisfies the following accuracy condition:

$$\sqrt{\sigma}\|[h_1; h_2] - \mathcal{M}[y_{\rm tmp}^{k+1}; \bar{y}_{\rm tmp}^{k+1}]\| \le \varepsilon_k,$$

where $\{\varepsilon_k\}$ is a predefined summable sequence of nonnegative numbers. In [24], the linear system corresponding to $\mathcal{M}$ is $\mathcal{A}\mathcal{A}^* y = h_1$, and it is solved by a direct method based on sparse Cholesky factorization. Here, the inexact sGS-ADMM framework [2] we have employed gives us the flexibility to solve the linear system approximately by an iterative solver such as the preconditioned conjugate gradient method, while not affecting the convergence of the algorithm. Such a flexibility is obviously critical to the computational efficiency of the algorithm when the sparse Cholesky factorization of $\mathcal{M}$ is impossible to compute for a very large linear system. Step 2b. Let $R_1^{k+1} = \mathcal{A}^*(y_{\rm tmp}^{k+1}) + \mathcal{B}^*(\bar{y}_{\rm tmp}^{k+1}) + S^k + Z^{k+1} - C + \sigma^{-1}X^k$. Compute

$$S^{k+1} = \operatorname{argmin}\left\{\delta_{\mathbb{S}_+^n}(S) + \frac{\sigma}{2}\|S - S^k + R_1^{k+1}\|^2\right\} = \Pi_{\mathbb{S}_+^n}(S^k - R_1^{k+1}).$$

Step 2c. Let $h_1^{\rm new} := h_1 - \mathcal{A}(S^{k+1} - S^k)$, and $h_2^{\rm new} := h_2 - \mathcal{B}(S^{k+1} - S^k)$. Set $(y^{k+1}, \bar{y}^{k+1}) = (y_{\rm tmp}^{k+1}, \bar{y}_{\rm tmp}^{k+1})$ if

$$\sqrt{\sigma}\left\|[h_1^{\rm new}; h_2^{\rm new}] - \mathcal{M}[y_{\rm tmp}^{k+1}; \bar{y}_{\rm tmp}^{k+1}]\right\| \le 10\varepsilon_k;$$

otherwise solve (7) with the vector h_1 replaced by $h_1^{\rm new}$ and h_2 replaced by $h_2^{\rm new}$, and the approximate solution $(y^{k+1}, \bar{y}^{k+1})$ should satisfy the above accuracy condition.
Step 3. Let $R_{D,1}^{k+1} = \mathcal{A}^*(y^{k+1}) + \mathcal{B}^*(\bar{y}^{k+1}) + S^{k+1} + Z^{k+1} - C$ and $R_{D,2}^{k+1} = v^{k+1} - \bar{y}^{k+1}$. Compute

$$X^{k+1} = X^k + \tau\sigma R_{D,1}^{k+1}, \quad s^{k+1} = s^k + \tau\sigma R_{D,2}^{k+1},$$

where $\tau \in (0, (1+\sqrt{5})/2)$ is the steplength which is typically chosen to be 1.618.

We note that by [2], the computation in Step 2a–2c is equivalent to solving the subproblem:

$$(S^{k+1}, y^{k+1}, \bar{y}^{k+1}) = \operatorname{argmin}\left\{\begin{array}{l} L_\sigma(Z^{k+1}, v^{k+1}, S, y, \bar{y}; X^k, s^k) \\ \quad + \dfrac{\sigma}{2}\|(S; y; \bar{y}) - (S^k; y^k; \bar{y}^k)\|_{\mathcal{H}}^2 \end{array}\right\},$$

where $\mathcal{H}$ is the symmetric Gauss-Seidel decomposition linear operator associated with the linear operator $(\mathcal{I}; \mathcal{A}; \mathcal{B})(\mathcal{I}, \mathcal{A}^*, \mathcal{B}^*) + \operatorname{diag}(0, 0, \mathcal{I})$, i.e.,

$$\mathcal{H} = \begin{bmatrix} (\mathcal{A}^*,\ \mathcal{B}^*)\mathcal{D}^{-1}(\mathcal{A}; \mathcal{B}) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{with} \quad \mathcal{D} = \begin{bmatrix} \mathcal{A}\mathcal{A}^* & \mathcal{A}\mathcal{B}^* \\ \mathcal{B}\mathcal{A}^* & \mathcal{B}\mathcal{B}^* + \mathcal{I} \end{bmatrix}.$$

There are numerous implementation issues which are addressed in SDPNAL+ to make the above skeletal algorithm practically efficient and robust. A detailed description of how the issues are addressed is beyond the scope of this paper. Hence we shall only briefly mention the most crucial ones.

(1) Dynamic adjustment of the penalty parameter σ, which is equivalent to restarting the algorithm with a new parameter by using the most recent iterate as the initial starting point.
(2) Initial scaling of the data, and dynamic scaling of the data.
(3) The efficient implementation of the PCG method to compute an approximate solution for (7).
(4) Efficient computation of the iterate S^{k+1} by using partial eigenvalue decomposition whenever it is expected to be more economical than a full eigenvalue decomposition.
(5) Efficient evaluation of the residual measure η defined in (4).

The algorithm in the second phase of SDPNAL+ is designed based on the following convergent inexact sPADMM algorithm (or the sPALM algorithm if the bound constraints are absent). After presenting the algorithm, we will explain the changes we made in this algorithm compared to that developed in [24].

Second-phase algorithm. Given an initial iterate $(Z^0, v^0, S^0, y^0, \bar{y}^0, X^0, s^0)$ generated in the first phase, perform the following steps in each iteration.

Step 1. Compute (Z^{k+1}, v^{k+1}) as in Step 1 of the first-phase algorithm.

Step 2. Compute

$$(y^{k+1}, \bar{y}^{k+1}, S^{k+1}) \approx \operatorname{argmin} L_\sigma(Z^{k+1}, v^{k+1}, S, y, \bar{y}; X^k, s^k)$$

by using the semismooth Newton-CG (SNCG) method which has been described in detail in [25] such that the following accuracy condition is met:

$$\sqrt{\sigma}\max\{\|b - \mathcal{A}\Pi_{\mathbb{S}^n_+}(W^{k+1})\|, \|\mathcal{B}\Pi_{\mathbb{S}^n_+}(W^{k+1}) - s^k + \sigma(\bar{y}^{k+1} - v^{k+1})\|\} \le \varepsilon_k,$$

where $W^{k+1} := \mathcal{A}^* y^{k+1} + \mathcal{B}^* \bar{y}^{k+1} + S^k + Z^{k+1} - C + \sigma^{-1} X^k$, and $\{\varepsilon_k\}$ is a predefined summable sequence of nonnegative numbers.

Step 3. Compute (X^{k+1}, s^{k+1}) as in Step 3 of the first-phase algorithm.

As one may observe, the difference between the first-phase and the second-phase algorithms lies in the construction of $(y^{k+1}, \bar{y}^{k+1}, S^{k+1})$ in Step 2 of the algorithms. In the first phase, the iterate is generated by adding the semi-proximal term $(\sigma/2)\|(S; y; \bar{y}) - (S^k; y^k; \bar{y}^k)\|^2_{\mathcal{H}}$ to the augmented Lagrangian function $L_\sigma(Z^{k+1}, v^{k+1}, S, y, \bar{y}; X^k, s^k)$. For the second phase, no such a semi-proximal term is required though one may still add a small semi-proximal term to the augmented Lagrangian function to ensure that the subproblems are well defined. As our goal is to minimize the augmented Lagrangian function $L_\sigma(Z, v, S, y, \bar{y}; X^k, s^k)$ for each pair of given (X^k, s^k), it is thus clear that Step 2 of the second-phase algorithm is closer to that goal compared to Step 2 of the first-phase algorithm. Of course, the price to pay is that the subproblem in Step 2 of the second-phase algorithm is more complicated to solve.

