Semidefinite programming via `Projective Cutting-Planes` for dense (easily-feasible) instances

November 27, 2025

Abstract

The cone of positive semi-definite (SDP) matrices can be described by an infinite number of linear constraints. It is well-known that one can optimize over such a feasible area by standard `Cutting-Planes`, but work on this idea remains a rare sight, likely due to its limited practical appeal compared to Interior Point Methods (IPMs). We extend the standard `Cutting-Planes` by upgrading the widely-used separation sub-problem to the following *projection sub-problem*: given an SDP matrix $X \succeq \mathbf{0}$ and a direction $D \in \mathbb{R}^{n\times n}$, find the maximum step length t such that $X + tD \succeq \mathbf{0}$, *i.e.*, such that $X + tD$ stays in the SDP cone. Exploiting this sub-problem, we use a projective `Cutting-Planes` variant that possesses a capability not inherent to standard `Cutting-Planes`: generate feasible solutions along the iterations. In fact, the proposed SDP algorithm can actually construct both a primal-feasible and a dual-feasible SDP solution at each iteration, which is a feature that does not exist (built-in) in standard IPMs. The main theoretical challenge was to design a very fast algorithm to calculate this non-orthogonal projection in the SDP cone. Compared to other methods, `Projective Cutting-Planes` performs best on (easily-feasible) SDP programs with large dense $n \times n$ matrices (*e.g.*, $n \geq 1000$), because in such cases many IPMs seem to become computationally slower and memory-intensive. An interesting new feature is the natural integration of re-optimization: in contrast to IPMs, `Projective Cutting-Planes` does not need to restart from scratch when a new linear constraint is added after solving an initial SDP program. This enabled us to apply `Branch and Bound` on a binary SDP program in a streamlined manner: we construct the branching tree by *incrementally solving a sequence of slightly different SDP programs*, since each branching rule reduces to imposing a specific constraint on the relaxed continuous SDP program.

1 Introduction

We consider solving an optimization problem of the following form.

$$(SDP_{\mathscr{C}})\begin{cases}\max\limits_{\mathbf{y}\in\mathbb{R}^k} & \mathbf{b}^\top\mathbf{y} & (1.A.1)\\ s.t & \mathcal{A}^\top\mathbf{y} \preceq A_0 & (1.A.2)\\ & \mathbf{a}^\top\mathbf{y} \leq c_a \ \forall(\mathbf{a},c_a)\in\mathscr{C} & (1.A.3)\end{cases} \text{feasible area } \mathscr{P}^{\text{SDP}}$$

where $\mathcal{A}^\top\mathbf{y} = \sum_{i=1}^k A_i y_i$; $A_1,\ A_2, \ldots, A_k$ and A_0 are symmetric $n \times n$ matrices. The set $\mathscr{C}$ in (1.A.3) may include $\mathbf{y} \geq \mathbf{0}$ (*i.e.*, one can enforce $y_i \geq 0$ by taking $a_i = -1$ and $a_j = 0 \forall j \neq i$ and $c_a = 0$); it generally contains a number of initial linear constraints. In Section 5.5, $\mathscr{C}$ will evolve to contain prohibitively many robust inequalities derived according to a well-known robust optimization paradigm. The variables $\mathbf{y}$ are generally continuous, but we will also consider $\mathbf{y} \in \{0,1\}^k$ in Section 5.6.

We also recall the SDP dual below.

"$= b_i$" becomes "$\geq b_i$" if (1.A.3) contains $y_i \geq 0$

$$(DSDP_{\mathscr{C}})\begin{cases}\min & C \bullet S + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} c_a x_a & (1.B.1)\\ s.t. & A_i \bullet S + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} a_i x_a = b_i \ \forall i \in [1..k] & (1.B.2)\\ & S \succeq \mathbf{0}, \mathbf{x} \geq \mathbf{0}, & (1.B.3)\end{cases}$$

where the operator $\bullet$ denotes the matrix inner product $X \bullet Y = \sum_{i=1}^n \sum_{j=1}^n X_{ij} Y_{ij}\ \forall X, Y \in \mathbb{R}^{n\times n}$.

To recast the problem in a form more suitable to `Cutting-Planes`, we re-write (1.A.1)-(1.A.3) as below; if we have $\mathscr{D} = \mathbb{R}^n$ in (1.C.4), we obtain an equivalent reformulation since in such case (1.C.4) actually reduces to $X \succeq \mathbf{0}$ for $X = A_0 - \mathcal{A}^\top \mathbf{y}$. We used the well-known fact that $X \succeq \mathbf{0} \iff X \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \; \forall \mathbf{d} \in \mathbb{R}^n$, where recall $X \bullet \mathbf{d}\mathbf{d}^\top = \mathbf{d}^\top X \mathbf{d}$.

$$(SDP - LP_{\mathscr{C},\mathscr{D}}) \begin{cases} \max\limits_{\mathbf{y}\in\mathbb{R}^k} \; \mathbf{b}^\top \mathbf{y} & (1.C.1) \\ s.t \;\; X = A_0 - \mathcal{A}^\top \mathbf{y} = A_0 - \sum\limits_{i=1}^{k} A_i y_i \\ \quad \mathbf{a}^\top \mathbf{y} \leq c_a \; \forall (\mathbf{a}, c_a) \in \mathscr{C} \\ \quad X \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \; \forall \mathbf{d} \in \mathscr{D} \subseteq \mathbb{R}^n \end{cases} \begin{matrix} (1.C.2) \\ \\ (1.C.3) \\ (1.C.4) \end{matrix} \Big\} \text{outer approximation of } \mathscr{P}^{\text{SDP}}$$

The above Linear Program (LP) will be integrated into a `Cutting-Planes` logic, generating the constraints $\mathscr{D}$ (and $\mathscr{C}$) on the fly. The constraints $\mathscr{D}$ are variously termed eigenvalue-cuts [3, eq. (26)], eigenvector-cuts, or simply eigen-cuts [19, § 1.1]. Since we cannot enumerate them all and reach $\mathscr{D} = \mathbb{R}^n$, one difficulty will be to control the growth of the real set $\mathscr{D}$ along the iterations. We have to make the above program reach the optimum of the SDP program (1.A.1)-(1.A.3) without any explosion of the constraint count $|\mathscr{D}|$. Applying the classical LP duality, we provide below the dual of above $(SDP{-}LP_{\mathscr{C},\mathscr{D}})$, using a variable $\lambda_{\mathbf{d}}$ for each $\mathbf{d} \in \mathscr{D}$ and a variable x_a for each $(\mathbf{a}, c_a) \in \mathscr{C}$.

"$= b_i$" becomes "$\geq b_i$" if (1.C.3) contains $y_i \geq 0$

$$(DSDP2_{\mathscr{C},\mathscr{D}}) \begin{cases} \min \; A_0 \bullet S + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} c_a x_a & (1.D.1) \\ s.t. \; A_i \bullet S + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} a_i x_a = b_i \; \forall i \in [1..k] & (1.D.2) \\ \quad S = \sum\limits_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top & (1.D.3) \\ \quad \mathbf{x} \geq \mathbf{0}, \lambda_{\mathbf{d}} \geq 0 \; \forall \mathbf{d} \in \mathscr{D} \subseteq \mathbb{R}^n & (1.D.4) \end{cases}$$

Given a fixed $\mathscr{D}$ obtained at some `Cutting-Planes` iteration working on $(SDP{-}LP_{\mathscr{C},\mathscr{D}})$, the dual multipliers λ_d (with $d \in \mathscr{D}$) can be employed to generate an SDP matrix $S = \sum_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top \succeq \mathbf{0}$ as in (1.D.3). The set of matrices that can be generated this way for a *fixed* $\mathscr{D}$ can cover only a subset of the SDP cone. Any such matrix S is a dual feasible SDP solution in (1.B.1)-(1.B.3). We will use this property in Remark 2.B (p. 7) to construct both a primal and dual solution of (1.A.1)-(1.A.3) at each iteration, see also [29, Theorem 9].[1]

We will mainly work on (1.C.1)-(1.C.4), which is a semi-infinite LP formulation with k variables; the lower the value of k, the faster the underlying LP solver for optimizing (1.C.1)-(1.C.4) for each (finite) $\mathscr{D}$ constructed along the iterations. An alternative semi-infinite LP formulation could have been established by working on the SDP dual (1.B.1)-(1.B.3), using $\frac{n\cdot(n+1)}{2}$ variables corresponding to all free elements of a symmetric matrix of size $n \times n$. This would require considering the lower triangular part of S as linear variables, followed by replacing $S \succeq \mathbf{0}$ with $S \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \; \forall \mathbf{d} \in \mathbb{R}^n$. If k is larger than $\frac{n\cdot(n+1)}{2}$, one could ease the work of the LP solver by applying (`Projective`) `Cutting-Planes` on this alternative LP formulation. A `Cutting-Planes` algorithm for such (quite rare) instances was developed in [16].

1.1 The main steps of the SDP `Projective Cutting-Planes`

Although `Projective Cutting-Planes` (`Proj-Cut-Pl` for short) was previously only used on polyhedral problems [24, 26, 27], we will show how it can be used to optimize over a non-polyhedral feasible area $\mathscr{P}^{\text{SDP}}$. The most difficult task is designing an algorithm to solve the projection problem over $\mathscr{P}^{\text{SDP}}$ as rapidly as possible. But once this has been accomplished, the generalization simply relies on interpreting the feasible area $\mathscr{P}^{\text{SDP}}$ as a polyhedron with infinitely-many constraints; it is impossible to record anyways an infinite number of constraints on a computer with finite memory resources.

[1] Program (1.D.1)-(1.D.4) can be exploited in a different manner: there may be other sets $\overline{\mathscr{D}}$ so that $S = \sum_{\overline{\mathbf{d}}\in\overline{\mathscr{D}}} \lambda_{\overline{\mathbf{d}}} \overline{\mathbf{d}}\,\overline{\mathbf{d}}^\top$, *e.g.*, obtained by computing the eigendecomposition of S. We could thus express S in a different basis that generates a different set of constraints $X \bullet \overline{\mathbf{d}}\,\overline{\mathbf{d}}^\top \geq 0$ that are tight for X in (1.C.4).

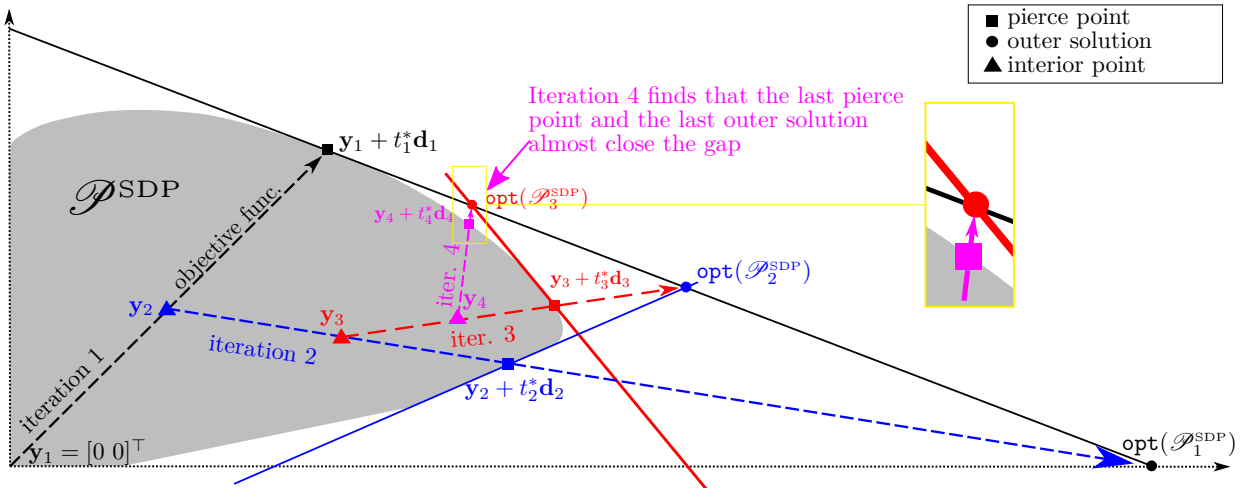


Figure 1.A: `Projective Cutting-Planes` illustration for $k = 2$. The red and black constraints are tangents to the non-linear SDP feasible area determined by $\mathcal{A}^\top\mathbf{y} \preceq A_0$. The blue constraint is linear, and so, it belongs to the set $\mathscr{C}$. The first iteration projects $\mathbf{y}_1 = \mathbf{0} = [0\ 0]^\top$ along the objective function, as depicted by the black dashed arrow. At the end of iteration 1, the outer approximation $\mathscr{P}_1^{\text{SDP}} \supset \mathscr{P}^{\text{SDP}}$ only contains the largest triangle. Iteration 2 projects the midpoint $\mathbf{y}_2$ of this black arrow towards the optimal outer solution $\texttt{opt}(\mathscr{P}_1^{\text{SDP}})$. This generates a second facet (solid blue line) that is added to the facets of $\mathscr{P}_1^{\text{SDP}}$ to construct $\mathscr{P}_2^{\text{SDP}}$. The process is repeated until finding a very small gap at iteration 4. The objective value of the last pierce point is hardly distinguishable from the optimal one in this figure.

We now present the key operations, working on (1.C.1)-(1.C.4). To start, `Proj-Cut-Pl` requires an initial feasible point $\mathbf{y}_1$, that may be generated as we will describe in Section 2.2, *i.e.*, either using Weyl's inequalities or by seeking to project an exterior point towards the SDP cone (in a repeated manner). The first iteration projects this point $\mathbf{y}_1$ along the objective function $\mathbf{d}_1 = \mathbf{b}$, so as to advance along the direction of greatest rate of improvement in objective value. This generates a first pierce point $\mathbf{y}_1 + t_1^*\mathbf{d}_1$ and a first constraint of $\mathscr{P}^{\text{SDP}}$, which is used to construct a first polyhedral outer approximation $\mathscr{P}_1^{\text{SDP}}$ of $\mathscr{P}^{\text{SDP}}$. Iteration 2 chooses an inner point $\mathbf{y}_2$ between $\mathbf{y}_1$ and $\mathbf{y}_1 + t_1^*\mathbf{d}_1$ and it projects $\mathbf{y}_2$ along a direction $\mathbf{d}_2$ pointing towards the current outer optimal solution $\texttt{opt}(\mathscr{P}_1^{\text{SDP}})$. This generates a new pierce point $\mathbf{y}_2 + t_2^*\mathbf{d}_2$ and a new constraint of $\mathscr{P}^{\text{SDP}}$ that separates $\texttt{opt}(\mathscr{P}_1^{\text{SDP}})$; this constraint also serves to refine $\mathscr{P}_1^{\text{SDP}}$ and to construct a tighter outer approximation $\mathscr{P}_2^{\text{SDP}} \subseteq \mathscr{P}_1^{\text{SDP}}$ of $\mathscr{P}^{\text{SDP}}$.

Figure 1.A illustrates this process over multiple iterations. At each iteration $\texttt{it}$, an inner solution $\mathbf{y}_{\texttt{it}} \in \mathscr{P}^{\text{SDP}}$ is projected towards the direction $\mathbf{d}_{\texttt{it}}$ of the current optimal outer solution $\texttt{opt}(\mathscr{P}_{\texttt{it}-1}^{\text{SDP}})$, *i.e.*, we take $\mathbf{d}_{\texttt{it}} = \texttt{opt}(\mathscr{P}_{\texttt{it}-1}^{\text{SDP}}) - \mathbf{y}_{\texttt{it}}$. The (non-orthogonal) projection sub-problem asks to determine $t_{\texttt{it}}^* = \max\{t : \mathbf{y}_{\texttt{it}} + t\mathbf{d}_{\texttt{it}} \in \mathscr{P}^{\text{SDP}}\}$. For this, one has to find the pierce (hit) point $\mathbf{y}_{\texttt{it}} + t_{\texttt{it}}^*\mathbf{d}_{\texttt{it}}$ and a (first-hit) constraint of $\mathscr{P}^{\text{SDP}}$, which is added to the constraints of $\mathscr{P}_{\texttt{it}-1}^{\text{SDP}}$ to construct $\mathscr{P}_{\texttt{it}}^{\text{SDP}}$. This implicitly solves the separation sub-problem for all points $\mathbf{y}_{\texttt{it}} + t\mathbf{d}_{\texttt{it}}$ with $t \in \mathbb{R}_+$, because the above first-hit constraint separates all solutions $\mathbf{y}_{\texttt{it}} + t\mathbf{d}_{\texttt{it}}$ with $t > t_{\texttt{it}}^*$ and proves $\mathbf{y}_{\texttt{it}} + t_{\texttt{it}}^*\mathbf{d}_{\texttt{it}} \in \mathscr{P}^{\text{SDP}}\ \forall t \in [0, t_{\texttt{it}}^*]$. At the next iteration $\texttt{it}+1$, `Projective Cutting-Planes` takes a new interior point $\mathbf{y}_{\texttt{it}+1}$ on the segment joining $\mathbf{y}_{\texttt{it}}$ and $\mathbf{y}_{\texttt{it}} + t_{\texttt{it}}^*\mathbf{d}_{\texttt{it}}$ and projects $\mathbf{y}_{\texttt{it}+1}$ along $\mathbf{d}_{\texttt{it}+1} = \texttt{opt}(\mathscr{P}_{\texttt{it}}^{\text{SDP}}) - \mathbf{y}_{\texttt{it}+1}$.

A series of papers on linear problems [24, 26, 27, 25] describe how the projection logic can offer a few advantages that do not exist in standard `Cutting-Planes`, pushing its potential to a higher level of (practical) performance. In addition to generating feasible solutions along the iterations, it can identify tighter constraints and improve the general dynamics of the convergent process. For instance, it can eliminate the "yo-yo" effects (heavy oscillation) of the intermediate outer solutions that arise very often in many `Cutting-Planes` [26, Figure 3]. The degeneracy risks can be reduced (see [27, Remark 2]). We refer to the aforementioned papers for more literature related to projection ideas (or line-search or in-out separation) in the context of polyhedral problems.

1.2 New contributions in SDP optimization

1.2.1 Positioning `Proj-Cut-Pl` among other SDP methods

While the general intuitive idea behind the proposed (non-orthogonal) projection is not new, the overall SDP optimization algorithm built upon it is, to the best of our knowledge, original in SDP programming. Although it constructs a sequence of interior points converging towards the optimal solution like an Interior Point Method (IPM), it is powered by a different idea. Recall an IPM moves from solution to solution by advancing along a Newton direction at each iteration, in an attempt to solve first order optimality conditions. Advancing along a Newton direction is not really equivalent to performing a projection. A projection executes a full step-length (advancing up to the pierce point) while a Newton step in an IPM does not even advance to fully solve the first order conditions – which actually correspond to a primal objective function penalized by a barrier term that only vanishes at the last iteration.

This work actually originated from the empirical conjecture that IPMs might run out of gas for $n \geq 2000$ when using dense matrices (or perhaps $n \geq 7000$ for sparse matrices). While IPMs can offer everything one can desire in theory, they need to iteratively solve (very) large Newton systems. For this purpose, the fastest IPM variants end up computing a positive-definite (Schur complement) matrix of size $k \times k$ at each iteration [34, (23)]; the calculation of each element of this matrix involves multiplying several matrices [10, (5.b)], introducing an important overall computational bottleneck. The whole operation may require $O(k^2n^2 + kn^3)$ operations, see, *e.g.*, [12, p. 66], [29, p. 26], or point 1 at the end of [31, §4]. However, to our knowledge, considerable development efforts (many man-hours) have been devoted to accelerating such calculations and practical IPM software can seriously reduce this very high theoretical $O(k^2n^2 + kn^3)$ complexity.[2]

Different well-established methods in general conic or non-linear optimization have also been applied to SDP programming. This include first-order approaches that mainly exploit gradient-based information, such as the augmented Lagrangian method or the alternating direction method of multipliers. The augmented Lagrangian method (also called the method of multipliers) works on (1.B.1)–(1.B.3) and replaces each constraint (1.B.2) with a double penalty in the objective: a standard linear Lagrangian penalty and a second one that measures the Euclidean norm of the infeasibility of the constraint [2, § 3.1]. Many highly-successful solvers exploit different low-rank properties. In the above study [2], an optimal SDP matrix S in (1.B.1)–(1.B.3) may have a form $S = RR^\top$, where R has n rows but very few columns, which leads to the reformulation of the objective as a quadratic function with a (very) limited number of variables. The IPM from [18] exploits the fact that the constraints (1.B.2) can have rank 1 if they are associated to sum-of-square decompositions in polynomial optimization. In the IPM framework from [10], the input matrices are low-rank in the sense that they have a few large outlying eigenvalues.

Another highly-successful (first-order) SDP approach is the `ConicBundle` method [13, 11], that reformulates the semidefinite program as an eigenvalue optimization problem, which is then solved by a subgradient method. This eigenvalue optimization problem arises as follows. The `ConicBundle` requires a constant trace constraint in the dual (1.B.1)-(1.B.3). Consider the expression $\mathcal{A}^\top \mathbf{y} = \sum_{i=1}^{k} A_i y_i$ from (1.C.2) and suppose we have $A_k = I_n$ and $b_k > 0$, which reduces to adding $\texttt{trace}(S) = b_k$ to the dual (1.B.1)-(1.B.3). The optimal manner to enforce $A_0 - \mathcal{A}^\top \mathbf{y} \succeq \mathbf{0}$ and maximize the objective is to set $y_k = \lambda_{\min}\left(A_0 - \sum_{i=1}^{k-1} A_i y_i\right)$. Isolating this variable from the other $k-1$ variables, the goal is to maximize $b_k y_k$ which reduces to maximizing this minimum eigenvalue function over the decision variables $y_1, y_2, \ldots, y_{k-1}$. This method has to maintain a cutting model that underestimates the concave non-smooth minimum eigenvalue function.

`Projective Cutting-Planes` was deliberately designed to be more lightweight than most (if not all) existing SDP optimization methods. Although this means it is intended to be very fast, it may also offer certain benefits beyond standard running-time competition. For instance, it is very naturally adaptable to re-optimization tasks such as the following: after solving a (1.C.1)-(1.C.4) program, solve the same program again after adding a new linear constraint, imposed, *e.g.*, by a branching rule in a `Branch and Bound` (as in Section 5.6).[3] Likewise, what happens if the coefficients of the given nominal linear constraints (1.C.3) vary according to a robust optimization paradigm and produce a prohibitively-large set $\mathscr{C}$ of robust linear inequalities (like in Section 5.5)? The projection algorithm for the associated robust LP (ignoring the SDP

[2] Regarding `Mosek`, search "Schur" in https://docs.mosek.com/slides/2015/ismp-2015-andersen-linear.pdf from ISMP 2015.

[3] While work on warm-starting IPMs does exist, the IPM paradigm that does not lend itself very easily or naturally to warm starting or re-optimization. We are not aware of any currently-available SDP software that can solve a sequence of slightly modified SDP programs without restarting from scratch.

part) was already presented in [27]. To make `Proj-Cut-Pl` work for the resulting *robust SDP program*, it needs to call two projection algorithms, for two different sub-problems, one for the prohibitively-many linear robust inequalities $\mathscr{C}$ from (1.C.3) and one for the SDP component $\mathscr{D}$ from (1.C.4).

1.2.2 Main SDP novelties

Although the basic projection idea was first developed in a polyhedral LP context, the overall project involved overcoming many SDP-specific obstacles. We present here a few SDP novelties resulting from this endeavor.

