Semidefinite optimization and (robust) reoptimization via projective cutting planes

Daniel Porumbel

August 9, 2023

Abstract

Contents

1 Introduction

We focus on solving the following SDP.

$$(SDP_{\mathscr{C}}) \begin{cases} \max & \mathbf{b}^\top \mathbf{y} & \text{(1.1a)} \\ s.t & \mathcal{A}^\top \mathbf{y} \preceq C & \text{(1.1b)} \\ & \mathbf{a}^\top \mathbf{y} \leq c_a \ \forall (\mathbf{a}, c_a) \in \mathscr{C} & \text{(1.1c)} \\ & \mathbf{y} \in \mathbb{R}^k, & \text{(1.1d)} \end{cases}$$

where $\mathcal{A}^\top \mathbf{y}$ stands for $\sum_{i=1}^{k} A_i y_i$ and $A_1,\ A_2, \ldots, A_k \in \mathbb{R}^{n \times n}$; $C \in \mathbb{R}^{n \times n}$. Constraints (1.1c) could be incorporated into (1.1b) by enlarging the matrices $A_1,\ A_2, \ldots, A_k$ and C; yet, this is a theoretical property that will not be used in practice because it would introduce slow down all algorithms (unless $\mathscr{C}$ is very small).

These constraints (1.1b) can simply impose non-negativities $y_i \geq 0$, for various $i \in [1..n]$

The set of linear constraints $\mathscr{C}$ can be empty, reasonably sized or prohibitively large. In the latter case, we may impose some of these constraints from the very beginning, but they are are too numerous to be all

enumerated in practice. We will add them one by one on the fly, *i.e.*, using constraint generation scheme. We will also allow column generation as follows: we may increase k so that $\mathcal{A}^\top \mathbf{y}$ becomes for $\sum_{i=1}^{k+1} A_i y_i$, but without touching on the fly to (1.1c) by column generation.

In fact, for now, we will only illustrate the constraint generation scheme in a robust optimization setting: given a number of nominal constraints, we will make each one become a set of constraints by allowing the coefficients to vary according to some robust rules (and an uncertainty budget).

The dual is:

"$= b_i$" becomes "$\geq b_i$" if you add $y_i \geq 0$ to (1.1c)

$$(DSDP_{\mathscr{C}}) \begin{cases} \min\ C \bullet X + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} c_a x_i & (1.2a) \\ s.t.\ A_i \bullet X + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} a_i x_i = b_i\ \forall i \in [1..k] & (1.2b) \\ X \succeq \mathbf{0}, \mathbf{x} \geq \mathbf{0} & (1.2c) \end{cases}$$

If $\mathscr{C}$ contain some $(\mathbf{a}, c_a)$ so that $c_a = 0$, $a_i = -1$ and $a_j = 0\ \forall j \in [1..k], j \neq i$ for a fixed i, then $\mathbf{a}^\top \mathbf{y} \leq c_a$ reduces to $y_i \geq 0$. The sum in (1.2b) will contain a corresponding term $-x_i$. Since $x_i \geq 0$ in (1.2c), this means that the equality (1.2b) becomes an inequality. This explain the arrow pointing to the equality sign in (1.2b). In the rest of this paper, we will write our models using an equality constraint, which can be seen as an inequality one if the sum in (1.2b) does not cover the above $(\mathbf{a}, c_a)$. If the some does cover it, the inequality is implicit.

Strong duality is not a must in this work; we focus on (SDP).

2 Linear models used to solve the SDP program

It is more convenient to isolate the SDP features in (1.1) and reformulate (SDP) as follows (also omitting superfluous $\mathbf{y} \in \mathbb{R}^n$):

$$(SDP2_{\mathscr{C}}) \begin{cases} \max\ \mathbf{b}^\top \mathbf{y} & (2.1a) \\ s.t\ \ S = C - \mathcal{A}^\top \mathbf{y} & (2.1b) \\ \mathbf{a}^\top \mathbf{y} \leq c_a\ \forall (\mathbf{a}, c_a) \in \mathscr{C} & (2.1c) \\ S \succeq \mathbf{0} \iff S \bullet \mathbf{d}\mathbf{d}^\top \geq 0\ \forall \mathbf{d} \in \mathbb{R}^n & (2.1d) \end{cases}$$

Given the equivalence from the last constraint, this model can be seen as an LP with prohibitively many constraints; the last constraint makes it a semi-infinite LP.

A relaxation of this last program is obtained by replacing $\mathbf{d} \in \mathbb{R}^n$ in the last constraint (2.1d) with $\mathbf{d} \in \mathscr{D}$, for some $\mathscr{D} \subsetneq \mathbb{R}^n$. Thus, (SDP2) is relaxed into its relaxed version:

$$(SDP2_{\mathscr{C},\mathscr{D}}) \begin{cases} \max\ \mathbf{b}^\top \mathbf{y} & (2.2a) \\ s.t\ \ S = C - \mathcal{A}^\top \mathbf{y} & (2.2b) \\ \mathbf{a}^\top \mathbf{y} \leq c_a\ \forall (\mathbf{a}, c_a) \in \mathscr{C} & (2.2c) \\ S \bullet \mathbf{d}\mathbf{d}^\top \geq 0\ \forall \mathbf{d} \in \mathscr{D} & (2.2d) \end{cases}$$

We can now write the dual of this program using LP duality, using a variable $\lambda_{\mathbf{d}}$ for each $\mathbf{d} \in \mathscr{D}$. We strengthen the feasible area of (1.2) obtaining $(DSDP2_{\mathscr{C},\mathscr{D}})$.

$$(DSDP2_{\mathscr{C},\mathscr{D}}) \begin{cases} \min\ C \bullet X + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} c_a x_i & (2.3a) \\ s.t.\ A_i \bullet X + \sum_{(\mathbf{a},c_a)\in\mathscr{C}} a_i x_i = b_i\ \forall i \in [1..k] & (2.3b) \\ X = \sum_{\mathbf{d}\in\mathscr{D}} \lambda_{\mathbf{d}} \mathbf{d}\mathbf{d}^\top & (2.3c) \\ \mathbf{x} \geq \mathbf{0}, \lambda_{\mathbf{d}} \geq 0\ \forall \mathbf{d} \in \mathscr{D} & (2.3d) \end{cases}$$

Given a fixed $\mathscr{D}$ obtained from the saturated constraints in the dual, you obtain some feasible X. Keeping fixed the null part of X (meaning the eigenvectors $\mathbf{u}$ associated to null eigenvalues), you can still solve (DSDP–S) by forgetting that X has to stay SDP (and maybe keeping the other variables x_i fixed). You

obtain an optimal X' with the same null space like X. You project $X \to X'$ and you obtain new eigenvectors that become constraints on $\mathbf{y}$.

You may not necessarily keep $\mathbf{x}$ fixed. You still get an optimal $(X, \mathbf{x})$ and shooting toward that direction is ok. The end point $(X', \mathbf{x}')$ is feasible w.r.t. all constraints except $X \succeq \mathbf{0}$. If the projection $X \to X'$ return some $t \in [0, 1]$, this means $(X + t(X' - X), \mathbf{x} + t(\mathbf{x}' - \mathbf{x}))$ is feasible. Then eigendecompose the resulting X and use the eigenvectors as new cuts in the $\mathbf{y}$ program.

Whenever you compute a relaxation, we have to keep some initial constraints that replace the constraint that S or resp (X) is SDP.

You may have $S \bullet X = 0$ (equiv. $SX = 0$) but X is SDP feasible while S is not SDP (except at optimality of the $\mathbf{y}$ primal).

You will play with $(DSDP2_{\mathscr{C},\mathscr{D}})$ and $(SDP2_{\mathscr{C},\mathscr{D}})$, alternating between them and projecting in both (primal and dual spaces), using threads. Optimizing the dual may be a computational bottleneck because it has a huge size, X has n^2 variables. Since all $S_{\mathtt{it}}$ will be infeasible as outer solutions, think of using infeasible interior point algorithms.