Now we highlight the differences between the above inexact sPADMM algorithm and the majorized semismooth Newton-CG (MSNCG) augmented Lagrangian method developed in [24]. First, the algorithm in [24] is designed to solve (**SDP**) with only linear equality constraints while the algorithm here is for the general problem with additional linear inequality constraints. Even when we specialize the algorithm here to the problem with only linear equality constraints, our algorithm here is also different from the one in [24]

which we will now explain. For the case when only linear equality constraints are present, the augmented Lagrangian function associated with the dual of that problem is given by

$$L_\sigma(Z,S,y;X) = \delta^*_{\mathcal{P}}(-Z) + \langle -b, y\rangle + \delta_{\mathbb{S}^n_+}(S) + \frac{\sigma}{2}\|\mathcal{A}^*y + S + Z - C + \sigma^{-1}X\|^2 - \frac{1}{2\sigma}\|X\|^2.$$

At the kth iteration of the MSNCG augmented Lagrangian method, the following subproblem must be solved:

$$\min_{y,S,Z}\left\{L_\sigma(Z,S,y;X^k)\right\},$$

and theoretically it is solved by the MSNCG method until a certain stopping condition is satisfied. However, in the practical implementation, only one step of the MSNCG method is applied to solve the subproblem and the stopping condition is not strictly enforced. Thus there is a gap between the theoretical algorithm and the practical algorithm implemented in [24]. But for the convergent inexact sPADMM algorithm employed in this paper, its practical implementation follows closely the steps described in the second-phase algorithm. Thus the practical algorithm presented in this paper is based on rigorous stopping conditions in each iteration to guarantee its overall convergence.

4. Interface

In this section, we will present a basic interface for our SDPNAL+ solver. First, we show how to use it via a small SDP example given as follows:

$$\begin{aligned}
\min\quad & \mathrm{trace}(X^{(1)}) + \mathrm{trace}(X^{(2)}) + \mathrm{sum}(X^{(3)}) \\
\text{s.t.}\quad & -X^{(1)}_{12} + 2X^{(2)}_{33} + 2X^{(3)}_{2} = 4, \\
& 2X^{(1)}_{23} + X^{(2)}_{42} - X^{(3)}_{4} = 3, \\
& 2 \le -X^{(1)}_{12} - 2X^{(2)}_{33} + 2X^{(3)}_{2} \le 7, \\
& X^{(1)} \in \mathbb{S}^6_+,\ X^{(2)} \in \mathbb{R}^{5\times 5},\ X^{(3)} \in \mathbb{R}^7_+, \\
& 0 \le X^{(1)} \le 10E_6,\ 0 \le X^{(2)} \le 8E_5,
\end{aligned} \tag{8}$$

where E_n denotes the $n \times n$ matrix of all ones. In the notation of (1), the problem (8) has three blocks of variables $X^{(1)}, X^{(2)}, X^{(3)}$. The first linear map $\mathcal{A}^{(1)}$ contains two constraint matrices $A^{(1)}_1, A^{(1)}_2 \in \mathbb{S}^6$ whose nonzero elements are given by

$$(A^{(1)}_1)_{12} = (A^{(1)}_1)_{21} = -0.5, \quad (A^{(1)}_2)_{23} = (A^{(1)}_2)_{32} = 1.$$

With the above constraint matrices, we get $\langle A^{(1)}_1, X^{(1)}\rangle = -X^{(1)}_{12}$ and $\langle A^{(1)}_2, X^{(1)}\rangle = 2X^{(1)}_{23}$.

The second linear map $\mathcal{A}^{(2)}$ contains two constraint matrices $A_1^{(2)}, A_2^{(2)} \in \mathbb{R}^{5\times 5}$ whose nonzero elements are given by

$$(A_1^{(2)})_{33} = 2, \quad (A_2^{(2)})_{42} = 1.$$

Since the third variable $X^{(3)}$ is a vector, the third linear map $\mathcal{A}^{(3)}$ is a constraint matrix $A^{(3)} \in \mathbb{R}^{2\times 7}$ whose nonzero elements are given by

$$(A^{(3)})_{12} = 2, \quad (A^{(3)})_{24} = -1.$$

In a similar fashion, one can identify the matrices for the linear maps $\mathcal{B}^{(1)}, \mathcal{B}^{(2)}$, and $\mathcal{B}^{(3)}$.

The example (8) can be coded using our interface as follows:

Listing 1. Example (8).

```
n1 = 6; n2 = 5; n3 = 7;
mymodel = ccp_model('Example_simple');
X1 = var_sdp(n1,n1);
X2 = var_nn(n2,n2);
X3 = var_nn(n3);
mymodel.add_variable(X1,X2,X3);
mymodel.minimize(trace(X1) + trace(X2) + sum(X3));
mymodel.add_affine_constraint(-X1(1,2)+2*X2(3,3)+2*X3(2) == 4);
mymodel.add_affine_constraint(2*X1(2,3)+X2(4,2)-X3(4) == 3);
mymodel.add_affine_constraint(2<=-X1(1,2)-2*X2(3,3)+2*X3(2)<=7);
mymodel.add_affine_constraint(0 <= X1 <= 10);
mymodel.add_affine_constraint(X2 <= 8);
mymodel.solve;
```

Note that although the commands

```
mymodel.add_affine_constraint(-X1(1,2)+2*X2(3,3)+2*X3(2)==4);
mymodel.add_affine_constraint(2*X1(2,3)+X2(4,2)-X3(4)==3);
```

are convenient to use for a small example, it may become tedious if there are many such constraints. In general, it is more economical to encode numerous such constraints by using the constraint matrices of the linear maps $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \mathcal{A}^{(3)}$, which we illustrate below:

Listing 2. Example (8) with constraints specified via linear maps as cell arrays.

```
A1 = {sparse(n1,n1); sparse(n1,n1)}; A2 = {sparse(n2,n2); sparse(n2,n2)};
A3 = sparse(2,n3);
A1{1}(1,2) = -1; A2{1}(3,3) = 2; A3(1,2) = 2;
A1{2}(2,3) = 2;  A2{2}(4,2) = 1; A3(2,4) = -1;
b = [4;3];
mymodel.add_affine_constraint(A1*X1 + A2*X2 + A3*X3 == b);
```

As the reader may have noticed, in constructing the matrix `A1{1}` corresponding to the constraint matrix $A_1^{(1)}$, we set `A1{1}(1,2) = -1` instead of `A1{1}(1,2) = -0.5; A1{1}(2,1) = -0.5`. Both ways of inputing `A1{1}` are acceptable as internally, we will symmetrize the matrix `A1{1}`.

In following subsections, we will discuss the details of the interface.

4.1. Creating a ccp model

Before declaring `variables`, `constraints` and setting `parameters`, we need to create a `ccp_model` class first. This is done via the command:

```
mymodel = ccp_model(model_name);
```

The string `model_name` is the name of the created `ccp_model`. If no model name is specified, the default name is `'Default'`.

After solving the created `mymodel`, we save all the relevant information in the file `'model_name.mat'`. It contains two structure arrays, `input_data` and `solution`, which store all the input data and solution information, respectively.

4.2. Delcaring variables

Variables in SDPNAL+ can be real vectors or matrices. Currently, our interface supports four types of variables: free variables, variables in SDP cones, nonnegative variables and variables which are symmetric matrices. Next, we introduce them in details.

(1) **Free variables.** One can declare a free variable $\mathtt{X} \in \mathbb{R}^{m\times n}$ via the command:

$$\mathtt{X} = \textbf{var\_free}(\mathtt{m}, \mathtt{n});$$

where the parameters `m` and `n` specify the dimensions of `X`. One can also declare a column vector variable $\mathtt{Y} \in \mathbb{R}^n$ simply via the command:

$$\mathtt{Y} = \textbf{var\_free}(\mathtt{n});$$

(2) **Variables in SDP cones.** A variable $\mathtt{X} \in \mathbb{S}^n_+$ can be declared via the command:

$$\mathtt{X} = \textbf{var\_sdp}(\mathtt{n}, \mathtt{n});$$

In this case, the variable must be a square matrix, so $\mathtt{X} = \textbf{var\_sdp}(\mathtt{m},\mathtt{n})$ with $\mathtt{m} \neq \mathtt{n}$ is invalid.