- We introduce an SDP-specific pseudo-code of `Proj-Cut-Pl` (Section 2) that extends previous algorithmic descriptions developed in a purely polyhedral context, by leveraging all structural features and mathematical intricacies specific to SDP optimization. As a novelty compared to IPMs, it constructs both a primal-feasible and a dual-feasible SDP solution at each iteration. This may even enable `Proj-Cut-Pl` to finish in few iterations if the goal is to only reach a very loose gap (*e.g.*, the same rounded-down value of the lower and upper bounds) like in the `Branch and Bound` for binary SDP optimization from Section 5.6. Yet the underlying `Cutting-Planes` paradigm has an important inherent weakness: it may need to generate too many linear constraints to obtain a sufficiently tight and descriptive polyhedral approximation of the SDP cone (around the global optimal solution).
- We design a numerically-exact SDP algorithm to solve the projection under ideal exact arithmetic conditions (Section 3). It proved to be (much) faster than adapting existing related methods for the same task. This projection algorithm may be useful beyond solving SDP programs, because related problems have been considered before in other areas, *e.g.*, under other names like "restoring definiteness" or "shrinking" (see Section 4.2).
- We also present a numerically-inexact algorithm computing the projection up to a precision. It returns an underestimated step-length and a constraint that is not tangent to the SDP cone at the true pierce point (Section 4.1); yet, it can sometimes be faster and more useful for our most practical needs.
- The overall project encompasses a variety of additional intricate adaptations and engineering complexities that will be progressively revealed throughout the ensuing discussion. Perhaps a novel contribution is the design of a (first) heuristic for the SDP feasibility problem that (still) uses a form of projection, but along an exterior-to-interior direction (Section 2.2). We do need such heuristic, because the overall approach requires a feasible solution to start from. The `Matlab` code source contains roughly 7000 lines.

The reminder is organized as follows. The next section presents the overall `Proj-Cut-Pl` for SDP optimization, considering the projection sub-problem as a black box. Section 3 is devoted to solving the SDP projection in exact arithmetic. Section 4 describes a numerically-inexact but practical (fast) projection approach using tolerance windows. Sections 3 and 4 both end by discussing related literature (Section 3.5 and 4.2). Multiple numerical results are provided in Section 5, before concluding in Section 6. An appendix presents the results of an implementation based on sparse matrix representation with n going up to 30000.

2 Specification of `Proj-Cut-Pl` in SDP programming

2.1 The overall algorithmic logic

2.1.1 The `Cutting-Planes` base framework

`Proj-Cut-Pl` still adheres to the following basic `Cutting-Planes` logic. At each iteration `it`, it constructs an outer approximation $\mathscr{P}^{\text{SDP}}_{\texttt{it}} \supset \mathscr{P}^{\text{SDP}}$ of $\mathscr{P}^{\text{SDP}}$ obtained by keeping only a manageable subset of the otherwise prohibitively-many constraints of $\mathscr{P}^{\text{SDP}}$. As in a standard `Cutting-Planes`, each intermediate outer optimal solution $\mathbf{y}_{\text{out}} = \texttt{opt}(\mathscr{P}^{\text{SDP}}_{\texttt{it}})$ is separated at iteration $\texttt{it}+1$; however, this separation is calculated using a more general non-orthogonal projection. The constraint that separates $\mathbf{y}_{\text{out}}$ is used to enrich the current constraint set and compute a more refined outer approximation $\mathscr{P}^{\text{SDP}}_{\texttt{it}+1} \subsetneq \mathscr{P}^{\text{SDP}}_{\texttt{it}}$. The process is repeated by (re-)optimizing over $\mathscr{P}^{\text{SDP}}_{\texttt{it}+1}$ at the next iteration, until the intermediate optimal outer solution becomes non-separable, and thus, globally optimal. Unless otherwise indicated (for the robust optimization problem from Section 5.5), we hereafter consider the initial linear constraints $\mathscr{C}$ can be enumerated in practice and inserted into the description of the first initial outer approximation $\mathscr{P}^{\text{SDP}}_{0}$.

2.1.2 The projection

Definition 2.A. *(SDP projection sub-problem) Given an interior feasible point* $\mathbf{y} \in \mathscr{P}^{SDP} \subsetneq \mathbb{R}^k$ *and a direction* $\mathbf{d} \in \mathbb{R}^k$*, the projection sub-problem* $\mathbf{y} \to \mathbf{d}$ *asks to find:*

1) *the maximum step length* t^* *such that* $\mathbf{y} + t^*\mathbf{d}$ *is feasible in* $\mathscr{P}^{SDP}$*, i.e.,* $t^* = \max\{t \geq 0 : \mathbf{y} + t\mathbf{d} \in \mathscr{P}^{SDP}\}$*. The solution* $\mathbf{y} + t^*\mathbf{d}$ *is referred to as the pierce point. If* $\mathbf{y} + t\mathbf{d}$ *is a ray of* $\mathscr{P}^{SDP}$*, the sub-problem returns* $t^* = \infty$*. This question in the variables* $\mathbf{y}$ *corresponds to the following problem in the world of SDP matrices: project the SDP matrix* $X = A_0 - \mathcal{A}^\top\mathbf{y} \succeq \mathbf{0}$ *towards the direction* $D = -\mathcal{A}^\top\mathbf{d}$*. This reduces to finding the maximum* t^* *such that* $X + t^*D \succeq \mathbf{0}$*.*
2) *a first-hit constraint* $(\mathbf{a}, c_a) \in \mathscr{C} \cup \mathscr{D}$ *satisfied with equality by the pierce point, i.e., such that* $\mathbf{a}^\top(\mathbf{y} + t^*\mathbf{d}) = c_a$*; such a constraint certainly exists if* $t^* \neq \infty$*. It is meant to correspond to a supporting hyperplane of the feasible area, which is always possible in a pure LP, but may be impossible in some exceptional SDP cases (see below). In the world of n-by-n SDP matrices, this may translate to determining a vector* $\mathbf{v} \in \mathbb{R}^n$ *such that condition (2.A) below proves* $X + t^*D$ *is on the boundary of the SDP cone. Once* $\mathbf{v} \in \mathbb{R}^n$ *is determined, we can install a constraint (1.C.4) in the* $\mathbb{R}^k$ *space by setting* $c_a = A_0 \bullet \mathbf{v}\mathbf{v}^\top$ *and* $a_i = A_i \bullet \mathbf{v}\mathbf{v}^\top$ $\forall i \in [1..k]$*.*

$$(X + t^*D) \bullet \mathbf{v}\mathbf{v}^\top = 0 \text{ and } D \bullet \mathbf{v}\mathbf{v}^\top < 0. \tag{2.A}$$

We will pay attention to a degenerate case that illustrates a broader underlying phenomenon: if $X = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right]$ and $D = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, the projection sub-problem needs to return $t^* = 0$, but there is no $\mathbf{v} \in \mathbb{R}^2$ that satisfies (2.A) exactly, even if using an arbitrarily high precision. Indeed, the SDP cone does not admit a supporting hyperplane at X that separates all $X + tD$ $\forall t > 0$ according to (2.A); however, we can approach (2.A) asymptotically: for any $t^* > 0$, including when $t^* \to 0$, the vector $\mathbf{v} = [1 \ \frac{-1}{2t^*}]^\top$ satisfies (2.A) exactly.

2.1.3 The main steps

The proposed approach first applies Section 2.2 to find an initial feasible solution $\mathbf{y}_1$; if this is particularly difficult, the given instance is outside the scope of this paper as indicated by its very title. Section 2.3 is then used to generate some initial eigen-cuts $\mathscr{D}$ (or even some standard initial linear constraints) to define a first outer approximation $\mathscr{P}^{\text{SDP}}_0 \supsetneq \mathscr{P}^{\text{SDP}}$. The first projection is $\mathbf{y}_1 \to \mathbf{d}_1$ using $\mathbf{d}_1 = \mathbf{b}$. This leads to a first pierce point $\mathbf{y}_1 + t^*_1\mathbf{d}_1$ and a first-hit constraint $\mathbf{a}^\top\mathbf{y} \leq c_a$, which is added to the set of initial constraints that define $\mathscr{P}^{\text{SDP}}_0$ to construct $\mathscr{P}^{\text{SDP}}_1$. Finally, the following steps are executed from iteration $\mathtt{it} = 2$ on.

(1) Select an inner solution $\mathbf{y}_{\mathtt{it}}$ on the segment joining $\mathbf{y}_{\mathtt{it}-1}$ and $\mathbf{y}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$, *i.e.*, between the previous inner solution and the last pierce point.

(2) Take the direction $\mathbf{d}_{\mathtt{it}} = \mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}) - \mathbf{y}_{\mathtt{it}}$, so as to advance towards the current optimal (outer) solution $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1})$. We mention a degenerate case: if $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1})$ is an extreme ray $\mathbf{d}_\infty$ of $\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}$ such that $\{\mathbf{y} + \lambda\mathbf{d}_\infty : \lambda \geq 0\} \subset \mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}$ $\forall \mathbf{y} \in \mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}$, we take $\mathbf{d}_{\mathtt{it}} = \mathbf{d}_\infty$.

(3) Solve the projection sub-problem $\mathbf{y}_{\mathtt{it}} \to \mathbf{d}_{\mathtt{it}}$ to determine the maximum step length $t^*_{\mathtt{it}}$, the pierce point $\mathbf{y}_{\mathtt{it}} + t^*_{\mathtt{it}}\mathbf{d}_{\mathtt{it}}$, and a first-hit constraint $\mathbf{a}^\top\mathbf{y} \leq c_a$. If t^* is ∞, we conclude the SDP program is unbounded.

(4) If $t^*_{\mathtt{it}} = 1$, then $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1})$ is the sought optimal solution and the algorithm can stop.
If $t^*_{\mathtt{it}} < 1$, then the current outer solution $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1})$ can be separated and we perform the following:

- insert the first-hit constraint $\mathbf{a}^\top\mathbf{y} \leq c_a$ into the current constraint set, producing a more refined outer approximation $\mathscr{P}^{\text{SDP}}_{\mathtt{it}} \subsetneq \mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}$.
- calculate a new intermediate optimal outer solution $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}})$ by re-optimizing over $\mathscr{P}^{\text{SDP}}_{\mathtt{it}}$, which may only require performing a dual Simplex pivot.
- if $\mathbf{y}_{\mathtt{it}} + t^*_{\mathtt{it}}\mathbf{d}_{\mathtt{it}}$ and $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}})$ are close enough in terms of the objective value, stop and return $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}})$.
- update $\mathtt{it} \leftarrow \mathtt{it} + 1$ and go to Step (1).

A key question is the choice of the interior point $\mathbf{y}_{\mathtt{it}}$ at Step (1) of each iteration $\mathtt{it}$. This question was generally addressed in more detail in [24, 26, 27], *e.g.*, see a visual illustration comparing the effect of an aggressive *long-step* choice and of a "cautious" more centered *small-step* choice in [26, Fig. 2 (p. 1011)]. This previous work suggest it is often not a good idea to advance along a maximum step by choosing the best feasible solution found up to iteration $\mathtt{it}$ (the last pierce point) via $\mathbf{y}_{\mathtt{it}} = \mathbf{y}_{\mathtt{it}-1} + t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$. This greedy

choice can lead to poor results in the long run, even if it does offer an interesting advantage: each new iteration pushes the next inner solution $\mathbf{y}_{\mathtt{it}}$ to the maximum extent possible, leading to monotonically increasing lower bounds, as in [27, Fig. 3 (§ 3.2.2)]. However, the drawback is that $\mathbf{y}_{\mathtt{it}}$ can fluctuate too much from iteration to iteration, producing a bang-bang (heavy oscillation) effect.[4] We thus define $\mathbf{y}_{\mathtt{it}} = \mathbf{y}_{\mathtt{it}-1} + \alpha t^*_{\mathtt{it}-1}\mathbf{d}_{\mathtt{it}-1}$ with a cautious rather short step $\alpha = 0.3$. In a broad sense, this is reminiscent of an idea of IPMs, which is to avoid long steps or touching the boundary before fully converging.

2.1.4 The primal-dual SDP solutions produced at each iteration and the duality gap

The above steps are finitely convergent, if we consider a computing machine that records all numbers in limited precision, so that it can only generate finitely-many constraints. Since we implicitly solve a separation sub-problem on $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1})$ at each iteration $\mathtt{it}$, the outer solution changes at each iteration (except the last one). But recall each such outer solution is a vertex of polytope $\mathscr{P}^{\text{SDP}}_{\mathtt{it}-1}$, situated at the intersection of a subset of (the hyperplanes associated with) some constraints of the overall problem. Since these initial constraints are considered finite, the number of such intersection points is also finite.

This (unavoidable) idea of finite precision may render the separation of certain pathological SDP matrices quite challenging. We illustrate this issue on the program below whose feasible solutions are given by $y_1 = 0$ and $y_2 \geq 0$. Is it possible to use the cuts $X \bullet \mathbf{d}\mathbf{d}^\top \geq 0$ proposed in (1.C.4) to separate a solution with $y_1 = -1$? Restricting to the top-left 2×2 block, such a cut is equivalent to $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 2y_2 \end{smallmatrix}\right] \bullet [1\ \epsilon]^\top [1\ \epsilon] = 2\epsilon + 2\epsilon^2 y_2 \geq 0$ for some $\epsilon \in \mathbb{R}$. The tightest variant is achieved by assigning a negative value for ϵ; let us divide the constraint by $2\epsilon < 0$ to obtain $1 + \epsilon y_2 \leq 0$, or $|\epsilon| y_2 \geq 1$. When y_2 is very large, ϵ needs to be very close to 0 to separate it. If the machine has limited precision, it is in practice impossible to generate constraints that can separate all infeasible solutions of the form $(-1, y_2)$ with $y_2 \to \infty$. Avoiding such numerical imprecisions remains very challenging for many SDP solvers when dealing with such ill-conditioned input: notice that increasing X_{11} by any infinitesimal value changes the optimal solution from 0 to 1.

$$\begin{array}{ll} \max\limits_{y_1, y_2} & -y_1 \\ \text{s.t.} & X = \begin{bmatrix} 0 & -y_1 & 0 \\ -y_1 & 2y_2 & 0 \\ 0 & 0 & 1+y_1 \end{bmatrix} \succeq 0 \end{array}$$

Remark 2.B. *Each intermediate inner solution* $\mathbf{y}_{it}$ *and outer solution* $\mathtt{opt}(\mathscr{P}^{SDP}_{it})$ *discovered by* `Proj-Cut-P`l *satisfy the following:*

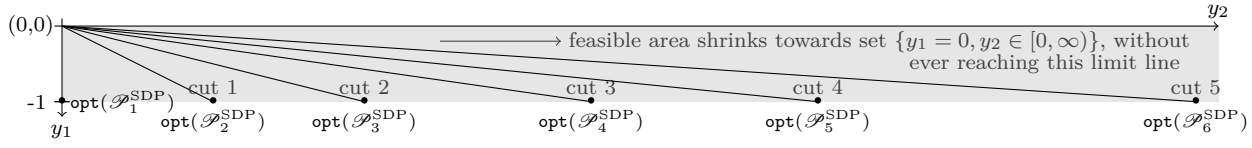
a. $\mathbf{y}_{it}$ *is a primal SDP solution;*

b. $\mathtt{opt}(\mathscr{P}^{SDP}_{it})$ *can be used to construct a dual SDP solution.*

(a) The projection sub-problem ensures by construction that $\mathbf{y}_{\mathtt{it}}$ is feasible in the primal SDP.

(b) Given the outer optimal solution $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_{\mathtt{it}})$ of the primal LP (1.C.1)-(1.C.4) associated with some generated constraints $\mathscr{C}$ and $\mathscr{D}$, we **cannot** construct a primal-feasible solution of the same objective value in the main primal SDP (1.A.1)-(1.A.3). Yet, we can determine a dual LP solution of the same objective value in the dual LP (1.D.1)-(1.D.4). Since any LP solver can also compute the optimal dual solution $\boldsymbol{\lambda}$ in (1.D.1)-(1.D.4), this $\boldsymbol{\lambda}$ generates an SDP matrix $S = \sum_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top$ as described in (1.D.3), so that S is also feasible in the main dual SDP (1.B.1)-(1.B.3). □

In the above example, the outer solutions $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_1)$, $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_2)$, $\mathtt{opt}(\mathscr{P}^{\text{SDP}}_3)$, ... correspond to a sequence of *dual SDP solutions* according to above remark. But there is a duality gap: the primal optimum is 0 and the dual optimum is 1. It is thus theoretically impossible to have enough cutting planes to make the outer solution reach an objective value below 1. The generated outer solutions satisfy $y_1 = -1$ and make y_2 grow to infinity, *i.e.*, without being able to separate all $(-1, y_2)$ when $y_2 \to \infty$. We illustrate below the first five cuts that may be generated this way, each one shrinking the feasible area to the right.



[4] As analyzed in [27, § 3.3], this aggressive variant proved useful only for the graph coloring `Column Generation` model from [26, § 4.2] and (to some extent) for the Multiple-Length Cutting-Stock `Column Generation` model from [27, § 3.2.2].

Yet, under the hypothesis that (1.C.1)-(1.C.4) only contains finitely-many constraints, `Proj-Cut-Pl` is convergent. From some iteration $\overline{\texttt{it}}$ on, no new constraint can be generated and $\texttt{opt}(\mathscr{P}^{\text{SDP}}_{\texttt{it}})$ will remain constant after $\texttt{it} \geq \overline{\texttt{it}}$, and so will remain each subsequent pierce point.

On the primal side, `Proj-Cut-Pl` can easily start with some inner point such that $y_1 = 0$. But each projection towards an outer solution with $y_1 = -1$ will stop at a null step $t^* = 0$ so that the initial feasible solution can never be improved.

2.2 Determining a first feasible inner point $\mathbf{y}_{\text{in}}$

While `Projective Cutting-Planes` was not primarily designed as a method for determining the feasibility status of an SDP program, this section does describe two ideas to find a first inner point, since it is needed to start the overall method. In a first stage, we use Weyl's inequalities to (try to) construct an LP that is included in the feasible area determined by $\mathcal{A}^\top \mathbf{y} \preceq A_0$. The second stage is based on a different idea. It is an iterative procedure that starts from an infeasible $\mathbf{y}$ and tries to follow exterior-to-interior projections to push $\mathbf{y}$ towards the interior of the SDP feasible area. It can thus be seen as a heuristic for the SDP feasibility problem, *i.e.*, it tries to solve the considered SDP program (1.A.1)-(1.A.3) with a null objective function $\mathbf{b} = \mathbf{0}_k$. To solve this variant with $\mathbf{b} = \mathbf{0}_k$, some Interior Point Methods (IPMs) may take up to a third of the computational time they need for a non-null $\mathbf{b} \in \mathbb{R}^k$.

Stage 1 A first inner point using Weyl's inequalities
To build an LP included in the feasible area of (1.A.1)-(1.A.3), it is enough to determine some linear constraints that ensure $\mathcal{A}^\top \mathbf{y} \preceq A_0$, *i.e.*, that guarantee $\lambda_{\max}(\mathcal{A}^\top \mathbf{y} - A_0) \leq 0$, where $\lambda_{\max}(\cdot)$ and $\lambda_{\max}(\cdot)$ are the maximum, resp., minimum eigenvalue function. We propose using Weyl's inequalities that imply $\lambda_{\max}(A + B) \leq \lambda_{\max}(A) + \lambda_{\max}(B)$ for any two symmetric matrices A and B. If we have $\mathbf{y} \geq \mathbf{0}_k$, these inequalities, coupled with $\lambda_{\max}(-A_0) = -\lambda_{\min}(A_0)$, would make the following relation sufficient to guarantee $\mathcal{A}^\top \mathbf{y} \preceq A_0$:

$$\sum_{i=1}^{k} \lambda_{\max}(A_i) \cdot y_i \leq \underbrace{\lambda_{\min}(A_0)}_{-\lambda_{\max}(-A_0)} . \tag{2.B}$$

In particular, if $\lambda_{\min}(A_0) \geq 0$, this makes $\mathbf{y} = \mathbf{0}_k$ feasible with regards to the SDP constraint $\mathcal{A}^\top \mathbf{y} \preceq A_0$. However, `Projective Cutting-Planes` does not necessarily start with $\mathbf{y}_{\text{in}} = \mathbf{0}_k$ when $\mathbf{0}_k$ is feasible. It constructs an LP by replacing (1.A.2) with (2.B) in (1.A.1)-(1.A.3). The optimal solution of this LP may have a higher quality than $\mathbf{0}_k$. But as stated above, this LP can be constructed this way only for $\mathbf{y} \geq \mathbf{0}_k$. Otherwise, if $\mathbf{y} \not\geq \mathbf{0}$, we extend the idea as follows. We first define k variables $\overline{\mathbf{y}}$ on which we impose $\overline{y}_i \geq \lambda_{\max}(A_i)y_i$ and $\overline{y}_i \geq \lambda_{\min}(A_i)y_i$, which suffices to ensure $\overline{y}_i \geq \lambda_{\max}(A_i y_i)$. The function $y_i \to \overline{y}_i$ is piecewise convex, similar to the absolute value function. Finally, we (try to) construct a first feasible solution by solving a similar LP but extended with variables $\overline{\mathbf{y}}$ and replacing (2.B) with $\sum \overline{y}_i \leq \lambda_{\min}(A_0)$.

It's only if $\lambda_{\min}(A_0) < 0$ that (2.B) cannot be used to generate a feasible solution using the above approach. This is the only case in which $\mathbf{y} = \mathbf{0}_k$ can be infeasible in (2.B), letting aside the initial linear constraints $\mathscr{C}$ from (1.C.3). In such scenario, it is even impossible to find a single $\mathbf{y}$ that satisfies (2.B) if we have $\lambda_{\max}(A_i) \geq 0 \ \forall i \in [1..k]$ when $\mathbf{y}$ is non-negative, or $\lambda_{\max}(A_i) = 0 \ \forall i \in [1..k]$ if $\mathbf{y}$ is free.

Stage 2: A heuristic for the feasibility SDP problem using exterior-to-interior projections
We next propose solving the feasibility problem (*i.e.*, considering $\mathbf{b} = \mathbf{0}_k$) by iteratively attempting to project a non-feasible SDP matrix X^{out} towards the SDP cone. We can start from an infeasible $\mathbf{y}_{\text{out}} = [y_1^{\text{out}} \ y_2^{\text{out}} \dots y_k^{\text{out}}]^\top$ that only has to satisfy the linear constraints $\mathscr{C}$ from (1.C.3), so that the associated matrix $X^{\text{out}} = A_0 - \sum_{i=1}^{k} A_i y_i^{\text{out}}$ from (1.C.2) is outside the SDP cone. This means X^{out} has a set of $\bar{j}$ negative eigenvalues $\lambda_1 \leq \lambda_2 \leq \lambda_3 \cdots \leq \lambda_{\bar{j}} < 0$ associated with some eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots \mathbf{v}_{\bar{j}}$ such that

$$f_j(\mathbf{y}_{\text{out}}) := \underbrace{\left(A_0 - \sum_{i=1}^{k} A_i y_i^{\text{out}}\right)}_{X^{\text{out}}} \bullet \mathbf{v}_j \mathbf{v}_j^\top = \lambda_j < 0, \ \forall j \in [1..\bar{j}].$$

It is not computationally difficult to calculate all products $A_i \bullet \mathbf{v}_j\mathbf{v}_j^\top$ for each $i \in [0..k]$ and $j \in [1..\bar{j}]$ to construct all coefficients of linear functions $f_j(\mathbf{y}) = A_0 \bullet \mathbf{v}_j\mathbf{v}_j^\top - \sum_{i=1}^{k} (A_i \bullet \mathbf{v}_j\mathbf{v}_j)\, y_i$. Each f_j with $j \in [1..\bar{j}]$ will satisfy $f_j(\mathbf{y}_{\text{in}}) \geq 0$ for any feasible optimal solution $\mathbf{y}_{\text{in}}$ of the initial SDP program. This implies that by advancing along the direction $\mathbf{y}_{\text{out}} \rightarrow \mathbf{y}_{\text{in}}$, each function f_j is strictly increasing. Thus, $\mathbf{y}_{\text{out}} \rightarrow \mathbf{y}_{\text{in}}$ may represent a projection from the exterior to the interior of the SDP cone. The core principle of the proposed heuristic is to iteratively identify and pursue such directions until reaching a point within the SDP cone. To determine each direction, we solve the following LP for some reasonable $\Delta = 100$ that simply defines a bounding box around $\mathbf{y}_{\text{out}}$:

$$\min_{t,\mathbf{y}} \left\{ t :\ t \geq f_j(\mathbf{y}) - f_j(\mathbf{y}_{\text{out}})\ \forall j \in [1..\bar{j}],\ \mathbf{y}_{\text{out}} - \Delta \leq \mathbf{y} \leq \mathbf{y}_{\text{out}} + \Delta,\ \mathbf{a}^\top\mathbf{y} \leq c_a\ \forall (\mathbf{a}, c_a) \in \mathscr{C} \right\}. \tag{2.C}$$

If the optimal value of this LP is not strictly positive, the overall SDP optimization algorithm stops by reporting the given instance is infeasible. This comes from the fact that any feasible solution $\mathbf{y}_{\text{in}}$ of the initial SDP model yields $f_j(\mathbf{y}_{\text{in}}) \geq 0\ \forall j \in [1..\bar{j}]$ so that each function f_j increases by advancing from $\mathbf{y}_{\text{out}}$ to this $\mathbf{y}_{\text{in}}$ that satisfies $f_j(\mathbf{y}_{\text{in}}) \geq 0 > f_j(\mathbf{y}_{\text{out}})\ \forall j \in [1..\bar{j}]$. Thus, $\mathbf{y} = \mathbf{y}_{\text{out}} + \tau(\mathbf{y}_{\text{in}} - \mathbf{y}_{\text{out}})$ remains feasible and attains a positive objective value in the above LP for some $\tau > 0$.

