Les Cahiers du GERAD ISSN: 0711–2440

IPNSDP: A solver for nonlinear semidefinite programming with automatic differentiation and chordal decomposition

A. G. Weldeyesus, M. F. Anjos

G–2026–06

February 2026

IPNSDP: A solver for nonlinear semidefinite programming with automatic differentiation and chordal decomposition

Alemseged G. Weldeyesus [a]

Miguel F. Anjos [a, b]

[a] *School of Mathematics and Maxwell Institute for Mathematical Sciences The University of Edinburgh, Edinburgh EH9 3FD, United Kingdom*

[b] *GERAD, Montréal (Qc), Canada, H3T 1J4*

njos@stanfordalumni.org

February 2026
Les Cahiers du GERAD
G–2026–06

Abstract : This paper introduces IPNSDP, a MATLAB solver for general Nonlinear Semidefinite Programming (NSDP) based on a second-order interior point method. Unlike linear SDP solvers, existing NSDP solvers often require users to provide difficult manual derivatives or rely on other modeling interfaces. IPNSDP addresses this by directly integrating with CasADi for automatic differentiation, significantly reducing user effort. For large-scale and structured sparse problems, it supports chordal decomposition, replacing large positive semidefinite constraints with smaller ones, enhanced by a merging strategy that combines submatrices with significant overlap to improve computational efficiency. By integrating automatic differentiation with optimized decomposition, IPNSDP offers a user-friendly scalable solution for a broad class of NSDP problems.

Keywords : Nonlinear semidefinite programming; interior point methods; tables; mathematics; chordal decomposition

Acknowledgements: We thank the School of Mathematics and the Maxwell Institute for Mathematical Sciences at the University of Edinburgh for providing institutional support and hosting for the project.

1 Introduction

This paper presents IPNSDP, a MATLAB-based general-purpose solver to tackle a broad class of Nonlinear Semidefinite Programming (NSDP) problems. IPNSDP implements a second-order primal-dual interior point method and leverages automatic differentiation from CasADi to automate the computation of all required derivatives. In addition, the solver exploits chordal decomposition to efficiently handle large-scale sparse and structured matrix inequalities, significantly enhancing computational scalability.

NSPD generalizes the well-established framework of linear semidefinite programming to accommodate nonlinear objectives and constraints. NSDP arises in diverse application areas, including finance [7], control theory [10], structural optimization [12, 27], polynomial optimization [20], machine learning [11] etc. However, the non-linear nature of these problems introduces challenges, particularly in derivative evaluation, which often involves a difficult matrix calculus.

While several robust and efficient solvers exist for linear semidefinite programming, such as SDPT3 [23], SeDuMi [22], and MOSEK [16], the tools and methods available for solving NSDPs remain relatively limited [31], although recent approaches based on sequential quadratic programming have been developed, for example, as proposed in [19]. In most cases, solvers for NSDPs such as PENNON [14] and PENLAB [4] have been used. These solvers are implemented using the augmented Lagrangian method and can handle a broad class of NSDPs. However, users are required to manually implement derivatives of nonlinear functions involving matrix calculus or to rely on interfaces such as YALMIP [15].

We developed IPNSDP as alternative for tackling NSDPs, particularly those exhibiting convexity. It combines a user-friendly MATLAB interface with CasADi [1] as automatic differentiation engine. Therefore, users are only required to define the objective and constraint functions. All relevant first- and second-order derivatives are computed internally. This substantially reduces the modeling complexity of NSDPs and simplifies the application of NSDP methods to practical problems.

Furthermore, IPNSDP employs a well-known and efficient chordal decomposition technique [5, 9, 24, 33] to transform positive semidefinite constraints on large, structured sparse matrices into multiple positive semidefinite constraints on smaller submatrices. Such structured and sparse NSDPs commonly arise in applications such as structural optimization [13, 27]. To further enhance the efficiency of this decomposition, a graph-based clique merging strategy [6] is utilized to combine submatrices with significant overlap. The overall processes of decomposition and merging are all customizable by the user through the options included in the solver. This combined approach substantially reduces the computational cost and improves the numerical tractability of many sparse and structured NSDPs.

The paper is organized as follows. Section 2 describes the general NSDP problem formulation addressed by IPNSDP. Section 3 explains the theoretical background and implementation of chordal decomposition and clique merging. The implemented primal-dual interior point method and its associated parameters are detailed in Section 4. Section 5 describes how to use the solver, including problem setup and customization options. Several examples from the literature are solved in Section 6. Finally, concluding remarks are provided in Section 7.

2 Problem formulation

The solver IPNSDP addresses a general NSDP problem of the form:

$$
\begin{aligned}
\min_{X_k \in \mathbb{S}^{d_k},\, u \in \mathbb{R}^n} \quad & f(X,u) \\
\text{subject to} \quad & \sum_{k=1}^{m_X} \langle A^{(k)}_{11,i}, X_k \rangle + A_{12,i} u = b_{1,i}, \quad i = 1,\dots,n_1 \\
& \sum_{k=1}^{m_X} \langle A^{(k)}_{21,j}, X_k \rangle + A_{22,j} u \le b_{2,j}, \quad j = 1,\dots,n_2 \\
& c_1(X,u) = 0 \\
& c_2(X,u) \le 0 \\
& \mathcal{A}_s(X,u) \succeq 0, \quad s = 1,\dots,n_{psd} \\
& \underline{u} \le u \le \bar{u} \\
& \underline{\rho}_k I_{d_k} \preceq X_k \preceq \bar{\rho}_k I_{d_k}, \quad k = 1,\dots,m_X.
\end{aligned}
\tag{1}
$$

The problem involves two sets of decision variables. The first one is a tuple of symmetric matrices $X = (X_1, \dots, X_{m_X})$, where each matrix X_k is in the space $\mathbb{S}^{d_k}$ and is constrained to be positive semidefinite. The constants $\underline{\rho}_k \ge 0$ and $\bar{\rho}_k$ define the lower and upper bound on the eigenvalues of X_k, and I_{d_k} denotes the $d_k \times d_k$ identity matrix. The other variable is a vector $u \in \mathbb{R}^n$ with $\underline{u}$ and $\bar{u}$ its lower and upper bounds. The objective function $f : \mathbb{S} \times \mathbb{R}^n \to \mathbb{R}$ and the nonlinear constraint functions $c_1 : \mathbb{S} \times \mathbb{R}^n \to \mathbb{R}^{q_1}$ and $c_2 : \mathbb{S} \times \mathbb{R}^n \to \mathbb{R}^{q_2}$ are assumed to be sufficiently smooth. The linear constraints involve the trace inner product,

$$
\langle A, X \rangle := \operatorname{trace}(AX). \tag{2}
$$

with symmetric coefficient matrices $A^{(k)}_{11,i} \in \mathbb{S}^{d_k}$ and $A^{(k)}_{21,j} \in \mathbb{S}^{d_k}$ for $i = 1,\dots,n_1$ and $j = 1,\dots,n_2$, respectively. Moreover, $A_{12,i} \in \mathbb{R}^{1\times n}$ and $A_{22,j} \in \mathbb{R}^{1\times n}$, and $b_{1,i}$ and $b_{2,j}$ are right hand side constants. Each map $\mathcal{A}_s : \mathbb{S} \times \mathbb{R}^n \to \mathbb{S}^{t_s}$ produces a symmetric positive matrix of size $t_s \times t_s$.

Remark 1. While Problem (1) includes explicit linear constraints, it may be simpler for the user to embed them within the nonlinear constraint function for IPNSDP.