(3) **Variables in nonnegative orthants.** To declare a nonnegative variable $\mathtt{X} \in \mathbb{R}^{m\times n}_+$, one can use the command:

$$\mathtt{X} = \textbf{var\_nn}(\mathtt{m}, \mathtt{n});$$

We can also use $\mathtt{Y} = \textbf{var\_nn}(\mathtt{n})$ to declare a vector variable $\mathtt{Y} \in \mathbb{R}^n_+$.

(4) **Variables which are symmetric matrices.** To declare a symmetric matrix variable $\mathtt{X} \in \mathbb{S}^n$, one can use the command:

$$\mathtt{X} = \textbf{var\_symm}(\mathtt{n}, \mathtt{n});$$

In this case, the variable must be a square matrix.

(5) **Adding declared variables into a model.** Before one can start to specify the objective function and constraints in a model, the variables, say `X` and `Y`, that we have declared must be added to the `ccp_model` class `mymodel` that we have created before. This step is simply done via the command:

```
mymodel.add_variable(X,Y);
```

Here `mymodel` is a class object and **add_variable** is a method in the class.

Table 1. Supported functions for specifying the objective function in a model.

Function	Description
inprod(C, X)	The inner product of a constant vector or matrix C and variable X of the same dimension.
trace(X)	The trace of a square matrix variable X.
sum(X)	The sum of all elements of a vector or matrix variable X.
l1_norm(X)	The ℓ_1 norm of a variable X.
l1_norm($\mathcal{A}$*X +b)	The ℓ_1 norm of an affine expression. For the exact meaning of the expression "$\mathcal{A}$*X", the reader can refer to (10).

4.3. Declaring the objective function

After creating the model `mymodel`, declaring variables (say `X` and `Y`) and adding them into `mymodel`, we can proceed to specify the objective function. Declaring an objective function requires the use of the functions (methods) **minimize** or **maximize**. There must be one and only one objective function in a model specification. In general, the objective function is specified through the sum or difference of the **inprod** function (inner product of two vectors or two matrices) which must have two input arguments in the form: **inprod**(`C`,`X`) where `X` must be a declared variable, and `C` must be a constant vector or matrix which is already available in the workspace and having the same dimension as `X`. The input `C` can also be a constant vector or matrix generated by some MATLAB built-in functions such as `speye(n,n)`.

Although we encourage users to specify an optimization problem in the standard form given in (1), as a user-friendly interface, we also provide some extra functions to help users to specify the objective function in a more natural way. We summarize these functions and their usages in Table 1.

For the class `mymodel` created in Listing 1, we can see that the objective function of (8) is specified via the command:

```
mymodel.minimize(trace(X1) + trace(X2) + sum(X3));
```

4.4. Adding affine constraints into the model

Affine constraints can be specified and added into `mymodel` after the relevant variables have been declared. This is done via the function (method) `add_affine_constraint`. The following constraint types are supported in the interface:

- Equality constraints $==$
- Less-or-equal inequality constraints $<=$
- Greater-or-equal inequality constraints $>=$

where the expressions on both the left and right-hand sides of the operands must be affine expressions. Strict inequalities $<$ and $>$ are **not** accepted. Inequality and equality constraints are applied in an elementwise fashion, matching the behaviour of MATLAB itself. For instance, if `U` and `X` are $m \times n$ matrices, then `X <= U` is interpreted as mn (scalar) inequalities `X(i,j) <= U(i,j)` for all $i = 1, \ldots, m, j = 1, \ldots, n$. When one side is a scalar and the other side is a variable, that value is replicated; for instance, `X >= 0` is interpreted as `X(i,j) >= 0` for all $i = 1, \ldots, m, j = 1, \ldots, n$.

In general, affine constraints have the following form

$$\mathcal{A}_1 * \mathtt{X}_1 + \mathcal{A}_2 * \mathtt{X}_2 + \cdots + \mathcal{A}_k * \mathtt{X}_k <= (>= \text{or} ==) \, b, \tag{9}$$

where $\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_k$ are declared variables, b is a constant matrix or vector, and $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ are linear maps whose descriptions will be given shortly.

Next, we illustrate how to add affine constraints into the model object `mymodel` in detail.

4.4.1. General affine constraints

In this section, we show users how to initialize the linear maps $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ in (9).

- If $\mathcal{A}_i = a_i$, is a scalar, then $a_i * \mathtt{X_i}$ has the same dimension as the variable $\mathtt{X_i}$.
- If $\mathtt{X_i}$ is an n-dimensional vector, then $\mathcal{A}_i$ must be a $p \times n$ constant matrix, and $\mathcal{A}_i * \mathtt{X_i}$ is in $\mathbb{R}^p$.
- If $\mathtt{X_i}$ is an $m \times n$ $(n > 1)$ matrix, then $\mathcal{A}_i * \mathtt{X_i}$ is interpreted as a linear map such that

$$\mathcal{A}_i * \mathtt{X_i} = \begin{bmatrix} \left\langle A_1^{(i)}, \mathtt{X_i} \right\rangle \\ \vdots \\ \left\langle A_p^{(i)}, \mathtt{X_i} \right\rangle \end{bmatrix} \in \mathbb{R}^p, \tag{10}$$

  where $A_1^{(i)}, \ldots, A_p^{(i)}$ are given $m \times n$ constant matrices. In this case, $\mathcal{A}_i$ is a $p \times 1$ constant cell array such that

$$\mathcal{A}_i\{j\} = A_j^{(i)}, \quad j = 1, \ldots, p.$$

4.4.2. Coordinate-wise affine constraints

Although users can model coordinate-wise affine constraints in the general form given in (9), we allow users to declare them in a more direct way as follows:

$$a_1 * \mathtt{X}_1(i_1, j_1) + a_2 * \mathtt{X}_2(i_2, j_2) + \cdots + a_k * \mathtt{X}_k(i_k, j_k) <= (>= \text{or} ==) \, b, \tag{11}$$

where $a_1, a_2, \ldots, a_k, b$ are scalars and $\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_k$ are declared variables. The index pairs $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ extract the corresponding elements in the variables. From Listing 1, we can see how a constraint of the form (11) is added, i.e. `mymodel`.add_affine_constraint$(2 * \mathtt{X}_1(2,3) + \mathtt{X}_2(4,2) - \mathtt{X}_3(4) == 3)$

Our interface also allows users to handle multiple index pairs. For example, if we have a declared variable $\mathtt{X} \in \mathbb{R}^{m \times n}$ and two index arrays

$$I = [i_1, i_2, \ldots, i_k], \quad J = [j_1, j_2, \ldots, j_k],$$

where $\max\{i_1, i_2, \ldots, i_k\} \leq m$ and $\max\{j_1, j_2, \ldots, j_k\} \leq n$, then $\mathtt{X}(I, J)$ is interpreted as

$$\mathtt{X}(I, J) = \begin{bmatrix} \mathtt{X}(i_1, j_1) \\ \mathtt{X}(i_2, j_2) \\ \vdots \\ \mathtt{X}(i_k, j_k) \end{bmatrix} \in \mathbb{R}^k.$$

An example of such a usage can be found in Listing 9.

Table 2. Supported predefined maps.

Function	Description	Dimension
inprod(C, X)	The inner product of a constant vector or matrix C and a variable X of the same dimension.	1×1
trace(X)	The trace of a square matrix variable X.	1×1
sum(X)	The sum of all elements of a vector or matrix variable X.	1×1
l1_norm(X)	The ℓ_1 norm of a variable X.	1×1
l1_norm($\mathcal{A}$*X + b)	The ℓ_1 norm of an affine expression.	1×1
map_diag(X)	Extract the main diagonal of an $n \times n$ matrix variable X.	$n \times 1$
map_svec(X)	For an $n \times n$ symmetric variable X, it returns the corresponding symmetric vectorization of X, as defined in (3).	$\frac{n(n+1)}{2} \times 1$
map_vec(X)	For a $m \times n$ matrix variable X, it returns the vectorization of X.	$mn \times 1$

4.4.3. Element-wise multiplication

In our interface, we also support element-wise multiplication $(\cdot*)$ between a declared variable X and a constant matrix A with the same dimension. Suppose

$$\mathsf{X} = \begin{bmatrix} \mathsf{X}_{11} & \cdots & \mathsf{X}_{1n} \\ \vdots & \ddots & \vdots \\ \mathsf{X}_{m1} & \cdots & \mathsf{X}_{mn} \end{bmatrix}, \quad A = \begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mn} \end{bmatrix}.$$

Then $A \cdot *\mathsf{X}$ is interpreted as

$$A \cdot *\mathsf{X} = \begin{bmatrix} A_{11} * \mathsf{X}_{11} & \cdots & A_{1n} * \mathsf{X}_{1n} \\ \vdots & \ddots & \vdots \\ A_{11} * \mathsf{X}_{m1} & \cdots & A_{mn} * \mathsf{X}_{mn} \end{bmatrix}.$$

4.4.4. Specifying affine constraints using predefined maps

For convenience, we also provide some predefined maps to help users to specify constraints in a more direct way. We summarize these maps and their usages in Table 2.