Based on these ideas, the proposed heuristic iterates the following two steps.

Step A (direction finding) Compute all $\bar{j}$ strictly negative eigenvalues of X^{out} and the associated eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{\bar{j}}$. If there is no negative eigenvalue (*i.e.*, if $\bar{j} = 0$), stop and report that X^{out} is SDP, meaning that the underlying $\mathbf{y}_{\text{out}}$ is also feasible. Otherwise, use these $\bar{j}$ vectors to compute functions $f_1, f_2, \ldots, f_{\bar{j}}$ and solve the above LP (2.C). Let $(\tau^*, \mathbf{y}^*)$ be its optimal solution. If $\tau^* \leq 0$, stop and report the instance is infeasible. Otherwise, $\tau^* > 0$ shows that the minimum eigenvalue of X^{out} can increase by replacing $\mathbf{y}_{\text{out}} \leftarrow \mathbf{y}_{\text{out}} + \tau(\mathbf{y}^* - \mathbf{y}_{\text{out}})$ for an appropriate $\tau > 0$. Ideally, $\mathbf{y}^*$ should point towards a feasible solution $\mathbf{y}_{\text{in}}$, but this is generally not the case.

Step B (determine the step length τ) Given a projection direction $\mathbf{y}_{\text{out}} \rightarrow \mathbf{y}^*$ computed above, the next question is to determine an appropriate step length τ. The ideal τ should maximize $g(\tau) := \lambda_{\min}(X^{\text{out}} + \tau D)$ where $D = -\sum_{i=1}^{k} (y_i^* - y_i^{\text{out}})\, A_i$. The sub-differential (the set of all subderivatives) of g at $\tau = 0$ is the set $-\sum_{i=1}^{k} (y_i^* - y_i^{\text{out}})\, A_i \bullet \mathbf{v}\mathbf{v}^\top$ where $\mathbf{v}$ belongs to the convex hull of the eigenvectors of X^{out} associated with the minimum eigenvalue λ_1. These eigenvectors surely contain the above $\mathbf{v}_1$ and possibly other vectors in subsequent positions among $\mathbf{v}_2, \mathbf{v}_3, \ldots \mathbf{v}_{\bar{j}}$. But all these eigenvectors are considered and included in the above LP (2.C), and so, all these subderivatives are strictly positive. This means we can always find a small-enough τ such that $g(\tau) > g(0)$: we can always increase the minimum eigenvalue by moving from $\mathbf{y}_{\text{out}}$ to $\mathbf{y}_{\text{out}} + \tau(\mathbf{y}^* - \mathbf{y}_{\text{out}})$, which can thus be seen as an exterior-to-interior projection.

In practice, the final τ is determined by evaluating the piecewise concave function g over multiple positive values $\tau_1, \tau_2, \tau_3, \ldots$ following a dichotomic logic. Evaluating a value of τ requires computing a minimum eigenvalue. The implemented dichotomic logic exploits the concavity of g; for instance, if we have $g(\tau_1) < g(\tau_2) < g(\tau_3) > g(\tau_4) > g(\tau_5)$ for some $\tau_1 < \tau_2 < \tau_3 < \tau_4 < \tau_5$, then the optimal τ surely belongs to the interval (τ_2, τ_4).

All the ideas above have been implemented into a practical algorithm that also integrates a few other engineering customizations to reduce the number of iterations. For instance, it is often useful to consider less than $\bar{j}$ eigenvectors, *i.e.*, only those associated with an eigenvalue not very far from λ_1. But towards the end of the algorithm, when λ_1 is close to 0, the opposite may be true, *i.e.*, it may be useful to consider more than $\bar{j}$ eigenvectors by integrating some eigenvectors associated with a positive close-to-zero eigenvalue. More customizations may be needed to make this dichotomic search reach its full potential, but such aspects could constitute alone the subject of a different short paper on a heuristic for the SDP feasibility problem.

Stage 3: Turning a non-SDP matrix into an SDP one by adding a penalizing artificial matrix For the sake of completeness, the implementation does try to solve any input instance. If the above two stages fail, we extend the initial program (1.C.1)-(1.C.4) with an artificial matrix $A_{k+1} = -I_n$ associated with a new variable $y_{k+1} \geq 0$ and a high penalty $b_{k+1} > 0$ in the objective, which can generate artificial feasibility as follows. Considering only the LP part of (1.C.1)-(1.C.4), one can find an $\mathbf{y} \in \mathbb{R}^k$ that satisfies all linear constraints $\mathscr{C}$ from (1.C.3), although $X = A_0 - \mathcal{A}^\top\mathbf{y}$ is not SDP. Setting $y_{k+1} = -\lambda_{\min}(X) > 0$, the minimum eigenvalue of $A_0 - \mathcal{A}^\top\mathbf{y} - y_{k+1}A_{k+1} = A_0 - \mathcal{A}^\top\mathbf{y} - \lambda_{\min}(X) \cdot I_n$ becomes zero, *i.e.*, the SDP constraint is satisfied by integrating this artificial term. A really-feasible SDP solution with $y_{k+1} = 0$ will

always be preferable to an artificially-feasible SDP solution with $y_{k+1} > 0$, provided the penalty b_{k+1} is high enough. We do not recomment using `Proj-Cut-Pl` if it needs a large penalty in this artificial procedure.

2.3 The initial outer approximation of the SDP cone

In the very beginning there may be no default constraint to describe even a loose outer approximation of $\mathscr{P}^{\text{SDP}}$. To avoid initiating the process with chaotic outer solutions, we add some initial constraints, to (try to) produce a reasonable first outer approximation $\mathscr{P}_0^{\text{SDP}} \supseteq \mathscr{P}^{\text{SDP}}$. The simplest constraint added for each $i \in [1..n]$ is $X_{ii} \geq 0$, since any SDP matrix has only non-negative elements on the diagonal. Expressing X in variables $\mathbf{y}$, this boils down to $[A_0]_{ii} - \sum_{j=1}^{k} [A_j]_{ii} y_j \geq 0 \ \forall i \in [1..n]$, which is the same as the unique constraint from [5, § 7.1]. This inequality can also be obtained from (1.C.4) by restricting it to binary eigen-cuts $\mathbf{d}$ with only one non-zero value.[5] Future work may include using similar valid inequalities from the literature, *e.g.*, from [19, § 2]

We now introduce a second type of initial constraints that only work if the variables $\mathbf{y}$ are non-negative (*i.e.*, $\mathbf{y} \geq \mathbf{0}_k$). One formulation of Weyl's inequalities imply that $\lambda_{\max}(A+B) \geq \lambda_{\min}(A) + \lambda_{\max}(B)$ for any two symmetric matrices A and B. We use this relation to derive the second inequality in the formula below.

$$0 \geq \lambda_{\max}(\mathcal{A}^\top \mathbf{y} - A_0) \geq \lambda_{\min}\left(\sum A_i y_i\right) + \lambda_{\max}(-A_0).$$

Still considering $\mathbf{y} \geq \mathbf{0}_k$, we can also apply yet another form of Weyl's inequalities, namely $\lambda_{\min}(A+B) \geq \lambda_{\min}(A) + \lambda_{\min}(B)$ to obtain:

$$\lambda_{\min}\left(\sum A_i y_i\right) + \lambda_{\max}(-A_0) \geq \sum \lambda_{\min}(A_i) y_i + \lambda_{\max}(-A_0)$$

Putting the two formulae above together, we obtain the following inequality that is integrated into $\mathscr{P}_0^{\text{SDP}}$.

$$\underbrace{\lambda_{\min}(A_0)}_{-\lambda_{\max}(-A_0)} \geq \sum \lambda_{\min}(A_i) y_i \tag{2.D}$$

Inserting the above constraint (2.D) makes the considered SDP program easier when it is defined in non-negative variables $\mathbf{y} \geq \mathbf{0}_k$. Moreover, (2.D) can be applied on any principal minor of the considered matrices, to obtain a larger family of constraints. Preliminary tests suggest this is computationally expensive without achieving an impressive speed-up.

This initial outer approximation for the case when the variables $\mathbf{y}$ are free is one of the greatest weaknesses of `Proj-Cut-Pl`. If this initial envelope is very inaccurate, important time will be wasted in the beginning, because the outer solution $\mathbf{y}_{\text{out}}$ can (chaotically) fluctuate too much from iteration to iteration, far from the optimal solution (bang-bang oscillation effects). Research-wise, devising a lightweight approach for generating a more accurate outer approximation $\mathscr{P}_0^{\text{SDP}}$ may offer an important general speed-up of `Proj-Cut-Pl`.

2.4 Pseudo-code and computational aspects

Algorithm 1 provides the pseudo-code of the SDP `Proj-Cut-Pl`; it goes beyond the overall algorithmic logic presented in Section 2.1, by taking into account the following specificities of the SDP geometry.

1. It is often useful to generate a second-hit vector $\mathbf{v}_2$, which can be characterized as a first-hit vector among the vectors $\mathbf{v}$ that are X-orthogonal to the first hit-vector $\mathbf{v}_1$, *i.e.*, such that $\mathbf{v}^\top X \mathbf{v}_1 = 0$. Formally, $\mathbf{v}_1$ and $\mathbf{v}_2$ shall satisfy the following

$$\mathbf{v}_1 \in \arg\max \left\{ t : \ (X + tD) \bullet \mathbf{v}\mathbf{v}^\top \geq 0 \forall \mathbf{v} \in \mathbb{R}^n \right\}$$
$$\mathbf{v}_2 \in \arg\max \left\{ t : \ (X + tD) \bullet \mathbf{v}\mathbf{v}^\top \geq 0 \forall \mathbf{v} \in \mathbb{R}^n \text{ s.t. } \mathbf{v}^\top X \mathbf{v}_1 = 0 \right\}.$$

[5] It may be useful to restrict (1.C.4) to binary eigen-cuts $\mathbf{d}$ with two non-zero values, but in such case a `Cutting-Planes` phase may be needed since integrating all such $\mathbf{d}$ is quite expensive. Preliminary tests suggest the idea of restricting to eigen-cuts with few non-zeros risks to be computationally taxing without warranting any impressive speed-up.

The step-length t_2^* associated with $\mathbf{v}_2$ satisfies $t_2^* \geq t^*$. We thus seek a form of second projection restricted to hit-vectors that belong to a space of size $n-1$, *i.e.*, to the null space of $X\mathbf{v}_1$. This may be useful because the above space of size $n-1$ may have its importance in the overall projection geometry. Returning $\mathbf{v}_2$ is optional in Line 9. The same logic can be applied to obtain a third vector $\mathbf{v}_3$, but using too many such constraints may slow down the LP solver.

2. It may sometimes (still) be useful to cut by separation the current outer solution $X+D = A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}}$. If the minimum eigenvalue of this outer solution is $\lambda_{\min} < 0$ and the associated normalized eigenvector is $\mathbf{v}_0$, we can say $\lambda_{\min} = (A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}}) \bullet \mathbf{v}_0\mathbf{v}_0^\top < 0$ is a measure of the infeasibility of the constraint $(A_0 - \mathcal{A}^\top \mathbf{y}) \bullet \mathbf{v}_0\mathbf{v}_0^\top \geq 0$ in $\mathbf{y}_{\text{out}}$. Similarly, the infeasibility of the constraint returned by projection is $(A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}}) \bullet \mathbf{v}_1\mathbf{v}_1^\top < 0$, which is equal to $c_a - \mathbf{a}^\top \mathbf{y}_{\text{out}}$ according to Lines 13–14. If this is 100 times weaker than the infeasibility measure $\lambda_{\min}$ obtained by separation, Lines 16-18 will add the cut discovered by separation. The running time for finding the minimum eigen-pair of $X + D$ is not completely wasted: it may be used for many other purposes, such as the practical projection using tolerance windows from Section 4 or (optionally) to stop earlier if $\lambda_{\min}$ is too close to zero even when t^* is much smaller than 1.[6]

3. As in many `Cutting-Planes` algorithms, certain linear constraints added in the beginning of the process are no longer useful in later stages, *i.e.*, they are never tight after a certain number of iterations. To avoid slowing down the LP solver with such inactive constraints, we implement the following check every 100 iterations: all constraints that did not enter the row basis for 50 iterations, are discarded (Line 22).

4. We will present two algorithms to compute the projection, one in Section 3 and the other in Section 4. The first one is numerically-exact, while the second one may return an underestimated step-length $t_-^* \leq t^*$ and an inexact hit vector $\widehat{\mathbf{v}}_1$, *i.e.*, so that the associated hyperplane is usually a bit distant from the true pierce point. Moreover, both algorithms actually support certain implementation design customizations, and so, choosing the best source-code to calculate the projection given various running conditions may be rather challenging in practice. This (prosaic) decision may depend on questions such as: is $\mathbf{y}_{\text{in}}$ close to zero? Is the artificial box over $\mathscr{P}^{\text{SDP}}$ from Remark. 2.C (Section 2.4.1) still in place? Does $\mathbf{y}_{\text{in}}$ include an artificial term that made it feasible at the cost of penalizing the objective (using Stage 3 of Section 2.2) ? An interesting option is to launch the slower algorithm in the background[7] and wait for it (or not!) depending on the running time and the results of the fastest numerically-inexact algorithm. The overall pseudo-code operates in the same manner whatever algorithm is called at Line 9, even if it returns an underestimated step-length t_-^*.

2.4.1 Engineering customizations and design options

As in many projects that imply software development, the overall pseudo-code provides a rather generic framework that leaves room for different (lower level) engineering adaptations. We here describe only a few (but not all) design options that may be important to make `Proj-Cut-Pl` reach its full potential.

Lines 6-7 involve computing two weighted-sums of the form $\mathcal{A}^\top \mathbf{y} = \sum_{i=1}^k A_i y_i$. This only requires multiplying some matrices with a scalar and summing them up, which may seem too simple to attract attention. But in the first implementation, we were surprised to notice that the calculation of these two sums was the primary computational bottleneck for $n \geq 1000$ and $k \geq 500$; it was slower than all the proposed projection algorithms. Calculating such sums in the standard schoolbook manner can be way too slow in `Matlab` (and also in Julia according to preliminary experiments). It is better to record (the lower triangular part of) each matrix A_i as a column vector of size $\frac{n(n+1)}{2}$ and to collect the k vectors into a matrix $\widetilde{A} \in \mathbb{R}^{\frac{n(n+1)}{2} \times k}$. This $\widetilde{A}$ is calculated only once, before starting the iterations. Any (lower triangular part of a) sum of matrices $\sum_{i=1}^k A_i y_i$ is then computed as matrix product $\widetilde{A}\mathbf{y}$, capitalizing on the elaborately-tuned matrix multiplication routines of `Matlab`. Many other calculations can be accelerated using this flat (vectorized) version of the A_i matrices.

[6] If $\mathbf{y}_{\text{in}} = \mathbf{0}$, projecting $\mathbf{0} \to D$ reduces to returning $t^* = \infty$ if $D \succeq \mathbf{0}$ or 0 otherwise. If $\mathbf{y}_{\text{in}}$ is just very close to $\mathbf{0}$ and $X + D$ sits close to the boundary of feasibility, the separation is more numerically-reliable than the projection.

[7] In `Matlab`, one can use the Parallel Computing Toolbox and call `parfeval` to run in the background a function that can be canceled at any time.

Algorithm 1 `Projective Cutting-Planes` for maximizing $\mathbf{b}^\top \mathbf{y}$ over $\mathscr{P}^{\text{SDP}}$ in (1.C.1)-(1.C.4)

1: Construct a first inner solution $\mathbf{y}_{\text{in}}$ using Section 2.2 and set $\mathbf{y}_{\text{best-in}} = \mathbf{y}_{\text{in}}$.
2: Build an initial outer approximation $\mathscr{P} = \mathscr{P}_0^{\text{SDP}}$ of the feasible area $\mathscr{P}^{\text{SDP}}$ using Section 2.3.
3: (optional presolve) Use standard `Cutting-Planes` to be sure the above outer approximation is not unbounded, removing from $\mathscr{P}$ the extreme rays that do not appear in $\mathscr{P}^{\text{SDP}}$ – see Remark 2.C (p 12).
4: **repeat**
5: $\quad \mathbf{y}_{\text{out}} \leftarrow \arg\max\{\mathbf{b}^\top \mathbf{y} : \mathbf{y} \in \mathscr{P}\}$ ▷ Call an LP-solver
6: $\quad X = A_0 - \mathcal{A}^\top \mathbf{y}_{\text{in}}$
7: $\quad D = A_0 - \mathcal{A}^\top \mathbf{y}_{\text{out}} - X$
8: $\quad \lambda_{\min}, \mathbf{v}_0 \leftarrow$ minimum eigenvalue and eigenvector of $X + D$
9: $\quad t^*, \mathbf{v}_1 \leftarrow$ projection sub-problem $X \to D$ ▷ We may also return a second-hit vector $\mathbf{v}_2$
10: $\quad \mathbf{v}_1 \leftarrow \dfrac{\mathbf{v}_1}{\texttt{2-norm}(\mathbf{v}_1)}$ ▷ Be sure the vector is normalized
11: $\quad \mathbf{y}_{\text{best-in}} \leftarrow \arg\max\{\mathbf{b}^\top \mathbf{y} : \ \mathbf{y} \in \{\mathbf{y}_{\text{best-in}}, \mathbf{y}_{\text{in}} + t^*(\mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}})\}\}$
12: $\quad \mathbf{y}_{\text{in}} \leftarrow \mathbf{y}_{\text{in}} + \alpha t^*(\mathbf{y}_{\text{out}} - \mathbf{y}_{\text{in}})$ ▷ we use $\alpha = \frac{2}{3}$
13: $\quad \mathbf{a}^\top \leftarrow [A_1 \bullet \mathbf{v}_1\mathbf{v}_1^\top, \ A_2 \bullet \mathbf{v}_1\mathbf{v}_1^\top, \ \ldots, \ A_k \bullet \mathbf{v}_1\mathbf{v}_1^\top]$
14: $\quad c_a \leftarrow A_0 \bullet \mathbf{v}_1\mathbf{v}_1^\top$
15: $\quad$ Add cut $\mathbf{a}^\top \mathbf{y} \le c_a$ to the description of $\mathscr{P}$
16: $\quad$ **if** $(0 > c_a - \mathbf{a}^\top \mathbf{y}_{\text{out}} > 0.01 \cdot \lambda_{\min})$ ▷ If very low impact in cutting $\mathbf{y}_{\text{out}}$
17: $\quad\quad$ Repeat Lines 13-15 with $\mathbf{v}_0$ instead of $\mathbf{v}_1$ ▷ Add one cut by separation
18: $\quad$ **end if**
19: $\quad$ **if** (Line 9 returned some $\mathbf{v}_2$)
20: $\quad\quad$ Repeat Lines 13-15 with $\mathbf{v}_2$ instead of $\mathbf{v}_1$ ▷ Add one cut using a second-hit vector
21: $\quad$ **end if**
22: $\quad$ drop all cuts that didn't enter the row basis for the last 50 iterations (check this every 100 iterations)
23: **until** $\frac{\mathbf{b}^\top \mathbf{y}_{\text{out}} - \mathbf{b}^\top \mathbf{y}_{\text{best-in}}}{10^{\lceil \log_{10} |\mathbf{b}^\top \mathbf{y}_{\text{out}}| \rceil}} < \epsilon_{\texttt{opt}}$ ▷ or $\lfloor \mathbf{b}^\top \mathbf{y}_{\text{out}} \rfloor = \lfloor \mathbf{b}^\top \mathbf{y}_{\text{best-in}} \rfloor$ if in a relaxation of a binary problem

This division turns a gap of 123.456 into 0.123456

The probability of generating inner or outer solutions of the form $\mathcal{A}^\top \mathbf{y} = \sum_{i=1}^k A_i y_i$ that are very sparse may decrease as k becomes very large. Yet we do implement this idea when the overall density is below 3%. We simply use the `Matlab` features for recording a matrix in a sparse way, either when recording each A_i as an n-by-n matrix or as a column vector of size $\frac{n(n+1)}{2}$ as above (vectorized version). This may even be necessary for some big instances, considering standard memory limitations.

It is very useful in practice to normalize the hit-vector $\mathbf{v}_1$ before computing $\mathbf{a}$ and c_a in Lines 13–14; that's why we divide all coefficients of $\mathbf{v}_1$ by $\texttt{2-norm}(\mathbf{v}_1)$ in Line 10. Without such normalization, the range of the constraint coefficients $a_i = A_i \bullet \mathbf{v}_1\mathbf{v}_1$ with $i \in [1..k]$ as computed in Line 13 can change too much from iteration to iteration, complicating the task of the LP solver. Even using this normalization, some of these coefficients can still explode in some (very) rare cases, *e.g.*, if one of the input matrices A_i with $i \in [1..k]$ contains too many huge values compared to the other matrices (meaning the program is ill-conditioned). A full solver needs to address many such rather prosaic aspects that we cannot even describe in full detail within a purely scientific paper of appropriate length; most of them may belong more appropriately to the software documentation.

Remark 2.C. *(optional presolve) The optimal solution of the polyhedron $\mathscr{P}_0^{SDP} \supsetneq \mathscr{P}^{SDP}$ constructed in Line 2 according to Section 2.3 may be an unbounded ray, especially if $\mathbf{y}$ is free. If this happens, we tentatively place an artificial box over all variables, limiting each y_i with $i \in [1..k]$ to the interval [-10000,10000]. By installing this limitation, the resulting outer LP will not contain unbounded rays but the new optimal outer solution $\mathbf{y}_{\text{out}}$ will touch the artificial box. Yet, this outer solution will also likely violate some eigenvalue-cuts in (1.C.4) and we use a standard separation to remove it. Each such iteration may reduce the number of components of $\mathbf{y}_{\text{out}}$ that touch the box. We thus implement a standard `Cutting-Planes` phase that is stopped as soon as we obtain an outer solution that does not touch the box or if the number of components of $\mathbf{y}_{\text{out}}$ that touch the box can no longer be decreased.*

In the latter (rare) case above, we retain the artificial box during the real `Proj-Cut-Pl` iterations. As long as the box is needed, we also keep adding a cut by separation, *i.e.*, the condition at Line 16 can be

extended in practice. If we ever generate some $\mathbf{y}_{\text{out}}$ that touches the box but cannot be separated, it is clear that we cannot declare it optimal. In such case, we multiply the box side by 10, *i.e.*, the allowed variable bounds (initially $[-10000, 10000]$) are multiplied by 10. The process is repeated whenever necessary. If the considered SDP program does have an unbounded ray, the box will be enlarged this way multiple times. After each box resize, we solve an additional projection sub-problem towards a truncated $\mathbf{y}_{\text{out}}$ that replaces all components of $\mathbf{y}_{\text{out}}$ not touching the box with zeros. If this projection returns $t^* = \infty$, we conclude the given problem is unbounded, provided this projection advances along an unbounded feasible ray. We conclude the same if the objective value of the inner point reaches a very large value (10^{10} in practice).