Note that if we replace the objective function $f(X,u)$ with a linear function and remove the nonlinear constraints in Problem (1), the formulation reduces to a well-known primal linear SDP formulation:

$$
\begin{aligned}
\underset{X_k \in \mathbb{S}^{d_k},\, u \in \mathbb{R}^n}{\text{minimize}} \quad & \sum_{k=1}^{m_X} \langle C_k, X_k \rangle + c_u^T u \\
\text{subject to} \quad & \\
& \sum_{k=1}^{m_X} \langle A^{(k)}_{11,i}, X_k \rangle + A_{12,i} u = b_{1,i}, \quad i = 1,\dots,n_1 \\
& \sum_{k=1}^{m_X} \langle A^{(k)}_{21,j}, X_k \rangle + A_{22,j} u \le b_{2,j}, \quad j = 1,\dots,n_2 \\
& \underline{u} \le u \le \bar{u} \\
& \underline{\rho}_k I_{d_k} \preceq X_k \preceq \bar{\rho}_k I_{d_k}, \quad k = 1,\dots,m_X.
\end{aligned}
\tag{3}
$$

Although the linear SDP Problem (3) can be solved using IPNSDP, we generally recommend other efficient solvers specifically designed for linear SDPs. IPNSDP is primarily intended as an alternative solver for NSDPs.

3 Chordal decomposition and merging

This section provides a brief overview of the chordal decomposition and merging strategy implemented in IPNSDP. The decomposition technique transforms each positive semidefinite constraint on a large sparse matrices into several positive semidefinite constraints on smaller submatrices. The effectiveness of this decomposition technique can be improved by combining submatrices with significant overlap. For a detailed theoretical discussion of chordal decomposition and merging techniques, we refer the reader to [5, 6, 9, 24, 33].

We closely follow the presentation and content of Section 3 of [27].

The chordal decomposition originates from graph theory, where the relevant definitions are as follows. An undirected graph $G = (\mathcal{V}, \mathcal{E})$ is defined by a finite set of vertices $\mathcal{V} = \{1, \dots, n\}$ and a set of undirected edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, where $(\nu_i, \nu_j) \in \mathcal{E}$ implies $(\nu_j, \nu_i) \in \mathcal{E}$. A *path* in G is a sequence of vertices $\{\nu_1, \nu_2, \dots, \nu_k\} \subseteq \mathcal{V}$ such that the consecutive edges $(\nu_1, \nu_2), (\nu_2, \nu_3), \dots, (\nu_{k-1}, \nu_k)$ all belong to $\mathcal{E}$. A path $\{\nu_1, \nu_2, \dots, \nu_k\}$, with $k \geq 3$, forms a *cycle* if $(\nu_k, \nu_1) \in \mathcal{E}$, i.e., if the sequence begins and ends at the same vertex. A *chord* in a cycle is an edge connecting two nonconsecutive vertices of the cycle. A graph is said to be *chordal* if every cycle of length at least four $(k \geq 4)$ contains a chord. A *clique* $\mathcal{C} \subseteq \mathcal{V}$ is a subset of vertices such that every pair of distinct vertices in $\mathcal{C}$ is connected by an edge in $\mathcal{E}$. A clique is *maximal* if it is not strictly contained in any other clique of the graph. In sparse semidefinite programming, such a graph $\mathcal{G}$ is typically used to encode the sparsity pattern of a symmetric matrix, where vertices correspond to matrix indices and edges indicate potentially nonzero off-diagonal entries, so that the maximal cliques of a chordal graph correspond to dense principal submatrices in the chordal decomposition.

To apply the chordal decomposition technique to the NSDP (1), we first introduce a positive semidefinite matrix variable S_s as slack variable for the semidefinite inequality:

$$\mathcal{A}_s(X, u) - S_s = 0. \tag{4}$$

Note that the matrix S_s inherits the sparsity pattern of $\mathcal{A}_s(X, u)$, and it is assumed to be sparse and structured, such as in Figure 2b, for example. To employ chordal decomposition, the graph induced by the sparsity pattern of S_s must be chordal. In general, this sparsity pattern is typically not chordal in the first instance. In such cases, a chordal extension S_s^H is constructed by adding non-zero (i.e., edges) to ensure chordality. Finding a minimum chordal extension is NP-complete [32], hence heuristic methods are employed in practice. IPNSDP adopts a Cholesky factorization with zero fill-in, following the procedure described in [2], see Algorithm 1.

Algorithm 1 Chordal extension and decomposition

```
Input: The sparsity pattern of matrix $S_s$. This is generated by the CasADi symbolic representation of the matrix $\mathcal{A}_s(X, u)$.
Output: Maximal cliques $\mathcal{C}_1, \dots, \mathcal{C}_r$.
1: Construct the graph induced by the sparsity pattern of $S_s$.
2: Compute the Laplacian $A_s$ of the graph and update $A_s \leftarrow A_s + I$.
3: Compute the Cholesky factorization $PA_sP^T = LL^T$, where $P$ is obtained from approximate minimum degree
   ordering (a vertex permutation designed to reduce fill-in in sparse Cholesky factorization.)
4: Form the chordal extension $S_s^H \leftarrow P^T(L + L^T)P$.
5: Extract the maximal cliques $\mathcal{C}_1, \dots, \mathcal{C}_r$ (using MATLAB code based on [3]).
```

Assume now that the chordal extension S_s^H has a chordal sparsity graph $G(\mathcal{V}, \mathcal{E})$ with (possibly merged) maximal cliques $\mathcal{C}_1, \cdots, \mathcal{C}_r$. Then, the inequality in (4) can be reformulated as

$$\begin{aligned} &\mathcal{A}_s(X, u) - S_s^H = 0 \\ &S_s^H = \sum_{k=1}^{r} E_{\mathcal{C}_k}^T S_{s,k} E_{\mathcal{C}_k} \\ &S_{s,k} \succeq 0, \ k = 1, \cdots, r, \end{aligned} \tag{5}$$

where $S_{s,k} \in \mathbb{S}^{|\mathcal{C}_k|}$ are the smaller positive semidefinite matrices and $E_{\mathcal{C}_k}$ are the $|\mathcal{C}_k| \times n$ entry-selector matrices given by

$$(E_{\mathcal{C}_k})_{ij} = \begin{cases} 1, & \text{if } \mathcal{C}_k(i) = j \\ 0, & \text{otherwise}, \end{cases} \tag{6}$$

where $\mathcal{C}_k(i)$ is the i-th vertex. In IPNSDP implementation, S_s^H is eliminated and the problem is solved in the form

$$\begin{aligned} &\mathcal{A}_s(X,u) - \sum_{k=1}^{r} E_{\mathcal{C}_k}^T S_{s,k} E_{\mathcal{C}_k} = 0 \\ &S_{s,k} \succeq 0,\ k = 1, \cdots, r. \end{aligned} \tag{7}$$

In this case, the fill-ins introduced in the chordal extension S_s^H are enforced to be zero via equality constraints.

3.1 Clique merging

The Cholesky-based chordal extension method described in Algorithm 1 produces a matrix S_s^H that decomposes into many small cliques, often with significant overlaps. To improve efficiency, a clique merging strategy can be used to combine and produce fewer submatrices of marginally larger sizes. IPNSDP employs a clique graph-based merging technique [6]. In this framework, for each pair of cliques, the weight is first computed as

$$w_{ij} = |\mathcal{C}_i|^3 + |\mathcal{C}_j|^3 - |\mathcal{C}_i \cup \mathcal{C}_j|^3, \quad i,j \in \{1, \cdots, s\}.$$

Cliques corresponding to the largest positive weights are merged iteratively until all positive weights vanish. This process reduces the number of submatrices, for example, by approximately 50% for some of the test cases considered in [27], with only a marginal increase in their size.

3.2 Customizing decomposition and merging

The user has full control over the chordal decomposition and merging processes via solver settings. These options include

- Enabling or disabling chordal decomposition.
- Enabling or disabling merging of cliques.
- Setting the minimum block size for decomposition.
- Defining the maximum density allowed in blocks for decomposition.
- Specifying the expected reduction in matrix size.

All parameters can be configured through the solver interface. For complete details, see Appendix A.

4 Primal-dual interior point method

In this section, we briefly outline the primal-dual interior point framework employed in IPNSDP to solve the NSDP Problem (1). The framework follows standard and straightforward techniques for interior point methods (see e.g. [26, 28, 30]). The presentation and content closely follow Section 4 of [25]. The main goal of this section is to explicitly describe the parameters involved in the method and to explain how customizing these algorithmic parameters through the solver options may influence the convergence behaviour.