Keep in mind that we also allow increasing k by adding on the fly constraints on X like $A_{k+1} \bullet X = b_{k+1}$. Such constraints do not will not interfere with $\mathscr{C}$, *i.e.*, we consider $a_{k+1} = 1 \; \forall (\mathbf{a}, c_a) \in \mathscr{C}$. The only impact in $(SDP2_{\mathscr{C},\mathscr{D}})$ will be that for each $\mathbf{d} \in \mathscr{D}$ associated to (2.2d), $(C - \mathcal{A}^\top \mathbf{y}) \bullet \mathbf{d}\mathbf{d}^\top \geq 0$ is turned into $(C - \mathcal{A}^\top \mathbf{y} - A_{i+1}) \bullet \mathbf{d}\mathbf{d}^\top \geq 0$. Updating all these constraints in the linear models is not too hard; it is even possible to keep the old constraints (with $y_{k+1} = 0$) and add a new one. In the implementation you may define some k' to represent the maximum number of columns to be added on the fly. All constraints you send to the LP solver before these columns are added will simply put zeros from position k to position $k + k'$. The $\mathscr{C}$ constraints have all the right to do this; the SDP linear cuts can be updated later when the on-the-fly columns are inserted (you update them as described above).

3 The projection sub-problem

Sometimes you may know from the start that $X\mathbf{d} = X'\mathbf{d} = 0$ for some set of vectors $\mathbf{d}$. The advantage of projecting this way is that you are sure you can at least advance some ϵ towards X', which means improving the current sol $X_{\mathtt{it}}$. Anyway, you have to impose $X' \bullet \mathbf{d}\mathbf{d}^\top \geq 0$ when you search for a new X whenever the initial one satisfies $X\mathbf{d} = 0$. Only this way you can be sure you will be able to advance towards X' at least some ϵ. You have to do this with regards to all $\mathbf{d}$ in the null space of X (or to a basis of them), but, even better and more difficult, with regards to the whole space generated by such D, which means that you have an SDP constraint only with regards to a subspace.

But you will have to do the same when you solve the outer program in variables $\mathbf{y}$, ensuring $S' \bullet \mathbf{d}\mathbf{d}^\top \geq 0$ for any $\mathbf{d}$ in the null space of the starting S.

Property 3.1. *We will first project $X \to D$ under the hypothesis $X\mathbf{d} = 0 \implies D\mathbf{d} = 0$. The null space of X is included in the null space of D.*

The proof relies on the notion of congruent matrices. We say X and X' are congruent if there is some non-singular M such that $X' = MXM^\top$. We write $X \equiv X'$. Two congruent matrices have the same SDP status (see, for example, [**?**, Prop 1.2.3.]), *i.e.*, $X \succeq \mathbf{0} \iff X' \succeq \mathbf{0}$.

Definition 3.2. *(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.*

We prove in two steps that X has the same SDP status as its extended congruent X'.

1. $X \succeq \mathbf{0} \implies X' \succeq \mathbf{0}$. Assume for the sake of contradiction this is not the case: $\exists \mathbf{v}' \in \mathbb{R}^{n'}$ such that $\mathbf{v}'^\top X' \mathbf{v}' < 0$. This implies $\mathbf{v}'^\top MXM^\top \mathbf{v}' < 0$ or $X \not\succeq \mathbf{0}$, contradiction.

2. $X' \succeq \mathbf{0} \implies X \succeq \mathbf{0}$ Assume for the sake of contradiction this is not the case: $\exists \mathbf{v} \in \mathbb{R}^n$ such $\mathbf{v}^\top X \mathbf{v} < 0$. But 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 $\mathbf{v}'^\top MXM^\top \mathbf{v}' < 0$ or $\mathbf{v}'^\top X' \mathbf{v}' < 0$, contradiction.

To find the first-hit cuts, you will have to take take some $\mathbf{d} \in \mathbb{R}^n$ such that $X \bullet \mathbf{d}\mathbf{d}^\top = 0$ and extend $\mathbf{d}$ to a null vector $\mathbf{d}'$ of X' so that $X' \bullet \mathbf{d}'\mathbf{d}'^\top = 0$. Since M has full rank R, there exists some $M_* \in \mathbb{R}^{n' \times n}$ so that $M_*^\top M = I_n$. We can prove

$$D \bullet X = (M_* D M_*^\top) \bullet (M X M^\top)$$

To prove the above, do not forget $D \bullet X = tr(DX) = tr(DM_*^\top MX) = tr(DM_*^\top MXM^\top M_*)$, where we for the last development we used $M^\top M_* = I_n$, which is the transposed of $M_*^\top M = I_n$. Now, we know $tr(AB) = tr(BA)$ and we jump M_* from the end to the beginning of the last term, obtaining the desired result: $tr(M_* D M_*^\top MXM^\top)$.[1] Using this proof, you do not really need to have a left term $\bar{D} = M_* D M_*^\top$. You only need some $\bar{D}$ so that $M^\top \bar{D} M = D$. Indeed

$$D \bullet X = tr(DX) = tr(M^\top \bar{D} MX) = tr(\bar{D} MXM^\top) = \bar{D} \bullet (MXM^\top)$$

To get a $\bar{D}$ you do not need to compute any M_*, but simply solve $D = M^\top \bar{D} M$ in variables $\bar{D}$. If M is a triangular matrix coming from a Cholesky decomposition, you do back substitution. Otherwise, if it comes from an eigen decomposition, you compute its partial inverse by using the orthogonal vectors of M as rows in $M_*^\top$.

A)

Property 3.3. *The case $X \succ \mathbf{0}$ is included Prop 3.1*

We want to find

$$\max \{t : X + tD \succeq \mathbf{0}\} \tag{3.1}$$

We apply the Cholesky decomposition on X, obtaining a unique non-singular K so that $X = KK^\top$.[2] We then solve $D = KD'K^\top$ by back substitution in $O(n^3)$ (you first solve in variables Y the equation $D = KY$ which can be done in $O(n^3)$ by backsubstitution because K is triangular). Thus, (3.1) can be written as:

$$\max \left\{t : KI_nK^\top + tKD'K^\top \succeq \mathbf{0}\right\}, \tag{3.2}$$

equivalent (by congruence) to

$$\max \{t : I_n + tD' \succeq \mathbf{0}\}. \tag{3.3}$$

And the maximum value of t is here: $-\frac{1}{\lambda_{\min}(D')}$.

You can do the same with the eigendecomposition. Let $X = \sum_{i=1}^n \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$ and construct matrix V from the n columns $\sqrt{\lambda_i}\mathbf{v}_i$. You get $X = VIV^\top$, given that $\lambda_i > 0 \; \forall i \in [1..n]$ when $X \succ \mathbf{0}$. Like in the Cholesky approach above, you now solve $D = VD'V^\top$. But you can use that V is almost orthonormal. The inverse of V has as rows the elements $\frac{1}{\sqrt{\lambda_i}}\mathbf{v}^\top$.

B)

Now let's suppose $X \not\succ \mathbf{0}$, but Prop. 1 still holds. Take any Cholesky decomposition $X = KK^\top$ and we can prove $D = KD'K^\top$ can still be solved. Take full rank submatrix K_r of K, so that K_r, K and X all have rank r. We define matrix M_r by extracting rows and columns $J \subset [1..n]$ out of M for any $M \in \mathbb{R}^{n \times n}$. We than solve by back substitution $D_r = K_r D'_r K_r^\top$. This system has full rank in the world of $r \times r$ matrices.