2.4.2 Addressing two critical computational bottlenecks

One operation used multiple times (in the projection algorithms from Sections 3 and 4) is computing minimum eigenvalues and eigenvectors. The software available during the current decade (2020-2030) seems highly-optimized for this type of matrix operations even for large values of $n > 2000$. After some preliminary tests, we decided do implement `Proj-Cut-Pl` in `Matlab`, because it is one of the best numeric computing environments for most matrix routines invoked by the projection algorithm, including the Cholesky and the QR decompositions. Perhaps the overall numerical results might have been worse in previous decades.

Another computationally-critical step is the LP solver (Line 5 in Alg. 1) that iteratively optimizes over a growing outer approximation $\mathscr{P} \supset \mathscr{P}^{\text{SDP}}$, especially if $\mathscr{P}$ becomes large enough to integrate thousands of constraints. This LP solver may be particularly slow if both k and the iteration counter `it` is larger than 1000, because it has to solve at each iteration an LP with thousands of variables and constraints. A single Simplex pivot step may require a running time directly proportional to both k and the number of constraints. This may easily become a performance-limiting factor, unless the structure of the constraint set simplifies somehow the problem, *e.g.*, if the optimal $\mathbf{y}$ has very few non-zero components.

The numerical effectiveness of `Proj-Cut-Pl` (Section 5) is tributary to the external software chosen for the two above operations. For instance, any future progress in (commercial) LP solvers may also accelerate the overall `Proj-Cut-Pl`.

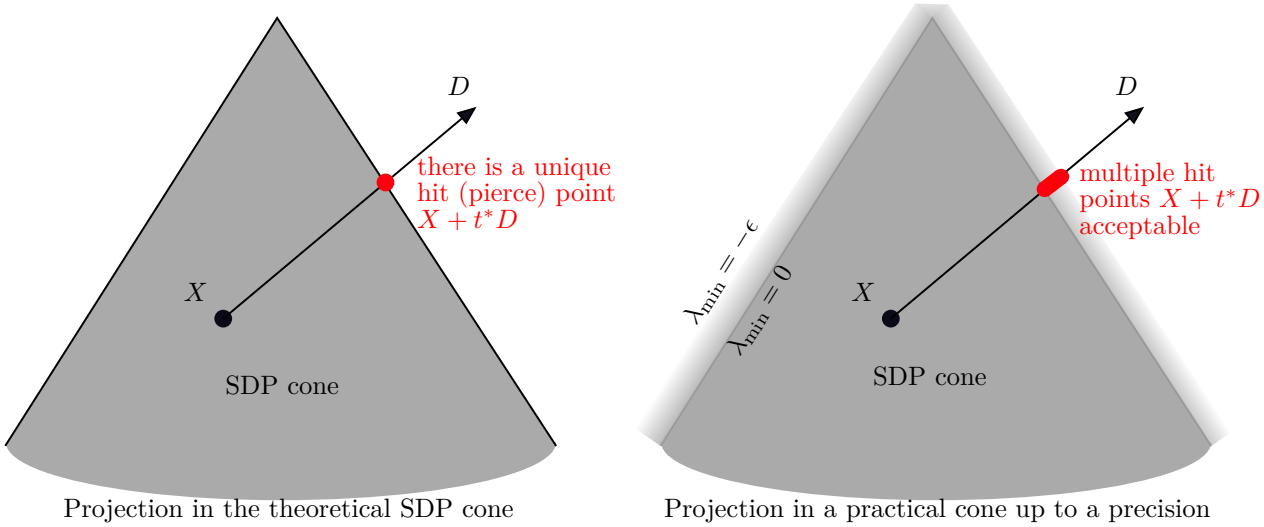
3 An SDP projection algorithm in ideal exact arithmetic

Most SDP algorithms need to take into account the following challenge that also complicates the projection-subproblem. It may be numerically very hard to decide if $\lambda_{\min}(S) > 0$ for some matrices S at the edge of the SDP cone. For example, if $\lambda_{\min}(S) = 2^{-99}$, most eigenvalue algorithms available today do not really have enough numerical accuracy to certify that S is SDP. To circumvent such imprecisions, most implementations (and most users) actually enlarge the SDP cone and consider S to be SDP if $\lambda_{\min}(S) > -\epsilon$ for some ϵ like $\epsilon = 10^{-6}$. In fact, even using $\lambda_{\min}(S) > -\epsilon$ is not enough to obtain a hard line of distinction, because this would only shift these precision problems. Yet we can be sure of this: thanks to using such condition, all matrices that satisfy $\lambda_{\min}(S) \geq 0$ will be classified as SDP. Among the matrices with $\lambda_{\min}(S)$ very close to $-\epsilon$, some will be considered SDP, others non-SDP; the probability of classifying S as SDP decreases as the real $\lambda_{\min}(S)$ decreases. In practice, there is a kind of soft gradual transition between the interior and the exterior of the SDP cone, as illustrated in Figure 3.A.

This Section 3 is devoted to a theoretical algorithm that computes the projection under ideal exact arithmetic conditions, providing only brief remarks on numerical stability. We will move to a numerical inexact but practical (often faster) algorithm using tolerance windows in Section 4. For now, we distinguish four cases in exact arithmetic addressed below by Sections 3.1, 3.2, 3.3 and 3.4, respectively.

3.1 Case (A): non-singular positive-definite $X \succ \mathbf{0}$

If X is non-singular, the problem is surprisingly easy. We apply the Cholesky decomposition to determine the unique non-singular K such that $X = KK^\top$. We then solve $D = KD'K^\top$ in variables D' by back substitution; this may require $O(n^3)$ in theory, but `Matlab` can compute it far more rapidly in practice, being recognized for its highly-optimized implementation of matrix calculations. The first two equations below are simply the same by the above substitution. The last two inequalities are equivalent because the involved $n \times n$ matrices are congruent, see, *e.g.*, [23, Prop 1.2.3.]; it is however easy to check that if K is



Projection in the theoretical SDP cone — Projection in a practical cone up to a precision

Figure 3.A: Projection in exact arithmetic with a hard line of separation (left) and in a practical world with a gradual line of separation (right). The level of gray is the probability of classifying a matrix S as belonging to the cone. The progressively whitening strips at right show how this probability decreases as the minimum eigenvalue goes below zero, up to vanishing when $\lambda_{\min}(S) \leq -\epsilon$. This complicates the projection sub-problem in practice since multiple pierce (hit) points may be more-or-less acceptable in practice.

non-singular, then $I_n + tD'$ and $K\left(I_n + tD'\right)K^T$ have the same SDP status.

$$\max\left\{t : X + tD \succeq \mathbf{0}\right\} \tag{3.A.1}$$

$$\max\left\{t : KI_nK^\top + tKD'K^\top \succeq \mathbf{0}\right\} \tag{3.A.2}$$

$$\max\left\{t : I_n + tD' \succeq \mathbf{0}\right\}. \tag{3.A.3}$$

The projection sub-problem has to return: (1) the pierce point and (2) the first-hit cut.

1. The pierce point is $X + t^*D$, where we determine $t^* = -\frac{1}{\lambda_{\min}(D')}$, or $t^* = \infty$ if $\lambda_{\min}(D') \geq 0$, according to (3.A.3).

2. According to Lines 13-15 of Alg. 1, the first-hit cut has the form $\left(A_1 \bullet \mathbf{v}\mathbf{v}^\top\right) y_1 + \left(A_2 \bullet \mathbf{v}\mathbf{v}^\top\right) y_2 + \cdots + \left(A_k \bullet \mathbf{v}\mathbf{v}^\top\right) y_k \leq A_0 \bullet \mathbf{v}\mathbf{v}^\top$. It has to be associated with a first-hit vector $\mathbf{v} \in \mathbb{R}^n$ that is an eigenvector of $K(I_n + t^*D')K^\top$ with an eigenvalue of 0. This means it has to satisfy $K(I_n + t^*D')K^\top\mathbf{v} = \mathbf{0}$; by left-multiplying with K^{-1}, this is equivalent to $(I_n + t^*D')K^\top\mathbf{v} = \mathbf{0}$. Thus, $\mathbf{u} = K^\top\mathbf{v}$ is an eigenvector of $I_n + t^*D'$ with an eigenvalue of 0. This $\mathbf{u}$ can be computed when determining $\lambda_{\min}(D') < 0$ above, because if $(\lambda_{\min}(D'), \mathbf{u})$ is an eigen-pair of D', we obtain $\left(I_n + t^*D'\right)^\top \mathbf{u} = \mathbf{u} + t^*\lambda_{\min}(D')\mathbf{u}$, which is equal to zero since recall $t^* = -\frac{1}{\lambda_{\min}(D')}$. The sought $\mathbf{v}$ solves $K^\top\mathbf{v} = \mathbf{u}$ and it can rapidly be computed by back-substitution. Notice one cannot advance by some ϵ beyond t^*: since $\mathbf{u}^\top D'\mathbf{u} = \lambda_{\min}(D') < 0$, we have $\mathbf{v}^\top KD'K^\top\mathbf{v} < 0 \implies \mathbf{v}^\top D\mathbf{v} < 0$, which leads to $\mathbf{v}^\top(X + (t^* + \epsilon)D)\mathbf{v} < 0$ for any $\epsilon > 0$.

As mentioned at point 1 of Section 2.4, it may be practically useful to find a second-hit pierce point $X + t_2^*D$ with $t_2^* \geq t^*$ associated with a kind of second-hit vector $\mathbf{v}_2$ such that: (i) $(X + t_2^*D) \bullet \mathbf{v}_2\mathbf{v}_2^\top = 0$ while $D \bullet \mathbf{v}_2\mathbf{v}_2^\top < 0$, and (ii) $\mathbf{v}_2^\top X\mathbf{v} = 0$. If the second smallest eigenvalue of D' satisfies $\lambda^2_{\min}(D') < 0$, we can define $t_2^* = -\frac{1}{\lambda^2_{\min}(D')}$. If the associated eigenvector is $\mathbf{u}_2$, you can check that the vector $\mathbf{v}_2$ that solves $K^\top\mathbf{v}_2 = \mathbf{u}_2$ satisfies the sought conditions (i)–(ii). First, $(X + t_2^*D) \bullet \mathbf{v}_2\mathbf{v}_2^\top = \mathbf{v}_2^\top(KK^\top + t_2^*KD'K^\top)\mathbf{v}_2 = \mathbf{u}_2^\top\mathbf{u}_2 + t_2^* \cdot \mathbf{u}_2^\top D'\mathbf{u}_2 = 1 + t_2^*\lambda^2_{\min}(D') = 0$, which is true for the considered t_2^*. Secondly, we can calculate $\mathbf{v}_2^\top X\mathbf{v} = \mathbf{v}_2^\top KK^\top\mathbf{v} = \mathbf{u}_2^\top\mathbf{u} = 0$, since the eigenvectors of D' are orthonormal.

A final note: if K contains some very small values (on the diagonal), the returned $\mathbf{v}$ (and $\mathbf{v}_2$) may contain some huge values in practice. This is why we do need the normalization in Line 10 (Alg. 1).

3.2 Case (b): D belongs to the image of X

We move to a generalized version of the above case, first formally fixing a few elementary properties.

Property 3.A. *(congruent expansion) We say that $X' \in \mathbb{R}^{n'\times n'}$ with $n' > n$ is a congruent expansion of $X \in \mathbb{R}^{n\times n}$ if and only if we can write $X' = MXM^\top$, for some $M \in \mathbb{R}^{n'\times n}$ of full rank n. X has the same SDP status as X'.*

Proof. We show both implications below.

1. $X \succeq \mathbf{0} \implies X' \succeq \mathbf{0}$. Assume the contrary for the sake of contradiction: $\exists \mathbf{v}' \in \mathbb{R}^{n'}$ such that $\mathbf{v}'^\top X' \mathbf{v}' < 0$. This implies $\mathbf{v}'^\top MXM^\top \mathbf{v}' < 0$, which is equivalent to $X \not\succeq \mathbf{0}$, contradiction.
2. $X' \succeq \mathbf{0} \implies X \succeq \mathbf{0}$ Assume the contrary: $\exists \mathbf{v} \in \mathbb{R}^n$ such $\mathbf{v}^\top X \mathbf{v} < 0$. We can surely write $\mathbf{v}^\top = \mathbf{v}'^\top M$ for some $\mathbf{v}' \in \mathbb{R}^{n'}$ because M has full rank. This means $\mathbf{v}'^\top MXM^\top \mathbf{v}' < 0$, equivalent to $\mathbf{v}'^\top X' \mathbf{v}' < 0$, contradiction. □

Property 3.B. *(D in the image of X) We say D belongs to the image of X if each column (and row, by symmetry) of D can be written as a linear combination of the columns (or rows, resp.) of X. We can equivalently say that the null space of X is included in the null space of D: $X\mathbf{d} = 0 \implies D\mathbf{d} = 0\ \forall \mathbf{d} \in \mathbb{R}^n$.*

Unlike in Section 3.1, we hereafter consider X singular. This means it has rank $c < n$, and so, X has to contain c independent rows (columns), referred to as *core* rows (columns); the other dependent rows (columns) are *non-core* positions. The first task is to separate these core positions from the others.

A fast method we identified for this task first constructs the LDL decomposition $X = L\mathtt{diag}(\mathbf{p})L^\top$, where $\mathbf{p} \geq \mathbf{0}_n$ and $L \in \mathbb{R}^{n\times n}$ is lower triangular. All $i \in [1..n]$ such that $p_i > 0$ constitute together the set of c core positions. The contribution of each p_i in $L\mathtt{diag}(\mathbf{p})L^\top$ is actually $p_i L_i L_i^\top$, where L_i is column i of L ($\forall i \in [1..n]$). The c core columns of L matter in the decomposition $X = L\mathtt{diag}(\mathbf{p})L^\top$ whereas the remaining $n - c$ non core columns of L vanish. Denoting these c core columns of L by L_{nc} and the non-zero elements of $\mathbf{p}$ by $\mathbf{p}_c$, we can re-write $X = L\mathtt{diag}(\mathbf{p})L^\top$ as $X = L_{nc}\mathtt{diag}(\mathbf{p}_c)L_{nc}^\top$. This leads to $X = L_{nc}\mathtt{diag}(\mathbf{p}_c)^{\frac{1}{2}}\mathtt{diag}(\mathbf{p}_c)^{\frac{1}{2}}L_{nc}^\top = K_{nc}K_{nc}^\top$ with $K_{nc} \in \mathbb{R}^{n\times c}$. The image (columns space) of K_{nc} is equal to the image of X, because both have the same null space in the last equality above.

Remark 3.C. *(Eliminating infinitesimal values for numerical stability) If all $n \times n$ elements of $p_i L_i L_i^\top$ are in absolute value below some precision threshold for some $i \in [1..n]$, we consider i to be a non-core position. Before computing K_{nc} as above, we reduce all such non-core positions p_i to zero because a smaller core leads to faster calculations. In fact, some elements of $\mathbf{p}$ may even be slightly negative (as returned by `Matlab`) in practice if X is borderline SDP, meaning that $\lambda_{\min}(X)$ is infinitesimally negative. It may be useful to remove such (often spurious) close-to-zero elements of $\mathbf{p}$ before projecting, although we need to take care not to introduce numerical instability later.*

We next solve $D = K_{nc}D'K_{nc}^\top$ in variables $D' \in \mathbb{R}^{c\times c}$. For this, we first reduce this system to work on $c \times c$ matrices, *i.e.*, we transform it into $D_{cc} = K_{cc}D'K_{cc}$ where K_{cc} is K_{nc} restricted to the c core rows and D_{cc} is D restricted to the $c \times c$ core rows and columns. To solve this square system, we apply back-substitution twice and this is very fast since K_{cc} is lower triangular (because so is L). If the resulting solution D' also satisfies $D = K_{nc}D'K_{nc}^\top$, this means D is in the image of X, which is equivalent to the image of K_{nc} (see above). This is how the algorithm detects if we are in this case or if it has to move to Section 3.3. The current case leads to a reduced-size version of (3.A.3) working in the space of $c \times c$ matrices:

$$\max\{t : I_c + tD' \succeq \mathbf{0}\}. \tag{3.B}$$

The maximum step-length is thus $t^* = -\frac{1}{\lambda_{\min}(D')}$, or $t^* = \infty$ if $\lambda_{\min}(D') \geq 0$.

Finally, we determine a first-hit vector $\mathbf{v}_c \in \mathbb{R}^c$ over the core rows and columns exactly as in the case of non-singular matrices from Section 3.1. To lift $\mathbf{v}_c$ to an n-dimensional hit vector $\mathbf{v} \in \mathbb{R}^n$, we construct $\mathbf{v}$ by inheriting the core positions from $\mathbf{v}_c$ and filling the non-core positions with zeros. The same lifting can be applied to a second-hit vector that can be determined as shown towards the end of Section 3.1.

3.3 Case (c): D has some components outside the image of X but they don't interact with the image of X in the projection geometry

We still use the decomposition $X = K_{nc}K_{nc}^\top$ computed above, but we suppose that the system $D = K_{nc}D'K_{nc}^\top$ has no solution in variables D'. This also means D does not belong to the image of K_{nc} or X.

We will express all columns of D as a linear combination of: (i) the columns of K_{nc} and (ii) a set of m columns of D referred to as active (stand-alone) columns. The fastest method (we) identified for this goal starts by applying the QR decomposition on matrix $[K_{nc}\ D] \in \mathbb{R}^{n\times(c+n)}$, writing $[K_{nc}\ D] = QR$, where $Q \in \mathbb{R}^{n\times(c+n)}$ is ortho-normal and $R \in \mathbb{R}^{(c+n)\times(c+n)}$ is upper triangular.[8] The QR decomposition algorithm scans the columns of $[K_{nc}\ D]$ one by one: at each step $i \in [1..c+n]$, the first i columns of Q are constructed to represent a basis for the space spanned by the first i columns of $[K_{nc}\ D]$. This will enable us to detect each column i of $[K_{nc}\ D]$ that does not expand the space spanned by the previously-scanned columns $1, 2, \dots i-1$ of $[K_{nc}\ D]$. This will equivalently identify each column $i-c$ of D that is merely a linear combination of K_{nc} and of the columns $1, 2 \dots, i-c-1$ of D; such a column will be termed inactive.

Let us focus on the first c columns R_{nc} of R so that $K_{nc} = QR_{nc}$. The only non-zeros of R_{nc} can be found in its $c \times c$ top block R_{cc} (in the c top rows of R_{nc}) because R is upper triangular. Denoting the first c columns of Q by Q_{nc}, we can write $K_{nc} = Q_{nc}R_{cc}$. Since K_{nc} is of full rank c, so must be R_{cc}, R_{nc} and even Q_{nc} as it can be written $Q_{nc} = K_{nc}R_{cc}^{-1}$. The matrix R_{cc} needs to have a non-zero diagonal to be non-singular, *i.e.*, $R_{jj} \neq 0$ for all $j \in [1..c]$.

Now focus on row $c+i$ of R for each $i \in [1..n]$. If all elements of this row are zero, column $c+i$ of Q has no contribution in the product QR. This also implies that column $c+i$ of $[K_{nc}\ D]$ can be expressed as a combination of the first $c+i-1$ columns of Q, because only column $c+i$ of R has an impact on column $c+i$ of $[K_{nc}\ D]$ and only the top $c+i-1$ elements of column $c+i$ of R are non-zero. This is how the QR decomposition enables us to detect that column i of D does not expand the space spanned by K_{nc} and by the first $i-1$ columns of D. The QR decomposition algorithm might have very well ignored constructing inactive column $c+i$ of Q in such case, because it has no impact in the product QR. In short, if row $c+i$ of R is zero, we say column $c+i$ of Q is inactive; otherwise, we say column $c+i$ is active.

Let N denote the matrix Q restricted to its $m > 0$ active columns detected above. An empty N with $m = 0$ would reduce to the case of Section 3.2 – since D would be in the image of the first c columns of Q, which is also the image of full-rank K_{nc} (or non-full-rank X). For $m > 0$, it is easy to see the following decomposition of X is valid, simply recalling $X = K_{nc}K_{nc}^\top$.

$$X = \underbrace{\begin{bmatrix}K_{nc} & N\end{bmatrix}}_{c\ +m} \begin{bmatrix}I_c & 0\\ 0 & 0\end{bmatrix} \begin{bmatrix}K_{nc}^\top\\ N^\top\end{bmatrix} \tag{3.C}$$

Next, we will show we can solve the system below in the variables F, G and E. This will express D as the sum of a part extracted from the image of X (*i.e.*, $K_{nc}FK_{nc}^\top$), a part that is completely outside this image (*i.e.*, $NEN^\top$) and a mix of the two (extracted using the terms that involve G).

$$D = \underbrace{\begin{bmatrix}K_{nc} & N\end{bmatrix}}_{c\ +\ m} \underbrace{\begin{bmatrix}F & G^\top\\ G & E\end{bmatrix}}_{D_{FGE}} \begin{bmatrix}K_{nc}^\top\\ N^\top\end{bmatrix}. \tag{3.D}$$

The hardest computational task is computing D_{FGE}, meaning F, G and E. A straightforward approach may be quite slow. We prefer to exploit again the information determined by the QR decomposition. We will modify both sides of the factorization $[K_{nc}\ D] = QR$ to make it similar to (3.D). To transform Q into $[K_{nc}\ N]$, we write $Q = [Q_{nc}\ Q_{n,c+1..c+n}]$, *i.e.*, we split its first c columns Q_{nc} from the last n columns $Q_{n,c+1..c+n}$. We can write $K_{nc} = Q_{nc}R_{cc}$, where recall R_{cc} is the $c \times c$ top-left block of R. Since this system is full rank, we obtain $Q_{nc} = K_{nc}R_{cc}^{-1}$. We can thus write $Q = [K_{nc}\ Q_{n,c+1..c+n}] \begin{bmatrix}R_{cc}^{-1} & \mathbf{0}\\ \mathbf{0} & I_n\end{bmatrix}$. Replacing this Q in $[K_{nc}\ D] = QR$, we obtain:

$$[K_{nc}\ D] = \underbrace{[K_{nc}\ Q_{n,c+1..c+n}] \begin{bmatrix}R_{cc}^{-1} & \mathbf{0}\\ \mathbf{0} & I_n\end{bmatrix}}_{Q} R.$$

[8] Technically, the standard QR factorization returns a matrix $Q \in \mathbb{R}^{n\times n}$ and a matrix $R \in \mathbb{R}^{n\times(c+n)}$, but we artificially extend Q with c *null columns* and R with c *null rows* to simplify notations.