4.4.5. Chained constraints

In our interface, one can add chained inequalities into the created `ccp_model mymodel`. In general, chained affine constraints have the form

$$\mathsf{L} <= \mathcal{A}_1 * \mathsf{X}_1 + \mathcal{A}_2 * \mathsf{X}_2 + \cdots + \mathcal{A}_k * \mathsf{X}_k <= \mathsf{U},$$

where L and U are scalars or constant matrices with having the same dimensions as the affine expression in the middle. As an example, one can add bound constraints for a declared variable X via the command:

```
mymodel.add_affine_constraint(L <= X <= U);
```

It is important to note that in chained inequality constraints, the affine expression in the middle should only contain declared variables but not constants.

4.5. Adding positive semidefinite constraints into the model

Positive semidefinite constraints can be added into a previously created object `mymodel` using the function (method) `add_psd_constraint`. Such a constraint is valid only

for a declared symmetric variable or positive semidefinite variable. In general, a positive semidefinite constraint has the form

$$a_1 * \mathtt{X}_1 + a_2 * \mathtt{X}_2 + \cdots + a_k * \mathtt{X}_k \succeq \mathtt{G}, \tag{12}$$

where $a_1, a_2, \ldots, a_k$ are scalars, and $\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_k$ are declared variables in symmetric matrix spaces or PSD cones, and G is a constant symmetric matrix. Note that one can also have the version '$\preccurlyeq$' in (12). We can add (12) into `mymodel` as follows:

`mymodel.add_psd_constraint(`$a_1 * \mathtt{X}_1 + \cdots + a_k * \mathtt{X}_k >= \mathtt{G}$`)`

Specially,

- For a variable $\mathtt{X} \in \mathbb{S}^n$, one can use `mymodel.add_psd_constraint(X>=0)` to specify the constraint $\mathtt{X} \succeq 0$ or $\mathtt{X} \in \mathbb{S}^n_+$.
- For a variable $\mathtt{X} \in \mathbb{S}^n$ and a constant matrix $\mathtt{G} \in \mathbb{S}^n$. One can use `mymodel.add_psd_constraint(X >= G)` and `mymodel.add_psd_constraint(X <= G)` to specify the constraint $\mathtt{X} \succeq \mathtt{G}$ and $\mathtt{X} \preceq \mathtt{G}$, respectively.

Similar to affine constraints, one can also use chained positive semidefinite constraints together. For example, for a variable $\mathtt{X} \in \mathbb{S}^n$ and two constant matrices $\mathtt{G1}, \mathtt{G2} \in \mathbb{S}^n$ ($\mathtt{G1} \preceq \mathtt{G2}$), one can specify $\mathtt{G1} \preceq \mathtt{X} \preceq \mathtt{G2}$ as

```
mymodel.add_psd_constraint(G1 <= X <= G2);
```

4.6. Setting parameters for SDPNAL+

As described in Section 2.4, there are mainly nine parameters in the parameter structure array `OPTIONS`. To allow users to set these parameters freely, we provide the function (method) `setparameter` for such a purpose. Parameters which are not specified are set to be the default values described in Section 2.4. Now, we describe the usage of `setparameter` in details.

Assume that we have created a `ccp_model` class called `mymodel`. Since `setparameter` is a method in the `ccp_model` class, so the usage of `setparameter` is simply

```
mymodel.setparameter('para_name',value)
```

In Table 3, we summarize the parameters which can be set in `setparameter`. Note that users can set more than one parameters at a time. For example, one can use

```
mymodel.setparameter('tol', 1e-4, 'maxiter', 2000);
```

to set the parameters `tol = 1e-4` and `maxiter = 2000`.

4.7. Solving a model and extracting solutions

After creating and initializing the class `mymodel`, one can call the method **solve** to solve the model as follow:

```
mymodel.solve
```

Table 3. Usage of `setparameter`.

Parameter Name	Usage	Default Value
`tol`	`mymodel.setparameter('tol', value)`	1e−6
`maxiter`	`mymodel.setparameter('maxiter', value)`	20000
`maxtime`	`mymodel.setparameter('maxtime', value)`	10000
`tolADM`	`mymodel.setparameter('tolADM', value)`	1e−4
`maxiterADM`	`mymodel.setparameter('maxiterADM', value)`	200
`printlevel`	`mymodel.setparameter('printlevel', value)`	1
`stopoption`	`mymodel.setparameter('stopoption', value)`	1
`AATsolve.method`	`mymodel.setparameter('AATsolve.method', value)`	'direct'
`BBTsolve.method`	`mymodel.setparameter('BBTsolve.method', value)`	'iterative'

After solving the SDP problem, one can extract the optimal solutions using the function **get_value**. For example, if `X1` is a declared variable, then one can extract the optimal value of `X1` by setting

```
get_value(X1)
```

Note that the input of the function `get_value` should be a declared variable.

4.8. Further remarks on the interface

Here we give some remarks to help users to input an SDP problem into our interface more efficiently.

- If a variable must satisfy a conic constraint, it would be more efficient to specify the conic constraint when declaring the variable rather than declaring the variable and imposing the constraint separately. For example, it is better to use `X = var_nn(m,n)` to indicate that the variable $\mathtt{X} \in \mathbb{R}^{m\times n}$ must be in the cone $\mathbb{R}^{m\times n}_+$ rather than separately declaring `X = var_free(m,n)` followed by setting
  `mymodel.add_affine_constraint(X >= 0);`
  Similarly, if a square matrix variable $\mathtt{Y} \in \mathbb{S}^n$ must satisfy the conic constraint that $\mathtt{Y} \in \mathbb{S}^n_+$, then it is better to declare it as `Y = var_sdp(n,n)` rather than separately declaring `Y = var_free(n,n)` followed by setting
  `mymodel.add_psd_constraint(Y >= 0);`
  The latter option is not preferred because we have to introduce extra constraints.
- When there is a large number of affine constraints, specifying them using a loop in MATLAB is generally time consuming. To make the task more efficient, if possible, always try to model the problem using our predefined functions

5. Examples on building SDP models using our interface

To solve SDP problems using SDPNAL+, the user must input the problem data corresponding to the form in (P). The file `SDPNALplusDemo.m` contains a few examples to illustrate how to generate the data of an SDP problem in the required format. Here we will present a few of those examples in detail. Note that the user can also store the problem data in either the SDPA or SeDuMi format, and then use the m-files to read `sdpa.m` or `sedumi.m` to convert the data for SDPNAL+.

We also illustrate how the SDP problems can be coded using our basic interface.

5.1. SDPs arsing from the nearest correlation matrix problems

To obtain a valid nearest correlation matrix (NCM) from a given incomplete sample correlation matrix $G \in \mathbb{S}^n$, one version of the NCM problem is to consider solving the following SDP:

$$\text{(NCM)} \quad \min \left\{ \|H \circ (X - G)\|_1 \mid \text{diag}(X) = e,\ X \in \mathbb{S}^n_+ \right\},$$

where $H \in \mathbb{S}^n$ is a nonnegative weight matrix and '$\circ$' denotes the elementwise product. Here for any $M \in \mathbb{S}^n$, $\|M\|_1 = \sum_{i,j=1}^n |M_{ij}|$.