Now X can be written $X = RX_rR^\top$ where $R \in \mathbb{R}^{n \times r}$ is an expansion full-rank matrix. If we restrict R to J we obtain I_r. The rest of the rows of R indicate how the remaining $n - r$ rows of X (and columns, by symmetry) are determined as a linear combination of the rows J of X (of X_r). You have to fully check: for $X \succeq \mathbf{0}$, $X\mathbf{d} = \mathbf{0} \implies K^\top \mathbf{d} = \mathbf{0}$ and $R^\top \mathbf{d}$ is also zero. This means that $D = RD_rR^\top$ and given $D_r = K_r D'_r K_r^\top$ (as solved above by back substitution) we have $D = RK_r D'_r K_r^\top R^\top$. We obtain that (3.1) reduces to

$$\max \left\{t : RK_rI_rK_r^\top R^\top + tRK_rD'_rK_r^\top R^\top \succeq \mathbf{0}\right\}, \tag{3.4}$$

---

[1] (if you only care for the zeros of the above, you can easily see $D \bullet X = 0 \implies DX = 0 \implies M_* DXM^\top = 0 \implies (M_* D M_*^\top)(MXM^\top) = 0 \implies (M_* D M_*^\top) \bullet (MXM^\top) = 0$)

[2] You can do the trick using the eigendecomposition: $X = \sum_{i=1}^n \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$. Construct V by putting as columns all n vectors $\sqrt{\lambda_i}\mathbf{v}_i$ and you get $X = VI_nV^\top$. Such V is almost orthonormal and easy to compute an inverse. Basically, the inverse has a rows $\frac{1}{\sqrt{\lambda_i}}\mathbf{v}_i^\top$

If the above $Z' = RK_r I_r K_r^\top R^\top + tRK_r D'_r K_r^\top R^\top$ is SDP for some t, then $Z = I_r + tD'_r$ is also SDP for that t, because Z' is a congruent expansion of Z (since RK_r has full rank r). It is thus enough to compute D'_r and t^* is $-\frac{1}{\lambda_{\min}(D'_r)}$ or ∞ if $D'_r \succeq \mathbf{0}$.

C)

I think we can generalize the above discussion if D has the form $D = \overline{D} + NEN^\top$ where $\overline{D}\mathbf{d} = 0 \forall \mathbf{d} \in \texttt{null}(X)$ and all columns of $N \in \mathbb{R}^{n\times r}$ are a basis for the null space of X (so that $XN = 0$). If $\lambda_{\min}(E) < 0$, you can not even advance a step length of ϵ along $X \to D$ and stay SDP (because: take any $\mathbf{r} \in \mathbb{R}^r$ such that $\mathbf{r}^\top E\mathbf{r} < 0$ and solve $\mathbf{r} = N^\top \mathbf{x}$ and $0 = D\mathbf{x}$ in variables $\mathbf{x} \in \mathbb{R}^n$. This system with n equations and n unknowns has to be feasible because N completes the vectors of the eigendecomposition of $\overline{D}$ associated to non-zero eigenvalues).

Else, if $E \succeq \mathbf{0}$, you can show the first hit constraint does not belong to N. In this case, it is enough to shoot towards $\overline{D}$ instead of D. Why? Let $\mathbf{v}+\mathbf{d}$ be the first hit constraint, where $\mathbf{d}$ belongs to the null space of X and $\mathbf{v}$ to the row space of X. We compute $(\mathbf{v}+\mathbf{d})^\top(\overline{D} + NEN^\top)(\mathbf{v}+\mathbf{d})$; since $\overline{D}\mathbf{d} = 0$ and $N^\top\mathbf{v} = 0$, this value is $\mathbf{v}\overline{D}\mathbf{v} + \mathbf{d}^\top NEN\mathbf{d} \geq \mathbf{v}\overline{D}\mathbf{v} = \mathbf{v}^\top(\overline{D} + NEN^\top)\mathbf{v}$. This means $\mathbf{v}$ is at least as good as $\mathbf{v}+\mathbf{d}$ and it also has to be a first-hit cut. And since $N^\top\mathbf{v} = 0$, you can only project towards $\overline{D}$ to find such $\mathbf{v}$

Now we go a bit beyond Prop. 3.1. We consider some $D = \overline{D} + NEN^\top$, where N contains as columns a basis of the null space of X.

If you can decompose $D = \overline{D} + NEN^\top$ where $\overline{D}\mathbf{d} = 0 \forall \mathbf{d} \in \texttt{null}(X)$, you'll end up with two choices: (1) $\lambda_{\min}(E) < 0$ and in this case you have $t^* = 0$ or (2) $E \succeq \mathbf{0}$ and in this case it is enough to solve $X \to \overline{D}$.

If such decomposition exists, then any $\mathbf{v}$ in the row image of X satisfies $\mathbf{v}^\top D\mathbf{d} = 0 \forall \mathbf{d} \in \texttt{null}(X)$, which is equivalent to $XD\mathbf{d} = 0$. This is a necessary condition and we will show it is sufficient. Let us consider the eigendecomposition of $X = \sum_{i=1}^r \lambda_i \mathbf{v}_i \mathbf{v}_i^\top$. We now construct matrix $V \in \mathbb{R}^{n\times r}$ by putting together the elements $\sqrt{\lambda_i}\mathbf{v}_i$. We construct $N \in \mathbb{R}^{n\times(n-r)}$ by collecting the $n-r$ eigenvectors of X associated zero eigenvalues. We obtain that $[\,V\ N\,]$ is non-singular in $\mathbb{R}^n$ and the dot product of any two columns is 0 (the matrix is almost orthogonal). We have by construction

$$X = \underbrace{\begin{bmatrix} V & N \end{bmatrix}}_{r+(n-r)} \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V^\top \\ N^\top \end{bmatrix} \tag{3.5}$$

and we can also compute (3.6)

$$\begin{aligned} D &= \underbrace{\begin{bmatrix} V & N \end{bmatrix}}_{r+(n-r)} \begin{bmatrix} F & G^\top \\ G & E \end{bmatrix} \begin{bmatrix} V^\top \\ N^\top \end{bmatrix} \qquad (3.7) \\ &= VFV^\top + NEN^\top + VG^\top N^\top + NGV^\top \end{aligned}$$

Suppose for the sake of contradiction that $G \neq 0$. We can compute $\mathbf{v}^\top D\mathbf{d} = \mathbf{v}^\top VG^\top N^\top \mathbf{d}$. Since $\mathbf{v}$ has rank r, we can make $\mathbf{v}^\top V$ be any (transposed) element $\overline{\mathbf{v}}$ of $\mathbb{R}^r$ (because otherwise V can not have rank r since the possible $\mathbf{v}$ considered here can actually span all the useful part of $\mathbb{R}^n$ since there is no use in covering anything that contain some portions of $\texttt{null}(X)$). Similarly, $N^\top\mathbf{d}$ can cover any element $\overline{\mathbf{d}}$ of $\mathbb{R}^{n-r}$. As such, if $G \in \mathbb{R}^{r\times(n-r)}$ is not null, we can make $\overline{\mathbf{v}}^\top G\overline{\mathbf{d}} \neq 0$. This is a contradiction of $\mathbf{v}^\top D\mathbf{d} = 0$ for any $\mathbf{v}$ in the row image of X and any $\mathbf{d} \in \texttt{null}(X)$. And this last condition would be implied by $XD\mathbf{d} = 0 \forall \mathbf{d} \in \texttt{null}(X)$ because $\mathbf{v}^\top = X\mathbf{v}'$ does have a solution in variables $\mathbf{v}'$.

We now obtain that $X + tD$ is congruent to $\left[\begin{smallmatrix} I_r + tF & 0 \\ 0 & tE \end{smallmatrix}\right]$. If $\lambda_{\min}(E) < 0$, we clearly have $t^* = 0$. Else, t^* is $-\frac{1}{\lambda_{\min}(F)}$ or ∞ if $\lambda_{\min}(F) \geq 0$.

This means that, beyond being able to project under the conditions of Prop. 3.1, you can also project under the following conditions.

Property 3.4. *We can project in reasonable time if* $X\mathbf{d} = 0 \implies XD\mathbf{d} = 0$.

It is not so hard computationally to determine F and E in (3.7). if $\left[\begin{smallmatrix} V^\top \\ N^\top \end{smallmatrix}\right]$ It is not so hard because it take $O(n^2)$ to compute the inverse of $[\,V\ N\,]$ since this matrix is almost orthogonal. This case (C) can even represent concurrence to case (B). You essentially use the eigendecomposition instead of Cholesky.