Restricting $[K_{nc}\ D]$ to its last n columns (*i.e.*, to D) and R to its last n columns (denoted by $R_{c+n,c+1..c+n}$), the above relation reduces to:

$$D = [K_{nc}\ Q_{n,c+1..c+n}] \underbrace{\begin{bmatrix} R_{cc}^{-1} & \mathbf{0} \\ \mathbf{0} & I_n \end{bmatrix} R_{c+n,c+1..c+n}}_{T}.$$

Using notation $T \in \mathbb{R}^{(c+n)\times n}$ with $T = \begin{bmatrix} R_{cc}^{-1} & \mathbf{0} \\ \mathbf{0} & I_n \end{bmatrix} R_{c+n,c+1..c+n}$ as above, we can extract the following:

$$D = [K_{nc}\ Q_{n,c+1..c+n}]T. \tag{3.E}$$

We can further restrict $Q_{n,c+1..c+n}$ to its m active columns identified above, *i.e.*, to N, because recall that an inactive column $c+i$ of Q has no contribution in the QR product since row $c+i$ of R is null. In other words, we reduce the left factor $[K_{nc}\ Q_{n,c+1..c+n}]$ of (3.E) to $[K_{nc}\ N]$ by removing the inactive columns of Q. The associated T factor in (3.E) is also reduced to some $\overline{T} \in \mathbb{R}^{(c+m)\times n}$ by removing its null rows that come from the null rows of R. We can re-write (3.E) as $D = [K_{nc}\ N] \cdot \overline{T}$, which is very similar to (3.D). We finally extract D_{FGE} from (3.D), by solving $D_{FGE} \begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix} = \overline{T}$ in variables D_{FGE}.[9]

We recall D_{FGE} has the following form:

$$D_{FGE} = \begin{bmatrix} F & G^\top \\ G & E \end{bmatrix}.$$

The case addressed in this sub-section is characterized by finding $G = \mathbf{0}$ when determining D_{FGE} above. Expanding (3.D) into $D = K_{nc}FK_{nc}^\top + NEN^\top + K_{nc}G^\top N^\top + NGK_{nc}^\top$, this means D actually has the form $D = K_{nc}FK_{nc}^\top + NEN^\top$. There is no interaction in (3.D) between the components of D inside the image of X and those outside, because the terms like $K_{nc}G^\top N^\top$ vanish when $G = \mathbf{0}$. Since N is orthogonal to K_{nc} by (the QR decomposition) construction, if we left-multiply this by $N^\top$ and right-multiply by N, we obtain:

$$E = N^\top DN. \tag{3.F}$$

Applying the congruence expansion Property 3.A on (3.C) and (3.D), the SDP status of $X + t \cdot D$ is the same as that of

$$\begin{bmatrix} I_c & 0 \\ 0 & 0 \end{bmatrix} + t \cdot \begin{bmatrix} F & \mathbf{0} \\ \mathbf{0} & E \end{bmatrix}. \tag{3.G}$$

Any hit-vector $\mathbf{v}' \in \mathbb{R}^{c+m}$ for the projection problem $\left[\begin{smallmatrix} I_c & 0 \\ 0 & 0 \end{smallmatrix}\right] \to \left[\begin{smallmatrix} F & \mathbf{0} \\ \mathbf{0} & E \end{smallmatrix}\right]$ can be lifted to a hit-vector $\mathbf{v} \in \mathbb{R}^n$ for the original projection $X \to D$ by finding a solution $\mathbf{v}$ of the underdetermined system $\mathbf{v}' = \begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix} \mathbf{v}$.

To project $\left[\begin{smallmatrix} I_c & 0 \\ 0 & 0 \end{smallmatrix}\right] \to \left[\begin{smallmatrix} F & \mathbf{0} \\ \mathbf{0} & E \end{smallmatrix}\right]$ we distinguish two cases:

$E \not\succeq \mathbf{0}$ The sought t^* is 0. It is straightforward to see in (3.G) that any $t > 0$ would generate in this case a non SDP matrix: if the bottom-right block is not SDP, the overall matrix is not SDP. Let $\mathbf{d}_m \in \mathbb{R}^m$ be an eigenvector of $E = N^\top DN$ of minimum eigenvalue, so that $\mathbf{d}_m^\top N^\top DN\mathbf{d}_m < 0$. Notice now that $\mathbf{d} = N\mathbf{d}_m$ satisfies $\mathbf{d}^\top D\mathbf{d} = D \bullet \mathbf{d}\mathbf{d}^\top < 0$. Since such $\mathbf{d}$ is in the image of N which is orthogonal to K_{nc} or X, we also obtain $X \bullet \mathbf{d}\mathbf{d}^\top = 0$. Thus, $\mathbf{d}$ is first-hit vector that separates $X + tD$ from the SDP cone for any $t > t^* = 0$, *i.e.*, it satisfies (2.A) for $t^* = 0$ (in Def. 2.A, p. 6).

$E \succeq \mathbf{0}$ The projection reduces to simply computing $t^* = \max\{t : I_c + tF \succeq \mathbf{0}\} = -\frac{1}{\lambda_{\min}(F)}$, or $t^* = \infty$ if $\lambda_{\min}(F) \geq 0$; this is a particularly fast calculation, especially *in the world of $c \times c$ matrices* when c is very small. Using $N^\top DN = E \in \mathbb{R}^{m\times m}$ from (3.F), we obtain that $E \succeq \mathbf{0}$ means D is SDP over $\{\mathbf{d} = N\mathbf{d}_m : \mathbf{d}_m \in \mathbb{R}^m\}$, *i.e.*, over the image of N. This N is outside the image of K_{nc} by construction. Recall from (3.D) that all $\mathbf{d} \in \mathbb{R}^n$ that are outside the image of both K_{nc} and N have to yield $D \bullet \mathbf{d}\mathbf{d}^\top = 0$. This means D is actually SDP over the full null space of X or K_{nc} *i.e.*, $D \bullet \mathbf{d}\mathbf{d}^\top \geq 0 \forall \mathbf{d} \in \mathbb{R}^n$ such that $X\mathbf{d} = 0$. The components of D that are outside the image of X (represented by this $E \succeq \mathbf{0}$) have no impact in the projection.

[9] This system always has a solution. It may be incompatible only if there were some $\mathbf{u} \in \mathbb{R}^n$ such that $\begin{bmatrix} K_{nc}^\top \\ N^\top \end{bmatrix} \mathbf{u} = \mathbf{0} \neq \overline{T}\mathbf{u}$. But that would imply $\mathbf{u}^\top [K_{nc}\ N] = 0$; and applying (3.E) after replacing all inactive columns of Q with zeros as they play no role in (3.E), this leads to $\mathbf{u}^\top D = \mathbf{0}$, equivalent to $D\mathbf{u} = \mathbf{0}$. Multiplying (3.E) this time on the right with $\mathbf{u}$, we obtain $D\mathbf{u} = \mathbf{0} = [K_{nc}\ Q_{n,c+1..c+n}]T\mathbf{u}$. Since Q is full rank and its first c columns cover the image of K_{nc}, matrix $[K_{nc}\ Q_{n,c+1..c+n}]$ must be full rank and invertible. This leads to $T\mathbf{u} = \mathbf{0}$, which is a contradiction since $\overline{T}$ contains a subset of the rows of T.

3.4 Case (d): the most general case

If all above cases fail, we still project towards a D_{FGE} matrix decomposed as above in sub-blocks F, G and E. We assume $E \succeq \mathbf{0}$, because otherwise we explained why t^* would be zero. We also consider that we have determined above a non-zero G, since otherwise we would have been in the previous case (c). This G generates in the expansion of (3.D) some non-zero terms $K_{nc}G^\top N^\top$ that combine certain components of D inside the image of X (and K_{nc}) with other components of D that are outside this image (embodied by N). The two spaces can no longer be separated. Thus, we have to find the maximum t for which the following generalization of (3.G) remains in the SDP cone:

$$\underbrace{\begin{bmatrix} I_c & 0 \\ 0 & 0 \end{bmatrix}}_{X_{c+m}\in\mathbb{R}^{(c+m)\times(c+m)}} + t \cdot \underbrace{\begin{bmatrix} F & G \\ G & E \end{bmatrix}}_{D_{FGE}\in\mathbb{R}^{(c+m)\times(c+m)}} \succeq \mathbf{0}. \tag{3.H}$$

Let us first focus on a tricky case, in which there is no hard line of separation associated with some fixed first-hit vector $\mathbf{v}$, but an asymptotic limiting behavior. If there is some $i \in [c+1..m]$ and some $j \in [1..c]$ such that the diagonal element (i,i) of D_{FGE} is zero while its non-diagonal element (i,j) is non-zero, then any $t > 0$ leads to $X_{c+m} + tD_{FGE} \not\succeq \mathbf{0}$. We return $t^* = 0$, but there is no hit vector $\mathbf{v}$ such that $(X_{c+m} + 0D_{FGE}) \bullet \mathbf{v}\mathbf{v}^\top = 0$ and $(X_{c+m} + tD_{FGE}) \bullet \mathbf{v}\mathbf{v}^\top < 0$ for any $t > 0$ no matter how small. Because if we reduce the whole projection to rows and columns i and j, there is no vector $\mathbf{v} \in \mathbb{R}^2$ such that $\left[\begin{smallmatrix} 1 & t \\ t & 0 \end{smallmatrix}\right] \bullet \mathbf{v}\mathbf{v}^\top < 0$ for any $t > 0$ no matter how small.

If all possibilities discussed up to this point fail, we solve the projection in two steps: (1) find a small underestimated step length $t_- < t^*$ such that $X_{c+m} + t_- D_{FGE} \succeq \mathbf{0}$ and (2) solve the projection $(X_{c+m} + t_- D_{FGE}) \to D_{FGE}$. In this second step, D_{FGE} belongs to the image of $X_{c+m} + t_- D_{FGE}$ (Prop 3.B satisfied) and we can use the techniques from Sections 3.1 and 3.2. We can find such a t_- by repeated separation, *i.e.*, we try a sequence of step lengths $t_1 > t_2 > t_3 \ldots$ until generating a first step length below t^*. However, this may be particularly expensive and quite error-prone when the sought t^* is close to zero, often requiring too many separations.

3.5 Computational aspects with respect to existing literature

Generalizing the separation, the SDP projection sub-problem is intrinsically more complex. While many ideas used in the above projection algorithms may seem to arise a bit out of the blue, they were designed following a conscious effort to reduce the runtime cost as much as possible; recall we aim to efficiently calculate projections $X \to D$ with $n \geq 1000$. Most prior work on related topics was not as strongly motivated or driven to the same extent by considerations of computational efficiency. We here describe related literature with this computational speed standpoint in mind. For instance, some mathematical details from Section 3.3 are a bit intricate, but the resulting computations may remain rather lightweight, especially when $m + c$ is very small compared to n, which may happen if both X and D have a (very) low rank and a large null space. Most projection-related approaches below do not capitalize on this kind of property to the extent we do.

3.5.1 Related ideas on separation and standard `Cutting-Planes`

Let us also briefly discuss the separation that can be used to find the minimum eigenvalue of a matrix $X = A_0 - \mathcal{A}^\top \mathbf{y}$. This is a very well-studied question: many software libraries (refined over decades) can solve it very rapidly. Despite this numerical efficiency, a canonical `Cutting-Planes` that optimizes over the outer approximation by repeated separation may need prohibitively-many iterations for large-scale SDP programs. As [5, § 7.5.2] put it, "*an exponential number of cutting planes is needed for an ϵ-approximation, see Braun et al. [19]. Depending on the problem type, this is also confirmed by slow performance in practice*".

We do mention a few `Cutting-Planes` implementations that follow this idea. We are aware of the algorithms from [16, 28] that often use $n \leq 100$ and operate on the dual (1.B.1)-(1.B.3). The PhD thesis [29] and two associated articles [30, 31] established one of the first unifying frameworks offering multiple valuable insights into the overall question, including the idea to work on the primal (1.A.1)-(1.A.3) instead of (1.B.1)-(1.B.3). The more recent study [4] develops interesting (theoretical approximation) results about different polyhedral approximations of the semidefinite cone.

3.5.2 Related projection approaches

The oldest related work traces back to the concept of *generalized eigenvalues*. Given matrices X and $M = -D$, we say λ (resp. $\mathbf{v} \neq \mathbf{0}$) is a generalized eigenvalue (resp. eigenvector) of $(X, M) = (X, -D)$ if $X\mathbf{v} = \lambda M\mathbf{v}$. This reduces to $(X+\lambda D)\mathbf{v} = \mathbf{0}$, implying that one of the eigenvalues of $X+tD$ must be zero. In this research thread dating back to the 1960s [6, Chapter 12], $X + \lambda M$ is called a *pencil*. Searching the web with this key-word can lead to many references with useful ideas.

The step-length t^* sought by the projection must therefore be a generalized eigenvalue of the pencil $(X, M) = (X, -D)$. Yet many generalized eigenvalues λ may satisfy $\lambda_{\min}(X + \lambda D) < 0$, and so, they have no impact in our projection question. Such λ are higher than t^*, *i.e.*, such that $X + \lambda D$ is beyond the pierce point, even if the corresponding eigenvector $\mathbf{v}$ satisfies $(X + \lambda D)\mathbf{v} = \mathbf{0}$. This last condition only certifies that one of the eigenvalues of $X + \lambda D$ is zero, even if $\lambda_{\min}(X + \lambda D) < 0$. We will see (Example 3.D below) that we actually seek a generalized eigenvalue t^* satisfying a rather peculiar property: t^* is the smallest real non-negative generalized eigenvalue of $(X, M) = (X, -D)$ associated with an eigenvector $\mathbf{v}$ such that $D \bullet \mathbf{v}\mathbf{v}^\top < 0$.[10]

Many programming languages (including `Matlab`) provide native implementations for computing the generalized eigenvalues. One way to solve the projection sub-problem would be to compute a generalized eigendecomposition and to select from the n generalized eigen-pairs the one that satisfies the above condition. This may make `Projective Cutting-Planes` easier to implement, but it is far too computationally expensive. Such an approach may even require considering complex generalized eigenvalues even if X and D are symmetric (see the example below). Moreover, it's absolutely natural to expect that computing a generalized eigendecomposition takes more time than a standard eigendecomposition; this latter operation was dismissed from the very beginning of this work as far too computationally demanding.

Example 3.D. *Any complex number can be a generalized eigenvalue of $(X, -D)$ for X and D below.*

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad\qquad D = \begin{bmatrix} -2 & -2 \\ -2 & -2 \end{bmatrix}$$

Projecting $X \to D$ yields $t^* = 0.5$ so that 0.5 is a generalized eigenvalue of $(X, -D)$. Yet, if we take $\mathbf{u} = [1 \ \ -1]^\top$, we obtain $\mathbf{0} = X\mathbf{u} = \lambda \cdot -D\mathbf{u}$ for absolutely any complex $\lambda \in \mathbb{C}$. We thus see a phenomenon that does not exist for standard eigenvalues: any number λ can be a generalized eigenvalue associated with a non-zero eigenvector $\mathbf{u}$. This shows how tricky the condition mentioned two paragraphs above can be: we seek the smallest real non-negative eigenvalue associated with an eigenvector $\mathbf{v}$ such that $D \bullet \mathbf{v}\mathbf{v}^\top < 0$. The above eigenvector $\mathbf{u}$ does not satisfy this condition for many values $\lambda \in \mathbb{C}$, including for all real $\lambda \in [0, t^*)$.

Since we need a rather particular generalized eigenvalue, we are a bit skeptical we can find off-the-shelf software (in the near future) that can solve the projection sub-problem rapidly enough by computing the generalized eigenvalues. Nevertheless, the techniques already in use for this purpose (ex, the Lanczos algorithm, the power method) may be used or adapted for our goal. The deepest algebra notions behind such algorithms constitute in themselves a field of mathematics that exists independently of SDP optimization. While most such aspects are beyond the scope of this paper, I do try to select below some ideas from the underlying theory that overlap the presented projection ideas.

The idea of exploiting congruence relations to show that (3.A.1), (3.A.2) and (3.A.3) are equivalent (Section 3.1) was certainly considered before in the context of generalized eigenvalues. As long as the input matrices are invertible and symmetric, it is rather straightforward to use a variant of (3.A.1)-(3.A.3) to reduce the generalized eigenvalue problem to a standard eigenvalue problem, see, *e.g.*, Section 5.2 "Transformation to Standard Problem" from [9].

Property 3.E. *Based on [21, §15.1], we say that pencils (A_1, M_1) and (A_2, M_2) are equivalent if there exists an invertible matrix E such that $A_2 = EA_1E^{-1}$ and $M_2 = EM_1E^{-1}$. If $E^{-1} = E^\top$, then the pencils are (also) equivalent by congruence. The eigenvalues of two equivalent pencils are the same and the eigenvectors are linked according to a procedure similar to that developed at point 2 from Section 3.1. In a projection context, this means that projections $A_1 \to (-M_1)$ and $A_2 \to (-M_2)$ are equivalent.*

[10] A similar but weaker condition (smallest non-zero real eigenvalue for a generalized eigenvalue problem) appeared in a question posted on *Matlab Answers* at the following URL, but the replies are not helpful for our context: mathworks.com/matlabcentral/answers/397765-smallest-non-zero-eigenvalue-for-a-generalized-eigenvalue-problem

The idea of simultaneously diagonalizing X and D may seem very tempting (in theory) because it reduces the question to a projection sub-problem with diagonal matrices, which is trivial. However, it seems this requires some computationally-expensive eigendecompositions. We extract one idea from [7, § 7.2] and (try to) express it in projection terms. Let us construct the eigendecomposition $X = P\texttt{diag}(\lambda_1,\ \lambda_2,\ \ldots \lambda_n)P^\top$, where matrix P is an orthonormal n-by-n factor and $\lambda_i \geq 0\ \forall i \in [1..n]$ since $X \succeq \mathbf{0}$. If X happens to be full-rank such that $X \succ \mathbf{0}$, let us use the notional shortcut $Q = P\texttt{diag}(\sqrt{\lambda_1},\ \sqrt{\lambda_2},\ \ldots \sqrt{\lambda_n})$, and notice that $X = QQ^\top$ leads to $Q^{-1}X(Q^{-1})^\top = I_n$. We now eigendecompose $Q^{-1}D(Q^{-1})^\top = RD_{\texttt{diag}}R^\top$, where $D_{\texttt{diag}}$ is diagonal and R is an orthonormal factor. If $S = R^{-1}Q^{-1}$, we obtain $SS^\top = I_n$ simply because $R^{-1} = R^\top$ and $Q^{-1} = Q^\top$; but we can further derive $SDS^\top = D_{\texttt{diag}}$ and $SXS^\top = R^{-1}(R^{-1})^\top = I_n$. By congruence, projecting $X \to D$ reduces to the trivial projection $I_n \to D_{\texttt{diag}}$.

An approach that uses only one eigendecomposition to achieve a similar diagonalization was presented in [15, p. 550]. Let us introduce the idea for the non-singular pencil case which is characterized by the fact that there is some $t > 0$ such that $X + tD \succ \mathbf{0}$: we apply the Cholesky decomposition $X + tD = KK^\top$ and write

$$K^{-1}\,(X + tD)\,K^{\top\,-1} = I_n. \tag{3.I}$$

Constructing the eigendecomposition of $K^{-1}DK^{\top\,-1}$ with an orthonormal $P \in \mathbb{R}^{n\times n}$ factor, we obtain that $PK^{-1}DK^{\top\,-1}P^\top = D_{\texttt{diag}}$ is diagonal. By left-multiplying and right-multiplying (3.I) with P and, resp., $P^\top$, we obtain that $PK^{-1}XK^{\top\,-1}P^\top = X_{\texttt{diag}}$ is also diagonal. Using Prop. 3.E above, the question is reduced to projecting $X_{\texttt{diag}} \to D_{\texttt{diag}}$. This approach can be generalized to the case in which X and D share a null space $U \in \mathbb{R}^{n\times m}$ with $m > 0$. In such case, we can never have $X + tD \succ \mathbf{0}$ for any t. But let $U_\perp \in \mathbb{R}^{n\times(n-m)}$ be the orthogonal complement of U, which is equivalent to the image of X and D. We can have $U_\perp^\top(X+tD)U_\perp \succ \mathbf{0}$ for some $t > 0$. Similar to the non-singular pencil case above, we can compute a non-singular $R \in \mathbb{R}^{(n-m)\times(n-m)}$ such that both $R^\top U_\perp^\top XU_\perp R = X_{\texttt{diag}}$ and $R^\top U_\perp^\top DU_\perp R = D_{\texttt{diag}}$ are diagonal. Exploiting $XU = DU = \mathbf{0}$, we can extend these expressions to obtain $\left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right]^\top [U_\perp U]^\top X [U_\perp U] \left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right] = \left[\begin{smallmatrix} X_{\texttt{diag}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{smallmatrix}\right]$, and, similarly, $\left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right]^\top [U_\perp U]^\top D [U_\perp U] \left[\begin{smallmatrix} R & \mathbf{0} \\ \mathbf{0} & I_m \end{smallmatrix}\right] = \left[\begin{smallmatrix} D_{\texttt{diag}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{smallmatrix}\right]$. Using Prop. 3.E again, the sought projection reduces to projecting $X_{\texttt{diag}} \to D_{\texttt{diag}}$.

Related decompositions can be found in multiple papers that address the Generalized Trust Region (GTR) subproblem using pencils. The dual of this GTR subproblem can be written as an SDP program with a constraint of the form $A - \lambda B \succeq \mathbf{0}$ [22, (2.2)], see also [15, (7)]. However, most of these studies are devoted to finding all eigenvalues and they do not have very similar computational speed concerns.

4 A numerically-inexact but often faster projection

Without addressing certain questions of numerical instability, it may sometimes be problematic in practice to apply the exact algorithms from Section 3. As described there, the most difficult projections arise when $\lambda_{\min}(X) = 0$. But practice introduces further complications: for most matrices X generated by numerical SDP software, the equality $\lambda_{\min}(X) = 0$ is most frequently only satisfied within a certain numerical precision.

We first describe a practical algorithm using tolerance windows in Section 4.1 and then compare it with related approaches in Section 4.2.

4.1 The projection approach up to a precision

Also referring to Figure 3.A, we introduce two prosaic but practical tolerance parameters:

$\epsilon_{\texttt{SDP}}$ We extend the notion of SDP cone to include all X that satisfy $\lambda_{\min}(X) \geq -\epsilon_{\texttt{SDP}}$. Thus, if X slightly outside the true SDP cone, we consider it SDP (our implementation usually uses $\epsilon_{\texttt{SDP}} = 10^{-6}$), as in many algorithms for SDP optimization.

$\epsilon_{\texttt{proj}}$ If $\lambda_{\min}(X) \geq \epsilon_{\texttt{proj}}$, the projection software module considers $X \succ \mathbf{0}$ and it applies the fastest approach from Section 3.1. We typically set $\epsilon_{\texttt{proj}} = 0.1 \cdot \epsilon_{\texttt{SDP}} = 10^{-5}$. Otherwise, this theoretical algorithm from Section 3.1 may be too error-prone: when $\lambda_{\min}(X) < \epsilon_{\texttt{proj}}$, the Cholesky factors K that yield $X = KK^\top$ have some very small diagonal values, leading to some prohibitively-large values in K^{-1}. Since the first-hit vector $\mathbf{v}$ satisfies $\mathbf{v} = K^{\top\,-1}\mathbf{u}$, this may induce too much numerical instability.