For clarity and notational simplicity, we write Problem (1) in a compact form where symmetric matrix variables are vectorized using the vectorization operator svec. Specifically, for every symmetric matrix $X_k \in \mathbb{S}^{d_k}$, we define the corresponding vector $x_k \in \mathbb{R}^{d_k(d_k+1)/2}$ such that

$$x_k = \operatorname{svec}(X_k),$$

where

$$\operatorname{svec}(X_k) := [\, X_{k,11},\, \sqrt{2}X_{k,21},\, \ldots,\, \sqrt{2}X_{k,d_k 1},\, X_{k,22},\, \sqrt{2}X_{k,32},\, \ldots,\, \sqrt{2}X_{k,d_k 2},\, \ldots,\, X_{k,d_k d_k}]^T.$$

Observe that $\operatorname{svec}(X_k)$ stacks the elements of X_k into a vector while preserving inner products

$$\langle A, X_k\rangle := \operatorname{trace}(AX_k) = \operatorname{svec}(A)\operatorname{svec}(X_k).$$

IPNSDP implicitly performs this transformation and carries out computations in the resulting space. However, to facilitate usability and reduce modeling effort, while still accommodating a broader class of NSDPs, the input is specified in the original form (2) involving the matrix X.

Next, let $x = (x_1^T, \cdots, x_{m_x}^T, u)$, $n_t = \sum_{k=1}^{m_x} \frac{d_k(d_k+1)}{2} + n$, $n_s = \sum_{s=1}^{n_{\text{psd}}} t_s + n_{bs}$ with n_{bs} the number of matrices X_k with finite upper bounds $\bar{\rho}_k$, and $n_g = n_2 + n_{bu}$ with n_{bu} the number of finite lower and upper bounds $\underline{u}, \bar{u}$ on u.

Then, formulation (2) can be written as

$$\begin{aligned} \underset{x\in\mathbb{R}^{n_t}}{\text{minimize}} \quad & f_0(x) \\ \text{subject to} \quad & h(x) = 0 \\ & g(x) \le 0 \\ & \mathcal{A}(x) \succeq 0, \end{aligned} \tag{8}$$

where $f_0 : \mathbb{R}^{n_t} \to \mathbb{R}$, $h : \mathbb{R}^{n_t} \to \mathbb{R}^{n_1}$ $g : \mathbb{R}^{n_t} \to \mathbb{R}^{n_2}$, and $\mathcal{A} : \mathbb{R}^{n_t} \to \mathbb{S}^{n_s}$ are twice continuously differentiable. To handle the inequality constraint $g(x) \le 0$, a slack variable $s \in \mathbb{R}_+^{n_g}$ is introduced such that

$$g(x) + s = 0, \quad s \ge 0.$$

Next, introducing the barrier parameter μ, the barrier subproblem is written as

$$\begin{aligned} \underset{x\in\mathbb{R}^{n_t},\, s\in\mathbb{R}^{n_g}}{\text{minimize}} \quad & f_0(x) - \mu \ln\det(\mathcal{A}(x)) - \mu \sum_{i=1}^{n_g} \ln(s_i) \\ \text{subject to} \quad & h(x) = 0, \quad g(x) + s = 0, \end{aligned} \tag{9}$$

and, the associated Lagrangian is given by

$$\mathcal{L}_\mu(x, s, \lambda, \nu) = f_0(x) - \mu \ln\det(\mathcal{A}(x)) - \mu \sum_{i=1}^{n_g} \ln(s_i) + \lambda^T h(x) + \nu^T (g(x) + s), \tag{10}$$

with multipliers $\lambda \in \mathbb{R}^{n_1}$, $\nu \in \mathbb{R}^{n_2}$. Defining the matrices

$$Z := \mu\mathcal{A}(x)^{-1}, \quad W := \mu S^{-1}, \quad S := \operatorname{diag}(s),$$

the KKT conditions for Problem (9) are written as

$$\nabla_x f_0(x) - \mathcal{G}(x)Z + \nabla h(x)^T\lambda + \nabla g(x)^T \nu = 0 \tag{11a}$$

$$-\mu S^{-1} e + \nu = 0 \tag{11b}$$

$$h(x) = 0 \tag{11c}$$

$$g(x) + s = 0 \tag{11d}$$
$$\mathcal{A}(x)Z - \mu I = 0 \tag{11e}$$
$$SWe - \mu e = 0, \tag{11f}$$

where e is a vector of one of appropriate size, and $\mathcal{G}(x)Z$ is defined as

$$\mathcal{G}(x)Z = \begin{pmatrix} \left\langle \frac{\partial \mathcal{A}(x)}{\partial x_1}, Z \right\rangle \\ \vdots \\ \left\langle \frac{\partial \mathcal{A}(x)}{\partial x_{n_t}}, Z \right\rangle \end{pmatrix}.$$

To apply Newton's method to solve the KKT conditions, symmetry is enforced by replacing $\mathcal{A}(x)Z = \mu I$ with

$$H_P(\mathcal{A}(x)Z) = \mu I$$

where

$$H_P(\mathcal{A}(x)Z) = \mu I, \quad H_P(Q) := \frac{1}{2}(PQP^{-1} + (PQP^{-1})^T).$$

The non-singular symmetric matrix P is chosen according to the Nesterov–Todd scaling [17, 18].

To compute the search direction $(\Delta x, \Delta s, \Delta\lambda, \Delta\nu, \Delta Z, \Delta w)$, we apply Newton's method is applied to the system (11):

$$\begin{aligned}
\nabla^2_{xx}\mathcal{L}_\mu \Delta x - \mathcal{G}(x)\Delta Z + \nabla h(x)^T \Delta\lambda + \nabla g(x)^T \Delta\nu &= -\nabla_x \mathcal{L}_\mu \\
\nabla h(x)\Delta x &= -h(x) \\
\nabla g(x)\Delta x + \Delta s &= -g(x) - s \\
S\Delta w + W\Delta s &= \mu e - SWe \\
\mathcal{E}\Delta X + \mathcal{F}\Delta Z &= \mu I - H_P(\mathcal{A}(x)Z),
\end{aligned} \tag{12}$$

where $\Delta X = \sum_{i=1}^{n_t} \Delta x_i \frac{\partial \mathcal{A}(x)}{\partial x_i}$.

Given a current iterate $(x, s, \lambda, \nu, Z, w)$ and search directions $(\Delta x, \Delta s, \Delta\lambda, \Delta\nu, \Delta Z, \Delta w)$, the primal step length $\tilde{\alpha}_p$ and dual step length $\tilde{\alpha}_d$ are determined as

$$\tilde{\alpha}_p = \max\{\alpha \in (0,1] : s + \alpha\Delta s \geq (1-\tau)s,\ w + \alpha\Delta w \geq (1-\tau)w\} \tag{13a}$$
$$\tilde{\alpha}_d = \max\{\alpha \in (0,1] : \nu + \alpha\Delta\nu \geq (1-\tau)\nu,\ Z + \alpha\Delta Z \succeq (1-\tau)Z\}, \tag{13b}$$

where $\tau \in (0,1)$ is the fraction-to-the-boundary parameter. The next iterate $(x^+, s^+, \lambda^+, \nu^+, Z^+, w^+)$ is determined by

$$x^+ = x + \gamma\tilde{\alpha}_p\Delta x, \quad s^+ = s + \gamma\tilde{\alpha}_p\Delta s, \quad w^+ = w + \gamma\tilde{\alpha}_p\Delta w$$
$$\lambda^+ = \lambda + \gamma\tilde{\alpha}_d\Delta\lambda, \quad \nu^+ = \nu + \gamma\tilde{\alpha}_d\Delta\nu, \quad {}^+Z = Z + \gamma\tilde{\alpha}_d\Delta Z,$$

where $\gamma \in (0,1)$ is a given constant.

The overall primal-dual interior point method is summarized in Algorithm 2. Given prescribed optimality tolerance $\epsilon^0 > 0$, feasibility tolerance $\epsilon^f > 0$, and maximum iteration count $k_{\max}$, the algorithm employs two nested stopping criteria.