D)

If all above conditions fail, consider (from the eigendecomposition of X) a basis N for the null space of X. We are in this case because $DN \neq \mathbf{0}$. But we do not know the value of $D \bullet \mathbf{d}\mathbf{d}^\top$ for the various columns of N. To answer this question, we check the SDP status of D with regards to $\texttt{null}(X)$. For this, it is enough to check if $N^\top DN$ is SDP or not in the world of matrices of order $n - r$. I see two cases:

1. if the answer is negative we return $t^* = 0$. Let $\mathbf{v} \in \mathbb{R}^{n-r}$ the eigenvector of minimum eigenvalue for $N^\top DN$. We have $\mathbf{v}^\top N^\top DN\mathbf{v}$. Thus, $N\mathbf{v}$ is the first-hit constraint returned by the projection $X \to D$. We surely have $\hat{X} = X + \epsilon D \not\succeq \mathbf{0}$ for however small an ϵ.

   - Particular case: if $\texttt{null}(X)$ contains only one vector $\mathbf{d}$, this case can be detected faster (not SDP status check) by testing that $D \bullet \mathbf{v}\mathbf{v}^\top$ is less than 0.

2. Otherwise, we can surely take a small enough ϵ so that $\hat{X} = X + \epsilon D \succeq \mathbf{0}$. The first-hit constraint of $X \to D$ will not belong to $\texttt{null}(X)$ and $t^* > 0$. You can project from $\hat{X}$ towards D and we have the property $\hat{X}\mathbf{d} = 0 \implies D\mathbf{d} = 0$. You first try a small ϵ and if $X + \epsilon D \not\succeq \mathbf{0}$, you decrease it, up to the moment when you get some $X + \epsilon D \succeq \mathbf{0}$. But before you decrease it, take the eigenvector of $X + \epsilon D$ of minimum negative eigenvalue and add it to the pool of SDP cuts to return. And now you can apply the approach from point (B) or (C). Using this infinitesimal step, you are sure that any $\mathbf{d} \in \texttt{null}(X) \setminus \texttt{null}(D)$ gets out of $\texttt{null}(X')$. Still, are you sure you do not to increase the null space of X when advancing this infinitesimal step? Can $X + \epsilon D$ produce by happenstance a new null vector $\bar{\mathbf{d}}$ such that $X\bar{\mathbf{d}} \neq 0$ but $\hat{X}\bar{\mathbf{d}} = 0$? To defend against such problem, it is enough to take all eigenvectors $\bar{\mathbf{d}}$ associated to a zero eigenvalue of $X + \epsilon D$ and check if they are (still) orthogonal to the image of X?

   - If this is the case, we surely have $\texttt{null}(X + \epsilon D) \subsetneq \texttt{null}(X)$. We can't have equality because we said there are some $\mathbf{d} \in \texttt{null}(X)$ such that $D\mathbf{d} \neq 0$ and such $\mathbf{d}$ will not belong to $\texttt{null}(X + \epsilon D)$.
   - If this is not the case, you have $X \bullet \bar{\mathbf{d}}\bar{\mathbf{d}}^\top > 0$ and $(X + \epsilon D) \bullet \bar{\mathbf{d}}\bar{\mathbf{d}}^\top = 0$. You can not move any further beyond ϵ because the last equality above will become an inequality. This means $t^* = \epsilon$ and $\bar{\mathbf{d}}$ is a first-hit cut.

Heuristic projection

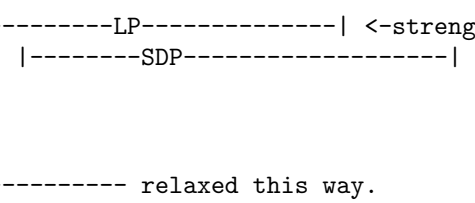
There may be many numerical problems if some eigenvalues of X are too small, especially when projecting from $X + \epsilon D$. In such case we may resort to a heuristic projection that essentially projects from some $\hat{X} = X + \sum \alpha_i \mathbf{v}_i \mathbf{v}_i^\top$, where the sum is carried over all eigenvectors $\mathbf{v}$ that have a very small (or null) eigenvalue. This may be better than using $\hat{X} = X + \epsilon D$ as above, because this last matrix will have infinitesimal eigenvalues.

4 Finding a feasible solution

4.1 Relaxing and strengthening the feasible area

Finding linear subsets of the feasible area is an important endeavour because we can thus generate feasible solutions.

You can somehow always strengthen and relax an LP area.



If you apply SDP cutting planes when optimizing over LP, you can be sure the relaxed part (at left in above figures) is separated.

4.2 A restricted subset of the feasible area based on $\lambda_{\max}(A+B) \leq \lambda_{\max}(A) + \lambda_{\max}(B)$

We need a constraint that would imply $\mathcal{A}^\top \mathbf{y} - C \leq 0$. This desired property is a consequence of $\lambda_{\max}(\mathcal{A}^\top \mathbf{y} - C) \leq 0$ and this later stuff is surely true if $\lambda_{\max}(\mathcal{A}^\top \mathbf{y}) + \lambda_{\max}(-C) \leq 0$. Using $\lambda_{\max}(-C) = -\lambda_{\min}(C)$ and after splitting $\mathcal{A}^\top \mathbf{y}$ considering $\mathbf{y} \geq 0$, the following implies the desired property.

$$\sum \lambda_{\max}(A_i) y_i \leq \lambda_{\min}(C)$$

Or

$$\sum \lambda_{\max}(P^\top A_i P) y_i \leq \lambda_{\min}(P^\top C P)$$

The whole idea of the initial development above is to try to make $(P^\top C P)$ become a diagonal with equal elements, because we hope it is better.

If $\lambda_{\min}(C) > 0$, you choose the P so that $P^\top C P = I_n$ and you obtain

$$\sum \lambda_{\max}(P^\top A_i P) y_i \leq 1 \tag{4.2.1}$$

If $\lambda_{\min}(C) \leq 0$ and you are obsessed to having I_n on the rhs, then consider $C' = \lambda_{\min}(C) I_n - \epsilon I_n$ as a sort of lower bottom of C. And we can say $C = C' + D$ The main SDP constraint is equivalent to $\mathcal{A}^\top \mathbf{y} - C' - D \preceq \mathbf{0}$ with $D \succ \mathbf{0}$. You can actually take any C' that makes D strictly SDP, for instance you can take $C' = \sum(\lambda_i - \epsilon)\mathbf{v}\mathbf{v}^\top$, summing over all negative eigenvalues λ_i.

Based on the eigendecomposition of any D, we can write it $D = P I_n P^\top$, where the columns of P are obtained by multiplying each column of $\bar{P}$ from the original eigendecomposition $D = \bar{P} diag(\lambda) \bar{P}^\top$ by $\sqrt{\lambda_i}$.

Then, the main SDP constraint becomes

$$\lambda_{\max}(\sum P^\top A_i P y_i - P^\top C' P - I_n) \leq 0$$

or

$$\lambda_{\max}(\sum P^\top A_i P y_i - P^\top C' P) \leq 1$$

If $\mathbf{y} \geq \mathbf{0}$, we can strengthen this into $\sum \lambda_{\max}(P^\top A_i P) y_i + \lambda_{\max}(-P^\top C' P) \leq 1$ based on Weyl's inequalities. This reduces to (4.2.1) if $C' = 0$, *i.e.*, if $C \succ \mathbf{0}$. If you find a solution $\mathbf{y}$ to this, this $\mathbf{y}$ is a feasible solution of the original problem.

If $\mathbf{y} \not\geq \mathbf{0}$, you can define variable $\overline{\mathbf{y}}$ so that $\overline{y}_i \geq \lambda_{\max}(P^\top A_i P) y_i$ and $\overline{y}_i \geq \lambda_{\min}(P^\top A_i P) y_i$. These two conditions will make sure that $\overline{y}_i \geq \lambda_{\max}(P^\top A_i P y_i)$. The function $y_i \to \overline{y}_i$ is piecewise convex, like a module. The main SDP constraint is strengthen into $\sum \overline{y}_i + \lambda_{\max}(-P^\top C P) \leq 1$.