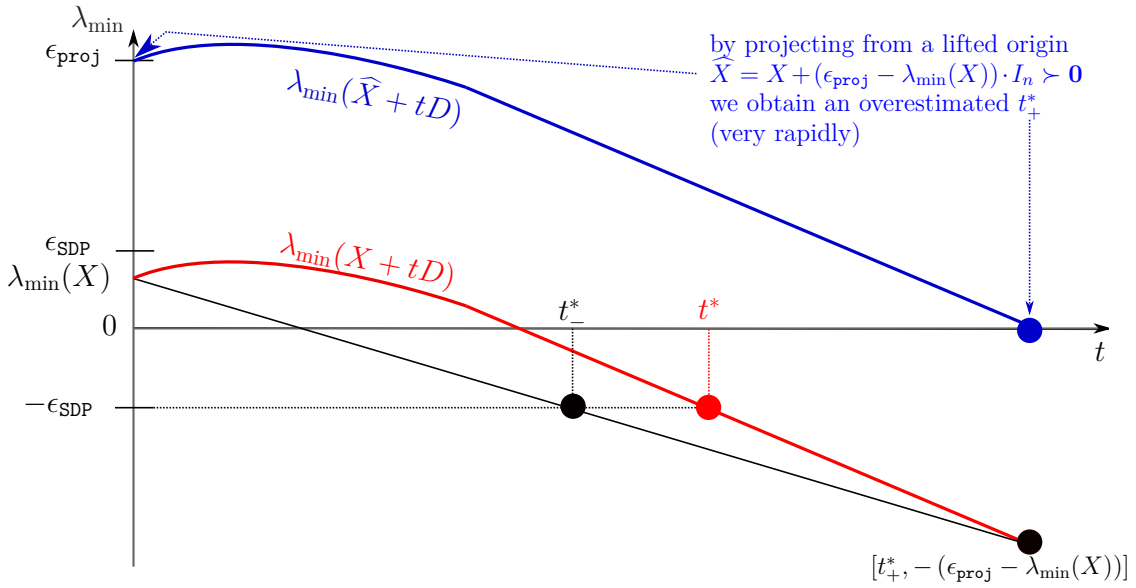


Figure 4.A: The sought t^* is marked in red and corresponds to the red disk where $\lambda_{\min}(X + t^*D) = -\epsilon_{\texttt{SDP}}$, given that we solve the projection with the $\epsilon_{\texttt{SDP}}$ tolerance window from (4.A). Since $\widehat{X} = X + \big(\epsilon_{\texttt{proj}} - \lambda_{\min}(X)\big) \cdot I_n$, the blue curve of the function $\lambda_{\min}(\widehat{X} + tD)$ is obtained by adding $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$ to the red curve representing function $\lambda_{\min}(X + tD)$.

This section is thus devoted to solving the projection sub-problem with an $\epsilon_{\texttt{SDP}}$ tolerance window:

$$t^* = \max \{t : \lambda_{\min}(X + tD) > -\epsilon_{\texttt{SDP}}\}. \tag{4.A}$$

Knowing that no algorithm for computing eigenvalues can exactly determine $\lambda_{\min}(X)$ in reasonable time, even the condition $\lambda_{\min}(X) > -\epsilon_{\texttt{SDP}}$ is not tested in exact arithmetic. This leads to a gradual line of distinction between SDP and non-SDP matrices, see the phenomenon in the right section of Figure 3.A (p. 14). The (unavoidable) infinitesimal inaccuracy of the LP solver certainly introduces some spurious data that complicates the projection sub-problem.

Figure 4.A illustrates the process for solving (4.A). Seeking to reduce the projection sub-problem to a form that can be solved with the fast algorithm from Section 3.1, we first translate the graph of the function $\lambda_{\min}(X + tD)$ upward by $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$. This leads to defining an SDP-lifted $\widehat{X} = X + (\epsilon_{\texttt{proj}} - \lambda_{\min}(X)) \cdot I_n$, such that $\lambda_{\min}(\widehat{X}) = \epsilon_{\texttt{proj}}$; the whole blue curve that starts at point $[0, \epsilon_{\texttt{proj}}]$ in the figure is simply obtained by shifting the red one upward by $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$. We then apply the fast algorithm from Section 3.1 to project from $\widehat{X}$ towards D, resulting in an overestimated $t^*_+ \geq t^*$. By subtracting $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$ to get back on the red curve, the point $[t^*_+, 0]$ is translated to $[t^*_+, -(\epsilon_{\texttt{proj}} - \lambda_{\min}(X))]$, see the black disk in the bottom-right corner. This implies that $\lambda_{\min}(X + t^*_+ D) = -(\epsilon_{\texttt{proj}} - \lambda_{\min}(X))$.

We now appeal to a very practical property a bit overlooked by related approaches that determine t^* by bisection or quasi-Newton methods: the $\lambda_{\min}$ function is *non-smooth concave*. This implies that the red curve is always above the black line that links points $[0, \lambda_{\min}(X)]$ and $[t^*_+, -\big(\epsilon_{\texttt{proj}} - \lambda_{\min}(X)\big)]$. Computing the intersection of this black line (a secant of the red graph) with the horizontal line of ordinate $y = -\epsilon_{\texttt{SDP}}$, we obtain a step length t^*_- that underestimates t^* (see the black disk), hence a sandwich relation:

$$t^*_- \leq t^* \leq t^*_+. \tag{4.B}$$

A second iteration can repeat the process after replacing $X \leftarrow X + t^*_- D$, since the sought t^* must be to the right of t^*_-. You notice in Figure 4.A a useful property: the red graph at the right of t^*_- is actually a straight line. This means that the proposed approach finds at this second iteration a second t^*_{-2} that is equal to the sought t^*. If Figure 4.A is a positive example, we will see (in figure) below a negative case-study for the proposed approach: if the red graph has high curvature at the right of t^*_-, the second iteration may be far from finding t^*. In such case, the same process may have to be repeated multiple iterations until hoping that (4.B) becomes very close to $t^*_- = t^* = t^*_+$.

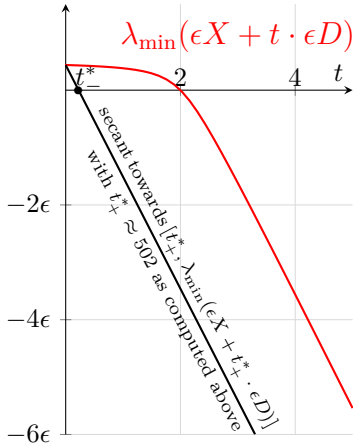
In practice, we stick to only one iteration, and the projection algorithm returns t^*_- to `Projective Cutting-Planes`. A disadvantage of this approach is that it is more difficult to find the appropriate hit-vector. Recall we actually applied the algorithm from Section 3.1 on $\widehat{X}$. This algorithm can provide a hit vector $\mathbf{v}_+$ such that $(\widehat{X} + t^*_+ D) \bullet \mathbf{v}_+\mathbf{v}_+^\top = 0$. This means that unless (4.B) is an equality, we will have $(\widehat{X} + t^*_- D) \bullet \mathbf{v}_+\mathbf{v}_+^\top > 0$.

In short, this approach returns to `Projective Cutting-Planes` an underestimated step length t^*_- and a first-hit constraint corresponding to an overestimated t^*_+ and a lifted X, generating imprecision. Such a numerically-inexact but faster solution may sometimes be preferable in practice to an exact but slow one.

However, in certain (few) cases the computed t^*_- may be too distant from t^*_+, rendering this approach quite unreliable. This may happen when X is very close to $\mathbf{0}$ or when X has a very large null space close to $\mathbb{R}^n$, because the above artificial up-shifting by $\epsilon_{\texttt{proj}} - \lambda_{\min}(X)$ may distort too much the overall projection geometry. We give an example to illustrate this unwanted distortion. If $X = \epsilon \begin{bmatrix} 3 & 2 \\ 2 & 2 \end{bmatrix}$ and $D = -\epsilon \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, the optimal step length is $t^* = 2$ and the real hit-vector is $\mathbf{v}^* = [0\ 1]^\top$ for any $\epsilon > 0$. If $\epsilon = \frac{\epsilon_{\texttt{proj}}}{1000}$, the above up-shifting adds to X a term $(\epsilon_{\texttt{proj}} - \lambda_{\min}(X)) \cdot I_n$ that will have more weight in the resulting projection than X itself. It thus leads to solving a very distorted projection from $\widehat{X} \approx \epsilon \begin{bmatrix} 1003 & 2 \\ 2 & 1002 \end{bmatrix}$ in the direction of the same D. This returns $t^*_+ \approx 502$, $t^*_- \approx 0.22$ and a hit-vector $\mathbf{v}_+$ that is virtually a multiple of $[0.999\ 1]^\top$. Notice t^*_- is far too distant from t^*_+.

If ϵ gets closer to $\epsilon_{\texttt{proj}}$, the distortion is reduced, but not enough:

- If $\epsilon = \frac{\epsilon_{\texttt{proj}}}{100}$, $\mathbf{v}_+$ becomes virtually a multiple of $[0.99\ 1]^\top$.
- If $\epsilon = \frac{\epsilon_{\texttt{proj}}}{10}$, $\mathbf{v}_+$ becomes virtually a multiple of $[0.91\ 1]^\top$.
- If $\epsilon = 1 \cdot \epsilon_{\texttt{proj}}$, $\mathbf{v}_+$ becomes virtually a multiple of $[0.36\ 1]^\top$, still far from the true hit-vector $\mathbf{v}^* = [0\ 1]^\top$.



Performing an additional iteration from $\epsilon X + t^*_- \cdot \epsilon D$ does not help too much; unlike in Figure 4.A the function $\lambda_{\min}(\epsilon X + t \cdot \epsilon D) = \epsilon \left(2.5 - t - \sqrt{(t-2)^2 + \frac{1}{4}}\right)$ is far from linear around $t^* = 2$. It actually has an important turning point at $t^* = 2$, decreasing almost linearly after $t^* = 2$. Thus, since t^*_- is always on a secant linking a top-left point of the red curve to a far lower right one (the black line is a first iteration), t^*_- will remain too distant from t^* even after multiple iterations.

4.2 Computational aspects with respect to existing literature

The above algorithm shares certain similarities with the standard bisection method, which can be employed to find (by repeated separation) a specific root $t^* \geq 0$ of the function $f(t) := \lambda_{\min}(X + tD)$. On the other hand, the idea of up-shifting this function by adding a multiple of I_n to X in order to simplify the projection is, to our knowledge, novel. However, generally-related dichotomic methods do exist, even if usually placing less emphasis on numerical speed.

Restoring definiteness [14, 33] asks to "shrink" a non-SDP matrix $X + D$ by moving from $X + D$ towards a target matrix $X \succeq \mathbf{0}$ up to the intersection with the SDP cone. If $X + D$ is a matrix that should have been SDP in theory but is indefinite in practice because of some noise, finding $t^* = \max\{t : X + t \cdot D \succeq \mathbf{0}\}$ may be seen as a kind of repairing $X + D$. When seen through such lenses, the projection sub-problem may be useful beyond SDP programming. Citing [14, § 1], the "restoration of definiteness is needed in a very wide variety of applications, of which some recent examples include modeling public health [8] and dietary intakes [36], determination of insurance premiums for crops [12], simulation of wireless links in vehicular networks [37], reservoir modeling [26], oceanography [32], and horse breeding [35]."

The above study discusses three approaches. The first one is a bisection method applied to the function $f(t) = \lambda_{\min}(X + tD)$. It starts with $t_l = 0$ and $t_r = 1$ so that $t_l \leq t^* \leq t_r$ and updates t_l and t_r using rules of the form $t_l = 0.5 \cdot (t_l + t_r)$ or $t_r = 0.5 \cdot (t_l + t_r)$, keeping each t_l at the left of t^* and each t_r at right of t^*. This reduces to a form of repeated separation or dichotomic search that iteratively evaluates f in points like t_l or t_r that get closer and closer. For faster convergence, a second approach consists of using

a standard Newton's method to find the root of f; computing the first derivative is easy, only requiring a few vector-matrix products. This leads to generating a sequence of points that all stay to the right of t^*, *i.e.*, $1 = t_0 > t_1 > t_2 \dots t_\iota = t^*$. The third approach from the above study relies on the notion of *generalized eigenvalues* that we already explored in Section 3.5.2.

The above approaches do not (need to) exploit the concavity of the minimum eigenvalue function, but we do it in our work: using the black secant from Figure 4.A (p. 21), the concavity enables us to determine an underestimate $t^*_- \leq t^*$ without evaluating $\lambda_{\min}(X + t^*_- D)$, leading to an interval $[t^*_-, t^*_+]$ certified to contain t^* from the very first iteration. We prefer to stick to one iteration because we aim to reduce the computational burden (almost) at all costs.

5 Numerical results

As indicated by the very title of this paper, `Proj-Cut-Pl` achieves the most favourable results against a very solid competition if the instance is rather dense and easily feasible; we often target very large values of n (*e.g.*, above 1000 and up to 30000) and limited values of k (even if we will also push k up to 10000 in Table 6). Only an independent external review covering many classes of instances can be very conclusive in a broad sense. First, let us discuss from the outset the cases in which `Proj-Cut-Pl` is not competitive.

- One should not bother (yet) with `Proj-Cut-Pl` if it is particularly challenging to find a single feasible solution, since it was not mainly designed to detect infeasibilities. If the first two stages for recovering feasibility from Section 2.2 fail, we use the third stage from Section 2.2 that simply adds an artificial term that can turn any non-SDP matrix into an SDP one at the cost of penalizing the objective. If `Proj-Cut-Pl` needs this third stage, the results will likely be not impressive.

- When k is too large (*e.g.*, in the thousands) and the optimal $\mathbf{y}$ has very few zero components, the LP solver may generate a significant slowdown when solving the outer approximations, as described in Section 2.4.2. On the other hand, if most components of $\mathbf{y}$ are zero at optimality, `Projective Cutting-Planes` may remain competitive even if k is large.[11]

- Many SDP solvers from the literature invested massive efforts to exploit sparsity, as sparsity can offer significant potential for speed-up within their frameworks. This is not so much the case for `Proj-Cut-Pl` because many inner points of the form $A_0 - \sum_{i=1}^k A_i y_i$ may not be (so) sparse for a large k even if the input $A_0,\ A_1, \dots A_k$ is sparse. While we did implement many sparsity features, this idea has not (yet) been pursued to its fullest extent. Most existing instances are actually extremely sparse in the (few) benchmarks available on-line, like `plato.asu.edu/ftp/sparse_sdp.html`; the very name of this URL suggests that it is devoted to sparse instances.[12]

Most instances available on-line do satisfy one or two of the above criteria. Yet, (my) practical experience suggests that comparing SDP optimization algorithms may be tricky, even more so than comparing LP algorithms. The performance of certain SDP software can seriously change by activating or deactivating some option or by slightly changing the nature of the instance. For instance, switching from $\mathbf{y} \geq \mathbf{0}$ to $\mathbf{y} \in \mathbb{R}^k$ can have a negative impact on `Projective Cutting-Planes`, because it is more difficult to find a good initial outer approximation (most of Section 2.3 only works if $\mathbf{y} \geq \mathbf{0}$). The same change seems to have a (very) positive on the `ConicBundle`, which is an algorithm based on a different philosophy.

Thus, the numerical results are not meant to show that `Proj-Cut-Pl` is generally superior to (all) other alternatives on the benchmark introduced in the ensuing discussion. While we do try to be very competitive in speed, this work is not (only) devoted to purely numerical competition.

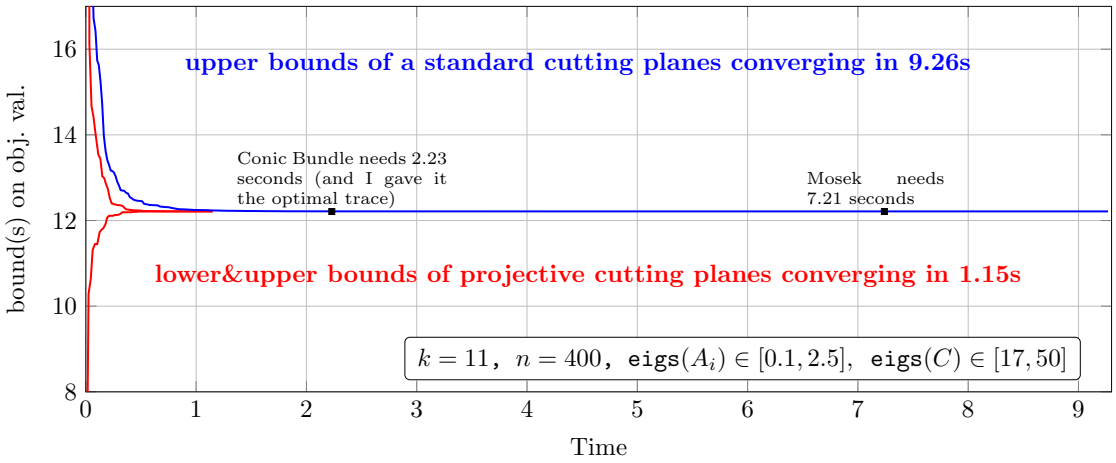
A preliminary glimpse in a world of strictly SDP matrices

Let us first present a short test on an instance with $A_0,\ A_1, \dots, A_k \succ \mathbf{0}$, *i.e.*, all the matrices involved are strictly SDP. To our knowledge, this case did not receive much attention in benchmarking SDP algorithms.

[11] An approach using principles of Column Generation may be used in the future to make the LP solver work only with a subset of the $\mathbf{y}$ variables (at each iteration), fixing many others at zero.

[12] For this reason, after a brief communication with Hans Mittelmann, I did not ask him to include my algorithm in this benchmarking tool.

Yet it may be interesting because such matrices are not a (very) rare sight in many applications. The projection algorithm becomes easy to implement (Section 3.1 suffices) and very fast. Each input matrix was generated by constructing an eigendecomposition $P\texttt{diag}(\lambda_1,\ \lambda_2, \ldots, \lambda_n)P^\top$ with a random orthonormal matrix P and eigenvalues $\lambda_1,\ \lambda_2, \ldots, \lambda_n$ randomly sampled within the ranges specified in the legend of the figure below. We use $\mathbf{b} = \mathbf{1}_k$, $k = 11$ and $n = 400$. This first experiment confirms `Proj-Cut-Pl` has the potential to compete with other methods. The standard `Cutting-Planes` needs almost 10 seconds while the gap reported by `Proj-Cut-Pl` after 0.33 seconds is hardly noticeable in the figure below.



5.1 Instances from existing benchmarks

Table 1 compares multiple solvers [20, 8, 10, 18, 1] on the (very) sparse instances `buck` and `vibra`. Notice that `Proj-Cut-Pl` can not compete with `Mosek` or `SeDuMi` on the largest sizes; when k becomes 544 (or, worse, 1200), it needs too many cutting planes to obtain a sufficiently tight and descriptive polyhedral outer approximation. On the other hand, `Proj-Cut-Pl` may compete well with some of the solvers listed in the bottom half of Table 1. However, this comparison is provided *for indication only*: similar to our algorithm, which is primarily designed to be most effective in certain cases (dense highly-feasible instances), some of these solvers are developed to be highly efficient in certain other cases (*e.g.*, low rank matrices); others target more general SDP problems (*e.g.*, with quadratic objective functions). They do not all have exactly the same stopping conditions (*e.g.*, optimality or feasibility tolerances) and they are not even written in or invoked using the same programming languages (we used `Matlab` for `Mosek` and `SeDuMi` and `Julia` for all others).

Software	Instance							
	`vibra1`	`buck1`	`vibra2`	`buck2`	`vibra3`	`buck3`	`vibra4`	`buck4`
	$k = 36$		$k = 144$		$k = 544$		$k = 1200$	
`Proj-Cut-Pl`	0.5+3.3	0.8+1.6	1.6+7.7	1.9+10.6	7.8 + 345	56+621	18+22243	155+12609
`Mosek`	0.2	0.2	0.8	1	20	19	181	163
`SeDuMi`	0.5	0.1	1.6	1.7	60	66	723	943
`Clarabel`	13.7	13.7	14.3	14.4	34.9	33.6	368	356
`Loraine`	24.6	24.6	27.6	27.5	128	127	1210	1149
`ClusteredLowRankSolver`	6.4	6.4	670	602	> 50000	> 50000	> 50000	> 50000
`CSDP`	10.3	10.3	384.9	234	out of memory on a comp. with 16GB RAM			
`COSMO`	55	2792	3727	4956	13349	> 50000	> 50000	> 50000

Table 1: CPU wall-clock time (seconds) of eight solvers on well-known instances of increasing size (mono-thread). The `Proj-Cut-Pl` line reports a sum that represents the CPU time used by the feasibility heuristic (from Stage 2 of Section 2.2) plus the CPU time of the real projective iterations.

We hereafter mainly compare to `Mosek`, as the fastest solver identified in Table 1. Since it is clear that `Proj-Cut-Pl` can never compete well on these very sparse instances, let us introduce a procedure to make them dense. We first generate a random ortho-normal matrix $M \in \mathbb{R}^{n\times n}$ and then transform the input

Instance	Original sparse instance				New densified instance	
	k	n	LP cuts	`Mosek` [secs]	`Mosek` [secs]	`Proj-Cut-Pl` [secs]
`vibra1`	36	49	36	0.2	0.3	2.2
`vibra2`	144	193	144	0.8	7.5	11
`vibra3`	544	641	544	20	1325	561
`vibra4`	1200	1345	1200	181	36555	24450
`buck1`	36	49	36	0.2	0.01	0.9
`buck2`	144	193	144	1	6	14.2
`buck3`	544	641	544	19	1320	828
`buck4`	1200	1345	1200	163	38696	15931

Table 2: CPU time (seconds) of `Mosek` and `Proj-Cut-Pl` on eight instances from the literature, considering both their initial sparse expression (Columns 2-5) and their densified variant (Columns 6-7). Column "LP cuts" reports the number of initial linear constraints, *i.e.*, the cardinal of $\mathscr{C}$ in (1.C.3).

matrices by replacing $A_i \leftarrow M^\top A_i M$ for all $i \in [0..k]$. The eigenvalues of $X = A_0 - \sum_{i=1}^k A_i y_i$ do not change after this substitution, which reduces to writing all matrices in a different basis. The eigenvalues remain the same and each eigenvector $\mathbf{v}$ of any original optimal matrix $X^{\text{opt}} = A_0 - \sum_{i=1}^k A_i y_i^{\text{opt}}$ is translated into an eigenvector $M^\top \mathbf{v}$ of a new densified optimal matrix $M X^{\text{opt}} M^\top$. Thus, this transformation does not change the optimal solution(s) $\mathbf{y}^{\text{opt}}$: it's an equivalent reformulation.[13]

Table 2 compares the results of `Mosek` and `Proj-Cut-Pl` on both the original and the dense variants of the instances `vibra` and `buck`. In fact, if we do not report (again) the results of `Proj-Cut-Pl` on the original sparse variants in Columns 2-5, it is because it cannot keep up the pace with `Mosek`: for the largest instances that `Mosek` solves in at most 3 minutes, `Proj-Cut-Pl` may need hours (see first data row in Table 1). The matrices $A_0, A_1, \ldots, A_k$ of these largest instances with $k = 1200$ and $n = 1345$ contain together only roughly 18.000 non-zero entries that fill only around 0.0018% of the total $k\frac{n(n+1)}{2} \approx 1.000.000.000$ available positions.

On the other hand, this very same Table 2 suggests that these large instances may become (very) difficult for `Mosek` when they are densified. Columns 6-7 show `Proj-Cut-Pl` does represent in this case a real competition for `Mosek`, and probably for many other (IPM) solvers.

We will next present (Table 3) a similar comparison on the densified variants of the even larger instances `hand` and `foot`. To show a more general trend over multiple values of k, we also use the procedure below to reduce the value of k using a compression factor $\texttt{k}_{\texttt{div}}$.