The first criterion applies to the inner loop, i.e., for a fixed barrier parameter μ_k. The inner loop terminates when

$$\left(\epsilon^0_\mu < \max\left(\frac{\sigma}{0.1}\mu_k,\ \epsilon^0 - \mu_k\right) \quad \text{and} \quad \epsilon^f_\mu < \max\left(\frac{\sigma}{0.1}\mu_k,\ \epsilon^f\right)\right) \quad \text{or} \quad i > k^{\text{inner}}_{\max}. \tag{14}$$

Remark 2. In (14), ϵ^0_μ and ϵ^f_μ denote the optimality and feasibility residuals, respectively. They are computed as the $\|\cdot\|_\infty$-norm of the KKT conditions in (11), where ϵ^0_μ corresponds to $\mu = 0$ and ϵ^f_μ corresponds to $\mu = \mu_k$.

The maximum inner iteration count $k^{\text{inner}}_{\max}$ is set to 3 for the first three outer iterations, corresponding to large barrier parameter values where high accuracy is not required, otherwise, it is set to 10.

The second criteria applies to the outer loop. This loop terminates when

$$\left(\epsilon^0_\mu < \epsilon^0 \quad \text{and} \quad \epsilon^f_\mu < \epsilon^f\right) \quad \text{or} \quad \mu < \mu_{\min} \quad \text{or} \quad k_{\text{total}} > k_{\max}. \tag{15}$$

Note that if the outer iteration stops due to one of the last two conditions in (15), it indicates that the problem was not solved to the desired optimality.

Remark 3. IPNSDP allows penalization of constraints in the objective function to enhance numerical stability. For example, nonlinear inequalities in (8) can be penalized element-wise using $\rho \max\{0, g_i(x)\}$ or quadratic penalties. Regardless, the original constraints remain explicitly enforced. The penalization process can be customized via solver options.

Algorithm 2 Outline of the primal-dual interior point Method

```
Input Initial feasible point $(x^0, s^0, \lambda^0, \nu^0, Z^0, w^0)$ with $s^0 > 0, Z^0 \succ 0, w^0 > 0$, optimality tolerance $\epsilon^0 > 0$, feasibility tolerance $\epsilon^f > 0$, maximum iteration count $k_{\max}$, fraction-to-the-boundary parameter $\tau \in (0,1)$, centrality parameter $\sigma \in (0,1)$, and step scaling parameter $\gamma \in (0,1)$.
Output Solution $(x, s, \lambda, \nu, Z, w)$.
 1: Compute initial barrier parameter
        $\mu_0 = \frac{1}{n}\langle \mathcal{A}(x^0), Z^0\rangle + \frac{1}{n_2}\langle s^0, w^0\rangle$
 2: Set iteration counter $k \leftarrow 0$
 3: while outer stopping criterion (15) not satisfied do
 4:     Set inner iteration counter $i \leftarrow 0$
 5:     while inner stopping criterion (14) not satisfied for $\mu = \mu_k$ do
 6:         Solve the Newton system (12) to obtain search directions
                $(\Delta x, \Delta s, \Delta\lambda, \Delta\nu, \Delta Z, \Delta w)$
 7:         Compute step lengths $\tilde{\alpha}_p, \tilde{\alpha}_d \in (0,1]$ as in (13)
 8:         Update variables
                $x \leftarrow x + \gamma\tilde{\alpha}_p\Delta x, \quad s \leftarrow s + \gamma\tilde{\alpha}_p\Delta s, \quad w \leftarrow w + \gamma\tilde{\alpha}_p\Delta w$
                $\lambda \leftarrow \lambda + \gamma\tilde{\alpha}_d\Delta\lambda, \quad \nu \leftarrow \nu + \gamma\tilde{\alpha}_d\Delta\nu, \quad Z \leftarrow Z + \gamma\tilde{\alpha}_d\Delta Z$
 9:         $i \leftarrow i + 1$
10:     end while
11:     Update barrier parameter
            $\mu_{k+i} = \sigma \cdot \left(\frac{1}{n}\langle \mathcal{A}(x), Z\rangle + \frac{1}{n_2}\langle s, w\rangle\right)$
12:     $k \leftarrow k + i$
13: end while
```

5 Usage and problem setup

5.1 Usage

To set up IPNSDP, first install the CasADi framework from https://web.casadi.org/get/. Next, download IPNSDP from https://github.com/AlemsegedWeldeyesus/IPNSDP and run the provided `install.m` script by executing `install()`. This adds the base directory and all subfolders to the MATLAB path. Finally, verify the installation by running `ipnsdp_test()`. If the test runs successfully, IPNSDP is ready for use.

Remark 4. When writing functions for the objective or nonlinear constraints, users should be aware that CasADi may requir the use of `SX` or `MX` data types. This is important when the implementation involves operations such as `for`-loops, since standard MATLAB data types are not compatible with CasADi framework.

5.2 Problem setup

To facilitate problem formulation, a structured MATLAB template, `ipnsdp_user_template.m`, provided with the IPNSDP solver and reproduced in Appendix A, serves as a basis for modeling a wide range of nonlinear semidefinite programming (SDP) problems.

5.2.1 Input

IPNSDP requires input for the formulation in (1). To specify the coefficient matrices for the linear constraints, for a given matrix $A \in \mathbb{R}^{n\times n}$, define the vector $\mathrm{vec}(A) \in \mathbb{R}^{n^2}$ as

$$\mathrm{vec}(A) := \left[a_{11},\, a_{21},\, \ldots,\, a_{n1},\, a_{12},\, a_{22},\, \ldots,\, a_{n2},\, \ldots,\, a_{1n},\, a_{2n},\, \ldots,\, a_{nn}\right]^T,$$

which is formed by stacking the columns of A into a single vector.

Now, following formulation (1), consider

$$\mathrm{vec}(X) = \left[\mathrm{vec}(X_1)^\top \quad \mathrm{vec}(X_2)^\top \quad \cdots \quad \mathrm{vec}(X_{m_X})^\top\right]^\top, \tag{16}$$

$$A_{11} = \begin{bmatrix} \mathrm{vec}\left(A_{11,1}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{11,1}^{(m_X)}\right)^\top \\ \mathrm{vec}\left(A_{11,2}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{11,2}^{(m_X)}\right)^\top \\ \vdots & \ddots & \vdots \\ \mathrm{vec}\left(A_{11,n_1}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{11,n_1}^{(m_X)}\right)^\top \end{bmatrix}, \quad A_{12} = \begin{bmatrix} A_{12,1}^\top \\ A_{12,2}^\top \\ \vdots \\ A_{12,n_1}^\top \end{bmatrix}, \tag{17}$$

$$A_{21} = \begin{bmatrix} \mathrm{vec}\left(A_{21,1}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{21,1}^{(m_X)}\right)^\top \\ \mathrm{vec}\left(A_{21,2}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{21,2}^{(m_X)}\right)^\top \\ \vdots & \ddots & \vdots \\ \mathrm{vec}\left(A_{21,n_2}^{(1)}\right)^\top & \cdots & \mathrm{vec}\left(A_{21,n_2}^{(m_X)}\right)^\top \end{bmatrix}, \quad A_{22} = \begin{bmatrix} A_{22,1}^\top \\ A_{22,2}^\top \\ \vdots \\ A_{22,n_2}^\top \end{bmatrix}. \tag{18}$$

The user can define the fields in the structure `prob` as listed below. Any optional inputs that are not specified by the user will take default values, which are described in Appendix A.

`Problem data`

- `prob.name`: (string) Problem name. (Optional)
- `prob.problem_data`: User-defined struct for passing data to other functions. (Optional)

`Matrix variables`

- `prob.nX`: m_x Number of matrix variables $X_1, \ldots, X_{m_X}$. Set to 0 if none.
- `prob.dimX`: $[d_1, ...d_k]$ Vector of dimensions of $X_1, \ldots, X_{m_X}$. Set to [] if none.
- `prob.lbX`: $[\underline{\rho}_1, ..., \underline{\rho}_k]$ Vector of lower bounds for constraints $\underline{\rho}_k I \preceq X_k$. Set to [] if all unbounded. Use `-Inf` for selectively unbounded entries. Defaults used if not provided.
- `prob.ubX`: $[\bar{\rho}_1, ..., \bar{\rho}_k]$ Vector of upper bounds for constraints $X_k \preceq \bar{\rho}_k I$. Set to [] if all unbounded. Use `Inf` for selectively unbounded entries. Defaults used if not provided.