4.2.1 Valid inequalities using Weyl's inequalities

For $\mathbf{y} \geq \mathbf{0}$, the following hold and has to stay subzero.

$$\lambda_{\max}(\mathcal{A}^\top \mathbf{y} - C) \geq \sum \lambda_{\min}(A_i) y_i + \lambda_{\min}(-C)$$

or, in a stronger version using Weyl

$$\lambda_{\max}(\mathcal{A}^\top \mathbf{y} - C) \geq \sum \lambda_{\min}(A_i) y_i + \lambda_{\max}(-C)$$

or you can replace any $\lambda_{\min}$ in the first equality by a $\lambda_{\max}$. Moving C to left, the subzero constraint becomes

$$\lambda_{\min}(C) \geq \sum \lambda_{\min}(A_i) y_i$$

So:

$$\sum \lambda_{\min}(A_i) y_i \leq \lambda_{\min}(C) \text{ relaxes the feasible area}$$

$$\sum \lambda_{\max}(A_i) y_i \leq \lambda_{\min}(C) \text{ from (4.2.1) restricts it}$$

And you can then apply this for any principal minor of all these matrices!

I think the most general is to define $\lambda_i(C)$ as the i^{highest} eigenvalue of C. And you can state

$$\lambda_{\max}(\mathcal{A}^\top \mathbf{y} - C) \geq \lambda_{p_0}(-C) + \sum_{i=1}^{k} \lambda_{p_i}(A_i) y_i$$

holds for any $p_0, p_1, \ldots p_k$ so that $\sum_{i=0}^{k} p_i \geq k * n + 1$. The second inequality above corresponds to $p_0 = 1$ and $p_i = n \ \forall i \in [1..n]$. A different choice is $p_0 = p + 1$, $p_i = n \ \forall i \in [1..n] \setminus \{j\}$ and $p_j = n - p$. You may get

$$\lambda_{\min}^{p+1}(C) \geq \sum_{i \neq j} \lambda_{\min}(A_i) y_i + \lambda_{\min}^{p+1}(A_j) y_j$$

You can determine more valid inequalities by increasing p in various ways, but also by considering various minors of the $n \times n$ matrices. If you left multiply with $P^\top$ and right multiply with P so that you have $P^\top C P = I_n$, then you have interest to increase p to $n - 1$. The above inequality will become:

$$1 \geq \sum_{i \neq j} \lambda_{\min}(P^\top A_i P) y_i + \lambda_{\max}(P^\top A_j P) y_j$$

You can do cutting planes, but maybe it is enough to do the above for each $k \in [1..n]$.

If you insist on cutting planes, Given $\mathbf{y}$ at current iteration, take as a good j, the one associated to the highest y_j. And then you can continue like in a maze, increasing p up to $n - 1$. If you left-multiply by $P^\top$ and right multiply by P so that $C = I$, you can have a fixed value on the left part of above. I do not find cutting planes here revolutionary; because the $p - 1$ up steps can only reduce a limited thing in the coefficients. But what if you go beyond $p - 1$ in illegal waters? I mean you may cut of some feasible S.

I find it hard to move to non-negative $\mathbf{y}$. But if you can consider $\mathbf{y} \geq -\Delta$, you can recast $\mathcal{A}^\top \mathbf{y} - C \preceq \mathbf{0}$ using non-negative variables so that $\mathbf{y} = \bar{\mathbf{y}} - \Delta$ where Δ is a vector having the same value Δ at each position, leading to

$$\mathcal{A}^\top \bar{\mathbf{y}} - C - \mathcal{A}^\top \Delta \preceq \mathbf{0}$$

You then get back to the original question only working with an updated $C = C - \mathcal{A}^\top \Delta$.

5 Projective Cutting Planes in DD restricted-relaxed spaces

5.1 A DD approach with σn additional variables

For large n being DD is too strong a condition. Instead of $S_{ii} > \sum_{j \neq i} |S_{ij}|$ (and do not forget $S = C - \mathcal{A}^\top \mathbf{y}$ or we can use even $S = PCP^\top - \sum P A_i P^\top y_i$ for some rang n matrix P), we propose

$$S_{ii} > \sum_{j \in smart} |S_i j|,$$

where *smart* can be all positions j that are within a distance δ to i, *i.e.*, $|i - j| \leq \delta$.

You choose above an P that will make the most SDP matrix diagonal with ones as positive values on the diagonal. It is easier to satisfy a DD constraint when all elements on the diagonal are positive. After reordering the elements you will have I_m in the top-left corner. You post and pre multiply all A_i by P. Then you perform the same thing focusing only on the most SDP matrix with regards the bottom-part matrix of size $(n - m) \times (n - m)$. You obtain P' to make that matrix with as many as possible ones on its diagonal. P' will have size $(n - m) \times (n - m)$. The final P will be (this is a **constructive heuristic** to be followed by LS)

$$P \begin{bmatrix} I_m & 0 \\ 0 & P' \end{bmatrix}$$

It is important to strive for an S with $S_{11} = S_{22} = S_{33} \cdots = S_{nn}$, because if you take $\delta = 1$, then the 2-local matrices are full (SDP). There is no SDP matrix with some $S_{ij} > S_{ii} = S_{jj}$. And optimizing over 2-local DD with cut planes can only be useful, you can not converge to some point that is not SDP as long as you work with matrices S produced by an $\mathbf{y}$ that yield an equal diagonal. Then you push δ a bit and hope for the best. This is surely much better for feasibility search than looking from some I_n. And if you do

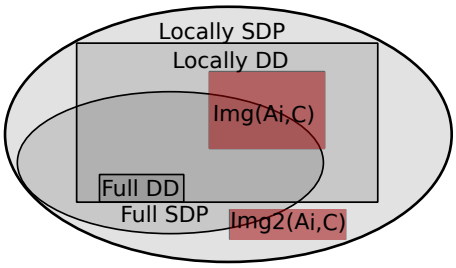
have an interior point you can always use projective cutting planes and produce SDP cuts and more interior points as you increase σ. So you have two cases in which the LP over σ-local DD matrices can not have a non-SDP optimal solution:

- $\delta = n - 1$, because you stay on DD SDP matrices
- $\delta = 1$ and equal diagonal

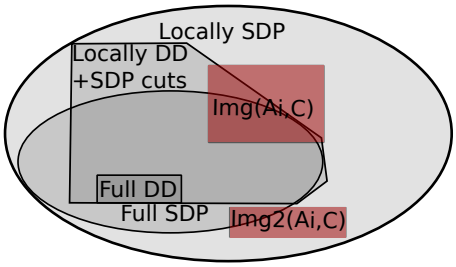
From some σ values on, your interior point will no longer be $\sigma-$DD, but you only have to push σ as high as you can. But you need an interior point and you add at worst a penalty param. In fact, you get the most SDP matrix A_i, back and left multiply with $P^\top$ and you get I_m in the top left corner. You add a second matrix with I_{n-m} in the bottom right corner.

You have to check that such a (δ-locally) DD matrix is SDP with regards to all vectors that contain non-null values only on the *smart* positions around each possible $i \in [1..n]$. When $\delta = 1$, (δ-locally) DD means that each diagonal element of S is larger than (the module of) its immediate neighbour in S. We ask for each 2×2 matrix that can be seen along the diagonal to be SDP because if $S_{11}, S_{22} \geq |S_{12}|$ in a 2×2 matrix S than S is SDP in the 2×2 world. But this linear condition is much easier to test than asking for the matrices along the diagonal to be SDP because that would involve a quadratic stuff $S_{11} \cdot S_{22} \geq |S_{12}|^2$.

We have $locally(DD) \subset locally(SDP)$ and $fully(SDP) \subset locally(SDP)$. We can solve something over $locally(DD)$ and add a few cutting planes, giving the figure below. The cutting planes you will add will be non-local. So instead of trying to generate all necessary SDP cuts, local or non-local, you generate only the non-local one. This is the advantage w.r.t., the standard method. If it is not very hard to generate all non-local cuts but it is hard to generate the local one, you may have at second image below a far easier problem. After enough non-local cuts you will end up inside $fully(SDP)$.