Remark 5.A. *(compressed instances) Given an initial instance and a compression factor* $\texttt{k}_{\texttt{div}}$*, we generate a compressed instance variant by applying the following substitutions:*

$$A_1 \leftarrow \sum_{i=1}^{\texttt{k}_{\texttt{div}}} A_i,\ A_2 \leftarrow \sum_{i=\texttt{k}_{\texttt{div}}+1}^{2\cdot\texttt{k}_{\texttt{div}}} A_i,\ \ldots; \textit{or generally } A_{\ell+1} = \sum_{i=\ell\cdot\texttt{k}_{\texttt{div}}+1}^{(\ell+1)\cdot\texttt{k}_{\texttt{div}}} A_i\ \forall \ell \in \left[0..\left\lceil \frac{k}{\texttt{k}_{\texttt{div}}} \right\rceil - 1\right].$$

We also apply the corresponding transformation of $\mathbf{b}$*, i.e.,* $b_{\ell+1} = \sum_{i=\ell\cdot\texttt{k}_{\texttt{div}}+1}^{(\ell+1)\cdot\texttt{k}_{\texttt{div}}} b_i$*. Actually, if* $(\ell+1)\cdot\texttt{k}_{\texttt{div}} > k$*, the range of* i *in the last sum above stops at* k*. Thus, the value of* k *evolves to* $\left\lceil \frac{k}{\texttt{k}_{\texttt{div}}} \right\rceil$*.*

Table 3 suggests that `Mosek` dominates `Proj-Cut-Pl` when both n and k are around 1000 (or a bit larger). This comes from the fact that modelling the outer approximation with linear constraints may require too many iterations. However, `Proj-Cut-Pl` becomes comparatively faster as we compress the instance to decrease k; this is more and more evident as k is reduced to less than 150, at which point `Proj-Cut-Pl` *can become up to four times faster.*

The first data row of Table 3 also indicates that `Proj-Cut-Pl` requires less memory than `Mosek`, which corroborates memory profiling observations from other IPM experiments, *e.g.*, see the "out mem." entries

[13] If the resulting instance still contains (too) many values close to zero for any reason, we also tested a different procedure to make it surely dense: subtract from (or add to) each $A_0,\ A_1,\ A_2, \ldots A_k$ some (full-rank) matrix T. We defined T with 0.01 in all off-diagonal positions and 0.07 on the diagonal (so that its image is $\mathbb{R}^n$). This transformation does change the optimal solution of the instance.

Instance				Results		
	$\mathtt{k_{div}}$	k	n	`Mosek` seconds	`Proj-Cut-Pl` seconds	`Proj-Cut-Pl` iterations
`foot`	1	2209	2208	128GB not enough	gap 0.7% after 360000s[100h]	
`foot`	4	553	2208	100280	108727	4322
`foot`	8	277	2208	26529	20116	3125
`foot`	12	185	2208	13192	8487	1961
`foot`	16	134	2208	15624	4625	797
`foot`	20	111	2208	12703	3773	860
`hand`	1	1297	1296	37533	162031	9129
`hand`	4	325	1296	4479	13958	5910
`hand`	8	163	1296	1915	2075	1596
`hand`	12	109	1296	1524	1012	926
`hand`	16	82	1296	1049	402	477
`hand`	20	65	1296	1330	282	385

Table 3: CPU time (seconds) of `Mosek` and `Proj-Cut-Pl` on the densified variant of instances `hand` and `foot` compressed as described by Remark 5.A with the factor $\mathtt{k_{div}}$ from Column 2 ($\mathtt{k_{div}} = 1$ corresponds to the original instance).

in the last two columns of Table 10 (p. 35). More exactly, a generous RAM amount of 128 Gigabytes is insufficient for `Mosek` to carry out a single iteration for $n \approx k \approx 2000$, whereas `Proj-Cut-Pl` can at least provide a duality gap using the same resources. Namely, it can produce (although after many hours) a primal and a dual solution with a gap below 1% for the densified uncompressed (original) `foot` instance.

We used in this section a relatively slow cluster: for now, it is our only computing resource with enough RAM (128 Gigabytes) to record the biggest matrices from the above benchmark. We plan to extend Table 2 with instances `buck5` and `vibra5` as soon as we find a (super-)computer with at least 512 Gigabytes of RAM. `Proj-Cut-Pl` was implemented in `Matlab` version R2024b. All comparisons are carried out in mono-thread mode. We have tried three LP solvers throughout this study: `Gorobi 12.01`, `Mosek` 11.0 and the latest `Cplex` version that supports `Matlab` (*i.e.*, `Cplex 12.10`). Various tests (see, *e.g.*, Table 6, p. 28) suggest that `Gorobi 12.01` is a more robust solution for a large $k > 1000$; for $k \leq 1000$, we always use `Cplex 12.10`.

From the next section on, we will switch to a faster mainstream laptop with an Intel `i7-8665U` processor clocked at 1.90GHz with 16 Gigabytes of RAM. Its operating system is Linux Mint and the Linux kernel version is 4.15.0.

5.2 The case of n in the order of thousands

We here present a very simple method to generate instances with a very large n. Let us consider $A_0 = 10000 \cdot I_n$ and generate each A_κ with $\kappa \in [1..k]$ by putting at positions (i, j) and (j, i) the value $(\kappa + i)^2 + j$ modulo 10, for any $i \leq j$. We set $\mathbf{b}_\kappa = \lfloor \sqrt[3]{\kappa} \rfloor$. It's more convenient to generate such very large instances this way, instead of sharing on-line files that can take up dozens of gigabytes (yet, we did share some at http://cedric.cnam.fr/~porumbed/sdp/).

Table 4 compares `Proj-Cut-Pl` and `Mosek` on these instances. The interior point method of `Mosek` may need $O(k^2n^2 + kn^3)$ operations at each iteration, as discussed in Section 1.2.1, probably inducing a large slowdown for $n, k \geq 1000$ when no sparsity can be exploited to reduce this time complexity. Thus, Table 4 suggests that `Projective Cutting-Planes` may be comparatively a better solution method for a dense SDP program with a value of n in the order of thousands. For such large n, `Mosek` can easily exceed a time cut-off limit of 10000 seconds; for $n = 10000$, it cannot perform a single iteration within this time limit.[14]

Another characteristic of such instances that makes them well-suited to `Projective Cutting-Planes` is that few variables $\mathbf{y}$ are non-zero at optimality. This means that the needed outer approximation is really

[14] We used the mainstream laptop described in Section 5.1 for $n < 5000$. For the few (six) data entries with $n \geq 5000$ in Table 4, we used the slower cluster from Section 5.1 because the matrices become too big matrices to be recorded on the laptop. The runtime needed on the cluster was normalized (scaled down) to estimate the runtime on the laptop (a multiplication by $\frac{2}{3}$ seems fine). The difference in performance between the two methods is so large that any reasonable imprecision on this scaling factor would not upset any key conclusion. The cells "no mem" indicate that even the 128GB RAM of the cluster is not enough.

n	k =10	k =100	k = 500	k =1000	k = 2000
100	0.4:0.2	0.5:1.2	0.9:8.2	1.5:22	2.5:93
500	0.7:8.7	1.8:70	7.3:507	15:1182	27:6436
1000	3.0:71	7.4:527	30:3444	50:6389	95:tm. out
5000	55:5264	171:tm. out	334:tm. out	733:tm. out	no mem.
10000	211:tm. out	988:tm. out	no mem.	no mem.	no mem.
Optimal values below					
100	44.523	89.104	155.93	219.75	267.31
500	8.8459	17.693	30.964	44.114	53.081
1000	4.4192	8.8389	15.468	22.067	26.516
5000	0.8832	1.7664	3.0913	4.4150	?
10000	0.4416	0.8831	?	?	?

Table 4: Run-time comparison on very dense instances (with n reaching up to 10000) generated as in Section 5.2. Each data entry (except the heading and Column 1) reports the CPU times $\texttt{tm}_p$ and $\texttt{tm}_m$ of Proj-Cut-Pl and resp. Mosek under the format $\texttt{tm}_p$:$\texttt{tm}_m$ (in seconds).

not expensive: a few generated linear constraints that mainly involve these few variables may be enough.

The short table below delves into the details of the very low number of iterations needed by Proj-Cut-Pl. We noticed that it always finishes by returning $t^* = 1$ after a few iterations, *i.e.*, the outer solution constructed in a few iterations is actually feasible and optimal. The initial outer approximation contains the $n = 1000$ constraints from the first paragraph of Section 2.3 and the one from (2.D), hence a 1001 term in the next-to-last row below. These 1001 constraints are not essential here: even without them, an outer approximation with a few constraints (a small set $\mathscr{D}$) would be enough. This means that the dual solution matrix $S = \sum_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top$ constructed as in (1.D.3) will also have a very low rank, which is a property often sought in certain applications.

	We fix n to 1000 and vary k				
	$k = 10$	$k = 100$	$k = 500$	$k = 1000$	$k = 2000$
Iterations of Proj-Cut-Pl	2	4	3	2	3
Non-zero $\mathbf{y}$ components at optimality	1	2	2	1	2
Constraints $\mathscr{D}$ in the final (1.C.1)-(1.C.4)	1001+2	1001+4	1001+3	1001+2	1001+3
Rank of the dual matrix in (1.D.1)-(1.D.4)	1	2	2	1	2

5.3 Comparing multiples solvers on new easily-feasible dense instances

We present a new way of constructing highly-feasible dense instances.[15] We first randomly generate $\frac{n}{5}$ vectors meant to cover the null space of all input matrices A_0, A_1, $A_2, \ldots A_k$. To avoid setting exactly the same null space to each matrix, each of these vectors is inserted into the null space of A_0 with a probability of 20% and into the null space of each A_i for $i > 0$ with a probability of 80%. Once a null space of size n_z is generated for any such matrix, we randomly sample $n - n_z$ orthonormal eigenvectors that will represent the image of the matrix as follows. We first collect all these vectors as columns of an orthonormal matrix $P \in \mathbb{R}^{n\times n-n_z}$. We then construct the final matrix by generating an eigendecomposition $P\texttt{diag}(\lambda_1, \lambda_2, \ldots \lambda_{n-n_z})P^\top$ where each λ_j with $j \in [1..n - n_z]$ is chosen at random (between 400 and 500 for A_0, or between 10 and 100 for A_i when $i > 0$). Since all these eigenvalues are non-negative, we have A_0, A_1, $A_2, \ldots A_k \succeq \mathbf{0}$; the probability of having any $A_i \succ \mathbf{0}$ (*i.e.*, $n_z = 0$) is almost zero. Each entry in the objective $\mathbf{b}$ is randomly chosen between 0.5 and 1.5 and we impose $\mathbf{y} \geq \mathbf{0}$. We thus obtain a kind of generalization of an LP with non-negative coefficients. Proj-Cut-Pl easily finds the starting feasible point $\mathbf{0}_k$ because $\lambda_{\min}(A_0)$ is zero in (2.B).

Table 5 compares Projective Cutting-Planes with seven other solvers on seven such instances, considering increasing values of k; each instance name follows the format highFsb$k$, knowing that we always consider $n = 10 \cdot k$. The first data row represents the standard Proj-Cut-Pl version with $\epsilon_{\texttt{opt}} = 10^{-5}$, *i.e.*, Alg. 1 (p. 12) stops when the five first digits are equal (see Line 23). The next data row shows how our method finishes more rapidly if we only require a modestly accurate solution (first two digits equal). As the instance grows larger, all Proj-Cut-Pl variants can still finish in a matter of seconds. Only Mosek

[15] Publicly available on-line at http://cedric.cnam.fr/~porumbed/sdp/ in the SDPA data format.

Software	Instance						
	highFsb5	highFsb10	highFsb15	highFsb20	highFsb25	highFsb30	highFsb50
	$n, k = 50, 5$	$n, k = 100, 10$	$n, k = 150, 15$	$n, k = 200, 20$	$n, k = 250, 25$	$n, k = 300, 30$	$n, k = 500, 50$
Proj-Cut-Pl $\epsilon_{\text{opt}} = 10^{-5}$	0.26	0.37	0.88	0.91	1.06	1.09	2.51
Proj-Cut-Pl $\epsilon_{\text{opt}} = 10^{-2}$	0.23	0.27	0.58	0.64	0.67	0.76	1.41
Proj-Cut-Pl $\epsilon_{\text{opt}} = 10^{-6}$	0.28	0.42	0.99	1.06	1.14	1.30	2.96
Mosek	0.04	0.17	0.77	2.39	5.34	11.7	79
SeDuMi	0.1	0.36	1.40	2.21	4.66	10.1	61
ConicBundle$\binom{\text{opt trace}}{\text{provided}}$	0.06	0.62	0.97	1.71	11	—	—
Clarabel	5.26	309	3346	—	—	—	—
Loraine	0.03	24	—	—	—	—	—
ClusteredLowRankSolver	8.97	79	—	—	—	—	—
CSDP	8.2	49	441	2215	—	—	—

Table 5: CPU wall-clock time (seconds) needed by various solvers on different sizes of the new *dense highly-feasible instances* generated in Section 5.3. The entries marked with a dash may indicate either slight inaccuracies or a time-out. The cut-off time limit is one hour.

and SeDuMi seem to be able to keep pace on larger sizes. Other preliminary experiments suggest that the ConicBundle can also be very fast if we keep $k \leq 20$ and let only n grow.

Table 6 reports results on instances of a similar nature but we fix $n = 100$ and introduce the following difference. We set $A_1 = I_n$, $A_2 = -I_n$, $b_1 = 1$ and $b_2 = -1$. This will actually force the trace of the dual solution to be 1. This information will be explicitly provided to the ConicBundle, because it does need it. We also add $10I_n$ to A_0, to make the instance even more largely feasible. The main goal is to see how the three best methods from Table 5 scale when increasing k, while also investigating the impact of a large k on various LP solvers that can be used by Proj-Cut-Pl (at Line 5 in Alg. 1, p. 12).

Let us focus on Column 2 compared to the last three columns of Table 6: when using Proj-Cut-Pl with the most efficient LP solver, it can be faster than the three best methods identified previously. Since $n = 100$ is small, the geometry of the outer approximation is somehow lighter for these instances, so that the number of Proj-Cut-Pl iterations stays within a limited magnitude (*i.e.*, at worse on the order of hundreds). A general runtime cost threat that may arise from a complex geometry of the outer approximation lies in the potential explosion of the number of iterations that Proj-Cut-Pl may need for modelling it.

k	Proj-Cut-Pl with each of the 3 LP solvers below			Mosek	SeDuMi	ConicBundle
	cplex 12.10	gurobi 8.11	gurobi 12.1			
100	1.9 (9%)	2.3 (13%)	2.5 (13%)	1.5	1.8	2.4
500	5.8 (12%)	8.7 (23%)	9.3 (23%)	13	22	25
1000	24 (14%)	27 (32%)	27 (32%)	36	77	72
1500	18 (16%)	33 (32%)	34 (34%)	86	175	353
2000	65 (37%)	66 (37%)	68 (37%)	285	384	803
2500	103 (31%)	170 (65%)	173 (66%)	539	416	tm. out
3000	180 (52%)	212 (66%)	212 (66%)	317	635	tm. out
4000	292 (55%)	329 (69%)	341 (71%)	592	1201	tm. out
5000	458 (68%)	796 (81%)	1209 (80%)	1005	2352	tm. out
7000	681 (72%)	1461 (86%)	1206 (85%)	2390	4589	tm. out
8500	1034 (68%)	1628 (83%)	1639 (85%)	3047	6987	tm. out
10000	1579 (79%)	3066 (90%)	4802 (88%)	3716	11210	tm. out

Table 6: The running time (secs.) of Projective Cutting-Planes and of the three fastest algorithms from Table 5 for a fixed $n = 100$ and increasing values of k. The information in parentheses is the proportion of CPU time spent on the LP solver. The cut-off time-out is the CPU time from Column 2 multiplied by 100.

Columns 2-4 of Table 6 report in parentheses the proportion of CPU time required by each of the three considered LP solvers. For large values of k, the LP solver step becomes the primary computational

Instance	Opt obj val	Proj-Cut-Pl					Mosek	SeDuMi
		Iterations	Time[s]	Proj	Sep	LP solver	Time[s]	Time[s]
highFsb5-lp	2.2211	2	0.15	17%	1%	2%	0.04	0.11
highFsb10-lp	5.1568	2	0.19	14%	1%	2%	0.17	0.19
highFsb15-lp	5.4538	24	0.81	49%	7%	5%	0.70	0.93
highFsb20-lp	5.5635	17	0.63	39%	6%	5%	1.52	2.32
highFsb25-lp	6.8495	20	0.85	41%	9%	5%	3.65	5.35
highFsb30-lp	4.6095	14	0.84	39%	7%	4%	6.37	9.40
highFsb50-lp	7.3828	8	1.61	55%	4%	1%	48.24	61.63
Instances above have $\mathbf{y} \geq 0$, while $\mathbf{y}$ can be either positive or negative ($\pm$) below								
highFsb5-lp$_\pm$	2.4149	9	0.17	17%	4%	7%	0.04	0.12
highFsb10-lp$_\pm$	7.6136	30+3	0.64	37%	6%	8%	0.18	0.30
highFsb15-lp$_\pm$	5.5436	83+7	2.24	56%	10%	7%	0.73	1.11
highFsb20-lp$_\pm$	9.0318	131+21	3.92	55%	12%	7%	2.88	2.20
highFsb25-lp$_\pm$	14.672	49+24	2.40	47%	10%	5%	4.17	5.26
highFsb30-lp$_\pm$	8.9571	124+16	5.33	47%	11%	6%	9.13	10.01
highFsb50-lp$_\pm$	16.5037*	117+27	13.9	60%	6%	3%	50.9	71.11

*This instance is actually unbounded. To keep it bounded, we impose $y_i \geq 0 \forall i \in [41..50]$.

Table 7: Comparing Projective Cutting-Planes with the two fastest IPM algorithms identified previously on the instances from Table 5 enriched with several initial (in)equalities, hence a suffix -lp in the instance name. Adding many inequalities would have no other impact on Proj-Cut-Pl than slowing down a bit the LP solver, while they may increase the size of the Newton systems in Interior Point Methods (IPMs).

bottleneck. If it does not take more than 90% of the total running time, it is because of Line 13 of Alg. 1 *i.e.*, it may also be expensive to compute the k coefficients of the new cut $\mathbf{a}^\top \mathbf{y} \leq c_a$ from the first-hit vector $\mathbf{v}_1$. This latter step could be parallelized since these k coefficients are independent. However, as it stands, these two steps (Lines 5 and 13) do take close to 90% of the total running time for $k \geq 7000$. The running time of the projection sub-problem becomes irrelevant in such cases. Perhaps a bit counter-intuitive, notice Gurobi 8.11 is perfectly fine for our LP solving needs – it's not necessarily slower than Gurobi 12.1.

5.4 Adding LP (in)equalities to the previous easily-feasible dense instances

We now consider the same instances from Table 5 but we introduce $\frac{k}{5}$ linear inequalities and one equality. The new instances will capture the whole model (1.C.1)-(1.C.4), *i.e.*, including a non-empty set of initial constraints $\mathscr{C}$ in (1.C.3). More exactly, we introduce inequalities $\sum_{i=1}^{5} y_{\kappa+i} \leq 2$ for each $\kappa \in 0, 5, 10, \ldots k-5$ and equality $y_1 + y_2 = 0$. The first benchmark sets $\mathbf{y} \geq \mathbf{0}_k$ while we allow $\mathbf{y}$ to be both positive and negative in a second instance set.

Table 7 presents the results on this new class of instances, providing more detail on Proj-Cut-Pl. Columns 1-2 provide the benchmark problem and its optimum value, using the same instance names as in Table 5, enriched with a suffix "-lp" to indicate the integration of the above LP component. Columns 3-7 are devoted to Proj-Cut-Pl. The number of iterations in Column 3 may be a sum where the second term is the number of iterations of the optional standard Cutting-Planes from Line 3 of Alg. 1 (the presolve from Remark 2.C, Section 2.4.1). Columns 5, 6 and, resp., 7 indicate the percentage of the total running time (from Column 4) spent on the projection sub-problem, the separation sub-problem and, resp, the LP solver. The time spent on the separation sub-problem involves determining the minimum eigenvalue in Line 8 (of Alg. 1) and optionally adding a cut by separation in Line 15. The second half of the table is devoted to the same instances as in the first half, only that $\mathbf{y}$ is allowed to be either positive or negative, hence a suffix "$_\pm$" in the instance name.

Table 7 shows that Proj-Cut-Pl can keep up with the fastest Interior Point Methods on (very) small instances, although it was slower. Yet, for the largest instance with $k = 50$ and $n = 500$, Proj-Cut-Pl is by an order of magnitude faster than the competition. When switching to a free $\mathbf{y}$, Proj-Cut-Pl can take 2-3 more time than in the non-negative case. This comes from the fact that the initial outer approximation from Section 2.3 is much weaker when $\mathbf{y}$ is free. This also leads to needing an artificial box meant to include the

Instance	Nominal Opt	Robust Opt	Proj-Cut-Pl Time (secs)	Iters	robust cuts	Mosek Time (sec)	Mosek solver calls	SeDuMi Time (secs)	SeDuMi solver calls
highFsb5-lp	2.2211	2.0192	0.19	4	3	0.06	2	0.13	2
highFsb10-lp	5.1568	4.6880	0.26	5	4	0.39	3	0.5	3
highFsb15-lp	5.4538	5.3580	0.80	19	6	2.61	4	4.01	4
highFsb20-lp	5.5635	5.5116	0.69	17	5	4.25	3	7.59	3
highFsb25-lp	6.8495	6.7921	0.95	17	5	9.9	3	16.4	3
highFsb30-lp	4.6095	4.5646	0.88	14	7	18	3	27	3
highFsb50-lp	7.3828	7.3299	1.6	8	5	145	3	276	3
Instances above have $\mathbf{y} \geq 0$, while $\mathbf{y}$ can be both positive and negative ($\pm$) below									
highFsb5-lp$_\pm$	2.4149	2.0192	0.25	7	6	0.13	5	0.36	5
highFsb10-lp$_\pm$	7.6136	7.1909	0.4	12+3	11	1.1	8	2	8
highFsb15-lp$_\pm$	5.5436	5.4246	1.6	58+7	18	2.2	4	4.4	4
highFsb20-lp$_\pm$	9.0318	7.9361	4.0	120+21	18	12.8	5	9.9	5
highFsb25-lp$_\pm$	14.672	12.5611	1.9	46+24	7	19	5	27	5
highFsb30-lp$_\pm$	8.9571	7.9725	4.86	134+16	26	47	6	61	6
highFsb50-lp$_\pm$	16.5037	15.3978	9.7	76+27	27	532	11	642	11

Table 8: Results for the robust version of the instances from Table 7. The Proj-Cut-Pl iterations in Column 5 are reported in the same manner as in Column 3 of Table 7.

feasible area (Remark 2.C, Section 2.4.1) before launching the main iterations, hence a number of presolve iterations reported as a second term in each sum from Column 3.

Columns 5 and 6 of Table 7 suggest the projection algorithm is a few times slower than the separation one. Since the separation reduces to finding the minimum eigen-pair of a matrix using very elaborately-tuned Matlab routines (refined over a few decades), the proposed projection approach seems quite reasonable in speed; it often takes less than half of the total running time.

5.5 Robust SDP optimization with prohibitively-many robust LP cuts

We here extend the linear constraints $\mathscr{C}$ from (1.C.3) in accordance to a robust optimization paradigm: all coefficients of a given nominal constraint $\mathbf{a}^\top \mathbf{y} \leq c_a$ can vary according to the following robust logic. We allow at maximum $\Gamma = 3$ of the nominal coefficients to go up or down by at most $\delta = 10\%$. The underlying assumption for using this uncertainty budget Γ is that in real life these coefficients cannot change all at the same time, always in an unfavorable manner. The key point is that this procedure can turn any nominal constraint into prohibitively-many linear robust cuts, so that the new set $\mathscr{C}$ can no longer be enumerated in practice.

Proj-Cut-Pl is essentially extended as follows: we call two projection algorithms at each iteration, one for the robust linear cuts $\mathscr{C}$ from (1.C.3) and one for the eigen-cuts $\mathscr{D}$ from (1.C.4). The robust projection is solved using the algorithm from [27, § 2.2.3]. The Cutting-Planes logic of these robust cuts is integrated very naturally into the overall Cutting-Planes framework of Proj-Cut-Pl. We are not aware of other SDP optimization technology that can so easily adapt to such a setting. To compare against Mosek and SeDuMi, we extend them as follows: (i) solve the nominal version of the problem, (ii) seek the most violated robust cut with regards to the *non-robust optimal* solution, (iii) re-solve the problem enriched with this most violated robust cut and determine a new *non-robust optimal* solution. The steps (ii)-(iii) are repeated in a Cutting-Planes fashion, *i.e.*, until no violated robust cut can be found at step (ii); in this latter case, this algorithm stops by reporting that the last solution computed at step (iii) is optimal in the full robust sense.