In order to express (NCM) in the form given in (P), we first write

$$\text{svec}(X) - \text{svec}(G) = x_+ - x_-,$$

where x_+ and x_- are two nonnegative vectors in $\mathbb{R}^{\bar{n}}$ ($\bar{n} = n(n+1)/2$). Then (NCM) can be reformulated as the following SDP with $m = n + \bar{n}$ equality constraints:

$$\begin{aligned} \min \quad & \langle \text{svec}(H), x_+ \rangle + \langle \text{svec}(H), x_- \rangle \\ \text{s.t.} \quad & \text{diag}(X) = e, \\ & \text{svec}(X) - x_+ + x_- = \text{svec}(G),\ X \in \mathbb{S}^n_+,\ x_+, x_- \in \mathbb{R}^{\bar{n}}_+. \end{aligned} \tag{13}$$

Given $G, H \in \mathbb{S}^n$, the SDP data for the above problem can be coded for SDPNAL+ as follows.

Listing 3. Generating the SDPNAL data for the NCM problem (13).

```
blk{1,1} = 's'; blk{1,2} = n;
n2 = n*(n+1)/2;
II = speye(n2); hh = svec(blk(1,:),H);

for k=1:n; Acell{k} = spconvert([k,k,1;n,n,0]); end
Atmp = svec(blk(1,:),Acell,1);
At{1,1} = [Atmp{1}, II];
At{2,1} = [sparse(n,n2), sparse(n,n2); -II, II]';

b = [ones(n,1); svec(blk(1,:),G)];
C{1,1} = sparse(n,n); C{2,1} = [hh; hh];
```

For more details, see the m-file `NCM.m` in the subdirectory `/util`.

Next, we show how to use our interface to solve the nearest correlation matrix problem (NCM). Given a data matrix $G \in \mathbb{S}^n$, we can solve the corresponding NCM problem using our interface as follows.

Listing 4. Solving a NCM problem with our interface.

```
 n = 100;
 G = randn(n,n);  G = 0.5*(G + G');
 H = rand(n);  H = 0.5*(H+H');
 model = ccp_model('Example_NCM');
    X = var_sdp(n,n);
    model.add_variable(X);
    model.minimize(l1_norm(H.*X - H.*G));
    model.add_affine_constraint(map_diag(X) == ones(n,1));
    model.setparameter('tol', 1e-6, 'maxiter', 2000);
 model.solve;
 Xval = get_value(X);
 dualinfo = get_dualinfo(model);
```

The last two lines in Listing 4 illustrate how we can extract the numerical value of the variable X and also the corresponding dual variables. Observe that with the help of our interface, users can input the problem into our solver very easily; see `Example_NCM.m` for more details.

5.2. SDP relaxations of the maximum stable set problems

Let G be an undirected graph with n nodes and edge set $\mathcal{E}$. Its stability number, $\alpha(G)$, is the cardinality of a maximal stable set of G, and it can be expressed as

$$\alpha(G) := \max\{e^T x : x_i x_j = 0,\ (i,j) \in \mathcal{E},\ x \in \{0,1\}^n\},$$

where $e \in \mathbb{R}^n$ is the vector of all ones. It is known that computing $\alpha(G)$ is NP-hard. But an upper bound $\theta(G)$, known as the Lovász theta number [9], can be computed as the optimal value of the following SDP problem:

$$\theta(G) := \max\left\{\left\langle ee^T, X\right\rangle \;\middle|\; \left\langle E^{ij}, X\right\rangle = 0\ \forall\ (i,j) \in \mathcal{E},\ \langle I, X\rangle = 1,\ X \in \mathbb{S}^n_+\right\}, \tag{14}$$

where $E^{ij} = e_i e_j^{\mathrm{T}} + e_j e_i^{\mathrm{T}}$ and e_i denotes the ith standard unit vector of $\mathbb{R}^n$. One can further tighten the upper bound to get $\alpha(G) \le \theta_+(G) \le \theta(G)$, where

$$\theta_+(G) := \max\left\{\left\langle ee^T, X\right\rangle \;\middle|\; \left\langle E^{ij}, X\right\rangle = 0\ \forall\ (i,j) \in \mathcal{E},\ \langle I, X\rangle = 1,\ X \in \mathbb{S}^n_+,\ X \ge 0\right\}. \tag{15}$$

In the subdirectory `/datafiles` of SDPNAL+, we provide a few SDP problems with data stored in the in SDPA or SeDuMi format, arising from computing $\theta(G)$ for a few graph instances. The segment below illustrates how one can solve the SDP problem, `theta8.dat-s`, to compute $\theta_+(G)$:

```
>> [blk,At,C,b] = read_sdpa('theta8.dat-s');
>> L = 0;
>> [obj,X,s,y,S,Z,ybar,v,info,runhist]
= sdpnalplus(blk,At,C,b,L);
```

To compute $\theta(G)$, one can simply set `L = []` to indicate that there is no lower bound constraint on X. In Listing 5, we illustrate how to use our interface to solve the θ_+ problem (15).

Listing 5. Solving the θ_+ problem (15) using our interface.

```
load theta6.mat
[IE,JE] = find(triu(G,1));
n = length(G);
model = ccp_model('Example_theta');
    X = var_sdp(n,n);
    model.add_variable(X);
    model.maximize(sum(X));
    model.add_affine_constraint(trace(X) == 1);
    model.add_affine_constraint(X(IE,JE) == 0);
    model.add_affine_constraint(X >= 0);
model.solve;
```

5.3. SDPs arising from the frequency assignment problems

Given a network represented by a graph G with n nodes and an edge set $\mathcal{E}$ together with an edge-weight matrix W, a certain type of frequency assignment problem on G can be relaxed into the following SDP (see [1, equation (5)]):

$$\begin{aligned} \text{(FAP)} \quad \max \quad & \left\langle \left(\frac{k-1}{2k}\right)\mathcal{L}(G,W) - \frac{1}{2}\mathrm{Diag}(We), X \right\rangle \\ \text{s.t.} \quad & \mathrm{diag}(X) = e,\ X \in \mathbb{S}^n_+, \\ & \left\langle -E^{ij}, X \right\rangle = 2/(k-1)\ \forall\ (i,j) \in \mathcal{U} \subseteq \mathcal{E}, \\ & \left\langle -E^{ij}, X \right\rangle \le 2/(k-1)\ \forall\ (i,j) \in \mathcal{E} \setminus \mathcal{U}, \end{aligned} \tag{16}$$

where $k > 1$ is a given integer, $\mathcal{U}$ is a given subset of $\mathcal{E}$, $\mathcal{L}((G,W) := \mathrm{Diag}(We) - W$ is the Laplacian matrix, $E^{ij} = e_i e_j^T + e_j e_i^T$. Note that (16) is equivalent to

$$\begin{aligned} \max \quad & \left\langle \left(\frac{k-1}{2k}\right)\mathcal{L}(G,W) - \frac{1}{2}\mathrm{Diag}(We), X \right\rangle \\ \text{s.t.} \quad & \mathrm{diag}(X) = e,\ X \in \mathbb{S}^n_+,\ L \le X \le U, \end{aligned} \tag{17}$$

where

$$L_{ij} = \begin{cases} -\dfrac{1}{k-1} & \forall (i,j) \in \mathcal{E}, \\ -\infty & \text{otherwise}, \end{cases} \qquad U_{ij} = \begin{cases} -\dfrac{1}{k-1} & \forall (i,j) \in \mathcal{U}, \\ \infty & \text{otherwise}. \end{cases}$$

Next, we show how to use our interface to solve the SDP problem (16). Assume that we have already computed the constant matrix $C := ((k-1)/2k)\mathcal{L}(G,W) - \frac{1}{2}\mathrm{Diag}(We)$ and saved it as `C` in the current workspace. Suppose `IU`, `JU` are two single column arrays storing the index pairs (i,j) corresponding to $\mathcal{U}$, and `IE`, `JE` are two single column arrays storing the index pairs (i,j) corresponding to $\mathcal{E}$. Assume that `IU`, `JU`, `IE`, `JE`, `n`, `kpara` are already stored in the current workspace. We can build the `ccp_model` for (16) using our interface as follows. More details can be seen in `Example_FAP.m`.