And after adding some SDP cuts



I think this can be useful when S has almost a block diagonal structure. If it is fully composed of diagonal block, we only need local SDP cuts. But since it is almost diagonal block, we may need a few non-local SDP cuts. In such case, the approach from this section will be much faster in finding a feasible solution. In a second stage, you go fully SDP. This sounds like a cool project: work with almost-block-diagonal matrices. In a first step you only have DD LPs with a few non-local SDP cuts. You get a feasible point and then you do ProjCutPlanes. This is a case in which the DD feasibility finder makes much sense!

5.2 A DD approach with $O(n^2)$ additional variables

AN EVEN BETTER local DD notion Let $z_j \geq S_{ij}, -S_{ij}$. We reduce $S_{ii} \geq \sum z_j$ to the following using the epigraph formulation.

$$S_{ii} \geq \delta q + \sum z_i^+, \text{where}$$

$$\mathbf{z}^+ \geq \mathbf{0}, z_i^+ \geq z_i - q, z_i^+ \geq -z_i - q,$$

or, more exactly, for each i you have:

$$S_{ii} \geq \delta q_i + \sum_{j \neq i} z_{ij}^+, \text{where}$$

$$\mathbf{z}^+ \geq \mathbf{0}, z_{ij}^+ \geq Q_{ij} - q_i, z_{ij}^+ \geq -Q_{ij} - q_i,$$

With this formulation, local δ-DD matrix is SDP with regards to any vector with δ non-zero values.

By first optimizing over δ-local DD matrices as discussed above you make the solver replace all SDP cuts coming from vectors with at most δ non-zero values with the δ-local DD LP cuts. This should be easier,

unless we have too many z_{ij}. The figure is good; a part of the circle becomes a square. Once you find a feasible solution in the δ local DD matrices, you should increase δ, to go towards the fully SDP cone (you are inside if $\delta = 1$). Once you find the highest δ with a $\delta-$local DD solution, you start optimizing via cut planes, because you still have no feasible solution. However, you can do the above stuff even if you do have a feasible solution. In case you are given a feasible SDP matrix, multiply $\mathcal{A}^\top \mathbf{y} \preceq C$ left and right with the matrix of eigenvectors, to a have a DD matrix.

The approach can work even if that initial matrix is almost SDP. The last line of the instance should give an $\bar{\mathbf{y}}$ so that $\mathcal{A}^\top \bar{\mathbf{y}}$ is a purely SDP as possible. As long as you do not have a feasible solution you are condemned to simple Cutting Planes. As long as you have it, you will be smarter with Projective Cutting Planes. For both phases (with δ-local DD matrices or not), Proj Cut Planes should be better.

In this section we restricted the DD condition $S_{ii} > \sum_{j \neq i} |S_{ij}|$ to a weaker one. A similar idea is followed in Section 5.1 by restricting it using an α. You can combine the two approaches or try them both.

You should first not use these ideas, but implement the simplified approach (with no $\alpha \neq 1$ from Section 5.1 or $\delta \neq n$, namely let the user provide an $\bar{\mathbf{y}}$ such that $\mathcal{A}^\top \mathbf{y} \succ \mathbf{0}$, as a last optional line in the instance file.

5.3 No longer useful because α is replaced by δ α-DD means SDP with regards to α-dominant vectors

A strengthening idea. Take any A_i that is strictly SDP or SDN. Write eigendecomposition $A_i = PDP^\top$. Then replace $\overline{A}^\top y \geq C$ with replace $P^\top(\overline{A}^\top y)P \geq P^\top CP$ and solve it using pure LP in the DD sense. You will have a strictly feasible solution with $y_i > 0$ if A_i is SDN or $y_i < 0$ if A_i is strictly SDN. In such case, you really have no need of introducing a penalizing term like $A_{k+1} = I_n$ with a huge b_{k+1}. The if you relax the DD concept to have $S_{ii} > 0.9 \sum |A_{ij}|$, you obtain 0.9-DD. If S is 0.9-DD, this means S is SDP with regards to all eigenvectors $\mathbf{v}$ such that the maximum element of $\mathbf{v}$ is more than $0.9\cdot$ any other element. You say S is $\alpha - DD$ with $\alpha = 0.9$. If you start with $\alpha = 0$, you basically want S to have only positive stuff on the diagonal. This will fail. You take an eigenvector $\mathbf{v}$ that shows the failing and get a new $\alpha = maxElemInModule(\mathbf{v})/secondMaxModule(\mathbf{v})$. In the initial feasibility search stage, you increase the value of α. But you may also add cuts like in the rest of the project to remove S with eigenvectors that do not satisfy $\alpha = maxElemInModule(\mathbf{v})/secondMaxModule(\mathbf{v})$. At first $\alpha = 0$ simply means that $S_i i > 0$ for all i. As you progressively increase α, you will eventually make your S enter the SDP cone (if it is possible to find an S that is DD). Otherwise, you stop and be happy with an S that is something like 0.7-DD. This operation should give you some feasible S, at worst by penalizing using term $A_{k+1} = I_n$. But in the very beginning, be sure to give him some matrices a linear combination $\bar{\mathbf{y}}$ provided by so that $\mathcal{A}\bar{\mathbf{y}} \succ \mathbf{0}$. We will eigendecompose $P\mathcal{A}^\top \bar{\mathbf{y}} P^\top$ and apply the above trick with matrix P, so that $P^\top(\mathcal{A}^\top \bar{\mathbf{y}})P - P^\top CP$ is DD (more exactly minus that stuff is DD). Keep only one matrix in $\mathbf{y}$ in the Matlab version. It may also be almost SDP, for instance with one A^i_{jj} not being DD, but you may hope to cover it using other matrices. You can try this with all A_i and then focus on the one that leads to a feasible solution with the highest α. If $\alpha = 1$, the matrix is surely SDP, but a matrix with $\alpha < 1$ may still be sometimes SDP and then have a better objective. A DD matrix may by too far inside the SDP cone. But then you can project and obtain a better one. You do not disclose the α idea before but you simply allow an optional line in the instance file where you provide $\bar{\mathbf{y}}$. This does not mean you give a interior solution, because you do solve a DD LP and you never know what that solution will be. You know there will be one!

6 Computational aspects and matrix libraries

We list here the critique routines (potential bottlenecks) in reversed order of their complexity. Our goal is to tackle SDP programs with $n = 1000$. The discussion in terms of timing does not aim at being complete and accurate. It is still enough to discover some approximate rules of thumb or approximate guidelines. Exact or strictly accurate findings are not necessary for our objectives. We will express all computational costs in milliseconds (ms). Generally speaking, we pay a few ms for quadratic complexity operations (like copying a matrix or multiplying it by a number)

There are half-dozen critical routines that require more than $O(n^2)$.

The **eigendecomposition** can be done in 60ms using matlab. Cholesky is 10-20 times less expensive in matlab so we prefer to avoid eigendecomposition. This is consistent with all theory.. Eigen needs 1500ms.

Lapcack needs 600ms.

The **minimum eigenvalue** of a random matrix of size 1000×1000 costs 30-40ms using `spectra` and maybe 10% less using `matlab`. Matlab can produce the eigendecomposition of such matrix in 65 ms. Julia is at 500ms for min eigenvalue. Eigen is at 1500ms as above, as it only knows computing the full eigendecomposition. Lapack is at 600ms as above, as it only knows computing the full eigendecomposition.