Table 8 compares these algorithms and it confirms the above advantages of Proj-Cut-Pl. It can be faster by a factor ranging from 10 to almost 100. This comes from the fact that the robust cuts are integrated seamlessly in the Proj-Cut-Pl framework, while they lead to calling the SDP solver too many times when using Mosek or SeDuMi. The number of solver calls from the last and the third-to-last columns also corresponds to the number of robust cuts needed by the corresponding solver. To the best of our knowledge, most IPMs have limited capability to take advantage of warm-starting strategies, *i.e.*, it is

difficult to start the process from some solution discovered before generating the last robust cut. In contrast to the Simplex algorithm, adding a new linear constraint to an SDP problem may require the IPM to restart from scratch. Or, at least, we are not aware of any off-the-shelf feature in `Mosek` or `SeDuMi` that performs such re-optimization.

5.6 A `Branch and Bound` powered by `Proj-Cut-Pl` for binary SDP optimization

Many SDP formulations have received considerable attention because they provide tighter bounds than a classical LP relaxation to certain integer or combinatorial (quadratic) optimization problems, see, *e.g.*, overview papers [17, 32]. One may sometimes solve a continuous SDP program only to obtain high-quality dual bounds during a `Branch and Bound` process, *i.e.*, the real goal is to solve the SDP program in mixed-integer variables. This is a more difficult NP-hard problem, which makes mixed-integer SDP optimization a very active area of research, see, *e.g.*, [3, 5, 19], to mention only a few more related papers from the last decade (many more references available therein).

Let us thus solve the same model (1.C.1)-(1.C.4) in binary variables $\mathbf{y} \in \{0,1\}^k$. We use a relatively simple `Branch and Bound` as a proof of concept; a more evolved variant may be the subject of a stand-alone future paper on this important topic. We branch on the natural order of variables, *i.e.*, first on y_1, then on y_2, y_3, etc. If no pruning is performed, the branching tree will have $\sum_{i=1}^{k} 2^i$ nodes besides the root one. Each generated node (or branch) fixes some variables $y_1, y_2, \dots y_\kappa$ to binary values and lets the remaining variables $y_{\kappa+1}, \dots y_k$ free, for some $\kappa \in [1..k]$. We compute a node-specific upper bound by solving a pure SDP program over the $k-\kappa$ unfixed variables. Using instances such that $A_1,\ A_2, \dots A_k \succeq \mathbf{0}$, we can always round down any $\mathbf{y}$ such that $\mathcal{A}^\top \mathbf{y} \preceq A_0$ and obtain a feasible solution. This procedure can turn the optimal fractional solution of a node-specific SDP program into a binary solution to the overall binary problem. A node is pruned if the rounded-down optimal value of its node-specific SDP program is no larger than the best binary feasible solution found so far. We also naturally prune a node if the fixed variables yield $\sum_{i=1}^{\kappa} A_i y_i \succ A_0$; in such case, the node-specific SDP program is infeasible.

We can even anticipate a few advantages of `Proj-Cut-Pl` without numerical results.

- Some branching nodes can be solved by performing a few projections and a few Simplex pivots that only update a previously solved model from a different branch. This is because we retain a unique `Cutting-Planes` model that is used to solve all SDP programs needed during the entire `Branch and Bound` process, integrating all eigen-cuts (1.C.3) ever generated. Only the bounds of the fixed variables $y_1, \dots, y_\kappa$ have to be changed when switching from one node to another.
- Since `Proj-Cut-Pl` generates an upper bound at each iteration, one can stop solving a node as soon as the rounded-down value of this upper bound is no larger than the best-known lower bound in binary variables. Even since the root node, this enables `Proj-Cut-Pl` to take advantage of a highly permissive stopping criterion: stop if the intermediate lower and upper bounds have the same rounded-down value.
- Since `Proj-Cut-Pl` generates a fractional feasible solution at each iteration, one can round it down to obtain a binary feasible solution; this increases the likelihood of finding heuristic solutions throughout all iterations of the node-solving process and not only at the last one (as in IPMs).

Instance	k	Opt	`Mosek`		`Proj-Cut-Pl`		
		obj val	Nodes	Time[s]	Nodes	Time[s]	Iterations
`highFsb5`	5	4	3	0.20	3	0.24	1+0
`highFsb10`	10	6	11	2.9	13	0.54	3+9
`highFsb15`	15	6	101	101	105	3.5	3+136
`highFsb20`	20	5	19	43	19	0.86	6+10
`highFsb25`	25	5	27	105	29	1.8	7+28
`highFsb30`	30	5	1	10	1	0.98	7+0
`highFsb50`	50	5	17	998	17	3.9	9+10

Table 9: A `Branch and Bound` powered by `Proj-Cut-Pl` against the same `Branch and Bound` powered by `Mosek` on the instances from Table 5, with a modified (integer) objective function $\mathbf{b} = \mathbf{1}_k$. Our method is faster because it retains a unique cutting planes model throughout the whole `Branch and Bound` process.

Table 9 reports the `Branch and Bound` results. The first three columns present the instance. Columns 4 and 5 provide the number of branching nodes, and, resp., the CPU time of the `Mosek`-powered `Branch and Bound`. Columns 6 and 7 report the same information for `Proj-Cut-Pl`. The last column provides the number of `Proj-Cut-Pl` iterations in the format $r + b$, where r is the number of iterations needed at the root node and b is the number of iterations executed while exploring the rest of the branching tree.

These results confirm the above theoretical advantages of `Proj-Cut-Pl`. If it is *an order of magnitude faster* than the `Mosek` solution, this is mainly because it does not solve each node-specific SDP program from scratch. The `Proj-Cut-Pl` approach retains all eigencuts ever generated throughout the entire process of constructing the branching tree. A few Simplex pivots may be enough to re-optimize when switching from one node to another. For the first instance, it was able to solve the node-specific SDP programs corresponding to $y_1 = 0$ and $y_1 = 1$ (the top two branches) without needing a single new eigen-cut: the outer approximation computed at the root node was sufficiently tight and descriptive to close the gap of these two child nodes.

The two methods do not necessarily construct the same number of nodes or the same branching tree because they differ in she way they determine binary SDP solutions $\mathbf{y} \in \{0,1\}^k$. While `Mosek` solves each node-specific SDP program to optimality, `Proj-Cut-Pl` can stop earlier, as soon as the rounded-down intermediate upper bound is small enough. Yet, the extra-work used to compute the final optimal solution of each branch does offer an interesting advantage to `Mosek`: by rounding down this optimal solution, it may eventually find a higher quality binary solution. This explains why `Mosek` needs a couple fewer branching nodes for `highFsb10`, `highFsb15` and `highFsb25`. If we replace objective function $\mathbf{b} = \mathbf{1}_k$ by $\mathbf{b} = 10 \cdot \mathbf{1}_k$, both methods require 567 nodes to solve `highFsb15`; they develop the same branching tree, because the rounding condition for stopping `Proj-Cut-Pl` earlier becomes more stringent. Yet we noticed that the `Mosek` approach needs 534 seconds to solve `highFsb15` in this case, while `Proj-Cut-Pl` can solve it in 14 seconds using 11+492 iterations.

Corroborating with the results from the previous Section 5.5, this last experiment confirms that `Proj-Cut-Pl` naturally has a higher efficiency in performing re-optimization. It can easily capitalize on not needing to restart from scratch when new constraints are generated. If the original model (1.C.1)-(1.C.4) contains an excessive number of linear constraints $\mathscr{C}$ but most of them are not active at optimality, `Proj-Cut-Pl` may generate only a part of them, on demand, whenever they are needed, *e.g.*, by a branching rule, by the robust optimization paradigm, or by any other logic that may imply generating cuts on the fly.

6 Conclusions and prospects

We have presented an algorithm powered by a new idea in SDP optimization. Its comparative effectiveness is maximized when applied to (easily-feasible) instances with n in the thousands, to the extent that other (IPM) algorithms may run out of gas for such sizes. The new method is less competitive for instances at the edge of feasibility or if k is too large; in the latter case, modelling the outer approximation with linear constraints may require too many cutting planes. This work is not primarily meant to be only devoted to numerical competition: the underlying implementation is a proof of concept and not (yet) a direct competitor to `Mosek` or to other well-established solvers in the view of the standard user. This is a future goal that involves many non-research engineering aspects, including software packages, documentation, additional testing protocols, usage examples, etc.

One advantage of `Proj-Cut-Pl` that goes beyond pure numerical speed is generating both a primal-feasible and a dual-feasible SDP solution at each iteration, a feature that does not exist (built-in) in most SDP optimization methodologies. This makes the new approach particularly interesting in applications that only require (very) modestly accurate optimal solutions, to the extent it can find such solutions in (very) few iterations. Another interesting feature is an intrinsic ability to accommodate re-optimization; unlike an IPM, `Proj-Cut-Pl` does not need to restart from scratch if the given SDP program is enriched with a new linear constraint. We capitalized on this feature in two contexts that require modifying the SDP program on the fly: a robust SDP program with prohibitively-many robust linear constraints (1.C.3) in Section 5.5 and a binary SDP program solved by `Branch and Bound` in Section 5.6. In the latter case, each branching rule simply imposes a specific linear constraint (1.C.3) on the SDP continuous relaxation.

More important for us, we hope the underlying idea may pave the way for solving more general conic problems, exploiting relatively-lightweight `Cutting-Planes` tools originating from linear programming. The idea is to use the projection tool to discover points inside the given cone and linear constraints to outer approximate the same cone. Whenever it is possible to design a sufficiently fast projection algorithm for

a particular cone, `Proj-Cut-Pl` may have the potential to become a practical lightweight solution to the associated conic problem.

References

[1] Brian Borchers. CSDP, a `C` library for semidefinite programming. *Optimization methods and Software*, 11(1-4):613–623, 1999.

[2] Samuel Burer and Renato D.C. Monteiro. A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. *Mathematical programming*, 95(2):329–357, 2003.

[3] Frank de Meijer and Renata Sotirov. The Chvátal–Gomory procedure for integer SDPs with applications in combinatorial optimization. *Mathematical Programming*, 209(1):323–395, 2025.

[4] Daniel de Roux, Robert Carr, and R Ravi. Instance-specific linear relaxations of semidefinite optimization problems. *Mathematical Programming Computation*, pages 1–51, 2025.

[5] Tristan Gally, Marc E Pfetsch, and Stefan Ulbrich. A framework for solving mixed-integer semidefinite programs. *Optimization Methods and Software*, 33(3):594–632, 2018.

[6] Felix Ruvimovich Gantmacher. *The Theory of Matrices*. Chelsea Publishing Company, New York, 1959.

[7] Benyamin Ghojogh, Fakhri Karray, and Mark Crowley. Eigenvalue and generalized eigenvalue problems: Tutorial. *arXiv preprint arXiv:1903.11240*, 2019.

[8] Paul J. Goulart and Yuwen Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives, 2024.

[9] Ming Gu, Axel Ruhe, Gerard Sleijpen, Henk Van Der Vorst, Zhaojun Bai, and Ruipeng Li. *Generalized Hermitian Eigenvalue Problems*, chapter 5 of book *Templates for the Solution of Algebraic Eigenvalue Problems*, pages 109–133. SIAM, 2000.

[10] Soodeh Habibi, Michal Kočvara, and Michael Stingl. Loraine–an interior-point solver for low-rank semidefinite programming. *Optimization Methods and Software*, 39(6):1185–1215, 2024.

[11] Christoph Helmberg. The `conicbundle` library for convex optimization. www-user.tu-chemnitz.de/~helmberg/ConicBundle/.

[12] Christoph Helmberg. Semidefinite programming for combinatorial optimization, 2000. Habilitation thesis (Habilitationsschrift), Technische Universität Berlin.

[13] Christoph Helmberg and Franz Rendl. A spectral bundle method for semidefinite programming. *SIAM Journal on Optimization*, 10(3):673–696, 2000.

[14] Nicholas J Higham, Natasa Strabic, and Vedran Sego. Restoring definiteness via shrinking, with an application to correlation matrices with a fixed block. *SIAM Review*, 58(2):245–263, 2016.

[15] Rujun Jiang, Duan Li, and Baiyi Wu. Socp reformulation for the generalized trust region subproblem via a canonical form of two symmetric matrices. *Mathematical Programming*, 169:531–563, 2018.

[16] Hiroshi Konno, Jun-ya Gotoh, Takeaki Uno, and Atsushi Yuki. A cutting plane algorithm for semidefinite programming problems with applications to failure discriminant analysis. *Journal of Computational and Applied Mathematics*, 146(1):141–154, 2002.

[17] Monique Laurent and Franz Rendl. Semidefinite programming and integer programming. In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 393–514. Elsevier, 2005.

[18] Nando Leijenhorst and David de Laat. Solving clustered low-rank semidefinite programs arising from polynomial optimization. *Mathematical Programming Computation*, 16(3):503–534, 2024.

[19] Frederic Matter and Marc E Pfetsch. Presolving for mixed-integer semidefinite optimization. *INFORMS Journal on Optimization*, 5(2):131–154, 2023.

[20] MOSEK ApS. *The MOSEK API for MATLAB 11.0.29*, 2025.

[21] Beresford N Parlett. *The symmetric eigenvalue problem*. SIAM, 1998.

[22] Ting Kei Pong and Henry Wolkowicz. The generalized trust region subproblem. *Computational optimization and applications*, 58(2):273–322, 2014.

[23] Daniel Porumbel. Demystifying the characterizations of sdp matrices in mathematical programming. expository paper not (yet) published, worked between 2018 and 2025 (arXiv:2210.13072).

[24] Daniel Porumbel. Ray projection for optimizing polytopes with prohibitively many constraints in set-covering column generation. *Mathematical Programming*, 155:147–197, 2016.

[25] Daniel Porumbel. From the separation to the intersection sub-problem in Benders decomposition models with prohibitively-many constraints. *Discrete Optimization*, 29:148–173, 2018.

[26] Daniel Porumbel. Projective cutting-planes. *SIAM Journal on Optimization*, 30(1):1007–1032, 2020.

[27] Daniel Porumbel. Projective cutting-planes for robust linear programming and cutting stock problems. *INFORMS Journal on Computing*, 34(5):2736–2753, 2022.

[28] Hanif D Sherali and Barbara M.P. Fraticelli. Enhancing RLT relaxations via a new class of semidefinite cuts. *Journal of Global Optimization*, 22:233–261, 2002.

[29] Kartik Krishnan Sivaramakrishnan. *Linear programming approaches to semidefinite programming problems*. PhD thesis, Rensselaer Polytechnic Institute, 2002. (`mitchjrpi.github.io/phdtheses/kartik/rpithes.pdf`).

[30] Kartik Krishnan Sivaramakrishnan and John E Mitchell. A unifying framework for several cutting plane methods for semidefinite programming. *Optimization methods and software*, 21(1):57–74, 2006.

[31] Kartik Krishnan Sivaramakrishnan and John E Mitchell. Properties of a cutting plane method for semidefinite programming. *Pacific Journal of Optimization*, 8:779–802, 2007.

[32] Renata Sotirov. SDP relaxations for some combinatorial optimization problems. In *Handbook on Semidefinite, Conic and Polynomial Optimization*, pages 795–819. Springer, 2012.

[33] Natasa Strabic. *Theory and algorithms for matrix problems with positive semidefinite constraints*. PhD thesis, University of Manchester, 2016.

[34] Michael J Todd, Kim-Chuan Toh, and Reha H Tütüncü. On the Nesterov–Todd direction in semidefinite programming. *SIAM Journal on Optimization*, 8(3):769–796, 1998.

A Towards $n = 30000$ using the `Max-Cut` SDP relaxation

This appendix provides an additional experiment to confirm a speed advantage conjectured since Section 1.2.1 on the basis of more theoretical arguments. Even by massively exploiting sparsity, for $n \geq 7000$, it may become too difficult for an IPM (`Mosek` or `SeDuMi`) to calculate the Schur matrix since that requires $O(k^2n^2 + kn^3)$ operations. For such instances (*low k, huge n and sparse matrices*), `Projective Cutting-Planes` is probably one of the best available solutions; we will see that on benchmark below it can be 20 or 30 times faster than `Mosek` for $n \geq 10000$, while requiring less memory.

We generated new instances starting from the following program with $n = k$, derived from the Goemans-Williamson relaxation of the `Max-Cut` problem, see, *e.g.*, [23, Prop. 7.1.1] or [12, § 6.1.1]. It was obtained by simple manipulations of the dual of $\max\left\{\frac{1}{4}A_0^L \bullet S : \texttt{diag}(S) = I_n,\ S \succeq \mathbf{0}\right\}$, where A_0^L denotes the Laplacian matrix of the graph (see below). One can find exactly the same program in equation (79) of [17].

n, k	Proj-Cut-Pl					**Sparse** Proj-Cut-Pl					Mosek	SeDuMi
	Iters	Time[s]	Proj.	Sep.	LP-solv.	Iters	Time[s]	Proj.	Sep.	LP-solv.	Time[s]	Time[s]
50, 50	249	2.3	0.7	0.3	0.6	249	2.2	0.7	0.2	0.5	0.04	0.08
100, 100	536	8	1.7	0.8	3.0	531	7.4	1.7	0.7	3.0	0.08	0.2
150, 150	833	27	3.4	1.1	13	850	23	3.2	1.3	14	0.1	0.4
200, 200	1761	117	9	2.5	74	1672	97	8	2.4	74	0.1	0.6
250, 250	2930	351	20	5	239	2789	269	15	4	226	0.2	1.19
Instances below are like the above ones, but k was reduced to 10 via substitutions like $A_1 \leftarrow A_1^L + A_2^L + \cdots + A_{\frac{k}{10}}^L$												
1000, 10	47	8	4	2	0.1	47	**5**	2	1	0.1	11	42
2000, 10	22	42	23	8	0.1	22	**10**	6	2	0.1	82	468
4000, 10	17	350	214	85	0.2	17	**78**	60	8	0.1	667	4083
6000, 10	14	796	514	184	0.2	14	**208**	190	16	0.1	2661	13490
8000, 10	12	1771	1110	443	0.2	12	**489**	436	37	0.1	6941	35414
10000, 10	9	2139	1310	509	0.2	9	**680**	601	54	0.1	14338	73151
12500, 10	5	2388	1551	450	0.1	5	**707**	629	30	0.1	23318	141606
15000, 10	5	4072	2627	821	0.2	5	**1206**	1206	59	0.1	41348	out mem.
17500, 10	5	5613	4205	1244	0.6	5	**1912**	1724	81	0.1	out mem.	out mem.
20000, 10	6	11702	7565	2400	0.5	5	**3645**	3320	211	0.3	out mem.	out mem.
25000, 10	out of memory					6	**7729**	6630	253	0.7	out mem.	out mem.
30000, 10	out of memory					6	**14740**	11762	434	1.4	out mem.	out mem.

Table 10: Comparison on the Goemans-Williamson SDP relaxation of Max-Cut. Columns 2-6 provide the results of the standard Proj-Cut-Pl. The next five columns are devoted to its sparse version, that simply records all matrices in a sparse format. The bottom section provides results over a compressed version of the SDP program (A.1)-(A.2) using substitutions like $A_1 \leftarrow A_1^L + A_2^L + \cdots + A_{\frac{k}{10}}^L$, or $A_2 \leftarrow A_{\frac{k}{10}+1}^L + A_{\frac{k}{10}+2}^L + \cdots + A_{2\frac{k}{10}}^L$, *i.e.*, using the compression procedure from Remark 5.A (p. 25) with $\mathtt{k_{div}} = \frac{k}{10}$.

$$\min \quad \sum_{i=1}^{k} y_i \tag{A.1}$$

$$s.t \quad \sum_{i=1}^{k} -A_i^L y_i \preceq -A_0^L \tag{A.2}$$

$$\mathbf{y} \geq 0. \tag{A.3}$$

Each A_i^L for $i \in [1..k]$ is an n-by-n matrix that contains only one non-zero element, namely the element at position (i,i) is 1. A_0^L is the Laplacian matrix of the input graph, *i.e.*, $A_0^L = \widehat{D} - \widehat{A}$, where $\widehat{D}$ is a diagonal matrix such that $\widehat{D}_{ii}$ is the degree of vertex i and $\widehat{A}$ is the adjacency matrix. In theory, $\mathbf{y}$ is free; but by restricting constraint (A.2) to the diagonal, you will notice we need to have $y_i \geq \widehat{D}_{ii} \geq 0$. Each intermediate feasible solution $\mathbf{y}_{\text{in}}$ of (A.1)-(A.3) constructed by Proj-Cut-Pl yields an upper bound $\mathbf{1}_k^\top \mathbf{y}_{\text{in}}$. Each outer solution $\mathbf{y}_{\text{out}}$ produces the lower bound $0.8785 \cdot \mathbf{1}_k^\top \mathbf{y}_{\text{out}}$ for the original binary Max-Cut problem, using the approximation factor of this relaxation.

Notice $\mathbf{y} = \mathbf{0}_k$ is in general not feasible since the right-hand term of the main SDP constraint (*i.e.*, $-A_0^L$) is often not SDP. This is one of the reasons why the first stage from Section 2.2 often fails. Yet, the second stage (based on iterative exterior-to-interior projections) can always produce a feasible solution in very few iterations. More generally, we can also exploit the particular structure of this program: notice $\sum_{i=1}^{k} -A_i^L \prec \mathbf{0}$ implies that $\mathbf{y} = \alpha \mathbf{1}_k$ is surely feasible for a large enough α.

Table 10 evaluates the standard Proj-Cut-Pl against its sparse version, as well as against the IPM competition. The sparse version simply encodes all involved matrices in a sparse format. Each instance is constructed from a graph of n vertices in which the edges are chosen randomly, subject to this condition: never exceed a degree of 20. Thus, the graph is quite sparse and so is A_0^L, *i.e.*, A_0^L contains n non-zero values on the diagonal and at most $20n$ non-zeros non-diagonal entries (since the underlying graph has maximum degree 20). Recall the matrices $A_1^L, A_2^L, \ldots A_k^L$ contain only one non-zero element, so that all matrices of the form $-A_0^L + \sum_{i=1}^{k} A_i^L y_i$ remain rather sparse.

The first half of Table 10 represents a negative case study for `Proj-Cut-Pl`. Given the large number of iterations needed to construct an expensive outer approximation, `Proj-Cut-Pl` is no match for `Mosek`, `SeDuMi`, or probably for other IPMs. However, if we may cite historical information beyond IPMs, [12, § 6.1] states that "before Interior Point Methods (IPMs) for semidefinite programming become available, problems on complete graphs with 40 nodes were considered unsolvable". At that time, even by using a slower or older LP solver, `Proj-Cut-Pl` could have been competitive.

The second half of Table 10 is a positive case study for `Projective Cutting-Planes`. Each considered instance is built from a `Max-Cut` instance with $n = k$ by applying the compression from Remark 5.A with $\mathtt{k_{div}} = \frac{k}{10}$, leading to a program with $k = 10$ and a very large n. The optimum value of this program still produces an upper bound for the original binary `Max-Cut` problem.

The sparse `Proj-Cut-Pl` variant can solve this problem with $k = 10$ roughly ten times more rapidly than `Mosek` for $n \in [2000, 7000]$; the speed-up factor increases as n becomes larger, up to reaching a speed-up factor of 20 for $n = 10000$ or 30 for $n = 15000$, while requiring less memory.

For $n > 15000$, `Mosek` and `SeDuMi` exceed the available amount of RAM memory which is a fairly standard one for a mainstream laptop (16 Gigabytes). This must happen even by massively exploiting the sparsity of the input n-by-n matrices that contain few non-zeros: A_0^L has at most $21n$ non-zero entries (see above) and each original matrix A_i^L with $i \in [1..k]$ contains only one non-zero. The input data can thus hardly require more than a few megabytes using a reasonable sparse data structure to record these at most $21n + k$ non-zeros. The memory demand of the IPMs must come from their internal functioning probably related to the Schur matrix.