Listing 6. Solving the FAP (16) using our interface.

```
model = ccp_model('Example_FAP');
   X = var_sdp(n,n);
   model.add_variable(X);
   model.maximize(inprod(C,X));
   model.add_affine_constraint(map_diag(X) == ones(n,1));
   const = -1/(kpara-1);
   model.add_affine_constraint(X(IU,JU) == const);
   model.add_affine_constraint(X(IE,JE) >= const);
model.solve;
```

One can also solve (FAP) using the equivalent formulation specified in (17). Assume that the matrices `L`, `U`, `C` and `n` have been computed in the current workspace, we can input the SDP problem (17) into our interface based on the above equivalent form as follows.

Listing 7. Solving the reformulated FAP (17).

```
model = sdp_model('Example_FAP2');
  X = var_sdp(n,n);
  model.add_variable(X);
  model.maximize(inprod(C,X));
  model.add_affine_constraint(map_diag(X) == ones(n,1));
  model.add_affine_constraint(L <= X <= U);
model.solve;
```

5.4. SDPs arising from Euclidean distance matrix problems

Consider a given undirected graph G with n nodes and edge set $\mathcal{E}$. Let $D = (d_{ij}) \in \mathbb{S}^n$ be a matrix whose elements are such that $d_{ij} > 0$ if $(i,j) \in \mathcal{E}$, and $d_{ij} = 0$ if $(i,j) \notin \mathcal{E}$. We seek points $x_1, x_2, \ldots, x_n$ in $\mathbb{R}^d$ such that $\|x_i - x_j\|$ is as close as possible to d_{ij} for all $(i,j) \in \mathcal{E}$. In particular, one may consider minimizing the L_1-error as follows:

$$\min \left\{ \sum_{(i,j)\in\mathcal{E}} |d_{ij}^2 - \|x_i - x_j\|^2| - \frac{\alpha}{2n} \sum_{i,j=1}^{n} \|x_i - x_j\|^2 \;\middle|\; \sum_{i=1}^{n} x_i = 0,\ x_1, \ldots, x_n \in \mathbb{R}^d \right\},$$

where the equality constraint is introduced to put the centre of mass of the points at the origin. The second term in the objective function is introduced to achieve the effect of spreading out the points instead of crowding together, and α is a given nonnegative parameter. Let $X = [x_1, \ldots, x_n] \in \mathbb{R}^{d\times n}$. Then $\|x_i - x_j\|^2 = e_{ij}^T X^T X e_{ij}$, where $e_{ij} = e_i - e_j$. The above nonconvex problem can be rewritten as (for more details, see [6]):

$$\min \quad \left\{ \sum_{(i,j)\in\mathcal{E}} |d_{ij}^2 - \left\langle e_{ij}e_{ij}^T, Y \right\rangle| - \alpha \langle I, Y\rangle \;\middle|\; \langle E, Y\rangle = 0,\ Y = X^T X,\ X \in \mathbb{R}^{d\times n} \right\}.$$

By relaxing the nonconvex constraint $Y = X^T X$ to $Y \in \mathbb{S}_+^n$, we obtain the following SDP problem:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in\mathcal{E}} x_{ij}^+ + x_{ij}^- - \alpha \langle I, Y\rangle \\ \text{s.t.} \quad & \left\langle e_{ij}e_{ij}^T, Y \right\rangle - x_{ij}^+ + x_{ij}^- = d_{ij}^2 \ \forall\, (i,j) \in \mathcal{E}, \\ & \langle E, Y\rangle = 0, \\ & Y \in \mathbb{S}_+^n,\ x_{ij}^+, x_{ij}^- \geq 0 \ \forall\, (i,j) \in \mathcal{E}. \end{aligned} \tag{18}$$

Note that the number of the equality constraints in (18) is $|\mathcal{E}| + 1$, and that the problem does not satisfy the Slater's condition because of the constraint $\langle E, Y\rangle = 0$. The problem (18) is typically highly degenerate and the optimal solution is not unique, which may result in high sensitivity to small perturbations in the data matrix D. Hence, the problem (18) can usually only be solved by SDPNAL+ to a moderate accuracy tolerance, say `OPTIONS.tol` $= 10^{-4}$. Given the data matrix $D \in \mathbb{S}^n$, and let $m = |\mathcal{E}|$, the SDP data for (18) can be coded as follows:

Listing 8. Generating the SDPNAL data for the EDM problem (18).

```
blk{1,1} = 's'; blk{1,2} = n;
Acell = cell(1,m+1); b = zeros(m+1,1); cnt = 0;
for i = 1:n
   for j = 1:n
      if (D(i,j) ~= 0)
         cnt = cnt + 1;
         Acell{cnt} = spconvert([i,i,1; i,j,-1; j,i,-1; j,j,1; n,n,0]);
         b(cnt) = D(i,j)^2;
      end
   end
end
Acell{m+1} = ones(n);
At(1) = svec(blk(1,:),Acell); C{1,1} = -alpha*speye(n,n);
blk{2,1} = 'l'; blk{2,2} = 2*m;
At{2,1} = [-speye(m), speye(m); sparse(1,2*m)]'; C{2,1} = ones(2*m,1);
```

Next, we show how to solve the EDM problem (18) using our interface. Assume that we have generated the data matrix $D \in \mathbb{S}^n$ such that $D_{ij} = d_{ij}$ for all $(i,j) \in \mathcal{E}$, and stored it in `data_randEDM.mat` together with a given α. As mentioned above, we set the accuracy tolerance to solve the problem as `1e-4`. Now we can input the SDP problem into our interface as follows.

Listing 9. Solving the EDM problem (18) using our interface.

```
load data_randEDM;
[ID, JD, val] = find(D);
dd = val.^2;
n1 = length(D);
n2 = length(ID);

model = ccp_model('Example_EDM');
  X1 = var_nn(n2,1);
  X2 = var_nn(n2,1);
  Y = var_sdp(n1,n1);
  model.add_variable(X1,X2,Y);
  model.minimize(sum(X1) + sum(X2) - alpha*trace(Y));
  model.add_affine_constraint(Y(ID,ID)+Y(JD,JD)-Y(ID,JD)-Y(JD,ID) -X1 +X2
       == dd);
  model.add_affine_constraint(sum(Y) == 0);
  model.setparameter('tol', 1e-4, 'maxiter', 2000);
model.solve;
```

5.5. SDPs arising from quadratic assignment problems

Let Π be the set of $n \times n$ permutation matrices. Given matrices $A, B \in \mathbb{R}^{n\times n}$, the associated quadratic assignment problem (QAP) is given by

$$v^*_{QAP} := \min\{\langle X, AXB\rangle : X \in \Pi\}. \tag{19}$$

For a matrix $X = [x_1, \ldots, x_n] \in \mathbb{R}^{n\times n}$, we will identify it with the n^2-dimensional vector $x = [x_1; \ldots; x_n]$. For a matrix $Y \in \mathbb{R}^{n^2\times n^2}$, we let Y^{ij} be the $n \times n$ block corresponding to $x_i x_j^T$ in the matrix xx^T. It is shown in [13] that v^*_{QAP} is bounded below by the following

number:

$$\begin{aligned} v := \min \quad & \langle B \otimes A, Y \rangle \\ \text{s.t.} \quad & \sum_{i=1}^{n} Y^{ii} = I, \ \left\langle I, Y^{ij} \right\rangle = \delta_{ij} \ \forall \ 1 \le i \le j \le n, \\ & \left\langle E, Y^{ij} \right\rangle = 1, \ \forall \ 1 \le i \le j \le n, \\ & Y \succeq 0, \ Y \ge 0, \end{aligned} \tag{20}$$

where E is the matrix of ones, and $\delta_{ij} = 1$ if $i = j$, and 0 otherwise. Note that there are $3n(n+1)/2$ equality constraints in (20). But two of them are actually redundant, and we remove them when solving the standard SDP generated from (20).

Now, we show an example of solving the SDP relaxation of the QAP problem `'chr12a'` via our interface.