`Scalar variables`

- `prob.nu`: n Number of scalar variables $u \in \mathbb{R}^n$. Set to 0 if none.
- `prob.lbu`: $\underline{u}$ Vector of lower bounds for constraints $\underline{u} \leq u$. Set to [] if all unbounded. Use `-Inf` for selectively unbounded entries. Defaults used if not provided.
- `prob.ubu`: $\bar{u}$ Vector of upper bounds for constraints $u \leq \bar{u}$. Set to [] if all unbounded. Use `Inf` for selectively unbounded entries. Defaults used if not provided.

`Nonlinear matrix inequality constraints`

- `prob.nPSDcon`: n_{psd} Number of nonlinear matrix inequality constraints, i.e., $\mathcal{A}_s(X, u) \succeq 0$. Set to 0 if none.

`Linear constraints`

- `prob.A11`: Coefficient matrix for $\text{vec}(X)$ as in (17). Set to [] or ignore if none.
- `prob.A12`: Coefficient matrix for u as in (17). Set to [] or ignore if none.
- `prob.A21`: Coefficient matrix for $\text{vec}(X)$ as in (18). Set to [] or ignore if none.
- `prob.A22`: Coefficient matrix for u as in (18). Set to [] or ignore if none.

`Function handles`

- `prob.obj`: Function handle for the objective, typically `@(x) objective(x, prob)`.
- `prob.nlcon`: Function handle for user-defined nonlinear constraints returning three outputs `[eq, ineq, psd]`, where
  - `eq`: Vector of nonlinear equality constraints. Set to [] if none.
  - `ineq`: Vector of nonlinear inequality constraints. Set to [] if none.
  - `psd`: Cell array of symmetric matrices subject to positive semidefinite constraints (e.g., `psd{1} =` $\mathcal{A}_1(X, u)$). Set to {} if none.

`Initial point`

- `prob.x0.X`: Cell array of initial points for $X_1, \ldots, X_{m_X}$ (e.g., `x0.X{1} = eye(d_1)`). Defaults are used if not provided.
- `prob.x0.u`: Vector of initial points for scalar variables u. Defaults used if not provided.

See also Example 1 for an illustration of this setup applied to the nearest correlation matrix problem.

5.2.2 Output

IPNSDP returns two outputs, `x` and `info`

$$[\texttt{x}, \texttt{info}] = \texttt{ipnsdp\_solve(prob)}.$$

Here, `x` contains all solution variables, including associated slack and dual variables, while `info` is a struct providing diagnostic information such as iteration history, residual norms, feasibility flags, and timing data.

5.2.3 Solver options

Many features of the interior point method implemented in IPNSDP can be customized through the `prob.options` field. For example,

- Chordal decomposition: Can be enabled or disabled via `chord_decomp = 'yes'/'no'`, with clique merging controlled by `clique_merge = 'yes'/'no'`.
- Constraint redundancy checks: Can be enabled or disabled through `linconCheck = 'yes'/'no'` and `nonlinconCheck = 'yes'/'no'`.

- Algorithmic parameters: Setting tolerances (`opt_tol`, `feas_tol`), centrality and fraction-to-the-boundary parameters (`sigma`, `tau`), and maximum iterations (`max_iter`).

For details on these and other options, such as the choice of solvers for the Newton system, step length strategies, and penalization schemes, see Appendix A.

6 Numerical examples

This section presents five examples of NSDP arising from practical applications that can be solved using IPNSDP. The complete code for these examples is included in the software distribution. All examples were solved on a PC equipped with a 13th Gen Intel(R) Core(TM) i5-1345U CPU running at 1.60 GHz with 32 GB of RAM, using an implementation in MATLAB R2023a. The algorithmic parameters for the interior point method were set to their default values as explicitly listed in Appendix A. The computational times reported do not include preprocessing or CasADi problem setup times.

Example 1. This example is the scaled nearest correlation matrix problem, that arises in financial applications [7]. It seeks a correlation matrix with a prescribed condition number κ that best approximates a given symmetric matrix $H \in \mathbb{S}^n$. The problem was solved in [4] as NSDP of the form

$$\begin{aligned} \min_{X,z} \quad & \|zX - H\|_F^2 \\ \text{s.t.} \quad & \operatorname{diag}(zX) = \mathbf{1} \\ & I \preceq X \preceq \kappa I, \end{aligned} \tag{19}$$

where $X \in \mathbb{S}^n$, $z \in \mathbb{R}$ are the variables, and the parameter $\kappa > 1$. This problem exhibits nonlinearity in both the objective function and the constraints.

Given H and κ, the problem can be solved using IPNSDP with the input shown in Figure 1.

Figure 1: Input to solve the scaled nearest correlation matrix problem in Example 1 via `IPNSDP`.

```
% Problem dimensions
prob.nX      = 1;              % One matrix variable X
prob.dimX    = [n];            % X in S^n
prob.nu      = 1;              % One scalar variable z
prob.lbX     = [1];            % Enforce X >= I
prob.ubX     = [kappa];        % Enforce X <= kappa*I
prob.nPSDcon = 0;              % no of PSD constraints,  A(X,u) >=0

% Objective function
prob.obj = @(x) norm(x.u * x.X{1} - H, 'fro')^2;

% Constraints: [eq, ineq, psd]
prob.nlcon = @(x) deal( ...
diag(x.u * x.X{1}) - 1, ...% diag(z*X) = 1
[], ...                    % no inequalities
{});                       % no PSD constraints

% Solve
[x, info] = ipnsdp_solve(prob);
```

For the matrix H and dimension $n = 6$ taken from [4], the code is provided in `ncm_c_b.m`, which also illustrates the consistency of the results. For other randomly generated instances of H and n, the code is available in `ncm_c.m`, and the corresponding computational statistics are presented in Table 1. In these instances, the matrix $H \in \mathbb{S}^n$, $n = 10, 20, 40, 80$ is constructed by adding symmetric Gaussian noise to a valid correlation matrix, followed by setting its diagonal entries to 1. The parameter κ is set to 10 in all cases.

Table 1: Computational results for Example 1.

Matrix size n	IPM iter	CPU (s)
10	20	¡1
20	21	¡1
40	22	11
80	25	610

Example 2. This example presents a problem arising in structural optimization. It involves optimizing both the topology and geometry of truss structures subject to global stability constraints. The design variables include the cross-sectional areas of the potential bars and the coordinates of the joints shown in Figure 2a. This problem demonstrates the effectiveness of the chordal decomposition technique in IPNSDP for solving problems with structured matrix inequality constraints, such as the one shown in Figure 2b. Mathematically, the problem is formulated as:

$$\begin{aligned} \underset{a,v,u}{\text{minimize}} \quad & l(v)^T a \\ \text{subject to} \quad & f^T u_j \leq \zeta \\ & K(a,v)u = f \\ & K(a,v) + \tau G(a,v,u) \succeq 0 \\ & v \in \mathcal{V} \\ & a \geq 0, \end{aligned} \tag{20}$$

where

- a is a vector of decision variables for the cross-sectional areas.
- v is a vector of decision variables for the nodal coordinates.
- u is a vector of state variables for the nodal displacements.
- f is the external constant force.
- $\zeta > 0$ is a compliance bound.
- $\tau \geq 1$ is a loading factor.
- $l(v)$ is a vector of bar lengths such that

  $$l_i(v) = \|v_i^{(2)} - v_i^{(1)}\|, \tag{21}$$

  where $\|\cdot\|$ is Euclidean norm.
- $\mathcal{V}$ is a region defined by balls of radii r around the joints and is given by

  $$\mathcal{V}_1 = \{v \in \mathbb{R}^{d_0 N} \mid \|v_j - \bar{v}_j\|^2 \leq r_j^2,\ j = 1, \ldots, d_0\}. \tag{22}$$

- $K(a,v)$ is the global stiffness matrix, given by

  $$K(a,v) = \sum_{i=1}^{m} a_i \frac{E}{l_i(v)} \gamma_i(v) \gamma_i^T(v), \tag{23}$$