Cholesky Matlab requires 4ms for above matrix ($+10000I_n$ to become SDP). Julia needs 13ms for the same. C++ Lapack needs 38ms and the naive C++ code is at 150ms. Eigen needs 32ms. If you want Matlab Cholesky, you may need 10-20ms to retrieve a unique vector. You can not retrieve the whole matrix, so the whole useful approach is one in which you never retrieve a whole vector of size n, but only values of y. All interior points in the primal can be written as linear combinations of vectors y.

matrix multiplication can take 15-20ms in Matlab. Eigen may need 100-150ms **if** an eigendecomposition was requested before or 900ms if called only once. Maybe it is ok to perform Mv for vector v.

back substitution is quadratic which should be a few ms. A bit related, Eigen can call lldtOfA(A).solve(b) to solve $Ax = b$ using Cholesky. You want $S = KK \implies$ find D' so that $D = KD'K$. But lldtOfA(A) has nice features to permute the rows so as to put the zeros in the bottom-right part. We will be able to solve $D = KD'K$ only in the upper part. And then check that the rest of the matrix can be extended so that $D = KD'K$ hold everywhere.

Get the null space of S Maybe not actually required if you check the steps below, maybe except if you use point 2.(a).

In C++, you may need 1.2ms to scan a 1000×1000 array and perform a division and an addition per element (or even to copy the matrix). But if you move to float you get 0.3ms. On the other hand, a $O(n^2)$ manipulation like the following may take 16ms in Matlab or 1ms in $C++$.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j <n; j++)
            if(b[i][j]>0)
                s+=1.2/b[i][j];
            else
                s-=300.11312;
cout<<"s="<<setprecision(9)<<s<<endl;
```

You will distribute several version of the software, depending on how critical matrix routines are implemented. At least one version will contain only pure $C++$ code. You can then move to a version with Lapack (very popular) and then one linking it to Julia (at least free) Matlab (licence not free, but you can distribute the Matlab code that users can turn into an executable).

There are countless libraries that could be used for various $O(n^3)$ operations. I would like:

1. Eigen with $C++$ for general manipulations
2. Spectra for min eigenvalue (almost faster than matlab)
3. Matlab for Cholesky (6 times faster than Eigen) and full eigendecomposition (seem faster than all else)

Before that, you write a first Matlab prototype. You will compare with C++ (or even Julia if you really have time) on that prototype. Only after that you think how you continue. You only apply teps below. And some initial cuts and that's all. If you really have time, you first solve the SDP primal in the space of DD matrices. If, *e.g.*, $A_1 + A_2 + A_3 = I_n$, the primal does have a solution that is strictly interior. Better first solve problems for which it is not difficult to find strictly interior points.

STEPS:

1. Decompose $S = KK^\top$ and solve $D = KD'K^\top$ in variables D'. If this is possible and we get either case (A) if K is full rank or (B) otherwise. Case (B) is the same as (A) but you project from a strictly interior point with regards to the relative interior of the orthogonal space of the null space of S (the row image of S)
2. If above fails, test the SDP status of $S + \epsilon D$ or $\frac{1}{\epsilon} \cdot S + D$. If S has a very small non-zero eigenvalue with eigenvector $\mathbf{v}$ you can test the SDP status of $\frac{1}{\epsilon} \cdot S + D + \frac{1}{\epsilon'} \cdot \sum \mathbf{v}\mathbf{v}^\top$, where the sum is carried out over all such $\mathbf{v}$. If the resulting matrix is not SDP, you have $t^* = 0$ because there is a $\mathbf{d}$ in the null space of S such that $D \bullet \mathbf{d}\mathbf{d}^\top < 0$.

(a) You can also solve the same problem by testing the SDP status of $N^\top DN$, where N contains the null space of S as columns. I have a feeling this may be more expensive than one of the above approaches that essentially only perform addition (supposing that computing $\mathbf{v}\mathbf{v}^\top$ is much less expensive than matrix multiplication and supposing that getting such vectors with the eigendecomposition is not too expensive either).

3. If you get here, you have $t^* > 0$. You are in case (D) and you move first to $S' = S + \epsilon D$ and project using cases (A) or (B) from S' towards D.

You first do in C++ the first two above points, using Eigen for Cholesky. The next step is to use Matlab for Cholesky. If you ever need eigendecomposition (later, hopefully) you use Matlab. If you need matrix multiplication a better solution than eigen would be useful.

If you ever send matrices from one solver to the other, you only send the factors of the decompositions, like $\mathcal{A}^\top y$, by only sending y. You actually use a class `ProjMat` . Since you only have one class, declare it either: `ProjMat_n` (native), `ProjMat_m` (matlab), `ProjMat_e` (eigen), `ProjMat_l` (lapack), or `ProjMat_m1` (mix1 a mix of multiple choices of libraries). The Makefile offers multiple compilation targets, one for each datatype above. The user will know: I could compile that or that version.

This matrix class has to redefine many operators, read data, load external matrices, load external vectors. Some operators can be written in clear text (like $M + t \cdot N + In$), very rarely. Such a method can perform the calculation twice, once inside to compute a real S inside. But also outside, for instance in Matlab. This way, you will not have to send huge matrices to Matlab; Matlab will compute internally $S = S + t^* D$.

You only rarely have to return a full matrix so you only rarely write $M = M + t \cdot N + I_n$. In the primal program in y you never touch a matrix in the main $C++$ code. Other operators have to be written with a method name like $M.S\_gets\_S\_plus\_t\_D()$.