Listing 10. Solving the SDP relaxation of a QAP with our interface.

```
problem_name = 'chr12a';
[A, B] = qapread(strcat(problem_name, '.dat'));
%% Construct C
Ascale = max(1, norm(A, 'fro'));
Bscale = max(1, norm(B, 'fro'));
A = A/Ascale; B = B/Bscale;
C = kron(B, A); C = 0.5*(C + C');
nn = length(C);
n = length(A);

model = ccp_model(problem_name);
  Y = var_sdp(nn, nn);
  model.add_variable(Y);
  model.minimize(inprod(C, Y));
  model.add_affine_constraint(Y >= 0);
  II = speye(n); EE = ones(n);
  for i = 1:n-1
     for j = i:n
        Eij = sparse(i,j,1,n,n);
        if (i==j) const = 1; else, const = 0; end
        model.add_affine_constraint(inprod(kron(II,Eij), Y) == const);
        model.add_affine_constraint(inprod(kron(Eij,II), Y) == const);
        model.add_affine_constraint(inprod(kron(Eij,EE), Y) == 1);
     end
  end
  model.add_affine_constraint(inprod(kron(II,sparse(n,n,1,n,n)), Y) == 1)
      ;
  model.setparameter('maxiter', 5000);
model.solve;
```

5.6. Comparison of our basic interface with CVX and YALMIP

As mentioned in the Introduction, our new interface is motivated by the need to facilitate the modelling of an SDP problem for SDPNAL+ to directly accept inequality constraints of the form '$l \le \mathcal{B}(X) \le u$', and bound constraints of the form '$L \le X \le U$' in addition to equality constraints of the form '$\mathcal{A}(X) = b$'.

For the interfaces CVX [4] and YALMIP [8], one will need to first reformulate a problem with the above mentioned inequality constraints into the standard primal SDP form (for interior-point solvers) by converting the inequality constraints into equality constraints

Table 4. Time taken (in seconds) to generate the SDP data (and the corresponding problem sizes) by various interfaces for the QAP problem (20) with matrices A,B of dimension $n \times n$. Here m is the final number of equality constraints in the generated SDP data, sblk is the dimension of the positive semidefinite matrix block, lblk is the dimension of the nonnegative vector, ublk is the dimension of the unrestricted vector.

n	CVX	YALMIP	Sdpnal+
10	9.22 $m = 5213$ sblk $= 100$, lblk $= 5050$	2.55 $m = 10100$ sblk $= 100$ lblk $= 5050$ ublk $= 10163$	0.49 $m = 163$ sblk $= 100$, lblk $= 5050$
15	448 $m = 25783$ sblk $= 225$, lblk $= 25425$	3.52 $m = 50850$ sblk $= 225$ lblk $= 25425$ ublk $= 50983$	0.67 $m = 358$ sblk $= 225$ lblk $= 25425$
20	took too long to run	9.86 $m = 160400$ sblk $= 400$ lblk $= 80200$ ublk $= 160628$	0.73 $m = 628$ sblk $= 400$ lblk $= 80200$

through introducing extra nonnegative variables as follows:

$$\mathcal{B}(X) - s^{(1)} = l, \mathcal{B}(X) + s^{(2)} = u,\ X - X^{(1)} = L,\ X + X^{(2)} = U,$$
$$s^{(1)} \geq 0,\ s^{(2)} \geq 0,\ X^{(1)} \geq 0,\ X^{(2)} \geq 0.$$

The above conversion not only will add significant overheads when generating the SDP data in CVX or YALMIP, a much more serious computational issue is that it has created a large number of additional equality constraints in the formulation which would cause huge computational inefficiency when solving the problem. Moreover, the large number of additional equality constraints introduced will likely make the SDP solver to encounter various numerical difficulties when solving the resulting SDP problem.

In Table 4, we present the relevant information for the SDP data generated by various interfaces for the QAP problem (20) with matrices A,B of dimensions $n \times n$. As one can observe, CVX took an exceeding long time to generate the data compared to YALMIP and Sdpnal+. When the problem dimension n becomes larger, the ratio of the times taken by YALMIP and Sdpnal+ to generate the data also grows larger, and the ratio is more than 13 for $n = 20$. More alarmingly, the number of equality constraints generated by CVX or YALMIP is exceedingly large. For $n = 20$, the ratio of the number of equality constraints generated by YALMIP and Sdpnal+ is more than 255 ($\approx 160400/628$) times. Such a huge number of equality constraints generated by CVX or YALMIP is fatal for the computational efficiency of interior-point solvers, and also disadvantageous for Sdpnal+.

6. Summary of the numerical performance of Sdpnal+

We have tested our solver Sdpnal+ on 745 SDP instances arising from various sources, namely,

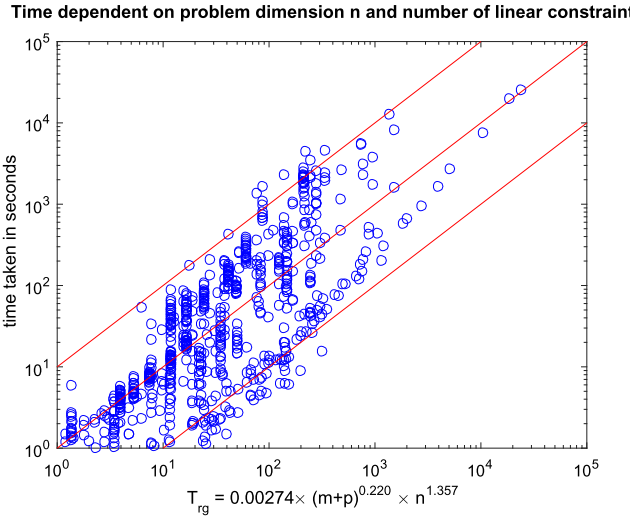

Figure 1. Time T taken to solve 707 SDP instances versus the times estimated based on regression $T_{\rm rg} = 0.00274\,(m+p)^{0.220}\,n^{1.357}$.

(1) 65 instances of DNN (doubly nonnegative) relaxation of maximum stable set problems from [14,17,19];
(2) 14 instances of SDP relaxation of frequency assignment problems (FAPs) [3];
(3) 94 instances of DNN relaxation of quadratic assignment problems (QAPs) [5];
(4) 165 instances of DNN relaxation of binary quadratic integer programming (BIQ) problems [22];
(5) 120 instances of DNN relaxation of clustering problems [12];
(6) 165 instances of DNN relaxation of BIQ problems with additional valid inequalities [16];
(7) 65 instances of SDP relaxation of maximum stable set problems from [14,17,19];
(8) 57 instances of SDP relaxation of best rank-one tensor approximation problems [11].

In total there are 623 SDP problems with simple polyhedral bound constraints on the matrix variable in addition to other linear constraints, and 122 standard SDP problems. The complete numerical results are available at

http://www.math.nus.edu.sg/ mattohkc/papers/SDPNALPtable-2017-Dec-18.pdf

Note that the results are obtained on a desktop computer having the following specification: Intel Xeon CPU E5-2680v3 @2.50 GHz with 12 cores, and 128GB of RAM. The extensive numerical experiments show that our SDPNAL+ solver is quite efficient and robust, in that it is able to solve 98.9% of the 745 instances of SDP problems arising from various applications listed above to the accuracy of less than 1.5×10^{-6} in the relative KKT residual η defined in (4).

In Figure 1, we plot the time T taken to solve a subset of 707 tested instances (with computation time of over one second each) versus the estimated times $T_{\rm rg} = 0.00274\,(m+p)^{0.220}\,n^{1.357}$, obtained based on the regression $\log_{10}(T) \approx \log_{10}(\kappa) + \alpha \log_{10}(m+p) + \beta \log_{10}(n)$. From the graph, one can observe that $T_{\rm rg}$ can estimate the actual time taken

Table 5. Summary of numerical results obtained by Sdpnal+ in solving 707 SDP problems (each with the computation time of more than one second). In each cell, the first number is the number of problems solved, and the second number is the average time taken to solve the problems. Here *K* means a thousand.