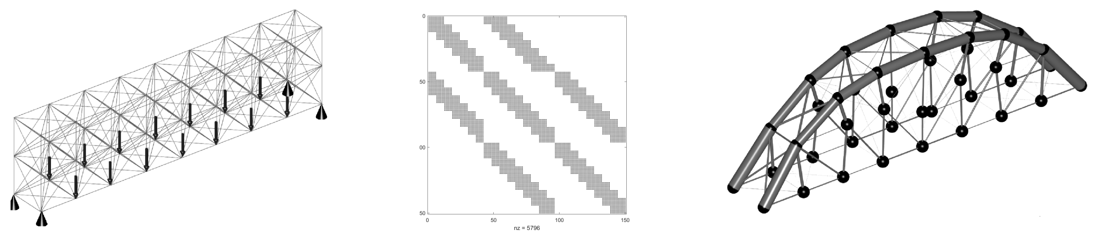
  where E is the Young's modulus of the material, and

  $$\gamma_i(v) = \frac{1}{l_i(v)} (v_i^{(2)} - v_i^{(1)},\ -(v_i^{(2)} - v_i^{(1)}))^T. \tag{24}$$

- $G(a,v,u_j)$ is the geometry stiffness matrix, given by

  $$G(a,v,u_j) = \sum_{i=1}^{m} \frac{a_i E \gamma_i(v)^T u_j}{l_i^2(v)} \left(\delta_i(v)\delta_i(v)^T + \eta_i(v)\eta_i(v)^T \right), \tag{25}$$

  where the vectors $\delta_i(v)$ and $\eta_i(v)$ must be computed such that $\gamma_i(v)$, $\delta_i(v)$, and $\eta_i(v)$ are mutually orthogonal.



(a) Loading condition. **(b) Sparsity of $K(a, v) + \tau G(a, v, u)$. (Before completion for chordality)** **(c) Optimal design.**

Figure 2: Example 2 – Loading condition, sparsity pattern of the resulting matrix, and optimal design.

The results are presented in Table 2. The table also illustrates the efficiency of the decomposition, showing that the computational time per iteration was reduced by a factor of approximately 53. This improvement was achieved by completing for chordality and decomposing the 150×150 matrix in Figure 2b into 8 blocks, each of size between 30 and 36. Note that this problem was previously solved in [27], where the derivatives were computed manually and the chordal decomposition was performed in advance, not by the solver. In contrast, in this paper, the derivatives are obtained via automatic differentiation, and the decomposition is carried out using the built-in feature of IPNSDP. The complete code for this example is provided in `t_buckgeo.m`.

Table 2: Computational results and comparison of solution times with and without chordal decomposition for Example 2. A single 150×150 matrix was decomposed into 8 blocks of size between 30 and 36.

Without chordal decomposition			With chordal decomposition		
f_val	IPM iter	CPU (s)	f_val	IPM iter	CPU (s)
4.3368	86	3166	4.3373	64	44

Example 3. This example presents a financial model that can be applied to classify failing and non-failing companies. The model is based on a Logit framework with a positive semidefinite quadratic discriminant function. A similar learning problem has been studied in [30]. The objective is to maximize the following function:

$$\begin{aligned} \max_{Q} \quad & \sum_{i=1}^{M} \left(y_i z(x_i) - \log\left(1 + e^{z(x_i)}\right) \right) \\ \text{subject to} \quad & Q \succeq 0, \end{aligned} \tag{26}$$

where, the parameters (variables) to be optimized are $a \in \mathbb{R}$, $b \in \mathbb{R}^q$, and $Q \in \mathbb{S}^q$. The function $z(x)$ is defined as

$$z(x) = a + b^T x + \frac{1}{2} x^T Q x$$

Here, $x_i \in \mathbb{R}^q$ represents the financial data for company i, and $y_i \in \{0, 1\}$ is a binary indicator ($y_i = 0$ for non-failure, $y_i = 1$ for failure). In practice, this problem involves a large number of learning data M and a small number of features q. Although the problem is nonlinear and potentially prone to numerical instability due to the exponential term in the objective function, the number of the optimization variables is relatively small, and the problem can typically be solved in a few seconds. However, for significantly large values of M, the setup phase in CasADi, can take minutes. For the numerical experiments, the entries of matrix M are generated from standard normal data adjusted using random average and spread vectors and then normalized between 0 and 1. See Table 3 for computational results. Note that for this example a 'separate' step strategy was used to determine the step length for the search directions. The complete code for this example is included in `f_d.m` in the distribution of the solver.

Table 3: Computational results for Example 3.

Sizes (M, q)	IPM iter	CPU (s)
(12,000,6)	21	¡1
(12,000,8)	19	3
(12,000,10)	46	13
(12,000,12)	27	13

Example 4. This example is adapted from [21]. It involves a bilinear matrix inequality and is formulated as

$$\begin{aligned} \min_{x\in\mathbb{R}^n,\,\lambda\in\mathbb{R}} \quad & \lambda \\ \text{s.t.} \quad & -|c^\ell| \le x^\ell \le |c^\ell|, \quad \ell = 1,\dots,n, \\ & B(x) = A_0 + \sum_{k=1}^{n} x_k A_k + \sum_{i=1}^{d}\sum_{j=1}^{d} x_i x_j K_{i,j} \preceq \lambda I_m, \end{aligned} \tag{11.1}$$

where $A_k \in \mathbb{S}^m$ for $k = 0,\dots,n$, $K_{i,j} \in \mathbb{S}^m$ for $i,j = 1,\dots,d$, and $0 < d \le n$. The bounds c^ℓ, $\ell = 1,\dots,n$, and the entries of the matrices A_k and $K_{i,j}$ are randomly drawn from the standard normal distribution. All matrices are approximately 20% dense; however, the aggregated matrix $B(x)$ is fully dense (100%), which makes the problem unsuitable for chordal decomposition, particularly in this setting. However, if $B(x)$ possesses a structured sparsity pattern, likely arising from specific applications, such problems could potentially be solved efficiently by IPNSDP. The numerical results are presented in Table 4. The implementation is provided in `bmi_1.m`, included in the solver distribution.

Table 4: Computational results for Example 4.

Sizes (n, d, m)	IPM iter	CPU (s)
(50,25,50)	25	6
(50,50,50)	28	7
(100,50,100)	28	99
(100,100,100)	33	162

Example 5. In this example, a problem known by a Gaussian channel capacity problem is solved. This has been solved in [29, 30]. It is formulated as

$$\begin{aligned} \underset{x_i,t_i}{\text{minimize}} \quad & \frac{1}{2}\sum_{i=1}^{n} \log(1+t_i) \\ \text{subject to} \quad & \frac{1}{n}\sum_{i=1}^{n} x_i \le 1 \\ & x_i \ge 0, \quad t_i \ge 0, \quad i = 1,\dots,n \\ & \begin{bmatrix} 1 - a_i t_i & \sqrt{r_i} \\ \sqrt{r_i} & a_i x_i + r_i \end{bmatrix} \succeq 0, \quad i = 1,\dots,n. \end{aligned}$$

The problem features a nonlinear objective function and a large number n of linear matrix inequality constraints, each of dimension 2×2. Due to the relatively small size of these matrix inequalities, the problem can be efficiently solved for larger instances compared to the other examples discussed in this report. For example, a problem involving 3200 such matrices was solved in under two minutes. For the results reported in Table 5, the parameters r_i and a_i, for $i = 1,\dots,n$, are randomly generated within the interval $[0, 1]$. The implementation code is provided in `g_c_cap.m`.

Remark 5. The IPNSDP software distribution includes additional examples, such as positive semidefinite programming extensions of selected test cases from the Hock and Schittkowski libraries [8] for testing nonlinear programming solvers and widely used in the NSDP literature [4, 30], the minimization of the minimum eigenvalue problem, and the primal and dual formulations of the well-known matrix completion problem.

Table 5: Computational results for Example 5.