```
%https://stackoverflow.com/questions/55379347/finding-k-smallest-eigen-values-and-its-corresponding-eig
%cica Lanczos e very effective.
%https://github.com/mrcdr/lambda-lanczos
%https://rcc.fsu.edu/software/arpack
%https://stackoverflow.com/questions/11064670/c-eigenvalue-vector-decomposition-only-need-first-n-vecto
%UPDATE: I managed to do it with ARPACK. I managed to compile it for windows, and even to use it. The r
%Spectra is a redesign of the ARPACK library using the C++ language.
%https://stackoverflow.com/questions/24468537/largest-eigenvalues-and-corresponding-eigenvectors-in-c
%http://eigen.tuxfamily.org/index.php?title=Main_Page
%https://github.com/yixuan/spectra/tree/master
%   - cica trebe numai header inclusion, including for eigen. maybe BEST
%                   NO: 100 slower then eig or eigs of matlab
%                  YES: when using -O3 is as good as matlab
%   - nu uita lapack
%maybe you install eigen, it contains some Arpack method that do work well for sym mats
%even if I am afraid I do need Arpack libraries
%https://scicomp.stackexchange.com/questions/26786/eigen-max-and-minimum-eigenvalues-of-a-sparse-matrix
%https://www.quora.com/What-is-the-C-program-to-find-eigenvalues-and-eigenvectors
%power method simple: https://www.codesansar.com/numerical-methods/power-method-using-cpp-output.htm
```

7 Three main stages

1. Get the most SDP matrix in the inner LP using heuristics.

   (a) You try the heuristic from Section 4.2. If success, maybe exit. Else remember the best SDP matrix of max `val`(S).

   (b) **Guessing** You define a set of directions, first starting with the n directions of the form $y_i = 1$ and $y_j = 0$ if $i \neq j$ for all $i \in [1..n]$. You can enrich the set of directions by solving the inner LP towards random objective functions. For each direction $\mathbf{y}$ you compute the SDP value of $S(\mathbf{y})$ as `val`(S):

   - sum of negative eigenvalues is these exists, or

- $nbOfStrictlyPositiveEigVals$ (which is $< n$) if the minimum eigenvalue if zero.
- $n + minEigenVal$ if $minEigenVal > 0$

I think in this heuristic stage we only maximize this $\texttt{val}(S)$.

From among all above matrices plus the optional $\bar{\mathbf{y}}$ submitted by the user, you take the most SDP one, of highest $\texttt{val}(S)$. Call it S_{bst}.

(c) **Constructive heuristic**

You left and right multiply $C - \mathcal{A}^\top \mathbf{y} \succeq \mathbf{0}$ with $P^\top$ and resp P, where P is the columns of the eigendecompositionof S_{bst}, after you modify P to transform that matrix into one with an equal diagonal. The DD cuts are more relevant on matrices with a more or less equal diagonal. You can even update P during the major stage 2 below before to make that stage work with more equal diagonals. It's easier to satisfy a DD cut if the data in the matrix is more or less uniform noisy when the matrix has a more or less an equal diagonal.

Apply the constructive heuristic as described when you search key word constructive heuristic, *i.e.*, you increase if you can the SDP dimension of $P^\top(C - \mathcal{A}^\top \bar{\mathbf{y}})P$, *i.e.*, the number of ones on the diagonal of this diagonal matrix. You repeat this as long as possible. P will be useful during stage 2.

(d) **Local Search** Then you move to an LS steps. You take random directions and you optimize the LP. For each direction hope to improve the SDP status of matrices like $S = S + step * direction$. You perform a number of steps in a simulated annealing manner, maximizing $\texttt{val}(S)$.

(e) $n-m$ **penalizing matrices** Let S_{bst} be the matrix of maximum `val` ever found above. Compute a P for it so that $P^\top S_{\text{bst}} P^\top$ is a diagonal matrix with some negative values on some diagonal positions and only ones as positive values on the diagonal. After left-multiplying $\mathcal{A}^\top \mathbf{y} - C$ with $P^\top$ and right multiplying with P, you will be able to construct a matrix that has m ones on the diagonal, but not everywhere. For each of the $n - m$ positions you consider a matrix that has zeros everywhere except one the diagonal element. These matrices are inserted in $\mathcal{A}^\top \mathbf{y}$ with huge penalties in the objective.

2. Optimize over σ-local SDP matrices, with decreasing the value of σ. Optimizing LPs over such sets will give you outer solutions. You project from the solution returned by Stage 1 to each outer solution. Optimize by Projective Cutting Planes the program associated with the best σ. The lower we make σ, the more cut planes work we have to do but the more chances there are to finish with a non-penalized SDP matrix. (when σ is too large, the σ-local DD LP area excludes too much of the SDP cone).

   - if $\sigma = n - 2$ a σ-DD matrix has very high chances to be SDP. When you optimize over $\sigma-$DD matrices, you only need to add cutting planes associated with vectors with more than $\sigma + 1$ non zero elements (σ-DD means you cover all vectors with something on diagonal S_{ii} larger than the sum of the σ highest value on row i of S), meaning at least $\sigma + 2 = n$ non zero elements. When implementing check if the resulting vectors do not always have a very low value somewhere.
   - if $\sigma = 0.1$, you have very very few SDP matrices that are not σ-DD.

   When σ is as low as 1 and you also impose an equal diagonal constraint, you have a particular phenomenon: the σ-local DD constraints cut nothing of the SDP cone. On the other hand, the equal diagonal constraint does cut a part of the SDP cone. When σ is large and you perform cut planes, you may end up with no non-penalized solution (see image 2 of A_i and C in the figure). For each sigma (but, most importantly, for $\sigma = 1$) you also try the equal diagonal constraint to see if you get a better matrix. But do not forget you can always project from the interior solution towards it. If you apply this stage with no feasible interior solution known, do not forget you can project towards it from an SDP solution that does not satisfy the whole inner LP, and maybe you get something.

   You will eventually reach $\sigma = 0$, which means you are at stage 3 below.

   Note: It is only here that you can think of using $n \times n$ variables (S or something else).

3. Full Proj Cut Planes, starting from the best interior point discovered above, and keeping all cut planes. But it is very useful to use above stuff because starting with a virgin set of SDP cuts is really not a good idea. This actually corresponds to $\sigma = 0$. If you have $\sigma = 0.1$, this means that $S_{ii} \geq 0.1\times$ the highest value among S_{ij} with $j \neq i$.

Again, with $\sigma = 0$ you include all the inner LP, the intersection of the SDP cone with image of A_i and C. But the resulting outer approximation that you have to cut with cutting planes is huge (ai mult de cioplit). As you increase σ (or as equal diagonal), you decrease that huge outer approximation, but you also lose some of the feasible area. Ai mai putin de cioplit (less to cutting planes work), but you are no longer sure to find a feasible solution. Still, you may find useful very global cuts. It would be shame to spend time with very local cuts during a bang-bang chaotic period in the beginning. Astea cizeleaza la sfarsit. When σ is high, the σ-local SDP cuts should be satisfied.

I think you should always maximize the original objective function because this way you will search the current feasible are of good solutions so that the generated cuts become relevant. It is true that a different function may push you towards the inner LP, but you can now do that using projections. Yet, since you are desperate for a feasible solution, you can try at stage 2 to optimize a different function from time to time.

7.1 How to implement it

The implementation is done in reversed order of the main steps, first testing $C = I_n$ and random matrices with all elements in $[0, 1]$, maybe with higher values on the diagonal. When you go back along the steps, you search for more particular matrices that may suit the objectives of stages 1 or 2. You also use larger arrays of variables to suit all you need, more and more.

Still, it is easy to implement 1.(a). You can even put some $y_i \geq 2$ and still find an interior solution with 1.(a). Because you do not want to say you start with $\mathbf{y} = \mathbf{0}$; the stuff may work even if $\mathbf{y}$ is not feasible.

When you work with random matrices, you can only project only using case (A) or a bit case (D).

Impose $y_i \geq 0 \forall i$ to use valid inequalities For this case you use Section 4.2.1. At the end of that section, you will see how you can generalize this for any $\mathbf{y}$. However, to make these things bring any result in practice you may need the A_i to have only positive eigenvalues. And these valid inequalities are of higher quality if the eigenvalues of each A_i are very grouped close together. Do not forget that Section 4.2 can be used both for strengthening and for relaxing the main feasible area. If necessary, try to obtain the relaxation as described in the last line of 4.2.1. Using $\mathbf{y} \geq 0$ makes you not require linearizing the module with new variables $\bar{\mathbf{y}}$ in the strengthening part.

- Think of using strengthening-relaxing (or restricting-relaxing approach) using this instead of doing that with costly DD matrices at main stage 2.

This, including the valid inequalities is the first implementation, in Matlab and $C++$ asking Matlab from time to time to provide a Cholesky. You work only with all full-dimensionally matrices.

The instance below seems interesting for the method that generates valid inequalities.

7.1.1 A modified instance to try to move to $n = 1000$, after identifying the bottleneck in the above fully-dimensional case

We will mainly work with Cholesky that is fast for $n = 1000$ Back-substitution is fast in matlabl (and in theory it has $O(n^2)$ serial complexity and $O(n)$ in for the parallel version), when solving something like $A\mathbf{v} = \mathbf{v}'$

In a second stage of the implementation, you start to compute minimum eigenvalues ideally like in case (B) on a matrix D'_r of a smaller order.

Imagine how this may work well when all matrices C and A_i have a large common null space. Or if at least one of them A_i does not have it, but you have to set $y_i = 0$ because of a huge negative objective coefficient b_i. You can start with a problem with $\mathbf{y} \geq \mathbf{0}$ and make it so the optimal solution is $y_1 = 15,\ y_2 = 0, y_3 = 0, \ldots, y_k = 0$. Maybe you need a unique projection (because the LP logic will not search to increase any other y_i). Start with the following with some random $\mathbf{v}_2, \mathbf{v}_3, \ldots \mathbf{v}_k$.

$$\max\{1000 \cdot y_1 - 1000 \sum_{i=2}^{k} y_i : y_i I + \sum_{i=2}^{k} y_i \mathbf{v}_i \mathbf{v}_i^\top \preceq 9 I_n, \mathbf{y} \geq \mathbf{0}\}$$

During the cutting-planes you only need computing the min eigenvalue (of D') and not the whole eigen-decomposition to find t^* using Lanczos.

You can naturally start from $\mathbf{y} = \mathbf{0}$ and project towards $\mathbf{b}$ if $\mathbf{y}$ is free. If you have $\mathbf{y} \geq 0$ you project towards $\bar{b}_1, \bar{b}_2, \ldots \bar{b}_n$ where $\bar{b}_i = b_i$ if $b_i \geq 0$ or $\bar{b}_i = 0$ otherwise. For above program, you would start towards the good direction $\bar{b}_1, \bar{b}_2, \ldots \bar{b}_n = 1000, 0, 0, \ldots, 0$.