$m+p$	$\leq 1K$	$(1K, 4K]$	$(4K, 16K]$	$(16K, 64K]$	$(64K, 256K]$	$(256K, 1024K]$	$> 1024K$
$n \leq 100$	36 2.04	16 2.16	11 8.74				
$100 < n \leq 200$	101 6.29	4 9.48	43 25.67	43 61.57			
$200 < n \leq 400$	127 29.17	8 14.07	15 3.88	40 95.25	20 269.95		
$400 < n \leq 800$	44 134.97	14 603.19	8 55.52	16 34.75	14 22.62	15 2067.98	
$800 < n \leq 1600$	10 168.62	56 871.80	22 496.73	7 115.58	10 250.14	5 172.92	
$1600 < n \leq 3200$		1 2672.64		8 2867.77	1 439.11	5 2291.35	1 966.32
$n > 3200$			1 12817.94				5 11512.53

to within a factor of about 20 for a given $(m + p, n)$. If we contrast the dependent of T_{rg} on $(m + p, n)$ with the $O((m + p)^2 n^2) + O((m + p)n^3) + O((m + p)^3)$ time complexity in an interior-point method such as those implemented in SDPT3 or SeDuMi, then we can immediately observe that the time complexity of SDPNAL+ is much better. In particular, the dependence on the number of linear constraints is only $(m + p)^{0.22}$ for a given matrix dimension n. This also explains why our solver can be so efficient in solving an SDP problem with a large number of linear constraints.

In Table 5, we give a summary of the numerical results obtained for the subset of 707 SDP problems mentioned in the last paragraph. Note that in the table, $m+p$ is the total number of linear constraints as specified by $\mathcal{A}$ and $\mathcal{B}$. The simple polyhedral bound constraints on the matrix variable are not counted in $m+p$. Thus even if $m+p$ is a modest number, say less than 1000, the number of actual polyhedral constraints in the problem can still be large. Observe that across each row in the table, the average time taken to solve the problems with different number of linear constraints does not depend strongly on $m+p$. However, across each column in the table, the dependence of the average time taken to solve the problems on the matrix dimension n is more significant, but it is still much weaker than the cubic exponent dependent on the matrix dimension.

7. Conclusion and future works

SDPNAL+ is designed to be a general purpose software for solving large scale SDP problems with bound constraints as well as having a large number of equality and/or inequality constraints. The solver has been demonstrated to be fairly robust and highly efficient in solving various classes of SDP problems arising from the relaxation of combinatorial optimization problems such as maximum stable set problems, quadratic assignment problems, frequency assignment problems, binary quadratic integer programming problems. It has also worked well on SDP problems arising from the relaxation of robust clustering problems, rank-one tensor approximation problems, as well as problems arising from electronic structure calculations in quantum chemistry.

Our solver is expected to work well on nondegenerate well-posed SDP problems, but much more future work must be done to make the solver to work well on degenerate and/or ill-posed problems. Currently our solver is not catered to problems with SOCP or exponential cone constraints. As an obvious extension, we are currently extending the solver to handle problems with the aforementioned cone constraints.

We have also designed a basic user friendly interface for the user to input their SDP model into the solver. One of our future works is to expand the flexibility and capability of the interface such as the ability to handle Hermitian matrices.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The research of first author is partially supported by a start-up research grant from the Hong Kong Polytechnic University. The research of second author is supported in part by the Ministry of Education, Singapore, Academic Research Fund under Grant R-146-000-256-114. The research of fourth author was supported by the National Natural Science Foundation of China under projects No. 11871002 and General projects of the Science and Technology Program of the Beijing Municipal Education Commission.

References

[1] S. Burer, R.D. Monteiro, and Y. Zhang, *A computational study of a gradient-based log-barrier algorithm for a class of large-scale SDPs*, Math. Program. 95 (2003), pp. 359–379.
[2] L. Chen, D.F. Sun, and K.C. Toh, *An efficient inexact symmetric Gauss-Seidel based majorized ADMM for high-dimensional convex composite conic programming*, Math. Program. 161 (2017), pp. 237–270.
[3] A. Eisenblätter, M. Grötschel, and A.M. Koster, *Frequency planning and ramifications of coloring*, Discuss. Math. Graph Theory 22 (2002), pp. 51–88.
[4] M. Grant and S. Boyd, *CVX: Matlab software for disciplined convex programming, version 2.1*, 2014. Available at http://cvxr.com/cvx.
[5] P. Hahn and M. Anjos, *QAPLIB – A quadratic assignment problem library*. Available at http://www.seas.upenn.edu/qaplib.
[6] N.-H.Z. Leung and K.C. Toh, *An SDP-based divide-and-conquer algorithm for large scale noisy anchor-free graph realization*, SIAM J. Sci. Comput. 31 (2009), pp. 4351–4372.
[7] X.D. Li, D.F. Sun, and K.C. Toh, *A Schur complement based semi-proximal ADMM for convex quadratic conic programming and extensions*, Math. Program. 155 (2016), pp. 333–373.
[8] J. Löfberg, *Yalmip: A tool box for modeling and optimization in matlab*, 2004 IEEE International Symposium on Computer Aided Control Systems Design, 2004.
[9] L. Lovasz, *On the shannon capacity of a graph*, IEEE Trans. Inform. Theory 25 (1979), pp. 1–7.
[10] R. Monteiro, C. Ortiz, and B. Svaiter, *A first-order block-decomposition method for solving two-easy-block structured semidefinite programs*, Math. Program. Comput. 6 (2014), pp. 103–150.
[11] J. Nie and L. Wang, *Semidefinite relaxations for best rank-1 tensor approximations*, SIAM J. Matrix Anal. Appl. 35 (2014), pp. 1155–1179.
[12] J. Peng and Y. Wei, *Approximating k-means-type clustering via semidefinite programming*, SIAM J. Optim. 18 (2007), pp. 186–205.
[13] J. Povh and F. Rendl, *Copositive and semidefinite relaxations of the quadratic assignment problem*, Discrete Optim. 6 (2009), pp. 231–241.
[14] N. Sloane, *Challenge problems: Independent sets in graphs*, 2005. Available at http://www.research.att.com/njas/doc/graphs.html.

[15] J.F. Sturm, *Using SeDuMi 1.02, a Matlab toolbox for optimization over symmetric cones*, Optim. Methods Softw. 11 (1999), pp. 625–653.
[16] D.F. Sun, K.C. Toh, and L.Q. Yang, *A convergent 3-block semi-proximal alternating direction method of multipliers for conic programming with 4-type constraints*, SIAM J. Optim. 25 (2015), pp. 882–915.
[17] K.C. Toh, *Solving large scale semidefinite programs via an iterative solver on the augmented systems*, SIAM J. Optim. 14 (2004), pp. 670–698.
[18] K.C. Toh, M.J. Todd, and R.H. Tutuncu, *SDPT3 – A Matlab software package for semidefinite programming*, Optim. Methods Softw. 11 (1999), pp. 545–581.
[19] M. Trick, V. Chvatal, B. Cook, D. Johnson, C. McGeoch, and R. Tarjan, *The second DIMACS implementation challenge – NP hard problems: Maximum clique, graph coloring, and satisfiability*, 1992. Available at http://dimacs.rutgers.edu/Challenges/.
[20] R.H. Tutuncu, K.C. Toh, and M.J. Todd, *Solving semidefinite-quadratic-linear programs using SDPT3*, Math. Program. 95 (2003), pp. 189–217.
[21] Z. Wen, D. Goldfarb, and W. Yin, *Alternating direction augmented Lagrangian methods for semidefinite programming*, Math. Program. Comput. 2 (2010), pp. 203–230.
[22] A. Wiegele, *Biq mac library*, 2007. Available at http://biqmac.uni-klu.ac.at/biqmaclib.html.
[23] M. Yamashita, K. Fujisawa, and M. Kojima, *Implementation and evaluation of SDPA 6.0 (semidefinite programming algorithm 6.0)*, Optim. Methods Softw. 18 (2003), pp. 491–505.
[24] L.Q. Yang, D.F. Sun, and K.C. Toh, *SDPNAL+: A majorized semismooth Newton-CG augmented Lagrangian method for semidefinite programming with nonnegative constraints*, Math. Program. Comput. 7 (2015), pp. 331–366.
[25] X.-Y. Zhao, D.F. Sun, and K.C. Toh, *A Newton-CG augmented Lagrangian method for semidefinite programming*, SIAM J. Optim. 20 (2010), pp. 1737–1765.