Number of matrices n	IPM iter	CPU (s)
100	24	¡1
200	27	¡1
400	27	2
800	31	6
1600	33	20
3200	35	74

Remark 6. In general, IPNSDP obtains solutions within a reasonable number of iterations when the problems exhibit convexity and when it is provided with feasible initial points. Although it includes features such as scaling and checks for linear constraint feasibility, it is still recommended to supply a feasible initial point. Furthermore, for some problems, performance can be improved by customizing the algorithmic settings or the penalization strategy through its options.

7 Conclusion

The paper introduces IPNSDP, an alternative MATLAB solver designed to address Nonlinear Semidefinite Programming (NSDP) problems using a second-order interior point method. IPNSDP aims to simplify the user experience by integrating with CasADi for automatic differentiation, thereby reducing the manual effort typically required for derivative computations involving matrix calculus.

For large-scale problems with structured sparse matrix constraints, IPNSDP leverages a chordal decomposition technique. This approach decomposes a single large positive semidefinite constraint into multiple smaller ones, significantly improving computational efficiency. To further improve the efficiency, the solver applies a clique merging strategy that combines submatrices with substantial overlap. This procedure is particularly beneficial for applications such as structural optimization, where these problem structures frequently occur.

Additionally, IPNSDP provides customization options for both the interior point algorithm and the decomposition procedures, enabling users to fine-tune the solver's performance. Overall, IPNSDP offers a practical and user-friendly framework that combines advanced algorithmic strategies with automated tools to effectively solve a wide class of NSDPs.

Appendix

A Complete IPNSDP problem setup template

```
function ipnsdp_input_template()
%IPNSDP_INPUT_TEMPLATE Example template for defining an NSDP in ipnsdp.
%
% This script sets up a generic Nonlinear Semidefinite Program (NSDP)
% of the form:
%
%   minimize   f(X, u)
%   subject to
%       sum_{k=1}^{m_X} <A11_i^{(k)}, X_k> + A12_i * u = b1_i, i = 1,...,n1
%       sum_{k=1}^{m_X} <A21_j^{(k)}, X_k> + A22_j * u <= b2_j, j = 1,...,n2
%       c1(X, u) = 0
%       c2(X, u) <= 0
%       A_s(X, u) >= 0  (PSD),   s = 1,...,n_psd
%       u_lower <= u <= u_upper
%       rho_k_lower * I_{d_k} <= X_k <= rho_k_upper * I_{d_k}, k = 1,...,m_X
%
% where
%   - X_k in S^{d_k}, k = 1,...,m_X, are symmetric matrix variables
%   - u in R^n is a vector of scalar variables
%   - <A, X> = trace(A' * X) denotes the Frobenius inner product
%
%% Problem Definition
% All fields are listed with empty values. Comments explain intended usage.

prob.name         = '';        % Name of the problem (optional)
prob.problem_data = struct(); % Struct to store user data (optional)

% Matrix variables X
prob.nX     = [];              % m_X: number of matrix variables X_k.
                               % Set to 0 if none.
prob.dimX   = [];              % [d_1,...,d_m_X]: sizes d_k of X_k.
                               % Set to [] if none.

% Bounds on Matrix variables X
prob.lbX   = [];               % [rho_1_lower,...,rho_m_X_lower]
                               % Set to [] if all unbounded.
                               % Use -Inf for selectively unbounded entries.
prob.ubX   = [];               % [rho_1_upper,...,rho_m_X_upper]
                               % Set to [] if all unbounded.
                               % Use +Inf for selectively unbounded entries.

% Scalar variables u
prob.nu     = [];              % n: dimension of vector u.
                               % Set to 0 if none.

% Bounds on scalar variables u
prob.lbu    = [];              % [u_lower_1;...;u_lower_n]
                               % Set to [] if all unbounded.
                               % Use -Inf for selectively unbounded entries.
prob.ubu    = [];              % [u_upper_1;...;u_upper_n]
                               % Set to [] if all unbounded.
                               % Use +Inf for selectively unbounded entries.

% nPSD constraints
prob.nPSDcon = [];             % n_psd: number of PSD constraints A_s(X,u) >= 0.
                               % Set to 0 if none.

% Linear constraints
% Note: You may embed these into nonlinear constraints if easier.
prob.A11 = [];      % [vec(A11_1_1)^T, ..., vec(A11_1_mX)^T;
                    %  ...;
                    %  vec(A11_n1_1)^T, ..., vec(A11_n1_mX)^T]
                    % Coefficient matrix for [vec(X_1); ...; vec(X_mX)]
                    % in linear equality constraints. [] if none.
prob.A12 = [];      % Coefficient matrix for scalar variables u in
                    % linear equalities. [] if none.
prob.b1 = [];       % RHS vector of linear equality constraints.
                    % [] if none.
prob.A21 = [];      % [vec(A21_1_1)^T, ..., vec(A21_1_mX)^T;
                    %  ...;
                    %  vec(A21_n2_1)^T, ..., vec(A21_n2_mX)^T]
                    % Coefficient matrix for [vec(X_1); ...; vec(X_mX)]
```

```
                   % in linear inequality constraints. [] if none.
prob.A22 = [];     % Coefficient matrix for scalar variables u in
                   % linear inequalities. [] if none.
prob.b2 = [];      % RHS vector of linear inequality constraints.
                   % [] if none.

% Objective function
prob.obj = @(x) objective(x, prob);
                             % User-defined objective function
% Nonlinear and PSD constraints
prob.nlcon = @(x) nonlinear_constraint(x, prob);
                   % User-defined nonlinear constraints
                   % Must return [eq, ineq, psd]:
                   % eq   = nonlinear equalities (vector)
                   % ineq = nonlinear inequalities (vector)
                   % psd  = cell array of PSD matrices

% Initial point (optional)
prob.x0.X = {};              % Cell array of initial guesses for X_k.
                             % Example: prob.x0.X{1} = eye(d_1), ...,
                             %          prob.x0.X{m_X} = eye(d_m_X)
                             % Defaults are used if not specified.
prob.x0.u = [];              % Column vector initial guess for u.
                             % Default is zero if not specified.

%% Algorithmic Options
% Default solver parameters (modifiable by the user)

prob.options.scale         = 'on';  % 'on'/'off': Enable/Disable scaling
prob.options.opt_tol       = 1e-6;  % Optimality tolerance
prob.options.feas_tol      = 1e-7;  % Feasibility tolerance
prob.options.max_iter      = 100;   % Maximum number of iterations
prob.options.min_mu        = 1e-9;  % Minimum barrier parameter
prob.options.sigma         = 0.4;   % Centrality parameter
prob.options.tau           = 0.9;   % Backtracking step size
prob.options.gamma         = 0.99;  % Step length scaling
prob.options.stepMode      = 'joint';% 'joint'/'separate': Step strategy
prob.options.chord_decomp  = 'yes'; % 'yes'/'no': Chordal decomposition
prob.options.clique_merge  = 'yes'; % 'yes'/'no': Clique merging
prob.options.density       = 0.7;   % Density threshold for decomposition
prob.options.min_block_size = 30;   % Minimum block size for decomposition
prob.options.rel_block_size = 0.6;  % Relative block size for decomposition
prob.options.linconCheck   = 'yes'; % 'yes'/'no': LinConst dependency test
prob.options.nonlinconCheckN = 'yes'; % 'yes'/'no': NonlinConst dependency test (Numeric)
prob.options.nonlinconCheckS = 'no'; % 'yes'/'no': NonlinConst dependency test (Symbolic)
prob.options.penlpMode     = 'off'; % 'off'/'on': Penalization
prob.options.penlp         = 0;     % Penalty parameter (0 = off)
prob.options.penldeg       = 2;     % 1 or 2: Penalty degree

%% Solve the problem
[x_sol, info] = ipnsdp_solve(prob);

%% Results (saved to ipnsdp_sol.mat)
x_sol.X            % Cell array of solution X_k matrices or slacks
x_sol.u            % Optimized scalar variables vector u
x_sol.Z            % Dual vars for PSD constraints X_k and A_s(X,u)
x_sol.lambda1      % Dual variables for linear equalities
x_sol.lambda2      % Dual variables for linear inequalities
x_sol.theta1       % Dual variables for nonlinear equalities
x_sol.theta2       % Dual variables for nonlinear inequalities
x_sol.slinear      % Slack variables for linear inequalities
x_sol.snonlinear   % Slack variables for nonlinear inequalities

info.obj_evals                  % objective function evaluations
info.grad_evals                 % gradient evaluations
info.nonlincon_evals            % nonlinear constraint evaluations
info.nonlincon_jacobi_evals     % nonlinear constraint Jacobian evals
info.Lag_hessian_evals          % Lagrangian Hessian evaluations
info.CPU                        % CPU time used (seconds)
info.iter                       % Number of iterations performed
info.obj_value                  % Final objective function value
info.opt_tol                    % Final optimality tolerance
info.primal_feasibility         % Final primal feasibility measure
info.dual_feasibility           % Final dual feasibility measure
info.complementarity            % Final complementarity gap
info.obj_val_hist               % History of objective values
info.kkt_optimality             % History of KKT conditions
info.exit                       % Solver exit message

end

%% Objective Function
function f_val = objective(x, prob)
% Computes the objective function value.
%
% Inputs:
%   x   : struct with fields:
%          x.X : cell array of matrix variables
%          x.u : vector of scalar variables
%   prob: problem-specific data struct
%
% Output:
%   f_val: scalar objective value
%
% ... (user-defined implementation)
end

%% Nonlinear and PSD Constraints
function [eq, ineq, psd] = nonlinear_constraint(x, prob)
% Computes nonlinear equality, inequality, and PSD constraints.
%
% Inputs:
%   x.X : cell array {X_1, ..., X_mX}, each X_k in S^{d_k}
%   x.u : vector of scalar variables u in R^n
%   prob: problem data struct
%
% Outputs:
%   eq   : nonlinear equality constraints vector (if none set to [])
%   ineq : nonlinear inequality constraints vector (if none set to [])
%   psd  : psd{1} = A_1(X, u),..., psd{n_psd} = A_n_psd(X, u)
%          cell array of PSD constraint matrices (if none set to {})
%
% ... (user-defined implementation)
end
```

Listing 1: `ipnsdp_user_template.m` – **user input template for IPNSDP**

References

[1] J.A.E. Andersson, J. Gillis, G. Horn, J.B. Rawlings, and M. Diehl, CasADi – A software framework for nonlinear optimization and optimal control, Mathematical Programming Computation 11 (2019):1–36.

[2] S.G. Constante F., J.C. López, and M.J. Rider, Optimal reactive power dispatch with discrete controllers using a branch−and−bound algorithm: A semidefinite relaxation approach, IEEE Transactions on Power Systems 36 (2021):4539–4550.

[3] D. Eppstein, M. Löffler, and D. Strash, Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time, in Algorithms and Computation, O. Cheong, K.Y. Chwa, and K. Park, eds., Berlin, Heidelberg. Springer Berlin Heidelberg, 2010:403–414.

[4] J. Fiala, M. Kočvara, and M. Stingl, PENLAB: a MATLAB solver for nonlinear semidefinite optimization (2013). Available at https://arxiv.org/abs/1311.5240.

[5] M. Fukuda, M. Kojima, K. Murota, and K. Nakata, Exploiting sparsity in semidefinite programming via matrix completion i: General framework, SIAM Journal on Optimization 11 (2001):647–674.

[6] M. Garstka, M. Cannon, and P. Goulart, A clique graph based merging strategy for decomposable SDPs, IFAC−PapersOnLine 53 (2020):7355–7361.

[7] N.J. Higham, Computing the nearest correlation matrix—a problem from finance, IMA Journal of Numerical Analysis 22 (2002):329–343.

[8] W. Hock and K. Schittkowski, Test Examples for Nonlinear Programming Codes, Springer-Verlag, Berlin, 1981.

[9] S. Kim, M. Kojima, M. Mevissen, and M. Yamashita, Exploiting sparsity in linear and nonlinear matrix inequalities via positive semidefinite matrix completion, Math. Program. 129 (2011):33–68.

[10] M. Kocvara, F. Leibfritz, M. Stingl, and D. Henrion, A nonlinear sdp algorithm for static output feedback problems in compleib, Proceedings of the 16th IFAC World Congress on Automatic Control, Prague 16 (2005).

[11] H. Konno, N. Kawadai, and D. Wu, Estimation of failure probability using semi-definite logit model, Computational Management Science 1 (2003):59–73.

[12] M. Kočvara, On the modelling and solving of the truss design problem with global stability constraints, Structural and Multidisciplinary Optimization 23 (2002):189–203.

[13] M. Kočvara, Decomposition of arrow type positive semidefinite matrices with application to topology optimization, Mathematical Programming 190 (2020):105–134.

[14] M. Kočvara and M. Stingl, Pennon: A code for convex nonlinear and semidefinite programming, Optimization Methods and Software 18 (2003):317–333.

[15] J. Lofberg, YALMIP : a toolbox for modeling and optimization in MATLAB, in 2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No.04CH37508). 2004:284–289.

[16] MOSEK ApS, Copenhagen, Denmark, MOSEK Optimization Toolbox for MATLAB Manual (2024). Available at https://docs.mosek.com/.

[17] Y.E. Nesterov and M.J. Todd, Self-scaled barriers and interior-point methods for convex programming, Mathematics of Operations Research 22 (1997):1–42.

[18] Y.E. Nesterov and M.J. Todd, Primal-dual interior-point methods for self-scaled cones, SIAM Journal on Optimization 8 (1998):324–364.

[19] K. Okabe, Y. Yamakawa, and E.H. Fukuda, A revised sequential quadratic semidefinite programming method for nonlinear semidefinite optimization, Journal of Industrial and Management Optimization 19 (2023):7777–7794.

[20] P.A. Parrilo, Semidefinite programming relaxations for semialgebraic problems, Mathematical Programming 96 (2003):293–320.

[21] M. Stingl, On the solution of nonlinear semidefinite programs by augmented Lagrangian method, Ph.D. diss., Institute of Applied Mathematics II, Friedrich-Alexander University of Erlangen-Nuremberg, 2006.

[22] J.F. Sturm, Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones, Optimization Methods and Software 11 (1999):625–653.

[23] K. Toh, M.J. Todd, and R.H. Tütüncü, SDPT3 — a Matlab software package for semidefinite programming, version 1.3, Optimization Methods and Software 11 (1999):545–581.

[24] L. Vandenberghe and M.S. Andersen, Chordal graphs and semidefinite optimization, Foundations and Trends®in Optimization 1 (2015):241–433.

[25] A. Weldeyesus, J. Gondzio, L. He, M. Gilbert, P. Shepherd, and A. Tyas, Truss geometry and topology optimization with global stability constraints, Structural and Multidisciplinary Optimization 62 (2020):1721–1737.

[26] A. Weldeyesus and M. Stolpe, A primal-dual interior point method for large-scale free material optimization, Computational Optimization and Applications 61 (2015):409–435.

[27] A.G. Weldeyesus, J. Gondzio, and M.F. Anjos, On the scalability of truss geometry and topology optimization with global stability constraints via chordal decomposition, Structural and Multidisciplinary Optimization 68 (2024), p. 7.

[28] S.J. Wright, Primal-Dual Interior-Point Methods, SIAM, Philadelphia, 1997.

[29] Y. Yamakawa and T. Okuno, A stabilized sequential quadratic semidefinite programming method for degenerate nonlinear semidefinite programs, Computational Optimization and Applications 83 (2022).

[30] H. Yamashita, H. Yabe, and K. Harada, A primal-dual interior point method for nonlinear semidefinite programming, Mathematical Programming 135 (2012):89–121.

[31] H. Yamashita and H. Yabe, A survey of numerical methods for nonlinear semidefinite programming, Journal of the Operations Research Society of Japan 58 (2015):24–60.

[32] M. Yannakakis, Computing the minimum fill-in is np−complete, SIAM Journal on Algebraic Discrete Methods 2 (1981):77–79.

[33] Y. Zheng, G. Fantuzzi, and A. Papachristodoulou, Chordal and factor-width decompositions for scalable semidefinite and polynomial optimization, Annual Reviews in Control 52 (2021):243–279.